QUERY SCHEDULING AND OPTIMIZATION IN PARALLEL AND MULTIMEDIA DATABASES

By

Minos N. Garofalakis

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCE)

at the

UNIVERSITY OF WISCONSIN – MADISON

1998

Abstract

Effective resource management support for parallelism and multimedia data is an important mandate for next-generation information systems. In this thesis, we address a number of resource scheduling issues that arise in the context of query processing and optimization in parallel and multimedia databases. Our contributions to the area of parallel databases include the development of a multi-dimensional framework and provably near-optimal algorithms for scheduling both Time-Shared and Space-Shared resources in hierarchical and shared-nothing architectures. We also present results from the implementation of our algorithms on top of a detailed simulation model that verify their effectiveness in a realistic setting. Based on our scheduling results, we identify a novel cost model for parallel query optimization that manages to capture all the important execution characteristics of a parallel query plan in only three "bulk" parameters. In the field of multimedia databases, our contributions span the areas of resource scheduling for composite multimedia objects, on-line admission control for multimedia databases, and scheduling support for periodic (i.e., "Pay-Per-View"-style) models of user service. In all cases, we propose novel algorithmic formulations and solutions and we study their performance both analytically and experimentally.

Acknowledgements

Looking back on my five-year graduate life (time-sliced between Madison and Murray Hill) the names of many people that, in one way or another, helped make this thesis possible come to mind. Among all these names, there is one that clearly stands out above the rest – that of my advisor Yannis Ioannidis. Yannis has been possibly the sole person responsible for some of the best decisions that I have made over the past five years, including choosing to join the UW-Madison graduate program and deciding to work in the area of database systems. As a researcher, Yannis has been a true source of inspiration throughout our collaboration. His depth and breadth of scientific knowledge and his hard work ethic, combined with his unique ability to just "take a step back" and make sense of *really* messy problems have never ceased to amaze me over the years. I can only hope that some of all his qualities have rubbed off on me... But Yannis has been a lot more to me than just an academic advisor and a hard-working collaborator – he has been a true friend. Our professional (and social) interactions (not to mention, our weekly basketball battles) are some of the things that I've really missed since Yannis' decision to move back to Greece. I am truly proud to be counted among his "academic children" and, hopefully, among his friends.

I would also like to extend my sincerest thanks to Banu Özden and Avi Silberschatz. During my extended visits to Bell Laboratories, Banu and Avi were gracious enough to take me "under their wing" and provide a challenging and, at the same time, friendly environment for me to work in. I am grateful to Banu and Avi for believing in me. I would also like to thank professors Jeff Naughton, Anne Condon, Raghu Ramakrishnan, and Rafael Lazimy for serving on my thesis committee.

Several friends deserve special mention in this thesis for different reasons. Andreas Moshovos, for always being there to answer my "low-level" systems-related questions, for countless basketball games, and for his hospitality during my last days in Madison. Rajeev Rastogi, Thimios Panagos, and Alex Biliris, for putting some interest to those long, dull New Jersey weekends. Vishy Poosala and Chee-Yong Chan, my officemates on the UW "database floor", for patiently sitting through several of my practice talks and putting up with my (many) idiosyncrasies. Dionysis and Natalia Pnevmatikatos, for freely offering me their help and friendship during my first years in Madison. Natassa Ailamaki, for providing the energy and willingness to help me out in various situations. Thank you all!

This thesis would not have been possible without the support and encouragement of three special people: my father Nikos, my mother Xanthippe, and my sister Katerina. Their unconditional love has been the one true constant throughout my life. This thesis is dedicated to them.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This thesis is concerned with resource scheduling issues that arise during query processing and query optimization in parallel and multimedia database systems. Judicious scheduling of available resources is crucial to obtaining fast response times and effective utilization in computing systems. As a consequence, algorithms for scheduling have been extensively researched since the 1950's both in theory and in practice. Parallel database systems attempt to exploit recent multiprocessor architectures and combine database management with parallel processing in order to build scalable high-performance database servers. Multimedia database systems take advantage of recent advances in computing, communication, and storage technologies to provide database functionality for multimedia data, such as images, graphics, audio, video, and animation, in a number of diverse application domains. Examples of such domains include digital libraries, satellite image archival and processing, training and education, entertainment, and medical databases containing X-rays and MRIs. The introduction of parallelism and multimedia data raises a host of novel resource scheduling challenges for the query processing and optimization component of a database system. In Sections 1.1 and 1.2, we concentrate on parallel and multimedia database systems, respectively, focusing on some of these novel challenges. Our goal is to explore the state of the art, identifying problems that motivated the work in this thesis, which is oultined in the subsequent sections.

1.1 Parallel Database Systems

Parallelism has been recognized as a powerful and cost-effective means of handling the projected increases in data size and query complexity in future database applications. Among all proposals, the shared-nothing [DG92] and, recently, the more general hierarchical (or, hybrid) [NZT96, BFV96] multiprocessor architectures have emerged as the most scalable to support very large database management. In these systems, each *site* consists of its own set of local resources and communicates with other sites only by message-passing (Figure 1). Despite the popularity of these architectures in both research and commercial environments, the development of effective and efficient query processing and optimization techniques to exploit their full potential still remains an issue of concern [GHK92, Val93]. To date, our work on parallel database systems has focused on the resource scheduling problems arising during the optimization of complex declarative queries for parallel execution. More specifically, we have concentrated on the design of *multi-dimensional* query scheduling algorithms for effective, coordinated allocation of multiple resources in hierarchical systems. In the remainder of this section, we describe the interaction between scheduling and query optimization in parallel database systems and we motivate the choice of a multi-dimensional problem framework. We also briefly describe the major shortcomings of earlier approaches. (Related work is discussed extensively in Chapter 2.)

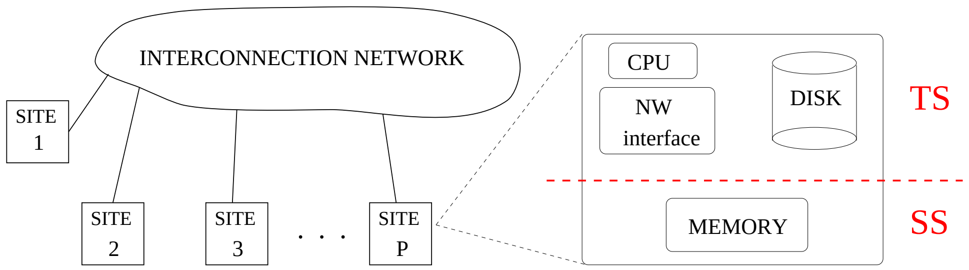


Figure 1: A hierarchical parallel database system

Perhaps the major difference between parallel query optimization and its well-understood centralized counterpart lies in the choice of *response time* as a more appropriate optimization metric. This choice of metric implies that the optimizer has to be able to quantify the impact of parallel execution on the response time of query execution plans. This suggests that it may not be a good idea to ignore resource scheduling during the optimization process. Prior work has demonstrated that divorcing the two will often result in a clearly suboptimal plan [JPS93, BFG$^+$95]. For example, using the traditional work (i.e., total resource consumption) metric can often result in plans that are inherently sequential and, consequently, unable to exploit the available parallelism.

On the other hand, using a detailed scheduling model can have a profound impact on optimizer complexity and optimization cost. To avoid this penalty, many systems have opted for a *two-phase* optimization approach. The idea is to divide the optimization process into a *join ordering* phase that creates a least work plan using conventional query optimization techniques, and a *parallelization* phase that schedules the plan at run-time [HS91, Hon92]. This approach obviously reduces the optimization cost significantly since the optimizer only needs to explore the parallelizations of a single plan instead the parallelizations of all possible plans. As we explained earlier, however, this reduction in optimization cost may come at the price of selecting highly suboptimal plans. Nevertheless, even in this case, effective query scheduling algorithms are still necessary for distributing the execution of the plan on the run-time environment during the parallelization phase [HM94, CHM95]. Hence, resource scheduling techniques form an important component of any approach to query processing and optimization in parallel database systems.

As a result, significant research effort has concentrated on the difficult problem of minimizing the response time of a single query through *parallelization* of an execution plan, i.e., scheduling of the plan's operators on the system's sites [CHM95, GW93, HM94, Hon92, HCY94, LCRY93]. Most of these efforts, however, are based on simplifying assumptions that limit their applicability. One of the main sources of complexity of query plan scheduling is the *multi-dimensionality* of the resource needs of database queries. That is, during their execution queries typically require multiple resources, such as memory buffers and CPU and disk bandwidth. This introduces a range of possibilities for effectively scheduling system resources among concurrent query operators, which can substantially increase the utilization of these resources and reduce the response time of the query. Moreover, system resources can be categorized into two radically different classes with respect to their mode of usage by query plan operators:

1. Time-Shared (TS) (or, preemptable) resources (e.g., CPUs, disks, network interfaces), that can be sliced between operators at very low overhead [GHK92]. For such resources, operators specify an amount of work (i.e., the effective time for which the resource is used) that can be stretched over the operator's execution time.

2. Space-Shared (SS) resources (e.g., memory buffers), whose time-sharing among operators introduces prohibitively high overheads [GHK92]. For such resources, operators typically specify rigid capacity requirements that must be satisfied throughout their execution.

Most previous work on parallel query scheduling has typically ignored the multi-dimensional nature of database queries. It has simplified the allocation of resources to a mere allocation of processors, hiding the multi-dimensionality of query operators under a scalar cost metric like "work" or "time" [CHM95, GW93, HM94, HCY94, LCRY93]. This one-dimensional model of scheduling is inadequate for database operations that impose a significant load on multiple system resources. With respect to SS resource allocation, all previous work has concentrated on simplified models, assuming that all SS resources are *globally accessible* to all tasks [GG75, ST94, NSHL95, CM96]. Clearly, such models do not account for the physical distribution of resource units or the possibilities of SS resource fragmentation. This limits the usefulness of these models to a shared-everything [DG92] context.

1.2 Multimedia Database Systems

With all the euphoria surrounding the potential benefits of the multimedia revolution, database researchers are faced with challenges that are pushing the current hardware and software technology to its limits. The fundamental problem in developing high-performance multimedia database servers is that images, audio, and other similar forms of data differ from conventional alphanumeric data in their characteristics, and hence require different techniques for their organization and management. A fundamental issue is that digital video and audio *streams* consist of a sequence of media quanta (video frames or audio samples) which convey meaning only when presented continuously in time. Hence, a multimedia database server needs to provide a guaranteed level of service for accessing such *continuous media* (CM) streams in order to satisfy their pre-specified real-time delivery rates and ensure *intra-media continuity*. Given the limited amount of server resources (e.g., memory, disk bandwidth, disk storage), it is a challenging problem to design effective resource scheduling algorithms that can provide on-demand support for a large number of concurrent continuous media clients. To date, our work on query processing in multimedia database systems has focused on (a) resource scheduling algorithms for composite multimedia objects, (b) on-line admission control for CM requests, and (c) scheduling solutions for supporting periodic (i.e., pay-per-view-like) service models.

1.2.1 Composite Multimedia Objects

An important requirement for multimedia database systems is the ability to dynamically compose new multimedia objects from an existing repository of CM streams. Temporal and spatial primitives specifying the relative timing and output layout of component CM streams provide perhaps the most powerful and natural method of authoring such *composite* multimedia presentations. Thus, to compose tailored multimedia presentations, a user might define temporal dependencies among multiple CM streams having various length and display bandwidth requirements. For example, a story for the evening news can start out by displaying a high resolution video clip with concurrent background music and narration added after an initial delay (Figure 2(a)). After some time into the story, the video screen is split and a new video clip starts playing on the left half of the screen. After the second video clip ends, the narration stops and the story comes to a conclusion with the display of the first clip and the background music.

In the presence of such composite multimedia objects, a scheduling algorithm must ensure that the *inter-media synchronization* constraints defined by the temporal relationships among CM components are met. Handling these synchronization constraints requires a task model

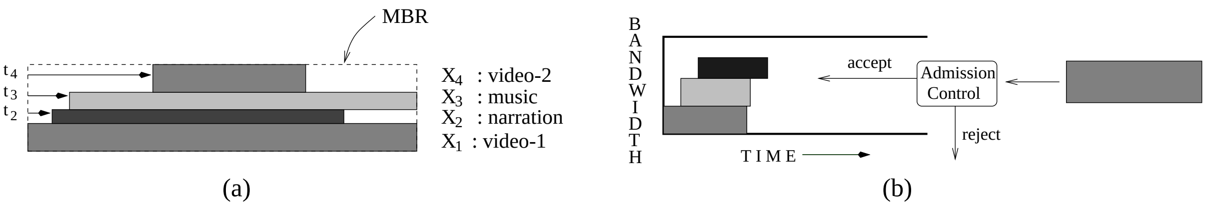


Figure 2: (a) A composite multimedia object. (b) The role of admission control.

that is significantly more complex than the models employed in scheduling theory and practice [CM96, GG75, GGJY76]. Furthermore, despite the obvious importance of the problem for multimedia database systems, our work appears to be the first systematic study of the problems involved in scheduling multiple composite multimedia objects. We suspect that this is due to the difficulty of the problems, most of which are non-trivial generalizations of $\mathcal{NP}$-hard optimization problems. Finally, note that although our discussion in this thesis is primarily geared towards composite objects, our task model also exactly captures the problem of scheduling the retrievals *variable bit rate* streams, that is, CM streams whose bandwidth requirements can vary over time. This is also a very important application of our scheduling framework, since real-life CM data is nearly always variable rate.

To the best of our knowledge, none of today's multimedia storage servers offer any clever scheduling support for composite multimedia presentations. The approach typically employed is to reserve server resources based on the *maximum (i.e., worst-case)* resource demand over the duration of a composite presentation. Examples of systems using this worst-case resource reservation method include the $\mathcal{F}$*ellini* and CineBlitz multimedia storage servers developed at Bell Labs [MNÖ+96], Starlight's StarWorks (`http://www.starlight.com/`), and Oracle's Media Server (`http://www.oracle.com/`). Conceptually, this approach is equivalent to identifying the resource requirements of the presentation over time with their enclosing *Minimum Bounding Rectangle (MBR)*. Although this simplification significantly reduces the complexity of the relevant scheduling problems, it suffers from two major deficiencies.

1. The *volume* (i.e., resource-time product [CM96]) in the enclosing MBR can be significantly larger than the actual requirements of the composite object. This can result in wasting large fractions of precious server resources, especially for relatively "sparse" composite objects.

2. The MBR simplification "hides" the timing structure of individual streams from the scheduler, making it impossible to improve the performance of a schedule through clever use of memory buffers.

1.2.2 On-line Admission Control

Consider a database server storing a collection of CM *clips* (i.e., contiguous portions of audio or video) and a set of on-demand clients issuing requests for the playback of specific CM clips. Given the limited amount of server resources, providing service level guarantees for CM data mandates an *admission control* mechanism, which is invoked whenever a new request arrives to decide whether to accept or reject the request (Figure 2(b)). By accepting a request, the server commits to satisfy the resource requirements (e.g., disk bandwidth, memory) of the corresponding playback stream throughout its execution, whereas rejected requests must pursue a different course of action (depending on the application)[1]. The effectiveness of the admission control component is of vital importance for the following reasons. First, the resource requirements of CM applications are high. Second, they require fractions of the server's resources to be reserved to meet their stringent performance requirements. Third, these applications tend to last for relatively long periods of time. Reserving large portions of the resources for long durations can result in drastic degradation of server utilization if the server makes wrong decisions whom to admit.

An important characteristic of admission control is the introduction of an *on-line decision making* element – the decision of whether to accept or reject a request has to be made without any knowledge of future requests, with the understanding that once a request is accepted, it is guaranteed a level of service throughout its duration (i.e., the schedule is *non-preemptive*). Despite its obvious implications, the on-line nature of the admission control problem has, for the most part, been ignored by prior work on multimedia servers.

1.2.3 Periodic Service

The finite amount of resources available in a multimedia database system obviously places a hard limit on the number of CM streams that can be simultaneously delivered. For many application domains (e.g., Movies-on-Demand), a multimedia server typically needs to sustain levels of concurrency that far exceed the limits imposed by available resource capacities. This means that the conventional *Random Access* (or, *Fully Interactive*) service model that places resource reservations to allocate independent physical channels for each client, cannot possibly provide cost-effective solutions in such environments [ÖBRS94, ÖBRS95]. As a result, several *data sharing* techniques have been proposed in the literature for increasing the number of concurrent clients beyond the capacity limitations of available resources:

[1] Our model corresponds to the *Full-VOD* service model [AGH95, BNGHA96, LV95]. Other service models have also been explored in the literature (see Section 1.2.3).

- *Batching* [AWY96a, AWY96b, DSS94, SY95] allows several clients waiting in the server's queue for the same CM clip to share the same stream.

- *Buffering* (or *Bridging*) [KRT95, ÖRSM95, SG97] uses extra memory buffers in a controlled fashion to allow requests that arrive with a small phase difference with respect to the start of a stream to fetch their data blocks directly from memory (i.e., with no disk access).

- *Piggybacking* [GLM96] allows clients to view the same clip at different display speeds so that, eventually, they can catch up with each other and share the same stream.

- *Enhanced Pay-Per-View (EPPV)* (or *Periodic*) service [ÖBRS94] assigns each clip a *retrieval period*, typically determined by the clip's popularity. Streams retrieving a clip are initiated periodically at offsets equal to the clip's retrieval period and multiple clients can share the same stream[2]. A graphical comparison of EPPV and Random Access service is depicted in Figure 3.

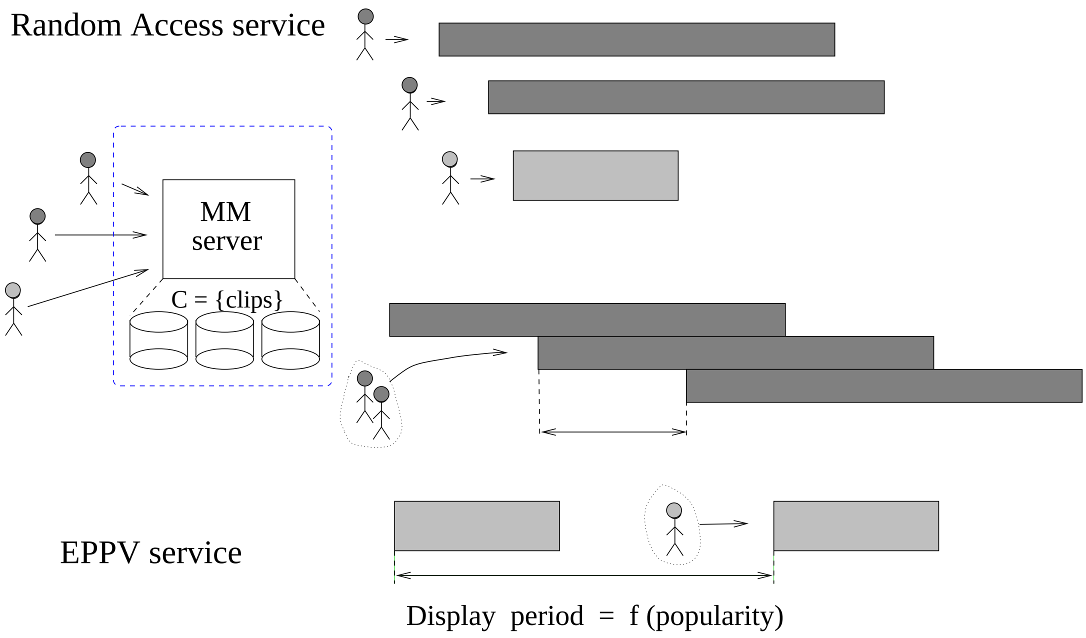


Figure 3: Random Access service vs. EPPV service

Work on batching has typically concentrated on different disciplines for scheduling requests from the server's queue. The goal is to strike a balance between: (1) fairness, (2) minimizing average waiting time, and (3) minimizing client reneging probability (i.e., the probability

[2]Depending on the underlying networking technology, clips can be delivered to clients via uni, multi, or broadcast channels.

that a client's request is cancelled due to excessive waiting). Scheduling disciplines like FCFS, Maximum Queue Length (MQL), and Maximum Factored Queue Length (MFQ) have been proposed and evaluated using simulation models [AWY96a, DSS94]. The main problem with these batching policies is that they are *unpredictable*, in the sense that they cannot offer a guaranteed upper bound on how long a client request must wait in the queue. Furthermore, results on the behavior of such policies (either analytical or by simulation) are typically based on specific probabilistic models of "customer reneging behavior" whose accuracy is often questionable in practice. Buffering is based on the idea that it is possible to trade extra memory for reduced bandwidth demand. This is a very general approach that is orthogonal to other data sharing schemes and, consequently, can be incorporated into batching, piggybacking, or EPPV, as an additional optimization (e.g., to facilitate VCR functionality). However, it should be noted that with current hardware pricing and stream parameters, trading memory for disk bandwidth is often a losing proposition [LLG97]. Piggybacking uses the fact that small differences in the display rates (e.g., deviations of at most 5% from the normal display rate) are usually not noticed by the typical viewer. However, difficult implementation issues arise when MPEG-like compression is used, because of inter-frame dependencies.

Compared to other data sharing schemes (most notably batching), EPPV service offers the advantage of *predictability* – the response time for transmission of a clip to a client is bounded by the clip's retrieval period[3]. This retrieval period is typically determined by the service provider, based on factors such as clip popularity, legal/financial constraints on the distribution rights of a clip, and specific programming choices. Because of the regular pattern of clip retrievals, clients can be informed of the exact time that a specific transmission will start. Thus, even when resources are scarce the EPPV service model can guarantee *predictable* response times for all incoming requests. An additional benefit of the regularity of EPPV service is that it can also support user interaction through VCR-like operations [AA96, ÖRS96b]. From the service provider's perspective, a desirable feature is that it simplifies the periodic scheduling of live events, such as news and sports events, into the program.

Because of the advantages outlined above and its potential to provide scalable, cost-effective CM offerings, EPPV is becoming the service model of choice for telecom, cable, broadcast, and content companies [PRE]. Realizing this potential, however, requires schemes for effectively scheduling the available disk bandwidth and storage capacity so that high levels of concurrency and system utilization can be sustained. Two phenomena make this a challenging problem — the periodic nature of EPPV service and the relatively high latencies of magnetic disk storage.

[3]In existing television terminology, the term "Pay-Per-View" refers to both a prescheduled playback program and a certain pricing mechanism. We refer to our service model as EPPV to emphasize the scheduling aspect of the service without constraining its pricing mechanism.

The periodicity of clip retrievals in EPPV servers generates a host of difficult periodic task scheduling problems that fall within the realm of hard real-time scheduling theory [LL73]. The high disk latencies complicate effective utilization of disk bandwidth and storage with reasonable amounts of buffer space, which is an important cost factor in multimedia server design [ÖRS95b]. The use of multiple disks to handle the high storage volume and bandwidth requirements of CM data exacerbates the problem. Thus, the need for intelligent scheduling mechanisms becomes more pronounced as the scale of the system increases.

1.3 Thesis Contributions

In this section, we outline the key research contributions of this thesis. We also identify the important dimensions of the search space of the query scheduling problem for parallel and multimedia databases and the portions of that search space explored in this thesis.

1.3.1 Parallel Query Scheduling and Optimization

For simplicity of presentation, we start out by presenting a framework for multi-dimensional resource scheduling in shared-nothing and hierarchical parallel database systems with sites consisting of TS *resources only*. Using this framework, we develop an approach that is significantly more general than earlier work, capturing all forms of intra-query parallelism and exploiting sharing of multi-dimensional TS resource sites among concurrent plan operators. This allows scheduling a set of independent query tasks (i.e., operator pipelines) to be seen as an instance of the *multi-dimensional bin-design problem* [CGJ84]. Using a novel quantification of coarse grain parallelism, we present a list scheduling heuristic algorithm that is *provably near-optimal* in the class of coarse grain parallel executions (with a worst-case performance ratio that depends on the number of resources per node and the granularity parameter). We then extend this algorithm to handle the operator precedence constraints in a bushy query plan by splitting the execution of the plan into synchronized phases. Experimental results based on analytical cost models for various database operators confirm the effectiveness of our scheduling algorithm compared both to previous approaches and the optimal solution. Finally, we consider the more general *malleable* scheduling problem [Lud95] in which the degree of parallelism for operators is no longer constrained by a coarse granularity condition. Building on the ideas of Turek et al. [TWY92], we propose a preprocessing step with which our list scheduling method is provably near-optimal in the space of all possible parallel schedules.

Having set the stage with the simple (TS only) case, we then go on to extend our problem formulation to address the co-existence of time- and space-sharing and present a general framework for TS *and* SS resource scheduling in hierarchical parallel database systems. We develop a general approach capturing the full complexity of scheduling distributed multi-dimensional resource units for all forms of parallelism within and across queries and operators. We present a level-based list scheduling heuristic algorithm for independent query tasks (i.e., operator pipelines) that is provably near-optimal for given degrees of partitioned parallelism (with a worst-case performance ratio that depends on the number of TS and SS resources per site and the granularity of the clones). We also provide extensions to handle precedence constraints in bushy query plans as well as on-line task arrivals (e.g., in a dynamic or multi-query execution environment). Once again, experimental results based on cost model computations confirm the effectiveness of our algorithms compared to the optimal solution. Based on our analytical and experimental results, we revisit the open problem of designing efficient cost models for parallel query optimization and propose a solution that captures all the important parameters of parallel execution.

Finally, we present a set of results from the implementation of our resource scheduling strategies within a detailed simulation environment for shared-nothing and hierarchical database systems based on the Gamma parallel database system [Bro94].

1.3.2 Scheduling Composite Multimedia Objects

We formulate the resource scheduling problems for composite multimedia objects and we develop novel efficient scheduling algorithms, drawing on a number of techniques from pattern matching and multiprocessor scheduling. Our formulation is based on a novel *sequence packing problem*, where the goal is to superimpose numeric sequences (representing the objects' resource needs as a function of time) within a fixed capacity bin (representing the server's resource capacity). Given the intractability of the problem, we propose heuristic solutions using a two-step approach. First, we present a "basic step" method for packing two composite object sequences into a single, combined sequence. Second, we show how this basic step can be employed within different scheduling algorithms to obtain a playout schedule for multiple objects. More specifically, we present an algorithm based on Graham's list scheduling method [Gra69, GG75] that is provably near-optimal for *monotonic* object sequences. We also suggest a number of optimizations on the base list scheduling scheme. Experimental results with randomly generated composite objects confirm the effectiveness of our approach.

1.3.3 Throughput-Competitive Admission Control

We explore the implications of the on-line nature of the admission control problem which have, for the most part, been ignored in the multimedia literature. Employing *competitive analysis* techniques [ST85], we address the problem in its most general form with the following key contributions: (1) we prove a tight upper bound on the competitive ratio of the conventional Work-Conserving ($\mathcal{WC}$) policy, showing that it is within a factor $\frac{1+\Delta}{1-\rho}$ of the optimal clairvoyant strategy that knows the entire request sequence in advance, where Δ is the ratio of the maximum to minimum request length (that is, time duration), and ρ is the maximum fraction of the server's bandwidth that a request can demand; (2) we prove a lower bound of $\Omega(\frac{\log \Delta}{1-\rho})$ on the competitive ratio of *any deterministic or randomized* admission control scheme, demonstrating an exponential gap between greedy and optimal on-line solutions; (3) we propose simple deterministic schemes based on the idea of *bandwidth prepartitioning* that guarantee competitive ratios within a small constant factor of $\log \Delta$ (i.e., they are *near-optimal*) for sufficiently large server bandwidth; (4) we introduce a novel admission control policy that partitions the server bandwidth based on the expected popularities of different request lengths and present experimental results that demonstrate the benefits of our policy compared to $\mathcal{WC}$.

1.3.4 Resource Scheduling Support for Periodic Service

We provide a comprehensive study of the resource scheduling problems associated with supporting EPPV for CM clips with (possibly) different display rates, frequencies, and lengths. In particular, given a collection of clips to be scheduled, we present schemes for determining a schedulable subset of clips under different assumptions about data layout (Clustering, Striping). Our main objective is to maximize the amount of disk bandwidth that is effectively scheduled under the given layout and storage constraints. Our formulation gives rise to $\mathcal{NP}$-hard combinatorial optimization problems that fall within the realm of hard real-time scheduling theory. Given the intractability of the problems, we propose novel heuristic solutions with polynomial-time complexity. We also present experimental results for the average case behavior of the proposed scheduling schemes and examine how they compare to each other under different workloads. A major contribution of our work is the introduction of a robust scheduling framework that, we believe, can provide solutions for a variety of realistic EPPV resource scheduling scenarios, as well as any scheduling problem involving regular, periodic use of a shared resource.

1.3.5 Parallel and Multimedia Query Scheduling: Problem Dimensions

Table 1 summarizes the important dimensions of the query scheduling problem for parallel and multimedia databases, indicating the regions of the search space explored in this thesis. In the remainder of this section, we provide some additional comments on specific entries of Table 1 that, we feel, deserve further explanation. Table 1 will be revisited in the closing chapter of this thesis, when summarizing our contributions and discussing future research directions.

	Parallel DB Systems	**Multimedia DB Systems**
Multi-dimensionality	$\surd$	$\surd$
Malleability	$\surd$	
On-line scheduling	$\surd$	$\surd$
Time-varying resource demands		$\surd$
Admission control		$\surd$
Periodic service		$\surd$
Impact on Query Optimization	$\surd$	

Table 1: Dimensions of the Query Scheduling Problem

The issue of *multi-dimensionality*, i.e., concurrent management and allocation of multiple systems resources, plays a central role in our work on parallel query scheduling and optimization. With respect to multimedia query scheduling, most of the results presented in this thesis focus on the disk bandwidth resource, which has typically been the bottleneck for multimedia server and database applications [RV91, ÖBRS94, ÖRS95a]. We do, however, explore some issues of concurrent disk bandwidth and memory allocation in Chapter 6.

Malleable resource demands describe a scheduling scenario in which the scheduling algorithm is allowed the additional flexibility of trading off resources for time or, in a more general setting, some types of resources for others. For example, when scheduling a parallel query plan operator it may be possible to increase its degree of parallelism and reduce its response time at the cost of increasing the operator's overall resource consumption (because of increased startup and communication overheads) [Lud95, TWY92]. We address some of the malleable resource scheduling problems in the context of parallel query scheduling in Chapter 3.

On-line scheduling algorithms are necessary to deal with situations where the collection of jobs to be scheduled arrives over time and is not entirely known beforehand. We explore on-line scheduling issues in the context of both parallel and multimedia query scheduling. It is also important to note that many of the suboptimality bounds proven in this thesis for an off-line setting can be directly extended to on-line job arrivals with only a factor of 2 increase in the bound, thanks to a recent result of Shmoys, Wein, and Williamson [SWW95].

Finally, the impact of query scheduling on *query optimization* is another important problem dimension that this thesis explores mostly in the context of parallel database systems. Research on query optimization for multimedia databases is still in its infancy, with no universally accepted problem models or solutions. Thus, it is very difficult to assess how our results on multimedia query scheduling will affect the query optimization process. We do, however, offer some ideas on this issue in the closing chapter of this thesis.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 reviews related work in the areas of database systems, scheduling, and on-line algorithms. In Chapter 3, we initiate our study of the parallel query scheduling problem by considering the simpler version of the problem that assumes *only* TS resources. The full version of the problem is then addressed in Chapter 4, where the implications of our results for parallel query optimization are also explored. Chapter 5 discusses our experimental findings from the implementation of our query scheduling algorithms in a detailed simulation model of a parallel database system. In Chapter 6, we present our work on the problem of composite multimedia object scheduling. Chapter 7 discusses our results on on-line admission control for CM database systems. In Chapter 8, we address the problem of supporting periodic service in CM databases. Finally, Chapter 9 concludes the thesis and identifies directions for future research. Proofs of theoretical results presented in this thesis can be found in Appendix A.

Chapter 2

Related Work

This chapter surveys related work in the areas of parallel and multimedia database systems, deterministic scheduling theory, and on-line algorithms, and discusses the relation of earlier results to the work presented in this thesis. Particular technical references are also given in specific chapters.

2.1 Parallel Database Systems

The problem of scheduling complex query plans on parallel machines has recently attracted a lot of attention from the database research community. Hasan and Motwani [HM94] study the tradeoff between pipelined parallelism and its communication overhead and develop near-optimal heuristics for scheduling a star or a path of pipelined relational operators on a multiprocessor architecture. Chekuri et al. [CHM95] extend these results to arbitrary pipelined operator trees. The heuristics proposed in these papers ignore both independent and partitioned parallelism. Ganguly and Wang [GW93] describe the design of a parallelizing scheduler for a tree of coarse grain operators. Based on a *one-dimensional* model of query operator costs, the authors show their scheduler to be near-optimal for a limited space of query plans (i.e., left-deep join trees with a single materialization point in any right subtree). Ganguly et al. [GGW95] obtain similar results for the problem of partitioning independent pipelines without the coarse granularity restriction. The benefits of resource sharing and the multi-dimensionality of query operators are not addressed in these papers. Furthermore, no experimental results are reported. Lo et al. [LCRY93] develop optimal schemes for assigning processors to the stages of a pipeline of hash-joins in a shared-disk environment. Their schemes are based on a *two-phase minimax* formulation of the problem that ignores communication costs and prevents processor sharing among stages. Moreover, no methods are proposed for handling multiple join pipelines (i.e., independent parallelism).

With the exception of the papers mentioned above, most efforts are experimental in nature and offer no theoretical justification for the algorithms that they propose. In addition, many proposals have simplified the scheduling issues by ignoring independent (bushy tree) parallelism;

these include the right-deep trees of Schneider [Sch90a] and the segmented right-deep trees of Chen et al. [CLYY92]. Nevertheless, the advantages offered by such parallelism, especially for large queries, have been demonstrated in prior research [CYW92].

Tan and Lu [TL93] and Niccum et al. [NSHL95] consider the general problem of scheduling bushy join plans on parallel machines exploiting all forms of intra-query parallelism and suggest heuristic methods of splitting the bushy plan into non-overlapping *shelves* of concurrent joins. For the same problem, Hsiao et al. [HCY94] propose a processor allocation scheme based on the concept of *synchronous execution time*: the set of processors allotted to a parent join pipeline are recursively partitioned among its subtrees in such a way that those subtrees can be completed at approximately the same time. For deep execution plans, there exists a point beyond which further partitioning is detrimental or even impossible, and serialization must be employed for better performance. Wolf et al. [WTCY94] present a (one-dimensional) hierarchical algorithm for scheduling multiple parallel queries. Their main idea is to collapse each query plan to a single "large" parallel job and then apply the known results for independent jobs. This has the serious drawback that some obvious, critical co-scheduling may be lost. For example, although it may be highly desirable to combine CPU-bound and I/O-bound tasks from different plans [Hon92], this may not be possible after the collapse. Wilschut et al. [WFA95] present a comparative performance evaluation of various multi-join execution strategies on the PRISMA/DB parallel main-memory database system. Mehta and DeWitt [MD95] and Rahm and Marek [RM95] present experimental evaluations of various heuristic strategies for determining the degree of intra-operation parallelism and assigning processors in shared-nothing database systems. Both of these papers avoid dealing with complex query scheduling issues by assuming workloads consisting of simple binary joins and/or OLTP transactions. Bouganim et al. [BFV96] propose methods for optimizing load-balancing on each site of a hierarchical architecture at run-time so that inter-site data transfers are minimized. In their model, the optimizer still has to determine the assignment of operators to sites and the run-time environment has to make up for optimizer inaccuracies. The issue of how the high-level mapping should be done at the optimizer is not addressed.

A common characteristic of all approaches described above is that they consider a one-dimensional model of resource allocation based on a scalar cost metric (e.g., "work"), which ignores any possibilities for effective resource sharing among concurrent operations. Perhaps the only exception is Hong's method for exploiting independent parallelism in the XPRS shared-memory database system [Hon92]. His approach is based on dynamically balancing resource use between one I/O-bound and one CPU-bound operator pipeline to ensure that the system always executes at its I/O-CPU balance point. However, the substantial cost of communication renders such a scheduling method impractical for shared-nothing or hierarchical systems.

Finally, it is worth noting that the two resource classes described in Section 1.1 have been identified in prior work, e.g., the "stretchable" and "non-stretchable" resources of Pirahesh et al. [PMC+90] and Ganguly et al. [GHK92]. However, the general problem of scheduling operator graphs with both types of resources has not been addressed in prior work on databases or deterministic scheduling theory.

2.2 Multimedia Database Systems

Most prior research in multimedia database systems and multimedia storage servers has concentrated on resource management schemes and data organization techniques for efficiently supporting the continuous retrieval of independent, atomic CM streams [BGMJ94, CKY93, Gem95, GC92, GI95, ÖRS95a, ÖRSM95, ÖRS96c, RV91, SG95]. A number of conceptual models have been developed for capturing the temporal aspects of CM data and complex multimedia presentations. They can be roughly classified into three categories, namely: *graph-based models* (e.g., object composition petri nets [LG90] and presentation graphs [NTB96]), *language-based models* (e.g., HyTime [NKN91] and MHEG [Pri93]), and *temporal abstraction models* (e.g., temporal intervals and relations [All83, LG93]). Candan et al. present a method based on linear difference constraints for defining flexible inter-media synchronization requirements and show how these constraints can be solved and/or modified to ensure consistency [CPS96a]. Thimm et al. describe a feedback-based architecture for adapting a multimedia presentation to the changes in resource availability by modifying the presentation quality [TK96].

The only work prior to ours to address some of the issues in scheduling composite multimedia objects is that of Chaudhuri et al [CGS95] and Shahabi et al. [SGC95]. However, their research on composite objects has focused on (a) the use of memory to resolve the problem of "internal contention", which occurs when the temporal synchronization constraints cause stream retrievals for a single object to collide; and, (b) the development of heuristic memory management policies to distribute a fixed amount of server memory among multiple competing objects. More specifically, Chaudhuri et al. suggest a conservative and a greedy heuristic for allocating memory among multiple *binary* composite objects, under the assumption of regular, round-robin striping of component streams [CGS95]. However, extending their heuristics to general, n-ary objects appears to be problematic [CGS95]. Shahabi et al. show how these conservative and greedy methods can be adapted to the problem of resolving internal contention in a single n-ary composite object, again assuming round-robin layout [SGC95]. Although the authors outline some ideas on how to actually schedule multiple composite objects, they offer no concrete algorithmic solutions for the problem. Furthermore, their development is based on the assumption of a round-robin distribution of stream fragments across disks, whereas we assume a more abstract "black-box" model of server disk bandwidth.

Prior research has proposed a number of admission control policies for multimedia servers that can provide service level guarantees that are either deterministic (i.e., based on worst-case assumptions) [ÖRS95a, ÖRS96a] or stochastic (i.e., based on statistical models of system behavior) [NMW97, VGGG94]. However, little attention has been paid in the multimedia literature to the on-line nature of the admission control problem for CM database servers. Long and Thakur [LT93] present simple adversary arguments to show that no on-line algorithm can achieve a *constant* competitive ratio in the context of the Swift distributed I/O architecture. Aggarwal et al. [AGH95, BNGHA96] present a competitiveness analysis for a different service model, termed *Shared Video-On-Demand*. Requests are notified of acceptance or rejection within a server-specified time interval (termed *notification interval*) from their arrival. Admitted requests waiting for the same clip, can be batched onto a single stream. They show that allowing for sufficiently large notification intervals (linear in the length of the clips) can guarantee constant competitive ratios for simple scheduling algorithms [AGH95, BNGHA96]. The Shared Video-On-Demand model is different from our model of CM service in the sense that it tries to capture the effects of wait tolerance and batching on the number of clients served. Therefore, their results can be viewed as orthogonal to ours. Furthermore, the corresponding analysis assumes that (a) all CM clips have the same length (i.e., time duration) and require the same amount of bandwidth, and (b) any two requests by the same client must be separated by at least the duration of a clip. These assumptions severely limit the applicability of their results to general CM servers.

Despite the importance of the resource scheduling problem for EPPV service, prior work has typically concentrated on other issues such as data layout schemes to efficiently support periodic retrieval [ÖBRS94, ÖBRS95], and support for VCR functionality under EPPV [AA96, LLG97]. The *matrix-based scheme* was designed to support periodic video retrieval for a given period while minimizing video buffering requirements [ÖBRS94, ÖBRS95]. Extensions to the base scheme that deal with the varying transfer rates of commonly used SCSI disks and different video display rates were presented in [ÖRS96b, ÖRS96c]. Abram-Profeta and Shin [APS97] used simple queueing models to solve the problem of assigning optimal retrieval periods to the set of clips stored at an EPPV server. Their models assume that all clips have the *same length and display rate requirement* and ignore multi-disk data organization issues. None of the above efforts has considered the general problem of EPPV resource scheduling and, consequently, they can be viewed as orthogonal to our work. Only in very recent work, Özden et al. [ÖRS97] presented schemes for the periodic retrieval of videos from disk arrays using striping. Their work, however, addressed only a restricted form of the EPPV resource scheduling problems that assumes all clips to have *identical* display rates. Furthermore, they assume specific conditions on the video lengths that limit the usefulness of their results. Clearly, it is crucial for a CM server to be able to retrieve clips with arbitrary retrieval periods based on their popularity without placing any restrictions on the lengths and display rates of clips.

2.3 Scheduling Theory and On-Line Algorithms

Moving away from the database field, there is a significant body of interesting work on task scheduling in the field of deterministic scheduling theory. Early theoretical work on scheduling focused mainly on the problem of allocating processors to computation-intensive tasks. Deterministic models for *resource-constrained* scheduling constitute a relatively recent line of theoretical research that strives to create more realistic models for scheduling problems through the introduction of additional scarce resources [BCSW86]. Since the overwhelming majority of task scheduling problems are $\mathcal{NP}$-hard [BLRK83, GLLRK79, HLvdV97, KSW97], most research efforts have concentrated on providing fast heuristics with provable worst case bounds on the suboptimality of the solution.

Parallel task scheduling (obviously related to our parallel query scheduling problem) has attracted significant research interest since Graham's seminal papers in the late 1960's [Gra66, Gra69]. The problem is known to be $\mathcal{NP}$-hard in the strong sense [DL89] and a number of heuristic approaches have been explored in the scheduling theory literature. However, scheduling query plans on shared-nothing or hierarchical architectures requires a significantly richer model of parallelization than what is assumed in the classical [Gra66, GG75, GGJY76, GLLRK79] or even more recent [BB90, BB91, KM92, TWY92, WC92, ST94, CM96] efforts in that field. To the best of our knowledge, there have been no theoretical results in the literature on parallel task scheduling that consider multiple TS system resources and explore sharing of such resources among concurrent tasks, or study the implications of pipelined parallelism and data communication costs. Perhaps most importantly, even the very recent results of Shachnai and Turek [ST94] and Chakrabarti and Muthukrishnan [CM96] on multi-resource scheduling are based on the assumption that all resources are *globally accessible* to all tasks. In contrast, our target architectures are characterized by a physical distribution of resource units and an *affinity* of system resources to sites: an operation scheduled at a particular site can only make use of the resources locally available to that operation. To the best of our knowledge, there are no previous theoretical results on multi-resource scheduling in this context.

Scheduling composite multimedia objects also introduces novel algorithmic challenges. More specifically, handling the inter-media synchronization constraints defined by the temporal relationships among CM components requires a task model that is significantly more complex than the models employed in scheduling theory and practice. Composite multimedia objects essentially correspond to resource-constrained tasks with *time-varying resource demands.* Resource constraints come from the limited amount of server resources available to satisfy the requirements of CM streams and time-variability stems from the user-defined inter-media synchronization requirements. To the best of our knowledge, this is a task model that has not been previously studied in the context of deterministic scheduling theory.

A significant body of research related to our on-line admission control problem has been conducted in the field of on-line algorithms for bandwidth allocation and circuit routing in communication networks. Lipton and Tomkins [LT94] study the competitiveness of randomized strategies for the non-preemptive On-line Interval Scheduling (OIS) problem, which essentially corresponds to on-line admission control in a server that can support a *single* playback stream. Under the assumption that the ratio Δ of longest to shortest interval is *not known a-priori*, they present an $O((\log\Delta)^{1+\epsilon})$-competitive randomized algorithm and show that no $O(\log\Delta)$-competitive algorithm can exist. Extensions to their randomized scheme are presented by Faigle et al. [FGK96]. Awerbuch et al. [ABFR94] examine the more general problem of non-preemptive circuit routing on tree-structured networks and propose a general randomized technique termed "Classify and Randomly Select". The main idea is to classify on-line events in disjoint classes and then consider only the events that are assigned to a *randomly selected* class. By averaging over all possible random choices, "Classify and Randomly Select" achieves logarithmic competitive ratios (in an expected sense). However, the idea of an admission control scheme that considers only one randomly selected class of user requests and simply ignores all others is obviously not very appealing for CM database servers, since it ignores fundamental requirements such as fairness. Our proposed schemes also employ the idea of on-line classification, but they also are completely *deterministic* without compromising near-optimal competitiveness (for sufficiently large server bandwidth). Awerbuch et al. [AAP93] consider non-preemptive circuit routing on general networks. They present a deterministic scheme (called ROUTE_OR_BLOCK) which, assuming that the bandwidth requested by a single circuit never exceeds an $O(\frac{1}{\log nT_{max}})$ fraction of edge capacity, achieves a competitive ratio of $O(\log nT_{max})$, where n is the number of nodes in the network and T_{max} is the maximum duration of a call. They also prove that their scheme is near-optimal for deterministic on-line routing. The ROUTE_OR_BLOCK algorithm is based on ideas developed for multicommodity network flow problems. Roughly speaking, the main idea is to assign each edge a "length" that is exponential in its current load and route an incoming circuit only if the length of the shortest routing path is less than the "benefit" associated with the circuit. However, as reported by Plotkin [Plo95] and Gawlick [Gaw95], ROUTE_OR_BLOCK exhibited consistently poor performance in an actual implementation. Ad-hoc changes in the algorithm's parameters were necessary to improve its behavior. Furthermore, the ROUTE_OR_BLOCK scheme itself is rather complex and unintuitive and it is not clear how it can benefit from the knowledge of statistical information, such as request popularities. Finally, we should note that allowing *preemption* of requests can lead to better competitive ratios for on-line scheduling and admission control problems [BKM+91, BNCK+95, FN95, GGK+97, KS91, Woe94]. However, the assumption of preemptability is unrealistic in the context of CM applications.

Chapter 3

Parallel Query Scheduling with Multiple Time-Shared Resources

In this chapter[1], we present a framework for multi-dimensional resource scheduling in shared-nothing and hierarchical parallel database systems with sites consisting of TS *resources only.* Building on the work of Ganguly et al. [GHK92], we represent query operator costs as *work vectors* with one dimension per TS resource. This allows a scheduler to explore the possibilities for TS resource sharing among concurrent operators and to accurately quantify the effects of this sharing on query response time. In order to account for the communication overhead of parallelism, we initially restrict our attention to operator parallelizations that are sufficiently coarse grain. We present a quantification of the notion of coarse granularity based on the relative costs of communication and computation and use it to derive the degree of partitioned parallelism.

Based on this framework, the problem of scheduling a collection of concurrently executed operators is reduced to an instance of the *multi-dimensional bin-design problem* [CGJ84] for work vector packings. Based on this observation, we develop a fast resource scheduling algorithm called OPERATORSCHED that belongs to the class of *list scheduling algorithms* [Gra66]. The *response time* (or, *makespan*) of the parallel schedule produced by OPERATORSCHED is analytically shown to be

(a) within $(2d+1)$ of the optimal schedule length for given degrees of partitioned parallelism, and

(b) within $(2d(fd+1)+1)$ of the optimal coarse grain schedule length,

where d is the dimensionality of the work vectors and f is a "small" parameter capturing the granularity of the parallel execution. We also extend the algorithm to handle the operator precedence constraints in a bushy query plan by splitting the execution of the plan into synchronized phases. The resulting algorithm, called TREESCHED, uses OPERATORSCHED as a subroutine

[1]Parts of this chapter have appeared in the *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)* [GI96].

to determine the scheduling of operators within each phase. Experimental results based on analytical cost models for various database operators confirm the effectiveness of these algorithms compared to previous one-dimensional approaches. In addition, our results show that the analytical worst-case bounds are rather pessimistic compared to the average performance, which is extremely close to optimal. Finally, we consider the more general *malleable* problem in which the solution is no longer constrained by a coarse granularity condition. Instead, the scheduler is free to determine the degrees of partitioned parallelism with the objective of minimizing response time over *all possible* parallel schedules. Building on the ideas of Turek et al. [TWY92], we present a technique that allows our list scheduling rule for independent operators to achieve a suboptimality bound of $(2d+1)$ for the malleable problem at the additional cost of a preprocessing parallelization step.

3.1 Problem Formulation

3.1.1 Definitions

We consider hierarchical parallel systems [BFV96] with *identical* multiprogrammed resource sites connected by an interconnection network. Each site is a collection of d system resources that are assumed to be *time-sliceable* or *preemptable*, in the sense that they can be time-shared among different operations at low overhead. Resources like the CPU(s), the disk(s), and the network interface(s) or communication processor(s) are preemptable, while memory is not.

An *operator tree* [GHK92, Hon92, Sch90a] is created as a "macro-expansion" of an execution plan tree by refining each node into a subtree of physical operator nodes, e.g., `scan`, `probe`, `build` (Figure 4(a,b)). Edges represent the flow of data as well as two forms of timing constraints between operators: pipelining (thin edges) and blocking (thick edges). With respect to blocking edges, not only do they imply that an operator cannot start execution until all its children via blocking edges have finished, but they occasionally imply that all these children *must* be executed in parallel and the parent operator *must* be executed immediately afterwards as well. This is for instance the case with the `build` operators of Figure 4(b), which must build their hash tables in memory in parallel, so that the corresponding `probe` operators, being executed immediately after them, find those tables in memory. A *query task* is a maximal subgraph of the operator tree containing only pipelining edges. A *query task tree* is created from an operator tree by representing query tasks as single nodes (Figure 4(c)).

The above trees clarify the definitions of the three forms of intra-query parallelism:

- *Partitioned parallelism:* A single node of the operator tree is executed on a set of sites by

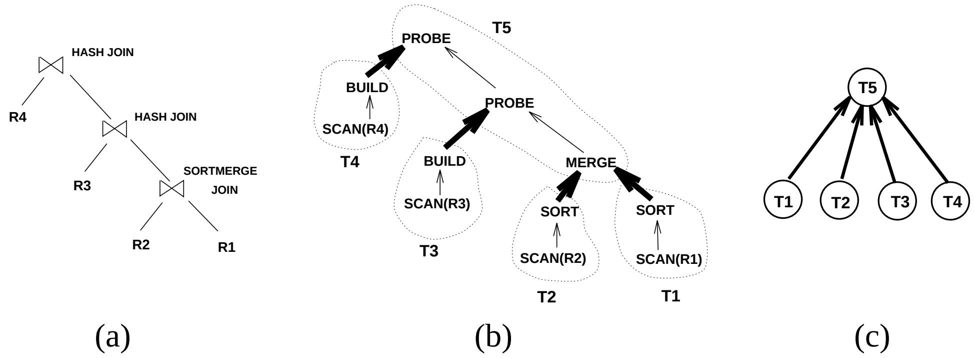


Figure 4: (a) An execution plan tree. (b) The corresponding operator tree. (c) The corresponding query task tree. The thick edges in (b) indicate blocking constraints.

appropriately partitioning its input data sets.

- *Pipelined parallelism:* The operators of a single node of the task tree are executed on a set of sites in a pipelined manner.

- *Independent parallelism:* Nodes of the task tree with no path between them are executed on a set of sites independent of each other (and as mentioned above, they occasionally *must* be executed in parallel). For example, in Figure 4, tasks T1-T4 may all be executed in parallel, with T3-T4 having no other choice but to be executed in parallel. Task T5 must await the completion of T1-T4 before it is executed, which should be immediately after T3-T4 complete.

The *home* of an operator is the set of sites allotted to its execution. Each operator is either *rooted*, if its home is fixed by data placement constraints (e.g., scanning the materialized result of a previous task), or *floating*, if the resource scheduler is free to determine its parallelization.

3.1.2 Overview

A *parallel schedule* consists of (1) an operator tree and (2) an allocation of system resources to operators. Given a query execution plan, our goal is to find a parallel schedule with minimal response time. To account for the communication overhead of parallelism, we initially restrict our attention to partitioned parallelism that is *coarse grain* [GW93, GY93]. That is, we ignore operator parallelizations whose ratio of computation costs to communication overhead is not sufficiently high, as most of them are bound to be ineffective.

Based on the above restriction, we devise an algorithm for scheduling bushy execution plan trees that consists of the following steps:

1. Construct the corresponding operator and task trees, and deterministically split the latter into synchronized phases [TL93], where each phase contains tasks with no (blocking) paths between them.

2. For each operator, determine its individual resource requirements using hardware parameters, DBMS statistics, and conventional optimizer cost models (e.g., [HCY94, SAC+79]).

3. For each floating operator, determine the degree of coarse grain parallelism based on the relative cost of computation and communication (partitioned parallelism).

4. For each phase of the task tree, schedule all floating operators on the set of available sites using a multi-dimensional list scheduling heuristic that is *provably near-optimal* in the space of coarse grain parallel executions (pipelined and independent parallelism).

We then propose a technique for selecting an operator parallelization that allows us to relax the coarse granularity restriction (Step 3). Combining this technique with our list scheduling rule for independent operators results in an algorithm that is provably near-optimal in the space of *all possible* parallel executions.

3.1.3 Assumptions

Our approach is based on the following set of assumptions:

A1. No Memory Limitations. An operator is always allotted sufficient memory buffers to allow the execution of an operator pipeline to proceed in a single phase. For example, when executing a pipeline of `probe` operators, the hash tables built on the inner relations are assumed to be memory-resident. To the best of our knowledge, developing an accurate memory usage model for parallel query optimization is an open problem; besides being non-preemptable, memory introduces an additional level of complexity since the resource requirements of an operator often depend on the amount of available memory.

A2. No Time-Sharing Overhead for TS Resources. Following Ganguly et al. [GHK92], slicing a preemptable resource among multiple operators introduces no additional resource costs.

A3. Uniform TS Resource Usage. Following Ganguly et al. [GHK92], usage of a preemptable resource by an operator is uniformly spread over the execution of the operator.

A4. Non-increasing Operator Execution Times. For the range of coarse grain parallelism considered, an operator's execution time is a non-increasing function of its degree of parallelism, i.e., allotting more sites cannot increase its response time.

A5. Dynamically Repartitioned Pipelined Outputs. The output of an operator in a pipeline is always repartitioned to serve as input to the next one. This is almost always accurate, e.g., when the join attributes of pipelined joins are different, the degrees of partitioned parallelism differ, or different declustering schemes must be used for load balancing.

3.2 Coarse Grain Parallelization of Operators

3.2.1 Resource Usage Model

Our treatment of resource usage is based on the model of preemptable resources proposed by Ganguly et al. [GHK92], which we briefly describe here. The usage of a single resource by an operator is modeled by two parameters, T and W, where T is the elapsed time after which the resource is freed (i.e., the response time of the operator) and W is the work measured as the effective time for which the resource is used by the operator. Intuitively, the resource is kept busy by the operator only W/T of the time. Although this abstraction can model the true utilization of a system resource, it does not allow us to predict exactly when the busy periods are. Thus, we make assumption A3 which, in conjunction with assumption A2, leads to straightforward quantification of the effects of resource sharing [GHK92].

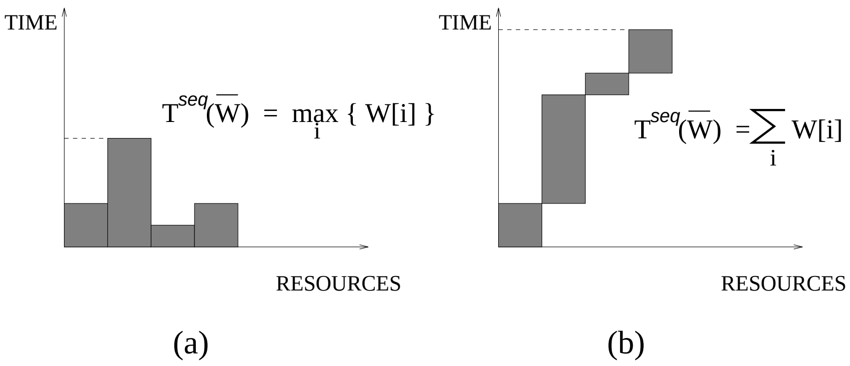


Figure 5: Extremes in usage of d-dimensional resource sites: (a) perfect overlap and (b) zero overlap.

We extend the model of Ganguly et al. [GHK92] and describe the usage by an isolated operator of a site comprising of d preemptable resources by the pair $(T^{seq}, \overline{W})$. Parameter T^{seq} is the *stand-alone* sequential execution time of the operator, while $\overline{W}$ is a d-dimensional *work vector* whose components denote the work done on individual resources. Our model assumes a fixed numbering of system resources for all sites; for example, dimensions 1, 2, 3, and 4 may correspond to CPU, disk-1, disk-2, and network interface, respectively. Time T^{seq} is actually a function of the operator's individual resource requirements, i.e., its work vector $\overline{W}$ (sometimes emphasized by using $T^{seq}(\overline{W})$ instead of T^{seq}), and the amount of *overlap* that can be achieved between processing at different resources. This overlap is a system parameter that depends on the hardware and software architecture of the resource sites (e.g., buffering architecture for disk I/O) as well as the algorithm implementing the operator. An important constraint for T^{seq}, however, is that it can never be less than the amount of work done on any single resource and it can never exceed the total work performed. As shown in Figure 5[2], this is more formally expressed as

$$\max_{1\leq i\leq d}\{W[i]\} \leq T^{seq}(\overline{W}) \leq \sum_{i=1}^{d} W[i].$$

3.2.2 Quantifying the Granularity of Parallel Execution

As is well known, increasing the parallelism of an operator reduces its execution time until a saturation point is reached, beyond which additional parallelism causes a speed-down, due to excessive communication startup and coordination overhead over too many sites [DGS+90]. To avoid operating beyond that point, we need to ensure that the granules of the parallel execution are sufficienty coarse. In particular, in the spirit of Stone [Sto87], we define the *granularity* of a d-dimensional parallel operator $\mathtt{op}$ as the ratio $W_p(\mathtt{op})/W_c(\mathtt{op}, N)$, where

- $W_p(\mathtt{op})$ denotes the total amount of work performed during the execution of $\mathtt{op}$ on a single site, when all its operands are locally resident (i.e., zero communication cost); it corresponds to the *processing area* [GW93] of $\mathtt{op}$ and is constant for all possible executions of $\mathtt{op}$; and

- $W_c(\mathtt{op}, N)$ denotes the total communication overhead incurred when the execution of $\mathtt{op}$ is partitioned among N clones; it corresponds to the *communication area* of the partitioned execution of $\mathtt{op}$ and is a non-decreasing function of N.

Using the above notions, we extend earlier quantifications of coarse grain parallelism [GW93] to our multi-dimensional operator model as follows:

Definition 3.2.1 A parallel execution of an operator $\mathtt{op}$ on N resource sites is *coarse grain with parameter f* (referred to as a CG_f execution) if the communication area of the execution is no more than f times the processing area of $\mathtt{op}$, that is, $W_c(\mathtt{op}, N) \leq f\, W_p(\mathtt{op})$. $\square$

[2] Figure 5 is actually a little misleading since, by assumption A3, the work performed on any resource should be *uniformly* spread over T^{seq}.

3.2.3 Degree of Partitioned Parallelism

Assuming zero communication costs, the resource requirements of the operator are described by a d-dimensional work vector $\overline{W}$ whose components can be derived from system parameters and traditional optimizer cost models [SAC+79]. By definition, the processing area of the operator $W_p(\mathtt{op})$ is simply the sum of $\overline{W}$'s components, i.e., $W_p(\mathtt{op}) = \sum_{i=1}^{d} W[i]$.

Let D denote the total size (in bytes) of the operator's input and output data set(s) that are transferred over the interconnect. We use a simple model of communication costs in which the total communication overhead for the parallel execution of an operator on N sites is estimated as:

$$W_c(\mathtt{op}, N) = \alpha\, N + \beta\, D,$$

where α, β are architecture-specific parameters specified as follows:

- α is the startup cost for each participating site, and
- β is the time spent at the network interface and/or communication processor per unit of data transferred.

This model of operator communication costs is substantiated by the experimental results of DeWitt et al. on the Gamma shared-nothing database machine [DGS+90], and simpler forms of this model have been adopted in previous studies of shared-nothing systems [GMSY93, WFA92].

Note that the startup cost cannot, in general, be distributed among the participating sites. Rather, it is inherently serial and is incurred at a single site (the designated "coordinator" for the parallel execution). This implies that there always exists some degree of parallelism beyond which the startup overhead at the coordinator dominates the actual processing time.

The following proposition is an immediate consequence of Definition 3.2.1 and our communication cost model.

Proposition 3.2.1 The maximum allowable degree of intra-operator parallelism for a CG_f execution of operator $\mathtt{op}$ is denoted by $N_{max}(\mathtt{op}, f)$ and is determined by the formula

$$N_{max}(\mathtt{op}, f) = \max\{\left\lfloor \frac{f\, W_p(\mathtt{op}) - \beta\, D}{\alpha} \right\rfloor\ ,\ 1\}$$

□

3.3 The Scheduling Algorithm

3.3.1 Notation

Table 2 summarizes the notation used in this section with a brief description of its semantics. Detailed definitions of some of these parameters are given below. Additional notation will be introduced when necessary.

Parameter	Semantics
P	Number of system sites
d	Site dimensionality (number of TS resources per site)
B_j	System site (i.e., "bin") number j ($j = 1, \ldots, P$)
B_j^W	Set of TS work vectors scheduled at site B_j
$T^{site}(B_j)$	Execution time for all operator clones at site B_j
M	Number of operators to be scheduled
$\mathtt{op}_i$	Operator, e.g., `scan`, `build` ($i = 1, \ldots, M$)
N_i	Degree of partitioned parallelism (number of clones) for $\mathtt{op}_i$
$\overline{W}_{\mathtt{op}_i}$	Work vector for $\mathtt{op}_i$ (including communication costs for N_i clones)
$T^{max}(\mathtt{op}_i, N_i)$	Maximum execution time among the N_i clones of $\mathtt{op}_i$ while alone in system
$T^{seq}(\overline{W})$	Time of sequential execution of operator with TS resource requirements $\overline{W}$ (Section 3.2.1)
S	Set of (floating) operator clones to be scheduled
S^W	Set of work vectors for all clones to be scheduled
$l(\overline{v}), l(S^v)$	Length of a vector $\overline{v}$ or set of vectors S^v

Table 2: Notation

Vector $\overline{W}_{\mathtt{op}_i}$ describes the total (i.e., processing and communication) TS resource requirements of $\mathtt{op}_i$, given its degree of parallelism N_i. Using the notions of communication and processing area defined in Section 3.2, the above is expressed as

$$\sum_{k=1}^{d} W_{\mathtt{op}_i}[k] = W_p(\mathtt{op}_i) + W_c(\mathtt{op}_i, N_i).$$

The individual components of $\overline{W}_{\mathtt{op}_i}$ are computed using architectural parameters and database statistics, as well as the SS allotment for $\mathtt{op}_i$ and our model for communication costs[3].

The *length of a n-dimensional vector $\overline{v}$* is its maximum component. The *length of a set S^v of n-dimensional vectors* is the maximum component in the vector sum of all the vectors in S^v. More formally,

$$l(\overline{v}) = \max_{1\leq k\leq n} \{v[k]\} \quad , \quad l(S^v) = \max_{1\leq k\leq n} \{\sum_{\overline{v}\in S^v} v[k]\}.$$

[3]The actual distribution of work among the vector's components is immaterial as far as our model is concerned.

The *performance ratio* of a scheduling algorithm is defined as the ratio of the response time of the schedule it generates over that of the optimal schedule. All the parallel query scheduling problems addressed in this thesis are non-trivial generalizations of traditional multiprocessor scheduling [GJ79] and, thus, they are clearly $\mathcal{NP}$-hard. Given the intractability of the problems, we focus on developing polynomial time heuristics that are *provably near-optimal*, i.e., with a constant bound on the performance ratio.

Since the parallelization of rooted operators is pre-determined, our algorithms are only concerned with the scheduling of floating operators. Also, for the purposes of this section, the degree of partitioned parallelism for all floating operators is determined based on a granularity condition, as shown in Proposition 3.2.1. In short, all algorithms presented in this section assume a pre-processing step that places rooted operator clones at their respective sites and computes the degree of coarse grain parallelism for all floating operators. Techniques to relax the coarse granularity restriction and deal with the more general *malleable* operator scheduling problem are discussed in Section 3.5.

3.3.2 Modeling Parallel Execution and Resource Sharing

We present a set of extensions to the (one-dimensional) cost model of a traditional DBMS based on the multi-dimensional TS resource usage formulation described in Section 3.2.1. Our extensions account for all forms of parallelism and quantify the effects of TS resource sharing on the response time of a parallel execution.

Partitioned Parallelism

In partitioned parallelism, the work vector of an operator is partitioned among a set of *operator clones* [GHK92]. Each clone executes on a single site and works on a portion of the operator's data. Consider an operator $\mathtt{op}_i$ that is distributed across N_i sites and runs in isolation, without experiencing resource contention. Partitioning $\overline{W}_{\mathtt{op}_i}$ into the work vectors for the operator clones is determined based on statistical information kept in the DBMS catalogs. Given such a partitioning $< \overline{W}_1, \overline{W}_2, \ldots, \overline{W}_{N_i} >$, where $\sum_{k=1}^{N_i} \overline{W}_k = \overline{W}_{\mathtt{op}_i}$, the parallel execution time for $\mathtt{op}_i$ can be expressed as the maximum of the sequential execution times of the N_i clones; that is,

$$T^{max}(\mathtt{op}_i, N_i) = \max_{1 \le k \le N_i} \{ \, T^{seq}(\overline{W}_k) \, \}. \tag{1}$$

Pipelined and Independent Parallelism

In the presence of multiple concurrent operators, we need a more precise definition of a parallel schedule.

Definition 3.3.1 Given a collection of M operators to be executed concurrently $\{\texttt{op}_i, i = 1 \ldots M\}$ and their respective degrees of partitioned parallelism $\{N_i, i = 1 \ldots M\}$, a *schedule* is a mapping of the $\sum_{i=1}^{M} N_i$ operator clones to the set of available sites such that no two clones of the same operator are mapped to the same site. $\square$

The constraint on the mapping of operator clones to sites ensures that N_i is the true degree of parallelism for $\texttt{op}_i$ so that Equation (1) is still valid.

The effects of time-sharing a site among many operators can be quantified as follows. Let B_j^W denote the set of all operator clones (or, equivalently, all work vectors) mapped to site B_j under a particular schedule. Since all resources are preemptable, the execution time for all the operator clones scheduled at B_j is determined by the ability to overlap the processing of resource requests by different operators. Specifically, under our model of preemptable resources described in Section 3.2.1, the execution time for all the operator clones scheduled at B_j is defined as

$$T^{site}(B_j) = \max\{ \max_{\overline{W} \in B_j^W} \{T^{seq}(\overline{W})\} \, , \, l(B_j^W) \, \}. \tag{2}$$

For example, consider two 2-dimensional operator clones with resource usage pairs $(T_1^{seq}, \overline{W}_1)$ = (22,[10,15]) and $(T_2^{seq}, \overline{W}_2)$ = (10,[10,5]) placed at B_j. In this case, $\overline{W}_1 + \overline{W}_2$ = [20,20], which means that the total requirements of the two clones ($l(\{\overline{W}_1, \overline{W}_2\}) = 20$) can be "squeezed" into the response time of the first clone ($T_1^{seq} = 22$), i.e., $T^{site}(B_j) = 22$. On the other hand, consider $(T_1^{seq}, \overline{W}_1)$ placed at B_j with $(T_3^{seq}, \overline{W}_3)$ = (10,[5,10]). In this case, $\overline{W}_1 + \overline{W}_3$ = [15,25], and the second resource gets congested, i.e., $T^{site}(B_j) = l(\{\overline{W}_1, \overline{W}_3\}) = 25$, while $\max\{T_1^{seq}, T_3^{seq}\} = 22$.

Let SCHED be a schedule for the parallel execution of $\{\texttt{op}_i, i = 1 \ldots M\}$ on a set of resource sites $\{B_j, j = 1 \ldots P\}$. Clearly, the response time of SCHED is determined by the most heavily loaded site. Thus, we can combine Equations (2) and (1) to estimate the response time as follows:

$$\begin{aligned} T^{par}(\text{SCHED}, P) &= \max_{1 \le j \le P} \{ \, T^{site}(B_j) \, \} \\ &= \max\{ \max_{1 \le j \le P} \{ \max_{\overline{W} \in B_j^W} \{T^{seq}(\overline{W})\}\} \, , \, \max_{1 \le j \le P} \{l(B_j^W)\} \, \}. \end{aligned}$$

Observe that the first term on the right-hand side takes the maximum over the sequential execution times of *all* concurrent operator clones. So, we can use Equation (1) to obtain

$$T^{par}(\text{SCHED}, P) = \max\{ \max_{1\le i\le M}\{T^{max}(\mathsf{op}_i, N_i)\} \ , \ \max_{1\le j\le P}\{l(B_j^W)\} \ \}. \tag{3}$$

Equation (3) defines the optimization metric for our scheduling algorithm, described in the next section. Intuitively the formula states that the response time of a parallel execution schedule is determined by either the slowest executing operator, or the load at the most heavily congested resource in the system, whichever is greater.

3.3.3 A Near-Optimal Heuristic for Independent Query Tasks

In this section, we develop a provably near-optimal heuristic for scheduling independent query tasks. In Section 3.3.4, we address the general query task tree scheduling problem. A collection of independent query tasks (pipelines) is essentially a collection of operators that can be executed concurrently. Operators within each task form producer-consumer pairs that communicate across the interconnection network, whereas operators in different tasks are completely independent.

More specifically, let S denote the set of all (floating) operators to be scheduled and let $N = \sum_{i=1}^{M} N_i$, where the degree of parallelism N_i is determined by Proposition 3.2.1 for $\mathsf{op}_i \in S$. As discussed earlier, the degree of parallelism and the mapping of rooted operators is fixed by data placement. Hence, depending on the operator type (rooted or floating), the left input of max in (3), i.e., $T^{max}(\mathsf{op}_i, N_i)$, is either fixed or minimized. Consequently, minimization of response time (equation (3)) translates to determining a mapping of the N work vectors obtained through the cloning of operators in S to the P d-dimensional sites, such that

(A) no two vectors from the same operator are mapped to the same site,

(B) data placement constraints for rooted operators are satisfied, and

(C) the maximum resource usage among all system resources, i.e., the right input of max in (3), is minimized.

This is essentially an instance of the *d-dimensional bin-design* problem (the dual of the d-dimensional vector-packing problem) [CGJ84]. In vector-packing terminology, our scheduling problem may be stated as follows:

> *Given a collection of positive d-dimensional vectors (the work vectors) and a set of P d-dimensional bins (the system sites), determine a* **packing** *of the vectors in the*

bins that obeys constraints **(A)** *and* **(B)** *and minimizes the required common bin capacity (the maximum* TS *resource usage in the system).*

This problem is clearly $\mathcal{NP}$-hard since it reduces to traditional multiprocessor scheduling for $d = 1$ and $N_i = 1$ for all i. Given the intractability of the problem, we develop an approximation algorithm, OPERATORSCHED , that runs in polynomial time and guarantees a constant bound on the performance ratio. OPERATORSCHED belongs to the class of *list scheduling* algorithms originally proposed by Graham [Gra66]. The algorithm begins by placing the work vectors of all rooted operators at their respective sites and computing the degree of coarse grain parallelism for all floating operators. It then proceeds to schedule floating operators according to the following *list scheduling rule*: Consider the list of work vectors resulting from the cloning of all floating operators in non-increasing order of their maximum component; at each step, pack the next vector in the least filled allowable bin/site (that is, pack the vector in the site s_j such that $l(work(s_j))$ is minimal among all bins not containing other vectors of that operator). OPERATORSCHED is depicted in Figure 6.

Algorithm OPERATORSCHED**(S, P)**
Input: A set of f-coarse grain floating operator clones S and a set of P sites $\{B_1, \ldots, B_P\}$.
Output: A mapping of the clones to sites $(\{B_j^W, j = 1, \ldots, P\})$ for the CG$_f$ execution of S satisfying (A)-(B). (Goal: Minimize response time.)

1. let $L_i =< \overline{W}_1, \ldots, \overline{W}_{N_i} >$ be the list of work vectors for $\texttt{op}_i$'s clones.
2. let $L =< \overline{W}_1, \ldots, \overline{W}_N >$ be the list of all floating work vectors in *non-increasing* order of $l(\overline{W}_i)$.
3. for $k = 1$ to N do
   - 3.1. let $\texttt{op}_i$ be the operator whose cloning produced $\overline{W}_k$.
   - 3.2. let B be a site with $B^W \cap L_i = \emptyset$ such that $l(B^W) = \min_{B_j : B_j^W \cap L_i = \emptyset} \{l(B_j^W)\}$.
   - 3.3. set $B^W = B^W \cup \{\overline{W}_k\}$.

Figure 6: Algorithm OPERATORSCHED

The following theorem establishes upper bounds on the time complexity and the worst-case performance ratio of our algorithm.

Theorem 3.3.1 OPERATORSCHED runs in time $O(MP(M + \log P))$, where M is the number of concurrent operators and P is the number of system sites. The parallel execution time of the schedule returned by OPERATORSCHED is

(a) within $(2d+1)$ of the length of the optimal schedule that uses the same degrees of intra-operator parallelism for all floating operators, and

(b) within $(2d(fd+1)+1)$ of the optimal CG_f schedule length.

$\square$

3.3.4 Handling Data Dependencies

Scheduling arbitrary query task trees must ensure that the blocking constraints specified by the tree's edges are satisfied. For this, we split a query task tree into synchronized phases or "shelves" [NSHL95, TL93]. Each phase contains independent tasks that are to be executed concurrently, after the completion of all tasks in the previous phase. The number of phases is equal to the height of the task tree and each task is scheduled in the phase closest to the root that does not violate the precedence constraints. For example, the plan in Figure 4 is executed in two distinct phases containing tasks T1-T4 and task T5, respectively. This corresponds to the *MinShelf* policy of Tan and Lu [TL93]. Resource scheduling within each phase is performed by the OperatorSched algorithm. The full algorithm, TreeSched is depicted in Figure 7. Note that by scheduling tasks according to their level TreeSched also satisfies the more "subtle" timing constraints discussed in Section 3.1.1. For example, all `build`-tasks in a join pipeline are executed together and the corresponding `probe`-task is scheduled in the immediately following phase.

Algorithm TreeSched(T, P, f)
Input: A query task tree $T=(V,E)$, a set of P sites $\{B_1,\ldots,B_P\}$, and a granularity parameter f.
Output: A schedule for the CG_f execution of T. (Goal: Minimize response time.)

1. for $i = height(T)$ downto 0 do
   - 1.1. $OP = \emptyset$.
   - 1.2. for each node $v \in V$ such that $level(v) = i$ do
     - 1.2.1. $OP = OP \cup \{$operators in task $v\}$.
   - 1.3. place all rooted clones in OP and determine the degree of parallelism for each floating operator in OP.
   - 1.4. call OperatorSched(OP, P).

Figure 7: Algorithm TreeSched

Observe that for any query execution plan the number of nodes in the operator tree is bounded by a small constant times the number of joins in the query, e.g., expanding a hash-join gives at most four operator nodes. Combining this observation with Theorem 3.3.1 gives the following complexity bound for TreeSched .

Proposition 3.3.1 TreeSched runs in time $O(JP(J + \log P))$, where J is the number of nodes in the query execution plan and P is the number of system sites. □

3.3.5 Comments on the Effectiveness of the Heuristics

Theorem 3.3.1 derives an upper bound on the worst-case performance ratio of the OperatorSched algorithm for scheduling a collection of CG_f concurrent operators. In general, the expected output quality of our heuristic should be much better than the worst-case bounds, especially for a set of operators with a good "mix" of resource requirements. This conjecture is supported by theoretical results on the expected performance of vector packing [KLMS84]. The big advantage of OperatorSched compared to previous approaches is its ability to explore resource sharing possibilities and balance the resource workloads at individual sites.

Deriving performance bounds for the schedule produced by the TreeSched algorithm is a much more difficult problem. Theorem 3.3.1 ensures that scheduling within each phase is near-optimal *given its data placement constraints*. When scheduling a query task tree, the scheduling decisions made at earlier phases may impose data placement constraints on the phases that follow. For example, the `build` and `probe` operators of a hash join belong to two adjacent phases because of their sequential dependency (the hash table has to be complete before probing can begin). Furthermore, the `probe` operator has to be executed at the set of sites that hold the hash table, that is, the home of the `build`. Such interdependencies between phases complicate any proof of suboptimality bounds for the TreeSched algorithm. At this point, we have not been able to obtain theoretical results on the quality of the schedule produced for the entire query task tree. However, given the load balancing capabilities of the OperatorSched algorithm, we feel confident that TreeSched will outperform previous approaches. Our conjectures for both OperatorSched and TreeSched are supported by the results of an experimental evaluation presented in the next section.

3.4 Experimental Performance Evaluation

In this section, we describe the results of several experiments we have conducted using analytical cost model computations for randomly generated query execution plans. A primary goal of our experimentation is to verify the benefits of our proposed multi-dimensional framework by comparing the average performance of our multi-dimensional scheduling algorithm with a one-dimensional "synchronous execution time" algorithm that we developed based on previous work [HCY94, LCRY93]. Another point of interest, given our worst-case analytical bounds, is examining how close the response time of the generated schedule is to that of the optimal coarse grain schedule *on the average*. We start by presenting our experimental testbed and methodology.

3.4.1 Experimental Testbed

We have experimented with the following algorithms:

- SYNCHRONOUS : Combination of the synchronous execution time method of Hsiao et al. [HCY94] for processor allocation for independent parallelism with the two-phase minimax technique of Lo et al. [LCRY93] for *optimally* distributing processors across the stages of a hash-join pipeline. Although these strategies were originally proposed for shared-disk systems, they were appropriately extended to account for the data redistribution costs in a shared-nothing environment.

- TREESCHED : Multi-dimensional list scheduling in synchronized phases.

- OPTBOUND : Hypothetical algorithm achieving a lower bound on the optimal response time. This bound was calculated as the maximum of the average work per site (see Lemma A.1.2) and the length of the critical path in the query execution plan.

We selected SYNCHRONOUS as a one-dimensional adversary since it is the "state-of-the-art" method for exploiting bushy tree parallelism in parallel query execution[4] [WFA95]. Prior research has demonstrated the advantages offered by such parallelism, especially for large queries [CYW92]. To the best of our knowledge, optimal processor distribution within general join pipelines remains an open problem. We therefore decided to restrict our experiments to bushy hash-join query plans so that the optimal technique of Lo et al. could be used in SYNCHRONOUS. We should stress, however, that TREESCHED is a general query scheduling algorithm that can be applied to *any* bushy plan.

Some additional assumptions were made to obtain a specific experimental model from the general parallel execution model described in Sections 3.2 and 3.3. These are briefly summarized below.

EA1. No Execution Skew: With the exception of startup cost, the work vector of an operator is distributed perfectly among all sites participating in its execution. Startup is added to

[4] The *Fully Parallel Execution Method* [WFA95] applies only to main-memory parallel database systems.

only one of these sites, the "coordinator site" for the parallel execution, and is equally divided between the coordinator's CPU and its network interface.

EA2. Uniform Resource Overlapping: The amount of overlap achieved between processing at different resources at a site can be characterized by a single system-wide parameter $\epsilon \in [0,1]$ for all query operators. This parameter allows us to express the response time of a work vector as a convex combination of the maximum and the sum of the vector components (see Section 3.2.1), i.e.,

$$T(\overline{W}) = \epsilon(\max_{1 \le i \le d}\{W[i]\}) + (1-\epsilon)\sum_{i=1}^{d} W[i].$$

Small values of ϵ imply limited overlap, whereas values closer to 1 imply a larger degree of overlap. In the extreme cases, $\epsilon = 1$ gives $T(\overline{W}) = \max_{1 \le i \le d}\{W[i]\}$ (perfect overlap), and $\epsilon = 0$ gives $T(\overline{W}) = \sum_{i=1}^{d} W[i]$ (zero overlap).

Finally, special precautions were taken to ensure that assumption A4 is not violated for any given value of the granularity parameter f. For each query operator, there exists an optimal degree of partitioned parallelism that minimizes the response time [WFA92], and beyond which startup costs will cause a speed-down. Our implementation makes sure that this optimal degree of parallelism is never exceeded for any operator.

We experimented with tree queries of 10, 20, 30, 40, and 50 joins. For each query size, twenty query graphs (trees) were randomly generated and for each graph a bushy execution plan was randomly selected. We assumed simple key join operations in which the size of the result relation is always equal to the size of the largest of the two join operands. The comparison metric was the *average response times* of the schedules produced by the algorithms over all queries of the same size. Experiments were conducted with the resource overlap parameter ϵ varying between 10% and 70% and the granularity parameter f varying between 0.3 and 0.9. (The results presented in the next section are indicative of the results obtained for all values of ϵ and f.)

In all experiments, we assumed a system consisting of 3-dimensional sites with one CPU, one disk unit, and one network interface. The work vector components for the CPU and the disk were estimated using the cost model equations given by Hsiao et al. [HCY94]. The communication costs were calculated using the model described in Section 3.2.2. The values of the cost model parameters were obtained from the literature [GW93, HCY94, WFA92] and are summarized in Table 3[5].

[5]The CPU speed and disk service rate were chosen so that the system is relatively balanced (i.e., not heavily CPU or I/O bound).

Configuration/DB Catalog Param.	Value
Number of Sites	10 - 140
CPU Speed	1 MIPS
Effective Disk Service Time per page	20 msec
Startup Cost per site (α)	15 msec
Network Transfer Cost per byte (β)	0.6 μsec
Tuple Size	128 bytes
Page Size	40 tuples
Relation Size	10^3 - 10^5 tuples

CPU Cost Parameters	No. Instr.
Read Page from Disk	5000
Write Page to Disk	5000
Extract Tuple	300
Hash Tuple	100
Probe Hash Table	200

Table 3: Cost Model Experiments: Parameter Settings

3.4.2 Experimental Results

The first set of experiments studied the effect of different values of the granularity parameter f on the performance of TreeSched compared to that of Synchronous (which is, of course, not affected by different values of f). The results for queries of 40 joins and a resource overlap of 30% (i.e., $\epsilon = 0.3$) are depicted in Figure 8(a). Clearly, for small values of f the coarse granularity condition is too restrictive, not allowing the execution system to fully exploit the available parallelism. As the value of f increases, the average plan response time drops substantially until the bound on operator parallelism is reached. As expected, the advantages of resource sharing are most evident for resource-limited situations (i.e., small parallel systems). Nevertheless, for sufficiently large values of f, our algorithm outperformed its one-dimensional adversary in the entire range of system and query sizes.

The second set of experiments studied the effect of the resource overlap parameter ϵ on the performance of the two algorithms, while the granularity parameter was kept constant. The performance results shown in Figure 8(b) (for queries of 40 joins) demonstrate that TreeSched consistently outperformed the Synchronous algorithm for various values of f. Clearly, the benefits of multi-dimensional scheduling are more significant for smaller values of the overlap parameter. The reason is that lower overlap results in longer idle periods for the individual resources which our algorithm can exploit through time-sharing with other operations.

The average performance of the two scheduling algorithms for different query sizes is depicted in Figure 9(a) for two different system sizes (20 and 80 sites) and overlap $\epsilon = 0.5$. For TreeSched we assume f to be fixed at 0.7. Note that, for a given system size, the relative improvement obtained with TreeSched increases monotonically with the query size and, as

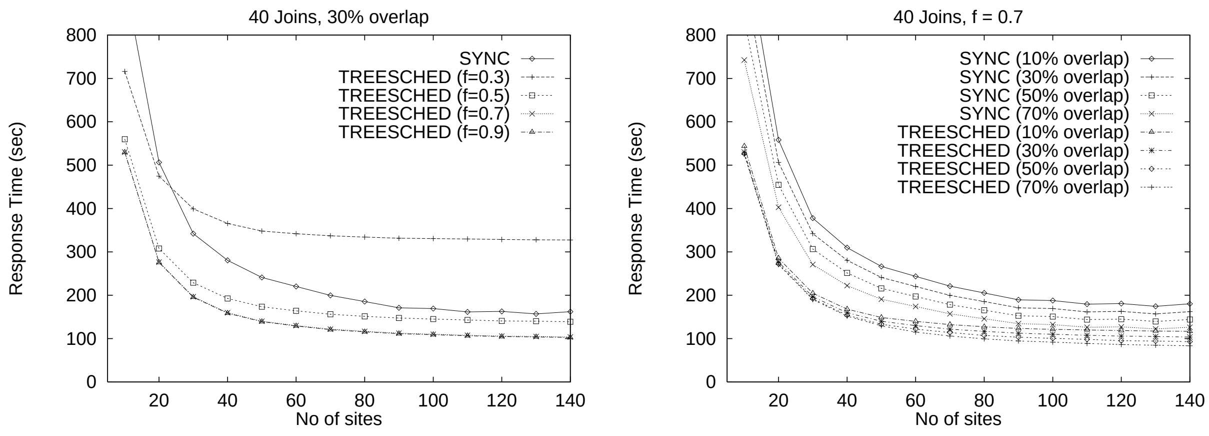


Figure 8: (a) Effect of the granularity parameter (f). (b) Effect of the resource overlap parameter (ϵ).

before, is higher for smaller systems and lower degrees of overlap.

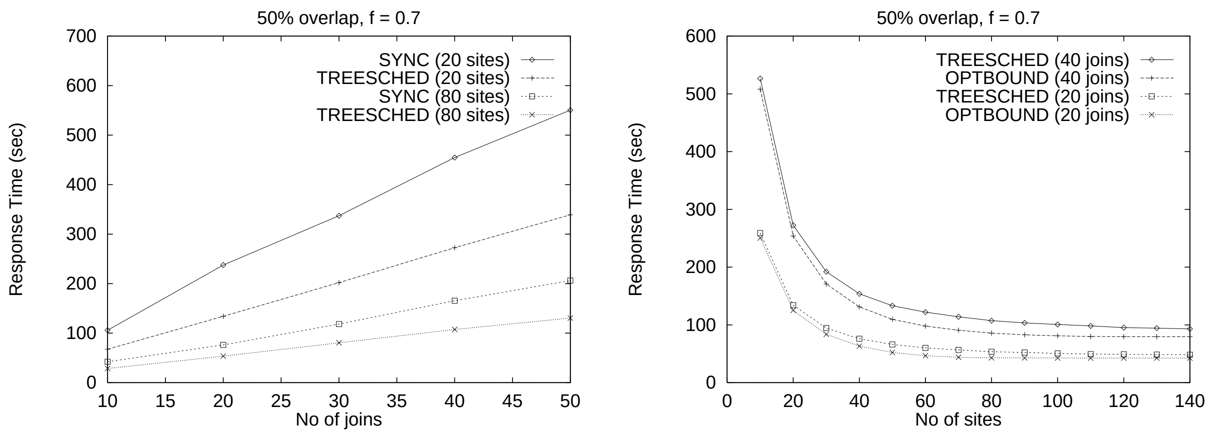


Figure 9: (a) Effect of query size. (b) Average Performance of TreeSched vs. Optimal.

We should also mention that the asymptotic time complexity of Synchronous is $O(JP\log(JP))$, where J is the number of joins in the query and P is the number of sites [LCRY93]. Thus, TreeSched appears to be slightly more expensive than Synchronous, being quadratic in the size of the query (Proposition 3.3.1). We believe that this is a small price to pay compared to the significant performance improvement offered by resource sharing, especially for large queries and/or resource-limited situations.

For our final set of experiments, we examined the average performance of TreeSched compared to a lower bound on the response time of the optimal CG_f execution for a constant value of f. This lower bound, OPTBOUND, was estimated using the formula

$$\text{OPTBOUND} = \max\{ \frac{l(S)}{P} , T(\text{CP}) \},$$

where

- S is the set of work vectors for all operators in the query execution plan *assuming zero communication costs for each operator*, and
- $T(\text{CP})$ is the total response time of the critical (i.e., most time-consuming) path in the plan *assuming the maximum allowable degree of coarse grain parallelism for each operator*.

By Lemma A.1.2 (Appendix A.1) and assumption A4, OPTBOUND is indeed a lower bound on the length of the optimal CG_f execution. The results for queries of 20 and 40 joins are shown in Figure 9(b) for $f = 0.7$ and overlap $\epsilon = 0.5$. These curves verified our expectations, showing that the average performance of TREESCHED is much closer to optimal than what we would expect from the worst-case bound derived in Theorem 3.3.1 for each plan phase. These results are in accordance with the theoretical results of Karp et al. [KLMS84] who used a probabilistic model to prove that even very simple vector-packing heuristics can be expected to produce packings in which very little of the capacity of the bins is wasted.

3.5 Extensions for Malleable Operators

In this section, we extend our list scheduling technique to handle the more general *malleable* scheduling problem. The difference with respect to our previous scheduling model is that the degree of parallelism for floating operators is no longer determined through a coarse granularity condition. Instead, floating operators are malleable, in the sense that the scheduler is free to determine their parallelization so that the execution time is minimized over all possible parallel schedules. Based on the work of Turek et al. [TWY92], we present a method that allows the $(2d + 1)$ suboptimality bound shown for the case of non-malleable independent operators (Theorem 3.3.1) to be duplicated for the more general malleable problem. Since rooted operators have no effect on the quality of the generated schedule (their scheduling is determined by data placement constraints) we will once again consider only floating operators in this section.

Let $\underline{N} = (N_1, \ldots, N_M)$ denote a parallelization (i.e., the degrees of parallelism) of a given set of independent operators, and let $S(\underline{N}) = (\overline{W}_{\mathtt{op}_1}(N_1), \ldots, \overline{W}_{\mathtt{op}_M}(N_M))$ be the set of (total) work vectors for the operators (including the communication costs for the given parallelization).

Finally, define $h(\underline{N}) = \max_{1\leq i\leq M}\{T^{max}(\mathsf{op}_i, N_i)\}$, i.e., the parallel execution time of the slowest operator. In proving the $(2d+1)$ suboptimality bound for OperatorSched (Appendix A.1), we actually show that the makespan of the schedule produced by our list scheduling rule for any given operator parallelization $\underline{N}$ satisfies the following inequality:

$$T^{par}(\text{SCHED}, P, \underline{N}) \leq (2d+1)\ \max\{\ \frac{l(S(\underline{N}))}{P}\ ,\ h(\underline{N})\ \},$$

where $LB(\underline{N}) = \max\{\ \frac{l(S(\underline{N}))}{P}\ ,\ h(\underline{N})\ \}$ is a lower bound on the optimal response time for the given parallelization.

Our goal is to determine a particular operator parallelization $\underline{N}$ such that when $\underline{N}$ is used as input to our list scheduling technique the resulting schedule is guaranteed to be within $(2d+1)$ of the optimal schedule (over all possible parallelizations). The following lemma formalizes our expectations.

Lemma 3.5.1 Let $\underline{N}^*$ denote the parallelization of operators in the optimal execution schedule. Let $\underline{N}$ be another (possibly identical) parallelization such that $LB(\underline{N}) \leq LB(\underline{N}^*)$. Then, applying our list scheduling rule to $\underline{N}$ will return an execution schedule whose length is within $(2d+1)$ of the optimal schedule length. □

We now present a greedy selection algorithm for generating a family of parallelizations. The algorithm is an adaptation of the **GF** method presented by Turek et al. [TWY92] based on the observation that in our work vector model, for any operator op, if $n \leq m$ then $\overline{W}_{\mathsf{op}}(n) \leq_d \overline{W}_{\mathsf{op}}(m)$[6] :

1. The first candidate parallelization is the minimum total work parallelization $\underline{N}^1 = (1, 1, \ldots, 1)$.
2. The k^{th} candidate parallelization is determined by the $(k-1)^{th}$ parallelization by first finding the operator whose execution time is equal to $h(\underline{N}^{k-1})$ and increasing its degree of parallelism by one.
3. The algorithm terminates when no more sites can be allotted to the largest operator.

Lemma 3.5.2 Let $\underline{N}^*$ denote the parallelization of operators in the optimal execution schedule. The above algorithm produces at least one parallelization $\underline{N}$ such that the following two properties hold:

1. $T^{max}(\mathsf{op}_i, N_i) \leq h(\underline{N}^*)$ for all i, and
2. $\overline{W}_{\mathsf{op}_i}(N_i) \leq_d \overline{W}_{\mathsf{op}_i}(N_i^*)$ for all i.

[6] $\leq_d$ stands for componentwise less-than, i.e., $\overline{W}_1 \leq_d \overline{W}_2$ iff $\overline{W}_1[i] \leq \overline{W}_2[i]$ for $i = 1, \ldots, d$.

□

From Lemma 3.5.2 and the definition of the lower bound $LB()$, at least one of the operator parallelizations produced by the algorithm will satisfy the conditions of Lemma 3.5.1.

Theorem 3.5.1 Let A be the family of parallelizations generated and let $\underline{N} \in A$ such that $LB(\underline{N}) = \min_{\underline{K} \in A}\{\ LB(\underline{K})\ \}$. Then, the schedule generated by our list scheduling rule for the parallelization $\underline{N}$ is within $(2d+1)$ of the optimal parallel schedule length. □

The number of parallelizations generated by our algorithm is bounded by $1+M(P-1)$ and so the complexity of selecting an operator parallelization is $O(MP\log M)$. Thus, this preprocessing step does not affect the asymptotic complexity of our scheduler. Also note that Theorem 3.5.1 does not depend on the non-increasing execution times assumption (A4) or any particular model for communication costs. The only assumption required is that of non-decreasing work vectors.

3.6 Conclusions

In this chapter, we have addressed the open problem of multi-dimensional TS resource scheduling for complex queries in parallel database systems. Our approach is based on (1) a model of resource usage that allows the scheduler to explore the possibilities for resource sharing among concurrent operations and quantify the effects of this sharing on the parallel execution time, and (2) a quantification of the notion of coarse grain parallelism for query plan operators. Using these tools we developed a vector-packing formulation of the resource scheduling problem for independent query tasks, and proposed OperatorSched, a fast list scheduling heuristic that is provably near-optimal in the class of coarse grain executions. We then extended our approach to handle the blocking constraints in a bushy query plan by splitting its execution into synchronized phases. The resulting algorithm, TreeSched, exploits all forms of intra-query parallelism and allows effective resource sharing among operators executing concurrently. We also verified the effectiveness of our scheduling methods compared to both previous (one-dimensional) approaches and the optimal solution through a series of experimental results. Finally, we proposed a technique that allows us to relax the coarse granularity restriction and obtain a provably near-optimal list scheduling method for the malleable independent operator scheduling problem. In practice, the coarse granularity condition provides a fast way of determining an efficient parallelization based on system parameters. The more sophisticated greedy selection technique can be used when the additional scheduling overhead is justified.

Chapter 4

Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources

In this chapter[1], we extend our earlier problem formulation to address the co-existence of time- and space-sharing and present a general framework for TS *and* SS resource scheduling in hierarchical parallel database systems. Building on our earlier results, we represent query operator costs as pairs of *work* and *demand* vectors with one dimension per TS and SS resource, respectively. We observe that the inclusion of the SS resource dimension(s) gives rise to certain interesting tradeoffs with respect to the degree of partitioned parallelism. Smaller degrees result in reduced communication overhead and, therefore, increased total work (i.e., TS resource requirements) for the operator execution (i.e., coarse grain parallel executions [GW93, GGS96]). On the other hand, larger degrees of parallelism in general imply smaller SS requirements for each operator clone, thus allowing for better load balancing opportunities and tighter schedulability conditions. The importance of such tradeoffs for parallel query processing and optimization has been stressed earlier [HFV96] and is addressed in this work.

Based on our multi-dimensional framework, we develop a fast list scheduling algorithm for operator pipelines called PIPESCHED and analytically show that it produces a parallel schedule with response time within $d(1 + \frac{s}{1-\lambda})$ of the optimal schedule length for given degrees of partitioned parallelism, where d and s are the dimensionalities of the TS and SS resource vectors respectively, and λ is an upper bound on the (normalized) SS demands of any clone in the pipeline. We then extend our approach to multiple independent pipelines, using a level-based scheduling algorithm [CGJT80, TWPY92] that treats PIPESCHED as a subroutine within each level. The resulting algorithm, termed LEVELSCHED, is analytically shown to be near-optimal for given degrees of operator parallelism. Furthermore, we show that LEVELSCHED can be readily extended to handle the operator precedence constraints in a bushy query plan as

[1] Parts of this chapter have appeared in the *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)* [GI97].

well as *on-line* task arrivals (e.g., in a dynamic or multi-query execution environment). Once again, experimental results based on cost model computations confirm the effectiveness of our algorithms compared to a lower bound on the optimal solution, showing that our analytical worst-case bounds are rather pessimistic compared to the average performance. Motivated by our scheduling results, we revisit the open problem of designing efficient cost models for parallel query optimization. In recent work, Ganguly et al. [GGS96] identified two important "bulk parameters" of a parallel query execution plan, namely average work and critical path length, that are crucial to characterizing its expected response time. Based on our analytical and experimental results, we identify a third parameter, the *average volume* (i.e., the resource-time product) for SS resources that is an essential component of query plan quality since it captures the constraints on query execution that derive from SS (i.e., memory) resources. Based on our results we believe that this 3-dimensional cost model is sufficient for efficient and accurate parallel query optimization.

4.1 Problem Formulation

We extend the parallel query scheduling model described in the previous chapter by introducing SS resources in our problem setup. More specifically, we view each site of a hierarchical parallel system as a collection of d TS resources (e.g., CPU(s), disk(s), and network interface(s) or communication processor(s)) and s SS resources (e.g., memory). Although memory is probably the only SS resource that comes to mind when discussing traditional database query operators, often the distinction between TS and SS resources depends on the needs of a particular application. For example, the playback of a digitized video from a disk requires a specific fraction of the disk bandwidth throughout its execution. Clearly such an operator views the disk as an SS resource although traditional database operators view it as a TS resource. For this reason, we decided to address the scheduling problems for general s rather than restricting our discussion to $s = 1$ (i.e., memory). An obvious advantage of this general formulation is that it allows us the flexibility to "draw the line" between TS and SS resources at any boundary, depending on factors such as application requirements or user view of resources.

Since the algorithms presented in this chapter explicitly account for the scheduling of memory resources, our results are no longer based on assumption A1 ("No Memory Limitations") employed in the previous chapter. We do, however, require the following (more realistic) assumption in addition to assumptions A2-A5 from Section 3.1.3.

A6. Constant SS Resource Demand. The total SS requirements of an operator are constant and independent of its degree of parallelism. For example, the total amount of memory required by all the clones of a `build` operator equals the size of a hash table on the build relation. Further, increasing the degree of parallelism does not increase the SS demands of individual clones.

4.1.1 Overview

Accounting for both TS and SS resource dimensions, our scheduling framework gives rise to interesting tradeoffs with respect to the degree of partitioned parallelism. Coarse grain operator parallelizations [GGS96, GW93] are desirable since they typically result in reduced communication overhead and effective parallel executions with respect to TS resource use. On the other hand, fine grain operator parallelizations are desirable since they imply smaller SS requirements for each clone thus allowing for better load balancing opportunities and tighter schedulability conditions. A quantification of these tradeoffs and our resolution for them are presented in Section 4.2.1.

We have devised an algorithm for scheduling bushy execution plan trees that consists of the following steps:

1. Construct the corresponding operator and task trees, and for each operator, determine its individual resource requirements using hardware parameters, DBMS statistics, and conventional optimizer cost models (e.g., [HCY94, SAC+79]).

2. For each floating operator, determine the degree of parallelism based on the TS vs. SS resource tradeoffs discussed above (partitioned parallelism).

3. Place the tasks corresponding to the leaf nodes of the task tree in the *ready list* L of the scheduler. While L is not empty, perform the following steps:

    3.1. Determine a batch of tasks from L that can be executed concurrently and schedule them using a *provably near-optimal* multi-dimensional list scheduling heuristic (pipelined and independent parallelism).

    3.2. If there are tasks in the tree whose execution is enabled after Step 3.1, place them in the ready list L.

We prove that our approach is *near-optimal* for scheduling multiple independent pipelines. Further, it can be readily used to handle *on-line* task arrivals (e.g., in a dynamic or multi-query execution environment).

4.2 Quantifying Partitioned Parallelism

4.2.1 Resource Usage Model

Extending the multi-dimensional model presented in Section 3.2.1, we describe the usage by an isolated operator of a site comprising d TS resources and s SS resources by the triple $(T^{seq}, \overline{W}, \overline{V})$, where:

- T^{seq} is the (stand-alone) sequential execution time of the operator,
- $\overline{W}$ is a d-dimensional *work vector* whose components denote the work done on individual TS resources, i.e., the effective time [GHK92] for which each resource is used by the operator; and
- $\overline{V}$ is an s-dimensional *demand vector* whose components denote the SS resource requirements of the operator throughout its execution. For notational convenience we assume that the dimensions of $\overline{V}$ are *normalized* using the corresponding SS capacities of a single site.

This generalized view of a system site is depicted in Figure 10.

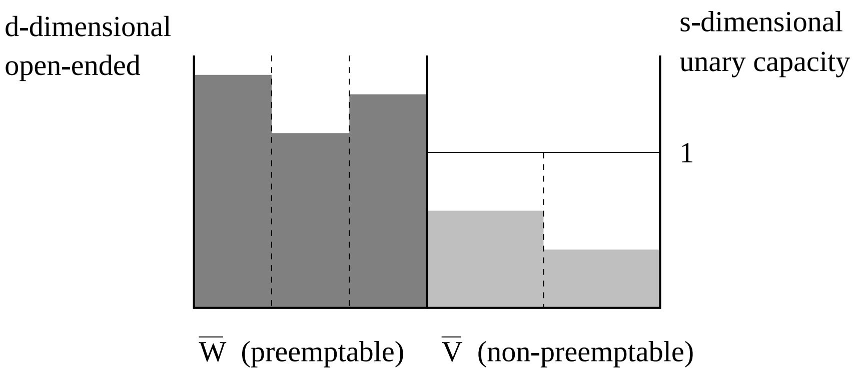


Figure 10: A site with preemptable and non-preemptable resources ($d = 3$, $s = 2$).

As explained earlier, time T^{seq} is actually a function of the operator's individual resource requirements, i.e., its work vector $\overline{W}$ (sometimes emphasized by using $T^{seq}(\overline{W})$ instead of T^{seq}), and the amount of *overlap* that can be achieved between processing at different resources. The operator's SS resource requirements ($\overline{V}$) depend primarily on the size of its inputs and the algorithm used to implement the operator. On the other hand, the operator's work requirements ($\overline{W}$) depend on both of these parameters as well as its SS resource allotment $\overline{V}$.

Note that, in this chapter, we are adopting a somewhat simplified view of the SS resource demands, assuming that components of $\overline{V}$ have fixed values determined by plan parameters.

In most real-life query execution engines, operator memory requirements are *malleable*, in the sense that they are typically specified as a *range* of possible memory allotments. This flexibility adds an extra level of difficulty to our scheduling problem. It means that the scheduler also has to select specific SS demand vectors $\overline{V}$ that minimize query response time over all possible $(\overline{W}, \overline{V})$ combinations. We plan to address this more general scheduling problem in our future work.

4.2.2 Quantifying Execution Granularity in the Presence of SS Resources

As explained in Section 3.2.2, employing coarse grain operator parallelizations ensures efficient (i.e., low overhead) parallel query execution. In the presence of SS resources, however, any scheduling method is restricted in its mapping of clones to sites by SS capacity constraints, i.e., it is not possible to concurrently execute a set of clones at a site if their total SS requirements exceed the site's capacity (in any of the s dimensions). Clearly, coarse operator clones imply that each clone has SS resource requirements that are relatively large. This means that when restricted to coarse grain operator executions a scheduling method can be severely limited in its ability to balance the total work across sites. Furthermore, coarse SS requests can cause severe fragmentation that may lead to underutilization of system resources. Thus, taking both TS and SS resources into account gives rise to interesting tradeoffs with respect to the granularity of operator clones. Our analytical results in Section 4.3 clearly demonstrate this effect.

To account for SS resource constraints, we view the granularity of a parallel operator $\mathtt{op}$ as a function of the ratio $\frac{W_p(\mathtt{op})}{W_c(\mathtt{op},N)}$ and $V(\mathtt{op}, N)$, where

- $W_p(\mathtt{op})$ and $W_c(\mathtt{op}, N)$ denote the processing and communication area of the execution of $\mathtt{op}$(Section 3.2.2); and
- $V(\mathtt{op}, N)$ denotes the maximum (normalized) SS resource requirement of any clone when the execution of $\mathtt{op}$ is partitioned among N clones; it corresponds to the SS *grain size* of the partitioned execution of $\mathtt{op}$ and is a non-increasing function of N.

Note that the execution of $\mathtt{op}$ with degree of partitioned parallelism equal to N is feasible only if $V(\mathtt{op}, N) \leq 1$; that is, the partitioning of $\mathtt{op}$ must be sufficiently fine-grain for each clone to be able to maintain its SS working set at a site. We only consider such "reasonable" operator parallelizations in the remainder of the chapter.

Definition 4.2.1 A parallel execution of an operator $\mathtt{op}$ with degree of partitioned parallelism equal to N is λ-*granular* if $V(\mathtt{op}, N) \leq \lambda$, where $\lambda \leq 1$. $\square$

The following quantification of coarse grain parallelism extends our earlier formulation.

Definition 4.2.2 A parallel execution of an operator $\texttt{op}$ with degree of partitioned parallelism equal to N is λ-*granular* CG_f, if the communication area of the execution is no more than f' times the processing area of $\texttt{op}$, i.e., $W_c(\texttt{op}, N) \leq f'\, W_p(\texttt{op})$, where f' is the minimum value larger than or equal to f such that $V(\texttt{op}, N) \leq \lambda$. □

The intuition behind this definition is that we may sometimes have to compromise our restrictions on communication overhead to ensure that the parallelization is in the λ-granular region. This is graphically demonstrated in Figure 11.

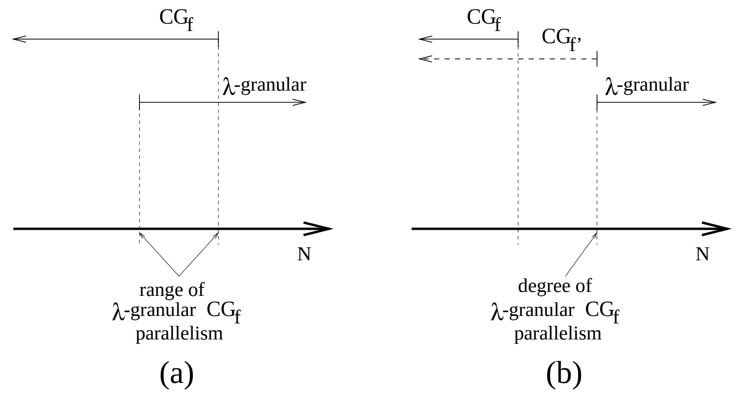


Figure 11: λ-granular CG$_f$ execution with (a) $f = f'$, and (b) $f < f'$.

4.2.3 Degree of Partitioned Parallelism

Assuming zero communication costs, the TS and SS resource requirements of an operator are described by a d-dimensional work vector $\overline{W}$ and an s-dimensional demand vector $\overline{V}$ whose components can be derived from system parameters and traditional optimizer cost models [SAC+79]. The processing area $W_p(\texttt{op})$ is simply the sum of $\overline{W}$'s components, and the communication area $W_c(\texttt{op}, N)$ is estimated using our linear model of communication costs (Section 3.2.3). The following proposition extends Proposition 3.2.1 and is an immediate consequence of Definition 4.2.2 and our communication cost model.

Proposition 4.2.1 The maximum allowable degree of partitioned parallelism for a λ-granular CG$_f$ execution of operator $\texttt{op}$ is denoted by $N_{max}(\texttt{op}, f, \lambda)$ and is determined by the formula

$$N_{max}(\texttt{op}, f, \lambda) = \max\{1, \left\lfloor \frac{f\, W_p(\texttt{op}) - \beta\, D}{\alpha} \right\rfloor,\ \min\{\, N : V(\texttt{op}, N) \leq \lambda \,\}\}.$$

□

4.3 The Scheduling Algorithm

4.3.1 Notation and Definitions

Table 4 summarizes the main extensions to the notation of the previous chapter (Section 3.3.1) to deal with the addition of SS resources. Detailed definitions of some of these parameters are given below. Additional notation will be introduced when necessary.

Parameter	Semantics
s	Number of SS resources per site
B_j^V	Set of SS demand vectors scheduled at site B_j
$\overline{V}_{\mathsf{op}_i}$	Demand vector for op_i
S^V	Set of demand vectors for all clones to be scheduled
S^{TV}	Set of *volume* (time $\times$ demand) vectors for all clones to be scheduled

Table 4: Additional Notation for SS Resources

Vector $\overline{V}_{\mathsf{op}_i}$ describes the total (normalized) SS resource requirements of op_i. The components of $\overline{V}_{\mathsf{op}_i}$ are computed using architectural parameters and database statistics. Note that these components are independent of the degree of partitioned parallelism N_i (assumption A6).

Given an operator clone with a (stand-alone) execution time of T and a SS demand of $\overline{V}$, we define the *volume vector* of the clone as the product $T \cdot \overline{V}$, i.e., the resource-time product[2] for the clone's execution [CM96]. S^W, S^V, and S^{TV} are used to denote the set of work, demand, and volume vectors (respectively) for the set S of all the clones to be scheduled. We use the W, V, and TV superscripts in this manner throughout this chapter.

4.3.2 Modeling Parallel Execution and Resource Sharing

In this section, we discuss the extensions necessary to our earlier multi-dimensional query execution model to account for the introduction of SS resources.

Partitioned and Independent Parallelism

Consider an operator op_i with degree of partitioned parallelism equal to N_i. The partitioning of $\overline{W}_{\mathsf{op}_i}$ and $\overline{V}_{\mathsf{op}_i}$ into work and demand vectors for operator clones is determined based on statistical information kept in the DBMS catalogs. Given such a partitioning $<(\overline{W}_1,\overline{V}_1),(\overline{W}_2,\overline{V}_2),\ldots,(\overline{W}_{N_i},\overline{V}_{N_i})>$, where $\sum_{k=1}^{N_i}\overline{W}_k = \overline{W}_{\mathsf{op}_i}$ and $\sum_{k=1}^{N_i}\overline{V}_k = \overline{V}_{\mathsf{op}_i}$, a lower

[2]The *volume* of an operator is defined as the product of the amount of resource(s) that the operator reserves during its execution and its execution time.

bound on the parallel execution time for $\mathtt{op}_i$ is the maximum of the sequential execution times of its N_i clones; that is, the parallel execution time for $\mathtt{op}_i$ is greater than or equal to

$$T^{max}(\mathtt{op}_i, N_i) = \max_{1 \le k \le N_i} \{ T^{seq}(\overline{W}_k) \}.$$

By our definitions of the TS and SS resource classes, it is obvious that a collection of operator clones $< (\overline{W}_1, \overline{V}_1), (\overline{W}_2, \overline{V}_2), \ldots, (\overline{W}_k, \overline{V}_k) >$ can be executed concurrently at some system site *only if* $l(\sum_1^k \overline{V}_i) \le 1$, i.e., their SS requirements do not exceed the capacity of the site. We call such clone collections *compatible*.

Definition 4.3.1 Given a collection of M independent operators $\{\mathtt{op}_i, i = 1 \ldots M\}$ and their respective degrees of partitioned parallelism $\{N_i, i = 1 \ldots M\}$, a *schedule* is a partitioning of the $\sum_{i=1}^M N_i$ operator clones into a collection of compatible subsets $S_1, \ldots, S_n$ followed by a mapping of these subsets to the set of available sites. □

Consider such a compatible subset of clones S_i and let S_i^W denote the set of work vectors for all clones in S_i. Following Equation 2, the execution time for all the operator clones in S_i is defined as

$$T(S_i) = \max\{ \max_{\overline{W} \in S_i^W} \{T^{seq}(\overline{W})\} , \, l(S_i^W) \}.$$

Thus, if we let $S(B_j)$ denote the collection of compatible subsets mapped to B_j under a given schedule SCHED, the execution time for site B_j is

$$T^{site}(B_j) = \sum_{S_i \in S(B_j)} T(S_i) = \sum_{S_i \in S(B_j)} \max\{ \max_{\overline{W} \in S_i^W} \{T^{seq}(\overline{W})\} , \, l(S_i^W) \}. \tag{4}$$

Clearly, the response time of SCHED is determined by the longest running site, that is, $T^{par}(\mathrm{SCHED}, P) = \max_{1 \le j \le P}\{ T^{site}(B_j) \}$.

Pipelined Parallelism

Pipelined parallelism introduces a co-scheduling requirement for query operators, requiring a collection of clones to execute in producer-consumer pairs using fine-grain/lock-step synchronization. The problems with load-balancing a pipelined execution have been identified in previous work [Gra93]. Compared to our model of a schedule for partitioned and independent parallelism (Definition 4.3.1), pipelined execution constrains the placement and execution of compatible clone subsets to ensure that all the clones in a pipe run concurrently – they all start and terminate at the same time [HM94]. This means that it is no longer possible to schedule resources at one site independent of the others, as we suggested in the previous section. Compatible subsets containing clones from the same pipeline must run concurrently. Furthermore, given that the scheduler is not allowed to modify the query plan, scheduling a pipeline is an "all-or-nothing" affair: either all clones will execute in parallel or none will. The implications of pipelined parallelism for our scheduling problem will be studied further in Section 4.3.4 where a near-optimal solution will be developed.

4.3.3 Scheduling Independent Operators

In this section, we extend our earlier lower bound on the optimal parallel execution time of independent operators (i.e., operators not in any pipeline) with a new term that accounts for the effect of SS resources. We then demonstrate that a heuristic based on Graham's LPT (Largest Processing Time) *list scheduling* method [Gra69] can guarantee near-optimal schedules for such operators.

Lemma 4.3.1 Let $\{\mathtt{op}_i, i = 1, \ldots M\}$ be a collection of independent operators with respective degrees of partitioned parallelism $\underline{N} = (N_1, N_2, \ldots, N_M)$. Let S be the corresponding collection of clones and define $T^{max}(S) = \max_{i=1,\ldots,M}\{T^{max}(\mathtt{op}_i, N_i)\}$. If $T^{par}(\mathrm{OPT}, P)$ is the response time of the optimal execution on P sites then $T^{par}(\mathrm{OPT}, P) \geq LB(S, P)$, where

$$LB(S,P) = \max\{\ T^{max}(S)\ ,\ \frac{l(S^W)}{P}\ ,\ \frac{l(S^{TV})}{P}\ \}.$$

□

Compared to our earlier results (Lemma A.1.2), the lower bound in Lemma 4.3.1 introduces a third term containing $l(S^{TV})$, i.e., the total *volume* of the parallel execution. We will see that this new parameter plays an important role in our analytical and experimental results.

The basic idea of our heuristic scheduling algorithm, termed OperatorSched, is to construct the partition of clones into compatible subsets incrementally, using a Next-Fit rule [CGJ84, CGJT80]. Specifically, OperatorSched scans the list of clones in non-increasing order of execution time. At each step, the clone selected is placed in the site B_i of minimal height $T^{site}(B_i)$ (see Equation 4). This placement is done as follows. Let S_{i,n_i} denote the topmost compatible subset in B_i. If the clone can fit in S_{i,n_i} without violating SS capacity constraints, then add the clone to S_{i,n_i} and update $T^{site}(B_i)$ accordingly. Otherwise, set $n_i = n_i + 1$, place the clone by itself in a new topmost subset S_{i,n_i}, and set $T^{site}(B_i)$ accordingly. The following theorem establishes an absolute performance bound of $d + 2s + 2$ for our heuristic.

Theorem 4.3.1 Given a set of clones S, OperatorSched runs in time $O(|S| \log |S|)$ and produces a schedule SCHED with response time

$$T^{par}(\text{SCHED}, P) \;<\; d \cdot \frac{l(S^W)}{P} \;+\; 2s \cdot \frac{l(S^{TV})}{P} \;+\; 2 \cdot T^{max}(S) \;\leq\; (d + 2s + 2) \cdot LB(S, P).$$

□

Theorem 4.3.1 guarantees an *asymptotic performance bound*[3] of $d+2s$ for OperatorSched. This bound gives us a feeling for the performance of the algorithm when the optimal response time is much larger that the longest execution time of all clones and is a better measure of performance when $|S|$ is large [CGJT80, BS83].

Note that our scheduling algorithm combines the list scheduling method of Graham [Gra69] with the Next-Fit Decreasing Height (NFDH) shelf-based algorithm of Coffman et al. [CGJT80]. Using the First-Fit Decreasing Height (FFDH) algorithm in place of NFDH, we can use the methodology of Coffman et al. to demonstrate a $d + 1.7s$ asymptotic performance bound.

4.3.4 Scheduling with Pipelining Constraints

The co-scheduling requirement of pipelined operator execution introduces an extra level of complexity that OperatorSched cannot address, namely the problem of deciding whether a pipeline is *schedulable* on a given number of sites. Given a collection of operator clones in a pipeline, the schedulability question poses an $\mathcal{NP}$-hard decision problem that essentially corresponds to the decision problem of s-dimensional vector packing [CGJ84]. Thus, it is highly unlikely that efficient (i.e., polynomial time) necessary and sufficient conditions for pipeline schedulability exist. Note that no such problems were raised in the previous section, since the clones were assumed to be feasible (i.e., 1-granular) and executing independently of each other.

In this section, we show that λ-granularity with $\lambda < 1$ for all operator parallelizations can provide an easily checkable sufficient condition for pipeline schedulability[4]. Once schedulability is ensured, balancing the work of the pipeline across sites to minimize its response time still poses an $\mathcal{NP}$-hard optimization problem. We present a polynomial time scheduling algorithm that is within a constant multiplicative factor of the response time lower bound for schedulable λ-granular pipelines. Further, we demonstrate that using a level-based approach, our methodology can be extended to provide a provably near-optimal solution for multiple independent pipelines (i.e., query tasks). Finally, we extend our techniques to handle the data dependencies in a bushy query plan and on-line task arrivals.

[3] An asymptotic performance bound characterizes the behavior of an algorithm as the ratio of the optimal makespan to the longest job processing time goes to infinity, i.e., when $\frac{T^{par}(\text{OPT},P)}{T^{max}(S)} \to \infty$.

[4] We use the term λ-*granular pipeline* to describe a pipeline in which all operator parallelizations are λ-granular.

Scheduling a Single λ-granular Pipeline

We present a near-optimal algorithm for scheduling a pipeline C consisting of λ-granular parallel operators, where $\lambda < 1$. Let S_C denote the collection of clones in C and define S_C^W, S_C^V, and $T^{max}(S_C)$ in the obvious manner. Note that, by our definitions, the pipeline C will require *at least* $l(S_C^V)$ sites for its execution (otherwise, λ would have to be greater than 1). The following lemma provides a sufficient condition for the schedulability of a λ-granular pipeline.

Lemma 4.3.2 The number of sites required to schedule a λ-granular pipeline C ($\lambda < 1$) is always less than or equal to $\frac{l(S_C^V)\cdot s}{1-\lambda}$. Furthermore, this bound is tight. □

Our heuristic, PIPESCHED, belongs to the family of list scheduling algorithms [Gra69]. PIPESCHED assumes that it is given a number of sites P_C that is sufficient for the scheduling of C, according to the condition of Lemma 4.3.2. The algorithm considers the clones in S_C in non-increasing order of their *work density ratio* $\frac{l(\overline{W}_i)}{l(\overline{V}_i)}$. At each step, the clone under consideration is placed in the least filled (i.e., least work) site that has sufficient SS resources to accommodate it; that is, clone $(\overline{W}_i, \overline{V}_i)$ is packed in bin B such that $l(B^W)$ is minimal among all sites B_j such that $l(B_j^V \cup \{\overline{V}_i\}) \leq 1$. The PIPESCHED algorithm is depicted in Figure 12.

Algorithm PIPESCHED(C, P_C)

Input: A set of λ-granular pipelined operator clones S_C and a set of P_C sites, where $P_C \geq \frac{l(S_C^V)\cdot s}{1-\lambda}$ (see Lemma 4.3.2).
Output: A mapping of the clones to sites that does not violate SS resource constraints. (Goal: Minimize response time.)

1. let $L =< (\overline{W}_1, \overline{V}_1), \ldots, (\overline{W}_N, \overline{V}_N) >$ be the list of all clones in *non-increasing* order of $\frac{l(\overline{W}_i)}{l(\overline{V}_i)}$.
2. for $k = 1$ to N do
   - 2.1. let $SB_k = \{B_j : l(B_j^V \cup \{\overline{V}_k\}) \leq 1\}$, i.e, the set of bins with sufficient SS resources for the k^{th} clone.
   - 2.2. let $B \in SB_k$ be a site such that $l(B^W) = \min_{B_j \in SB_k}\{l(B_j^W)\}$.
   - 2.3. place clone $(\overline{W}_N, \overline{V}_N)$ at site B and set $B^W = B \cup \{\overline{W}_N\}$, $B^V = B \cup \{\overline{V}_N\}$.

Figure 12: Algorithm PIPESCHED

The following theorem establishes an asymptotic upper bound of $d \cdot (1 + \frac{s}{1-\lambda})$ on the worst-case performance ratio of our algorithm.

Theorem 4.3.2 Given a λ-granular pipeline C ($\lambda < 1$) PIPESCHED runs in time $O(|S_C| \log |S_C|)$

and produces a schedule SCHED with response time

$$T^{par}(\text{SCHED}, P_C) \leq d(1+\frac{s}{1-\lambda}) \cdot \frac{l(S_C^W)}{P_C} + T^{max}(S_C) \leq [d(1+\frac{s}{1-\lambda})+1] \cdot LB(S_C, P_C).$$

□

Note that the volume term does not come into the expression for the performance bound of PipeSched. This is because, by definition, all the clones in S_C must execute in parallel and thus $l(S_C^V) \leq P_C$. Thus, for a single pipeline, we always have $\frac{l(S_C^{TV})}{P_C} \leq T^{max}(S_C) \cdot \frac{l(S_C^V)}{P_C} \leq T^{max}(S_C)$.

The bound established in Theorem 4.3.2 clearly captures the granularity tradeoffs identified in Section 4.2. Increasing the degree of partitioned parallelism decreases both $T^{max}(S_C)$ and λ allowing for a better asymptotic bound on the ratio and a smaller additive constant. On the other hand, it also increases the total amount of work $l(S_C^W)$ because of the overhead of parallelism. The importance of such work-space tradeoffs for parallel query processing and optimization has been stressed in recent work [HFV96].

Scheduling Multiple Independent λ-granular Pipelines

The basic observation here is that the PipeSched algorithm presented in the previous section can be used to schedule any collection of independent pipelines as long as schedulability is guaranteed by Lemma 4.3.2.

Our algorithm for scheduling multiple independent pipelines uses a Next-Fit Decreasing Height (NFDH) policy [CGJT80] in conjunction with Lemma 4.3.2 to identify pipelines that can be scheduled to execute concurrently on P sites (i.e., in one layer of execution). PipeSched is then used for determining the execution schedule within each layer. The overall algorithm, LevelSched, is formally outlined in Figure 13.

The following theorem gives an upper bound on the worst-case performance ratio of LevelSched. Note that the co-scheduling requirement for the clones in a pipe implies that the total volume for all the clones in $\{C_1, \ldots, C_N\}$ is $l(S^{TV}) = l(\sum_1^N T_{C_i}^{max} \cdot \sum_{\overline{v} \in S_{C_i}^V} \overline{v})$, since any clone in C_i will require its share of ss resources for at least $T_{C_i}^{max}$ time. The lower bound in Lemma 4.3.1 holds using the above definition of volume.

Theorem 4.3.3 Given a collection of N independent λ-granular pipelines ($\lambda < 1$) consisting of the set of clones S, LevelSched runs in time $O(N|S| \log P|S|)$ and produces a schedule SCHED with response time

$$\begin{aligned} T^{par}(\text{SCHED}, P) &< d^2(1+\frac{s}{1-\lambda}) \cdot \frac{l(S^W)}{P} + \frac{2s^2}{1-\lambda} \cdot \frac{l(S^{TV})}{P} + T^{max}(S) \\ &\leq [d^2(1+\frac{s}{1-\lambda}) + \frac{2s^2}{1-\lambda} + 1] \cdot LB(S, P). \end{aligned}$$

Algorithm LEVELSCHED($\{C_1, \ldots, C_N\}$, P)
Input: A set of λ-granular operator pipelines $\{C_1, \ldots, C_N\}$ and a set of P sites.
Output: A mapping of clones to sites that does not violate SS resource constraints or pipelining dependencies. (Goal: Minimize response time.)

1. Sort the pipelines in non-increasing order of T^{max}, i.e., let $L =< C_1, \ldots, C_N >$, where $T^{max}(S_{C_1}) \geq \ldots \geq T^{max}(S_{C_N})$.
2. Partition the list L in k maximal schedulable sublists: $L_1 =< C_1, \ldots, C_{i_1} >$, $L_2 =< C_{i_1+1}, \ldots, C_{i_2} >$, …, $L_k =< C_{i_{k-1}+1}, \ldots, C_N >$ based on Lemma 4.3.2. That is,
$$l(\cup_{C \in L_j} S_C^V) \leq \frac{P(1-\lambda)}{s} \quad \text{and} \quad l((\cup_{C \in L_j} S_C^V) \cup S_{C_{i_j+1}}^V) > \frac{P(1-\lambda)}{s}, \quad \text{for all } j = 1, \ldots, k-1.$$
3. for $j = 1, \ldots, k$ do
   3.1. call PIPESCHED($(\cup_{C \in L_j} C)$, P)

Figure 13: Algorithm LEVELSCHED

□

It is important to note that the lower bound estimated in Lemma 4.3.1 will, in most cases, significantly underestimate the optimal response time since it assumes that 100% utilization of system resources is always possible *independent of the given task list*. Thus, the quadratic multiplicative constants in Theorem 4.3.3 reflect only a worst case that is rather far from the average, as our experimental results have verified as well.

4.3.5 Handling Data Dependencies and On-Line Task Arrivals

Scheduling arbitrary query task trees must ensure that the blocking constraints specified by the tree's edges are satisfied. The LEVELSCHED algorithm can be readily extended to handle such constraints by ensuring that the (sorted) ready list of tasks L always contains the collection of query tasks that are ready for execution, i.e., they are not blocked waiting for the completion of some other (descendant) task in the task tree. In addition, as mentioned in Section 3.1, care must be taken to ensure that co-scheduling dependencies *across sibling tasks* and timing constraints across blocking edges are maintained. For example, the operators of all sibling `build`-tasks in the same join pipeline must be co-scheduled and those of the parent `probe`-task must be treated as rooted and scheduled in the immediately following shelf. All this is done by modifying LEVELSCHED as follows (see Figure 13):

1. Any sibling pipelines in the task tree with a co-scheduling dependency are treated as a unit, i.e., the way individual pipelines are treated in LevelSched. For the purposes of this algorithm, assume that the term "pipeline" is interpreted as such a unit.

2. Initially, the input set of pipelines $\{C_1, \ldots, C_N\}$ contains exactly the tasks at the leaf nodes of the query task tree.

3. After Step 3.1, determine the set of tasks $\mathbf{C}$ that have been enabled (i.e., are no longer blocked) because of the last invocation of PipeSched. If $\mathbf{C} \neq \emptyset$, then merge the tasks in $\mathbf{C}$ into the ready list L and go to Step 2. Otherwise, continue with the next invocation of PipeSched.

The exact same idea of dynamically updating and partitioning the ready list L can be used to handle *on-line* task arrivals in a dynamic or multi-query environment. Basically, newly arriving query tasks are immediately merged into L to participate in the partitioning of L into schedulable sublists right after the completion of the current execution layer. Thus, our layer-based approach provides a uniform scheduling framework for handling intra-query as well as inter-query parallelism.

As we have already indicated in Chapter 3, deriving performance bounds in the presence of data dependencies is a very difficult problem that continues to elude our efforts. The difficulty stems from the interdependencies between different execution layers: scheduling decisions made at earlier layers can impose data placement and operator execution constraints on the layers that follow. We leave this problem open for future research.

4.4 Experimental Performance Evaluation

In this section, we present the results of several experiments we have conducted using cost model computations for various query operators in order to examine the average-case performance of our scheduling algorithms. To the best of our knowledge, the issue of complex query scheduling with both TS and SS resources has not been addressed in prior work on databases or deterministic scheduling theory. Given this lack of adversaries, our experiments focus on how close the response time of the generated schedule is to a lower bound on the response time of the optimal execution schedule (Lemma 4.3.1) *on the average*. We start by presenting our experimental testbed and methodology.

4.4.1 Experimental Testbed

We have experimented with the following algorithms:

- LEVELSCHED : Level-based scheduling of multiple independent query tasks (i.e., operator pipelines).

- TREESCHED : Level-based scheduling of query task trees, observing blocking constraints in the tree as well as main-memory dependencies across execution levels (Section 4.3.5).

For both scheduling scenarios (independent tasks, task trees), we compared the average performance of our scheduling algorithms with a lower bound on the response time of the optimal execution schedule for the given degrees of partitioned parallelism (determined by the granularity parameters λ and f). These lower bounds for independent query tasks (LEVELBOUND) and query task trees (TREEBOUND) were estimated using the formulas:

$$\begin{aligned} \text{LevelBound} &= \max\{ \frac{l(S^W)}{P}, \frac{l(S^{TV})}{P}, T^{max}(S) \} \quad \text{and} \\ \text{TreeBound} &= \max\{ \frac{l(S^W)}{P}, \frac{l(S^{TV})}{P}, T(\text{CP}) \}, \end{aligned}$$

where S is the collection of all operator clones in the query tasks and task tree (respectively), $T^{max}(S)$ is the maximum stand-alone execution time among all clones in S, and $T(\text{CP})$ is the total response time of the critical (i.e., most time-consuming) path in the query task tree.

Once again, some additional assumptions were made to obtain a specific experimental model from the general parallel execution model described in Sections 4.2 and 4.3. More specifically, in addition to experimental assumptions **EA1** and **EA2** of Section 3.4, we also made the following assumption to estimate the SS demand vectors for join operators.

EA3. Simple Hash-Join Plan Nodes: The query tasks handed to our algorithms are generated by bushy hash-join plans, where the memory demand for each join equals the size of the inner relation times a "fudge factor" accounting for the hash table overhead. Note that even though it is possible to execute a hash-join with less memory [Sha86], such memory limitations complicate the processing of multi-join pipelines – since `probe` operators cannot keep their entire data sets (i.e., inner hash tables) in memory, it is no longer possible to execute the `probe` pipeline in one pass. This means that intermediate disk I/O has to be performed at one or more pipeline stages, essentially modifying the original plan with the addition of extra blocking and data dependencies. Various multi-pass schemes for pipelined hash-join execution under limited memory were studied in the work of Schneider [Sch90a]. As part of our future work, we plan to investigate the effects of such memory limitations on our scheduling methodology and results.

Finally, we should note that our implementations of TREESCHED and LEVELSCHED incorporated an additional optimization to the basic scheme shown in Figure 13: *After the placement* *of a schedulable sublist (Lemma 4.3.2) of ready pipelines, each remaining ready pipeline was checked (in non-increasing order of T^{max}) for possible inclusion in the current level before starting the next execution level.* Although this optimization does not help improve worst-case performance (since Lemma 4.3.2 is tight), we found that it really helped the average-case performance of the heuristics at the cost of a small increase in running time.

We experimented with tree queries of 10, 20, 30, 40, and 50 joins. For each query size, twenty query graphs (trees) were randomly generated and for each graph an execution plan was selected in a random manner from a bushy plan space. We assumed simple key join operations in which the size of the result relation is always equal to the size of the largest of the two join operands. For algorithm LEVELSCHED, all data dependencies in the query task trees were removed to obtain a collection of independent tasks from each tree query. We used two performance metrics in our study: (1) the *average performance ratio* defined as the response time of the schedules produced by our heuristics divided by the corresponding lower bound and averaged over all queries of the same size; and, (2) the *average response time* of the schedules produced by our heuristics over all queries of the same size. Experiments were conducted with the resource overlap parameter ϵ varying between 10% and 70%, the coarse granularity parameter f varying between 0.3 and 0.9, and the SS granularity parameter λ varying between 0.1 and 0.9. Since the effects of f and ϵ on scheduler performance were also studied in Section 3.4, the experiments discussed in this chapter mostly concentrate on the new parameter λ. (The results presented in the next section are indicative of the results obtained for all values of f and ϵ.)

In all experiments, we assumed system nodes consisting of $d = 3$ TS resources (one CPU, one disk unit, and one network interface) and $s = 1$ SS resource (memory). The work vector components for the CPU and the disk were estimated using the cost model equations given by Hsiao et al. [HCY94]. The communication costs were calculated using the model described in Section 4.2. The memory requirement of hash-join operators was estimated as F times the size of the inner join relation, where F is a "fudge factor" capturing the hash table overheads [Sha86]. The values of the cost model parameters were obtained from the literature [HCY94, GW93, WFA92] and are summarized in Table 5.

4.4.2 Experimental Results

The first set of experiments studied the average-case behavior of our level-based TREESCHED heuristic for different query, system, and memory sizes and various settings for the λ, f, and ϵ parameters. Figure 14(a) depicts the average performance ratio of TREESCHED for queries of 40 joins and 32MB of memory at each site, assuming a coarse-granularity parameter $f = 0.6$ and a resource overlap of 50% (i.e., $\epsilon = 0.5$). Note that our algorithm is consistently within a small

Configuration/DB Catalog Param.	Value
Number of Sites	10 – 120
CPU Speed	10 MIPS
Memory per Site	16 – 64 MB
Effective Disk Service Time per page	5 msec
Startup Cost per site (α)	15 msec
Network Transfer Cost per byte (β)	0.3 μsec
Fudge Factor (F)	1.4
Tuple Size	128 bytes
Page Size	32 tuples
Relation Size	10^4 – 10^6 tuples

CPU Cost Parameters	No. Instr.
Read Page from Disk	5000
Write Page to Disk	5000
Extract Tuple	300
Hash Tuple	100
Probe Hash Table	200

Table 5: Cost Model Experiments: Parameter Settings

constant factor (i.e., less than 2) of the lower bound on the optimal schedule length. Although the distance from the lower bound has certainly increased compared to our results for only TS resources, the results clearly demonstrate that the worst-case multiplicative factors derived in our analytical bounds are overly pessimistic as far as average performance is concerned.

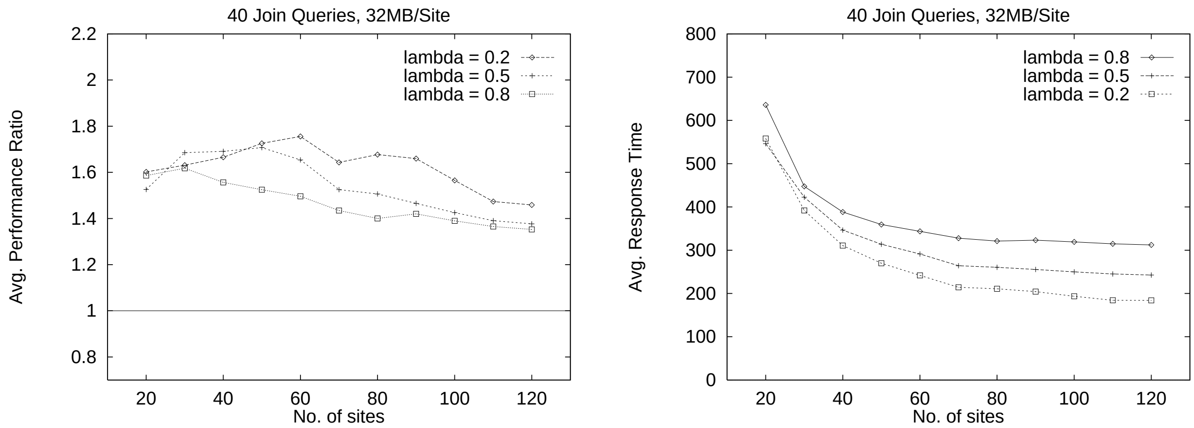


Figure 14: Effect of λ on (a) the average performance ratio of TreeSched, and (b) the average schedule response times obtained by TreeSched. ($f = 0.6$, $\epsilon = 0.5$)

Observing Figure 14(a), it appears that TreeSched performs better for larger values of the memory granularity parameter λ. This is slightly counterintuitive and seems to contradict Section 4.3.4: "finer" memory requirements should allow our schedulers to obtain better load balancing and, consequently, better schedules. However, as shown in Figure 14(b), although the *performance ratio* of the algorithms improves with larger values of λ, the *actual performance* (i.e., the response time) of the schedules deteriorates with larger λ, as expected. The explanation of this phenomenon lies in Figure 15 which shows how the three components of the TreeBound lower bound vary with the number of sites for our example set of 40-join queries and 16MB of memory per site. (We chose a smaller value for memory because it better illustrates the effect of the volume term for our system setting.)

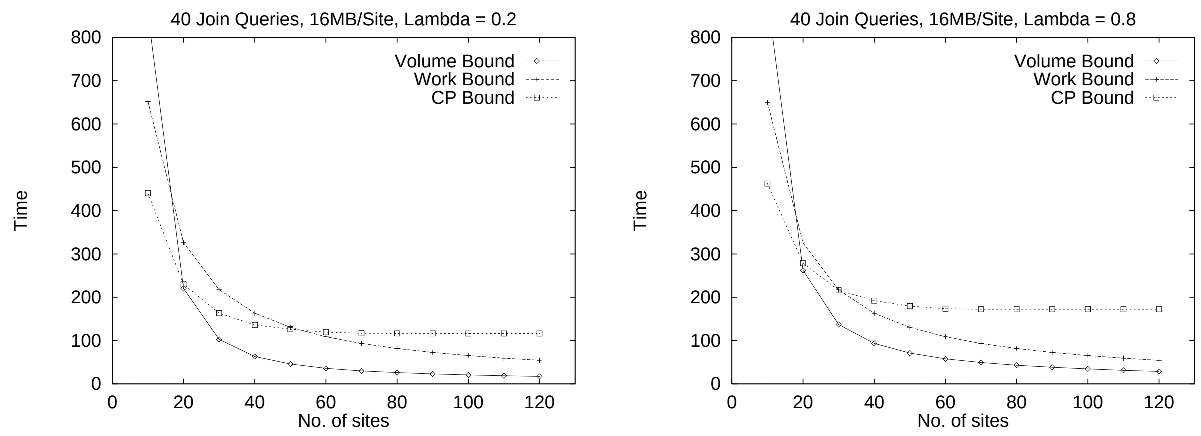


Figure 15: TreeBound components for (a) $\lambda = 0.2$, and (b) $\lambda = 0.8$. ($f = 0.6$, $\epsilon = 0.5$)

Note that for small values of the number of sites P the dominant factor in TreeBound is the average volume term ($\frac{l(S^{TV})}{P}$). For larger values of P (and, consequently, increased system memory), TreeBound is determined by the average work term ($\frac{l(S^{W})}{P}$). Eventually, as P continues to grow, the critical path term ($T(\mathrm{CP})$) starts dominating the other two terms in the bound. Also, note that as the critical path becomes the dominant factor in the query plan execution, our level-based methods become more accurate in approximating the lower bound. Intuitively, this is because the "height" of each execution level as determined by the plan's critical path will be sufficient to pack the work in that level and, thus, the resource loss due to "shelving" is not important. (This also explains why the average performance ratios for various values of λ all converge to a value close to unity as the number of sites is increased.) For the parameter settings in our experiments, larger values for the memory granularity λ typically imply lower degrees of parallelism for the operators in the plan, which means that the critical path will start dominating the other two factors in TreeBound much sooner. Furthermore, the aforementioned effect on the performance ratio becomes more pronounced since the critical path term will be significantly larger for larger λ (Figure 16). Consequently, larger values for λ

imply better performance ratios (Figure 14(a)), even though the actual schedule response times are worse (Figure 14(b)).

For our second set of experiments, we removed all the blocking and data dependencies from the query task trees and scheduled the resulting collection of independent query tasks using LevelSched. Again, we focused on the effect of λ on the average performance of our heuristic for different query and system sizes and different amounts of memory per site. The average performance ratio and average response times of the schedules obtained for various values of λ with $f = 0.6$ and $\epsilon = 0.5$ are shown in Figures 16(a) and 16(b), respectively. With respect to our previous discussion, the main observation here is that when all dependencies are removed, smaller values of λ (i.e., "finer" clones) result in both better response times *and* better performance ratios. The reason is that, in the absence of scheduling constraints, "finer" clones allow LevelSched to produce better packings and minimize resource wastage within each level. Also, note that both curves are significantly better (i.e., lower) than those of TreeSched, since the lack of dependencies allows LevelSched to obtain better execution schedules.

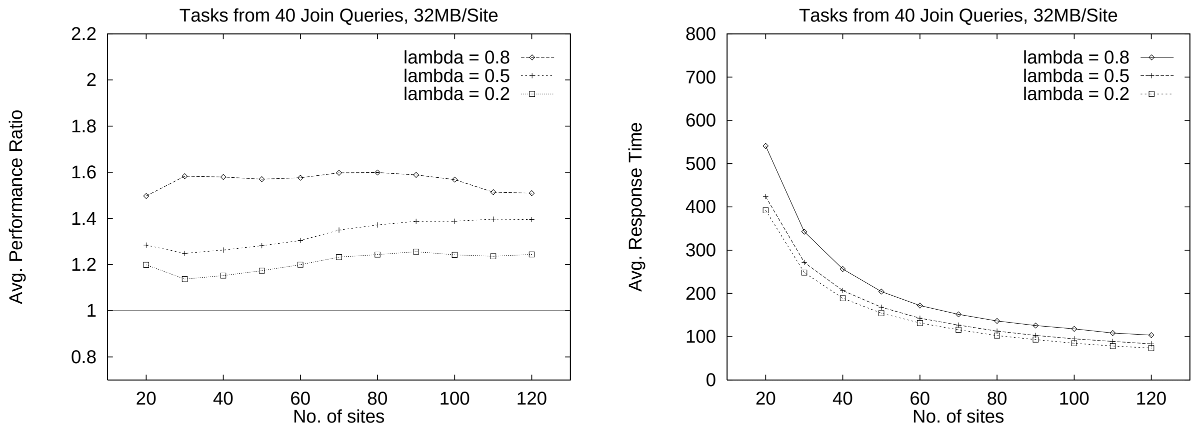


Figure 16: Effect of λ on (a) the average performance ratio of LevelSched, and (b) the average schedule response times obtained by LevelSched. ($f = 0.6$, $\epsilon = 0.5$)

4.5 Parallel Query Optimization

In this section, we study the implications of the analytical and experimental results presented in this chapter for the open problem of designing efficient cost models for parallel query optimization [DG92].

As noted in Section 1.1, the use of *response time* as optimization metric implies that a parallel query optimizer cannot afford to ignore resource scheduling during the optimization process. Prior work has demonstrated that a *two-phase* approach [HS91, Hon92] using the traditional work metric during the plan generation phase often results in plans that are inherently sequential and, consequently, unable to exploit the available parallelism [JPS93, BFG$^+$95]. On the other hand, using a detailed resource scheduling model during plan generation (as advocated by the *one-phase* approach [SE93, JPS93, LVZ93]) can have a tremendous impact on optimizer complexity and optimization cost. For example, a Dynamic Programming (DP) algorithm must use much stricter pruning criteria that account for the use of system resources [GHK92, LVZ93]. This leads to a combinatorial explosion in the state that must be maintained while building the DP tree, rendering the algorithm impractical even for small query sizes.

As in centralized query optimization, the role of the optimizer cost model is to provide an abstraction of the underlying execution system. In this respect, the one- and two-phase approaches lie at the two different ends of a spectrum, incorporating either detailed knowledge (one-phase) or no knowledge (two-phase) of the parallel execution environment in the optimizer cost metric. The goal is to devise cost metrics that are more realistic than total resource consumption, in the sense that they are cognizant of the available parallelism, and at the same time are sufficiently efficient to keep the optimization process tractable.

In recent work, Ganguly et al. [GGS96] suggested the use of a novel scalar cost metric for parallel query optimization. Their metric was defined as the maximum of two "bulk parameters" of a parallel query plan, namely the critical path length of the plan tree and the average work per site. Although the model used in the work of Ganguly et al. was one-dimensional, it is clear that the "critical path length" corresponds to the maximum, over all root-to-leaf paths, sum of T^{max}'s in the task tree, whereas the "average work" corresponds to $\frac{l(S^W)}{P}$ with S being all operator clones in the plan.

Based on our analytical and experimental results, there clearly exists a third parameter, namely the *average volume per site* $\frac{l(S^{TV})}{P}$ that is an essential component of query plan quality. The importance of this third parameter stems from the fact that it is the only one capturing the constraints on parallel execution that derive from SS (i.e., memory) resources.

We believe that the triple *(critical path, average work, average volume)* captures all the crucial aspects characterizing the expected response time of a parallel query execution plan. Consequently, we feel that these three components can provide the basis for an efficient and accurate cost model for parallel query optimizers. Finally, note that although Ganguly et al. [GGS96] suggested combining the plan parameters through a max{} function to produce a scalar metric, the way these parameters are used should depend on the optimization strategy.

For example, a DP-based parallel optimizer should use our three "bulk parameters" as a 3-dimensional vector and use a 3-dimensional "less than" to prune the search space [GHK92]. Clearly, using only three dimensions turns the Partial Order DP (PODP) approach of Ganguly et al. [GHK92] into a feasible and efficient paradigm for DP-based parallel query optimization.

4.6 Conclusions

The problem of scheduling complex queries in hierarchical parallel database systems of multiple time-shared and space-shared resources has been open for a long time both within the database field and the deterministic scheduling theory field. Despite the importance of such architectures in practice, the difficulties of the problem have led researchers in making various assumptions and simplifications that are not realistic. In this chapter, we have provided what we believe is the first comprehensive formal approach to the problem. We have established a model of resource usage that allows the scheduler to explore the possibilities for concurrent operations sharing both TS and SS resources and quantify the effects of this sharing on the parallel execution time. The inclusion of both types of resources has given rise to interesting tradeoffs with respect to the degree of partitioned parallelism, which are nicely exposed within our analytical models and results, and for which we have provided some effective resolutions. We have provided efficient, near-optimal heuristic algorithms for query scheduling in such parallel environments, paying special attention to various constraints that arise from the existence of SS resources, including the co-scheduling requirements of pipelined operator execution, which has been the most challenging to resolve. Our set of results apply to all types of query plans and even sets of plans that are either provided all at the beginning or arrive dynamically for scheduling. As a side-effect of our effort, we have identified an important parameter that captures one aspect of parallel query execution cost, which should play an important role in obtaining realistic cost models for parallel query optimization.

Chapter 5

Performance Evaluation Using Simulation

In this chapter, we present a performance study of our parallel query scheduling algorithms conducted using a detailed simulation model for a hierarchical/shared-nothing parallel database system [Bro94]. The simulator was written in the CSIM/C++ process-oriented simulation language [Sch90b] and was based on the Gamma parallel database machine [DGS$^+$90].

Figure 17 gives a high-level overview of the experimentation procedure employed in this chapter, in comparison with the analytical model experiments presented in earlier chapters. In both settings, the scheduling algorithms investigated receive as input a set of query execution plans along with schema and system configuration information (e.g., declustering of base relations, sizes of intermediate results, disk and CPU characteristics) and produce as output an execution schedule (i.e., a mapping of query plan operators onto system sites). The scheduling algorithm makes its mapping decisions *off-line*, based on its input and (possibly) on cost model computations. For example, LevelSched makes its scheduling choices using a cost model to determine the work and demand vectors for operator clones and taking the componentwise sum of vectors mapped to a specific site to estimate the expected TS and SS resource loads at that site. The next step is to determine the actual response time of the schedule. In our analytical model experiments, this was done simply by using the site workload estimates of the scheduling algorithms. In this chapter, however, the schedule response time is determined by *actually executing* the schedule on a detailed simulation model of a hierarchical parallel database system. As we will see, this introduces many elements of realism that were abstracted out in our analytical model and earlier experiments.

The following section describes the various components of our simulation model in detail, focusing on the differences between the simulator environment and the query execution environment assumed in the last two chapters. We then go on to describe our experimental testbed and discuss the simulation results.

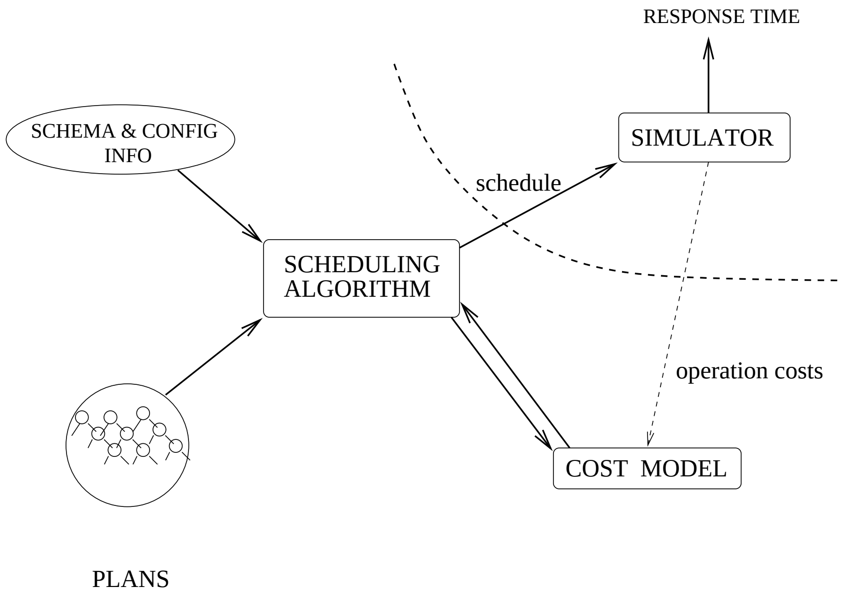


Figure 17: Experimentation Procedure.

5.1 Execution Environment

5.1.1 Query Processing Architecture

The system consists of a collection of *query processing sites*, comprising a CPU, memory, and one or more disk drives. The sites use an interconnection network for all communication. Query plans are submitted for execution from external terminals to a `GlobalScheduler` process running on a dedicated *scheduler site* that lies outside the set of query processing sites in the system. The `GlobalScheduler` determines a mapping of plan operators to sites, allocates the appropriate resources for the plan, and finally initiates a `QueryScheduler` process on the scheduler site that is responsible for executing the plan to completion. For each operator to be executed, the `QueryScheduler` creates an execution thread on every site executing a clone of the operator. For each such thread, appropriate communication channels are set up for the exchange of control messages between the `QueryScheduler` and the thread (e.g., transition from the `build`-phase to the `probe`-phase of a join pipeline, split table broadcasting, and "clone done" messages) and data messages between threads of neighboring operators in the plan. When a plan is run to completion the corresponding `QueryScheduler` dies and a reply message is sent to the submitting terminal. The `GlobalScheduler` may then choose to activate another query plan that had been waiting for some system resources to be released. Figure 18 provides a high-level view of the query processing architecture in our simulation environment.

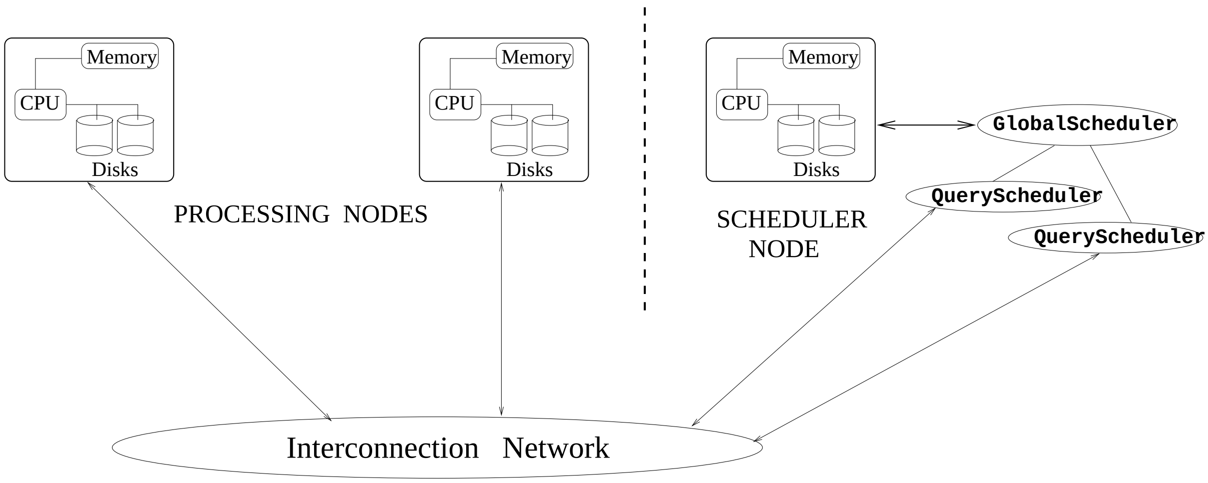


Figure 18: Simulator query processing architecture.

5.1.2 Hardware and Operating System Characteristics

Query processing sites in the system consist of a CPU, a buffer pool of 8 KByte pages, and one or more disk drives. The CPU uses a round-robin scheduling policy with a 5 msec timeslice. The buffer pool models a set of main memory page frames whose replacement is controlled by a 3-level LRU policy extended with *"love/hate"* hints [HCL+90]. In addition, a memory reservation mechanism under the control of the `GlobalScheduler` process allows memory to be reserved for a particular query operator. This mechanism is employed to ensure that hash table page frames allocated to join operators will not be stolen by other operators.

The simulated disks model a Fujitsu Model M2266 (1 GByte, 5.25") disk drive. This disk drive provides a cache that is divided into four 32 KByte cache contexts to effectively support up to four concurrent sequential prefetches. In the disk model, which slightly simplifies the actual operation of the disk, the cache is managed as follows. Each I/O request, along with the required page number, specifies whether or not prefetching is desired. If prefetching is requested, four additional pages are read from the disk into a cache context as part of transferring the originally requested page from disk to memory. Subsequent requests for one of the prefetched pages can then be satisfied without incurring an I/O operation. A simple round-robin policy is used to allocate cache contexts if the number of concurrent prefetch requests exceeds the number of available cache contexts. The disk queue is managed using an elevator algorithm [SG98].

The simulator models latency in the interconnection network, but essentially assumes infinite network bandwidth. Earlier studies have argued that this assumption is in agreement with most real parallel systems (e.g., iPSC/2, Hypercube, Paragon), where the interconnect rarely is the bottleneck during parallel processing. Besides the end-to-end latency, messages also require CPU processing at both the sender and receiver sites. Table 6 summarizes the execution environment parameters employed in our simulations. The CPU instruction counts for various database operations are based on earlier simulation studies [Meh94, PI96].

Configuration/Site Parameters	Value
Number of Sites	20 – 80
CPU Speed	20 MIPS
Number of Disks per Site	2
Memory per Site	8 – 16 MB
Page Size	8 KB
Latency for 8K Message	1.8 msec
Disk Seek Factor	0.617
Disk Rotation Time	16.667 msec
Disk Settle Time	2.0 msec
Disk Transfer Rate	3.09 MB/sec
Disk Cache Context Size	4 pages
Disk Cache Size	8 contexts
Disk Cylinder Size	83 pages
Buffer Manager	3-level LRU

CPU Cost Parameters	No. Instr.
Initiate Select	20000
Initiate Join	40000
Initiate Store	10000
Terminate Select	5000
Terminate Join	10000
Terminate Store	5000
Read Tuple	300
Write Tuple into Output Buffer	100
Probe Hash Table	200
Insert Tuple in Hash Table	100
Hash Tuple Using Split Table	500
Apply a Predicate	100
Copy 8K Message to Memory	10000
Message Protocol Costs	1000

Table 6: Simulation Parameter Settings

5.1.3 Comparison with our Earlier Model and Analysis

Using a detailed simulation environment to determine the execution time of a schedule for a given set of query plans factors in many elements of realism that were abstracted out in the analytical model experiments of Chapter 4. For example, as we already mentioned in Section 5.1.2, the simulator models the effects of buffer management, disk caching, and overheads (i.e., disk-arm contention) due to disk time-sharing at a considerable level of detail. Since most of these effects are extremely hard to capture in an analytical model (e.g., the seek overheads depend not only on the number of threads sharing a disk but also on the exact placement of blocks on the disk surface), our scheduling algorithm continues to make its mapping decisions (Figure 17) based on our original cost model with only minor modifications (e.g., using larger I/O transfer units to model prefetching for disk cache contexts). Of course, the resource requirements for the various operators (i.e., their work and demand vectors) were carefully estimated based on the optimizer's execution model and costs.

Another important difference between our simulation model and the problem model developed in Chapter 4 is the modeling of operator startup and communication costs. The simulator assumes the existence of a dedicated scheduler site that is not used for query processing and takes care of initiating and synchronizing all executing operator clones. Essentially, this means that the sequential startup cost for clones is not a "surcharge" to an operator's processing, as we assumed in the last two chapters. We clearly expect the effects of high startup costs for multiple clones to be less obvious under these assumptions. The simulator also models the interconnection network as an infinite bandwidth resource, rather than as a finite, per site TS resource. Thus, the network can never be the bottleneck resource during query execution, which led us to discard the network dimension from the query operators' work vectors. We also modified our definition of the coarse granularity parameter f slightly, defining a CG_f execution across N clones (Definition 3.2.1) as one which satisfies:

$$\frac{N \cdot \text{CloneOverhead}}{W_p(\texttt{op})} \leq f,$$

where $W_p(\texttt{op})$ is the processing area of `op` and "CloneOverhead" denotes the extra CPU processing required for initiating and terminating a clone, including control messages to and from the scheduler site. (The extension to λ-granular CG_f executions is obvious.) We should note that we decided to maintain the simulator's execution model intact even though we could have modified it to fit our earlier assumptions, since the same base simulation model has been used in a number of recent experimental studies on parallel query processing [MD95, MD97, Meh94, PI96]. In a real system, where some of the simulator's assumptions above may not hold, simple changes to our algorithm's cost model to improve effectiveness should be easily doable.

5.2 Experimental Testbed and Methodology

We have experimented with scheduling a collection of independent right-deep hash-join trees using a slightly modified version of our LevelSched algorithm. Briefly, our level-based scheduler continues to pack query tasks into disjoint execution "shelves" but views a right-deep join tree as a single query task rather than breaking up its execution across two neighboring shelves (one for the `build`-pipe(s) and one for the `probe`-pipe). LevelSched uses the combined work vector of the `build` and `probe` phase to make load-balancing decisions for joins, whereas the demand vector for allocating memory remains roughly the same across the two phases (modulo some very small number of pages for `select`'s and `store`'s). Although our original algorithm was designed having general task systems in mind, the version proposed above is probably better tuned to handle the specific form of dependency that exists between the `probe` task and its `build` children in a right-deep join tree.

Given the lack of adversary algorithms for complex query scheduling with both SS (memory) *and* TS (CPU and disk(s)) resources, we decided to compare the performance of LEVELSCHED to that of the standard query scheduling algorithm, termed ZETA, that had already been built-in the `GlobalScheduler` process of our detailed simulation model. ZETA simply runs each join operator at the set of sites where its left input (i.e., `build`-relation) is declustered. Since the performance of ZETA obviously depends on the declustering decisions made for the base relations, we experimented with various possible data placement strategies for the base relations of our workload queries:

- **Declust**: Every base relation is horizontally partitioned across all sites in the system.
- **Declust-1/4**: Every base relation is horizontally partitioned across the same subset of the system sites, comprising 1/4th of the entire system.
- **NoDeclust**: A base relation resides on a *single* system site, chosen randomly for every relation from the set of all sites.
- **NoDeclust-1/4**: A base relation resides on a *single* system site, chosen randomly for every relation from the same subset of the system sites, comprising 1/4th of the entire system.
- **Random**: Both the degree of declustering and the actual sites storing a relation are selected randomly from the underlying set of sites. To avoid partitioning small relations over too many sites (which is typically avoided in real systems), we placed an upper bound on the degree of declustering of a relation that is proportional to its size. (The upper bound for the largest relation in the system is the total number of sites.) The actual degree of declustering was then randomly chosen between 1 and that upper bound.
- **QueryBasedDeclust**: The degree of declustering for a base relation is equal to the minimum number of sites required to hold the entire `build` hash table in main memory. Furthermore, the actual set of sites given to a `build` relation is carefully selected so that *it does not overlap* with any of the homes of the other `build` relations in the same query.

Obviously, **Declust** and **NoDeclust** represent the two extreme choices in the space of data declustering strategies. The **1/4**-versions of these strategies were chosen to represent situations in which input queries will experience hotspots due to data placement choices. Although the choice of the fraction of the system used for mapping data (1/4) was arbitrary, we believe that our experimental observations will remain valid for other system fractions. The **Random** placement strategy tries to relieve query hotspots by mapping relation fragments to randomly selected sites. Such hotspots may still occur, however, based on the actual relations accessed by a query. For example, it may very well happen that the homes of the four `build` relations in a 4-join right-deep query overlap at one or more system sites. These sites will then be the bottlenecks for the `build` phase of the pipe. It is exactly such bottlenecks that our final and more sophisticated data placement strategy, **QueryBasedDeclust**, tries to avoid by explicitly examining the set of queries to be executed. Of course, determining the exact query workload may not always be possible (e.g., in an ad-hoc querying environment).

Note that the simulator implements a hybrid hash-join processing algorithm for right-deep trees, so it is possible for ZETA to schedule each join in a pipe to be executed with as little as the square root of its maximum memory demand [Sch90a, Sha86]. Of course, using less memory implies that the join pipeline is executed in multiple passes. (This is never the case for LEVELSCHED, since demand vectors are always determined using the maximum (i.e., one-pass) memory demand for a join.) We also implemented a scheduling switch that forces ZETA not to start the execution of a query until its maximum memory requirement can be satisfied, since our results showed that multi-pass join pipelining can really hurt response time performance. (Note, however, that this may not be possible given the degree of declustering of the `build` relations.)

We executed both scheduling algorithms (i.e., LEVELSCHED and ZETA) over the same set of input right-deep plans (submitted from the external terminals at time 0), for each of the aforementioned base relation placement strategies. Our workloads consisted of right-deep hash-join query plans, with the total number of joins in the workload varying between 8, 16, and 32. For each workload, we generated right-deep query plans of various sizes over randomly selected base relations. Given the total number of joins in the workload, different right-deep plan size combinations were tested (e.g., our 32-join workloads included one workload with two 8-join plans and four 4-join plans, one workload with four 8-join plans, etc.). For each workload combination, we executed five randomly generated workloads (by randomly selecting the participating base relations) and averaged the observed simulation times over the five runs. The results presented in the next section are indicative of the results obtained for all workload combinations. For the experiments presented in this chapter, there was *no base relation sharing across query plans*, i.e., each relation was part of at most one join in a workload. The purpose of this was to mitigate the effects of the buffer management algorithm on our scheduling results[1]. We plan to explore these effects in future work.

All our right-deep plans included a final step of storing the result relation across all sites using a single disk at each site. We decided to use this default strategy for `store`'s rather than

[1]Also note that allowing for such relation sharing across queries really complicates the definition of a query-based placement strategy like our **QueryBasedDeclust**.

treating them as floating operators since for many real-world queries the results do not actually need to be stored on disk, so we wanted to minimize the impact of `store`'s on the relative quality of the execution schedules. We also employed the simulator's built-in assumption of uniform join attribute distributions and decided to leave the study of join skew effects to future work. It should be clear, however, that skew can only increase the gains of our algorithm over ZETA, since by its definition ZETA is not allowed to re-balance the tuples of a `build` relation across sites. As previously, our basic comparison metric was the *response time* for a given workload. The values of our database schema and workload parameters are summarized in Table 7. Some motivation for our choices for the coarse granularity parameter f and the memory granularity parameter λ is provided in the next section. The *join probability* is the factor multipled by the cardinality of the right input to determine the result cardinality of a join, and the *projection factor* indicates the percentage of the sum of the left and right input tuple widths that should be retained in the result. We kept these join parameters fixed at 1.0 and 0.5, respectively during all our experiments.

Schema/Workload Parameters	Value
Tuple Size	200 bytes
Base Relation Size	1K – 100K tuples
Join Attribute Distribution	Uniform
Workload Size (Total No. of Joins)	8, 16, 32
Right-Deep Query Size	2, 4, 8 joins
Coarse Granularity Parameter (f)	0.001 – 0.1
Memory Granularity Parameter(λ)	0.15 – 0.25
Join Probability	1.0
Projection Factor	0.5

Table 7: Database Schema and Workload Parameters

5.3 Experimental Results

5.3.1 Tuning the Clone Granularity Parameters

Selecting effective values for the coarse granularity parameter f and the memory granularity parameter λ is a very difficult problem. The right choice depends on a number of parameters and characteristics of the underlying architecture, such as the CPU overhead of messages and clone synchronization and the speed of the interconnection network. Furthermore, as explained in Section 4.2.2, it is important to strike a balance between the overhead of parallelism and the ability to effectively "pack" memory demands at system sites.

In our simulation runs, we decided to choose a value for the memory granularity parameter λ between 0.15 and 0.25. This seemed like a reasonable choice, given that the largest relation in our experiments was approximately 20 MBytes and the smallest configuration tested had 20 sites with 8 MBytes each, for a total memory of 160 MBytes. (We would typically want the "large" relations to be declustered over a sufficiently large number of sites.) Another reason for choosing relatively small memory granularities comes from the small overhead of parallelism in our simulation model, due to the dedicated scheduling node and the infinite bandwidth interconnection network. As our results showed, the specific choice of λ between 0.15 and 0.25 does not really affect the performance of our LevelSched algorithm. Thus, we will only be presenting our results for $\lambda = 0.15$ in the remainder of this section.

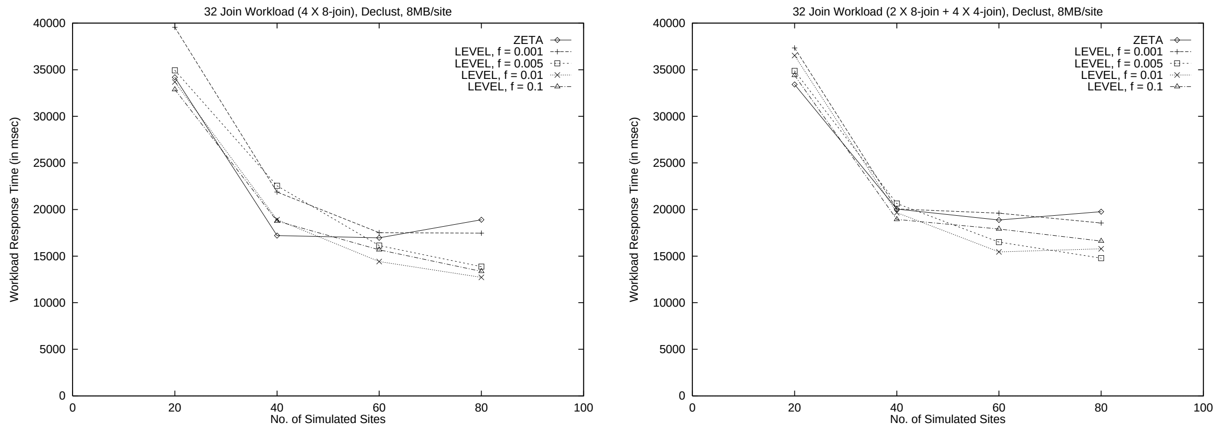


Figure 19: Effect of the coarse granularity parameter f on the performance of LevelSched for two 32-join workloads with fully declustered base relations ($\lambda = 0.15$).

Given the small overhead of parallelism in our simulation model and after observing some actual values for operator processing and communication costs during the operation of the simulator, we decided to vary the coarse granularity parameter f between the values 0.001 and 0.1. Our results for the various workloads showed that although a value of 0.001 was occasionally too small to exploit the available parallelism, values of f larger than 0.01 offered little or no benefits in terms of response time and occasionally gave rise to speed-downs for the larger configurations. An example of these trends is illustrated in Figure 19 for two 32-join workloads, one consisting of four 8-join queries and one consisting of two 8-join queries and four 4-join queries. (The performance of Zeta is also included in the plots for comparison purposes.) Consequently, in the remainder of this section we will be presenting our results for $f = 0.01$. We should once again stress, however, that the appropriate choices for λ and f could differ depending on system parameters and characteristics.

5.3.2 Effect of Data Placement Strategy

Figures 20–22 depict the performance of the LevelSched and Zeta scheduling algorithms for the various data placement strategies investigated in this chapter and site memory sizes of 8 and 16 MBytes. These results were obtained for 32-join workloads consisting of two 8-join queries and four 4-join queries.

As can be seen from Figure 20(a), the Zeta algorithm performs well under fully declustered data placement, managing to stay close to LevelSched especially for small system configurations. This was something we expected for the following reasons. First, given that all base relation scans for a pipeline occur concurrently at all sites and all result stores are again defaulted to execute on all sites, LevelSched really has no opportunities to exploit its multi-dimensional cost model in order to balance CPU and I/O processing across system sites. Second, since the overhead of parallelism is rather small in our simulator, the cost of cloning every join across all sites does not penalize performance, especially for small configurations. However, it is clear from Figure 20(a) that these penalties do appear for larger numbers of sites, whereas our algorithm can avoid these costs and still manage to balance execution loads effectively. We definitely expect these trends to become even more marked as the system size grows.

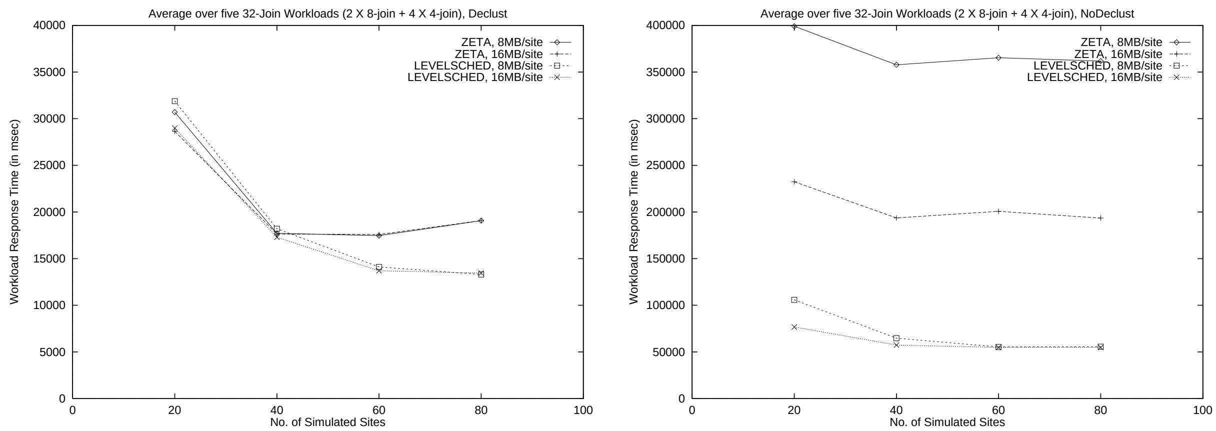


Figure 20: Performance of LevelSched and Zeta for (a) **Declust** and (b) **NoDeclust**.

As expected, LevelSched was the clear winner for the **NoDeclust**, **Declust-1/4**, and

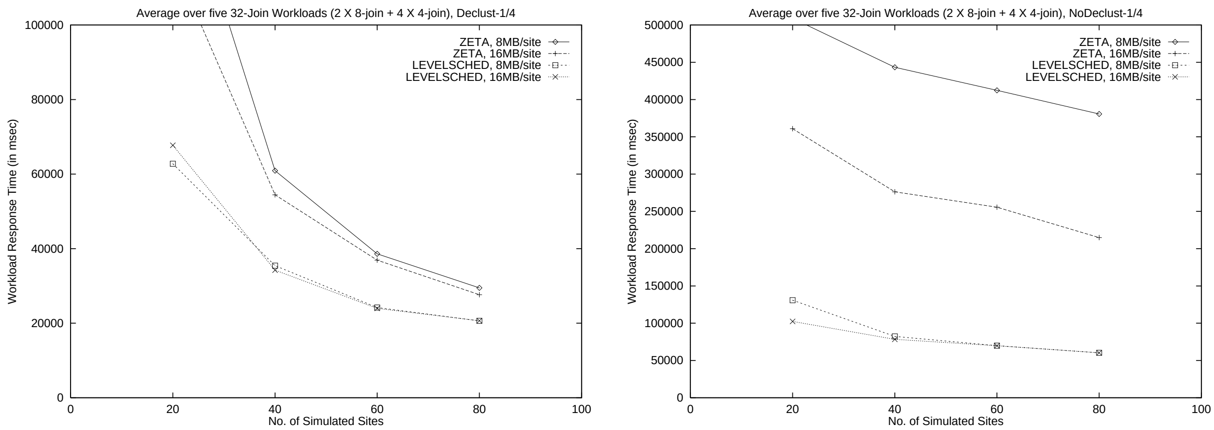


Figure 21: Performance of LevelSched and Zeta for (a) **Declust-1/4** and (b) **NoDeclust-1/4**.

NoDeclust-1/4 data placement policies, yielding substantial improvements over Zeta (Figures 20(b) and 21(a,b)). The main reason, of course, is that LevelSched manages to balance the memory demand for `build` relations across the system without being restricted by where these relations reside on disk.

Similar observations can be made for Figure 22(a), where LevelSched is shown to offer significant gains over Zeta for **Random** data placement. Figure 22(b) depicts the performance of the two algorithms for the **QueryBasedDeclust** data placement policy. In this particular experiment, we activated the "maximum memory" flag for the Zeta algorithm which ensures that a join will only be executed with its maximum memory allocation. We found that this really helped the response time performance of Zeta under **QueryBasedDeclust**. Nevertheless, LevelSched still manages to outperform Zeta by a significant margin across most of our system size range. A main reason for this phenomenon is that, by its data placement rule, **QueryBasedDeclust** essentially decides the degree of parallelism for a join by looking only at the `build` relation, which is not always a good predictor for the amount of work that needs to be done. For example, consider a very small `build` relation that is joined with a very large `probe` input. **QueryBasedDeclust** will probably assign only one site to that relation, which means that it will very likely be a bottleneck during the `probe` phase of the pipeline. LevelSched, on the other hand, considers both the memory *and* the work requirements of a join, by its combined memory granularity and coarse granularity condition. Needless to say, this observation places even more weight on the correct choice of λ and f.

Finally, Figure 23 gives a different view of the same results for a different 32-join workload

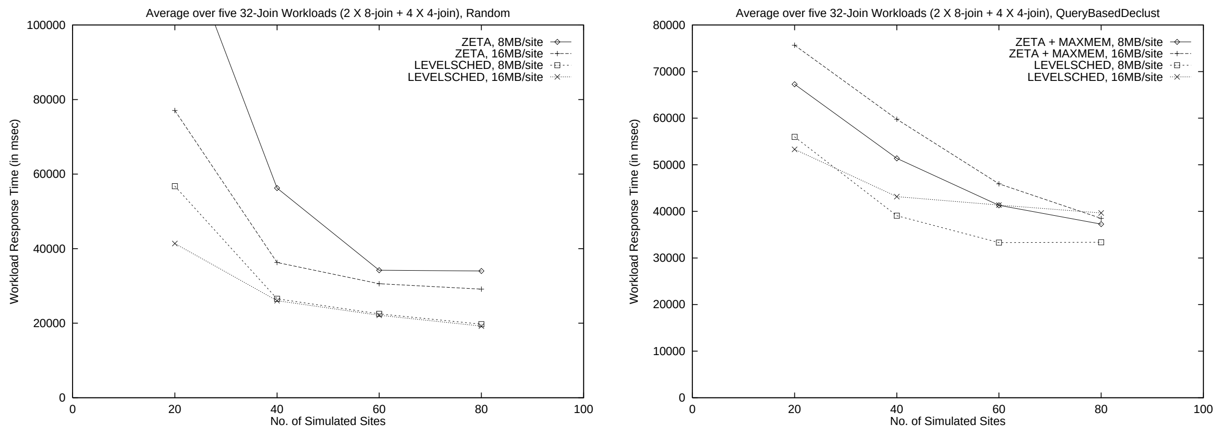


Figure 22: Performance of LevelSched and Zeta for (a) **Random** and (b) **QueryBased-Declust** (with the "max. memory" flag activated).

combination, consisting of four 8-join queries. The plots depict the observed response times for LevelSched and Zeta as a function of the input data placement, for two different system sizes (40 and 80 sites). The numbers clearly demonstrate the ability of our algorithm to distribute the query load across the system, independent of how "bad" the initial placement may be. The performance of LevelSched remains essentially unaffected by data placement choices.

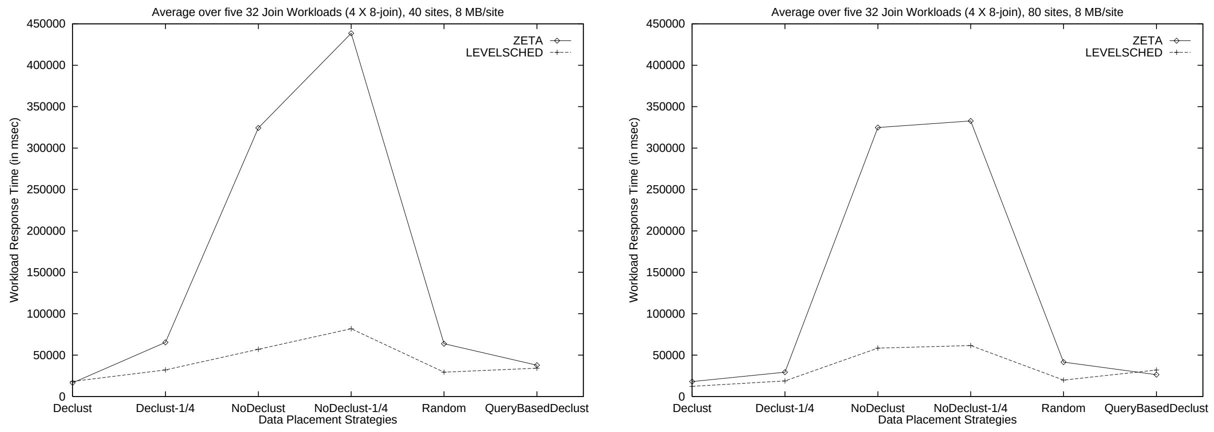


Figure 23: Performance of LevelSched and Zeta as a function of the data placement strategy for (a) 40 and (b) 80 system sites (8 MB/site).

5.4 Conclusions

In this chapter, we have presented a performance study of our multi-dimensional query scheduling algorithms using a detailed simulation model of a hierarchical parallel database system. Compared to our earlier analytical model results, this simulation study has added many elements of realism including a detailed model of disk drive operation and disk caching, and an LRU-type buffer manager. Although the simulation environment violated some of our earlier assumptions about communication and clone startup costs, the results of our simulations have shown that our algorithms can provide very effective load balancing in this environment. In the near future, we plan to experiment with workloads that allow for relation sharing across queries as well as bushy query plans.

Chapter 6

Resource Scheduling for Composite Multimedia Objects

In this chapter[1], we formulate the resource scheduling problems for composite multimedia objects and we develop novel efficient scheduling algorithms, drawing on a number of techniques from pattern matching and multiprocessor scheduling. Our formulation is based on a novel *sequence packing problem*, where the goal is to superimpose numeric sequences (representing the objects' resource needs as a function of time) within a fixed capacity bin (representing the server's resource capacity). We propose heuristic algorithms for the sequence packing problem using a two-step approach. First, we present a "basic step" method for packing two object sequences into a single, combined sequence. Second, we show how this basic step can be employed within different scheduling heuristics to obtain a playout schedule for multiple composite objects. More specifically, we examine greedy scheduling heuristics based on the general list-scheduling ($\mathcal{LS}$) methodology of Graham [Gra69, GG75]. We show that although $\mathcal{LS}$ schemes are *provably near-optimal* for packing *monotonic* sequences, they can have poor worst-case performance when the monotonicity assumption is violated. Based on this result, we: (1) suggest methods for improving the behavior of simple $\mathcal{LS}$ through the use of extra memory buffers; and, (2) propose a novel family of more clever scheduling algorithms, termed *list-scheduling with backtracking* ($\mathcal{LSB}$), that try to improve upon simple $\mathcal{LS}$ by occasional local improvements to the schedule. Experimental results with randomly generated composite objects show that our $\mathcal{LS}$ strategy offers excellent average-case performance compared to both an MBR-based approach and the optimal solution. We also briefly discuss our ongoing work on how the idea of *stream sharing* (i.e., allowing several presentations to share component streams) can be exploited to improve the quality of a schedule.

[1]Parts of this chapter have appeared in the *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB'98)* [GIÖ98].

6.1 Definitions and Problem Formulation

6.1.1 Composite Objects and Object Sequences

A composite multimedia object consists of multiple CM streams tied together through spatial and temporal primitives. Since the spatial layout of the output is predetermined by the user and does not affect the resource bandwidth requirements of CM streams, we concentrate on the temporal aspects of CM composition. Following the bulk of the multimedia systems literature, we also concentrate on the server disk bandwidth resource which is typically the bottleneck for multimedia applications [CGS95, RV91, ÖBRS94, SGC95]. To simplify the presentation, we assume that the stream resource demands have been normalized to $[0,1]$ using the aggregate disk bandwidth of the server B. We also assume that the time scale is *discrete* so that both the lengths of CM streams and their lag parameters have integer values. This is usually the case in practice, since most multimedia storage servers employ a *round-based* data retrieval scheme and thus timing can only be specified at the granularity of a round's length, which is typically very small (a few seconds) [RV91, ÖBRS94]. Of course, finer-grain synchronization can always be implemented using extra memory buffering [SGC95].

Following Chaudhuri et al. [CGS95], we define an n_i-ary composite multimedia object C_i as a $(2n_i-1)$-tuple $< X_{i_1}, X_{i_2}, \ldots, X_{i_{n_i}}, t_{i_2}, \ldots, t_{i_{n_i}} >$ where the X_{i_j}'s denote the component CM streams (in order of increasing start times) and t_{i_j} denotes the *lag* factor of X_{i_j} with respect to the beginning of the display of X_{i_1} (i.e., the beginning of the composite object). This definition covers the 13 qualitative temporal interval relationships of Allen [All83] and also allows us to specify quantitative temporal constraints. Figure 24(a) depicts a 4-ary object corresponding to the news story example mentioned in Section 1.2.1, consisting of two overlapping video clips with background music and narration. The height of each stream in Figure 24(a) corresponds to its bandwidth requirement, and the length corresponds to its duration (in general, the x-axis represents time).

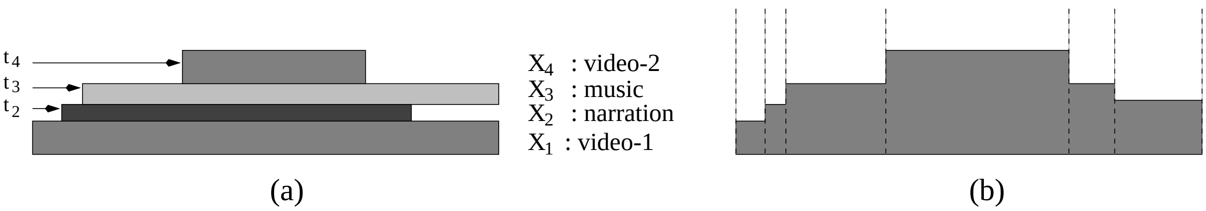


Figure 24: (a) A 4-ary composite multimedia object. (b) The corresponding object sequence.

For each component CM stream X_{i_j} of C_i, we let $l(X_{i_j})$ denote the time duration of the stream and $r(X_{i_j})$ denote its resource bandwidth requirements. Similarly, we let $l(C_i)$ denote the duration of the entire composite object C_i, i.e., $l(C_i) = \max_j\{t_{i_j} + l(X_{i_j})\}$, and $r(C_i, t)$ denote the bandwidth requirements of C_i at the t^{th} time slot after its start $(0 \leq t < l(C_i))$. Table 8 summarizes the notation used throughout this chapter with a brief description of its semantics. Detailed definitions of some of these parameters are given in the text. Additional notation will be introduced when necessary.

Parameter	Semantics
B	Aggregate server disk bandwidth (in bits per sec – bps)
T	Length of a time unit (i.e., round) (in sec)
C_i	Composite multimedia object
n_i	Number of component streams in C_i
$l(C_i)$	Length (i.e., time duration) of C_i
$r(C_i, t)$	Bandwidth requirement of C_i at time slot t $(0 \leq t < l(C_i))$
$r_{max}(C_i)$	Max. bandwidth requirement of C_i
X_{i_j}	The j^{th} component stream of C_i
$l(X_{i_j})$, $r(X_{i_j})$	Length and bandwidth requirement of stream X_{i_j}
k_i	Number of constant bandwidth "blocks" in the run-length compressed form of C_i
(l_{i_j}, r_{i_j})	Length and bandwidth requirement of the j^{th} run-length "block" of C_i
$V(C_i)$	Volume (i.e., total resource-time product) of C_i
$d(C_i)$	Density of C_i

Table 8: Notation

The bandwidth requirements of our example news story object can be represented as the *composite object sequence* depicted graphically in Figure 24(b), where each element of the sequence corresponds to the object's bandwidth demand at that point in time (i.e., during that time unit). Note that the rising and falling edges in a composite object sequence correspond to CM streams starting and ending, respectively. Essentially, the object sequence represents $r(C_i, t)$, that is, the (varying) bandwidth requirements of the object as a function of time t. Since our scheduling problem focuses on satisfying the bandwidth requirements of objects, we will treat the terms "composite object" and "sequence" as synonymous in the remainder of the chapter.

Typically, CM streams tend to last for long periods of time. This means that using the full-length, $l(C_i)$-element object sequence for representing and scheduling a composite multimedia object is a bad choice for the following two reasons. First, these full-length sequences will be very long (e.g., a 2-hour presentation will typically span thousands of rounds/time units). Second, full-length object sequences will be extremely redundant and repetitive since they only contain a small number of transition points. For our purposes, a more compact representation of composite objects can be obtained by using the *run-length compressed* form of the object sequences [AG97]. Essentially, the idea is to partition the composite object into "blocks" of constant bandwidth requirement and represent each such block by a pair (l_{i_j}, r_{i_j}), where r_{i_j} represents the constant requirement of the object over a duration of l_{i_j} time units. This process is shown graphically in Figure 24(b). Thus, we can represent the n-ary composite object C_i in a compact manner by the sequence: $< (l_{i_1}, r_{i_1}), \ldots, (l_{i_{k_i}}, r_{i_{k_i}}) >$, where $k_i << l(C_i)$. In fact, $k_i \leq 2 \cdot n_i - 1$, where n_i is the number of component CM streams in C_i.

Following the terminology of Chapter 4, we define the *volume (V)* of a composite object C_i as the total resource-time product over the duration of C_i. More formally, $V(C_i) = \sum_{j=1}^{k_i} l_{i_j} r_{i_j}$. The *density (d)* of a composite object C_i is defined as the ratio of the object's volume to the volume of its MBR, i.e., $d(C_i) = \frac{V(C_i)}{l(C_i) \cdot r_{max}(C_i)}$.

6.1.2 Using Memory to Change Object Sequences: Stream Sliding

Although inter-media synchronization constraints completely specify the relative timing of streams at presentation time, the scheduler can use extra memory buffers[2] to alter an object's retrieval sequence. The idea is to use additional memory to buffer parts of streams that have been retrieved before they are actually needed in the presentation and play them out *from memory* at the appropriate time. This method, termed *stream sliding*, was originally introduced by Chaudhuri et al. for resolving internal contention under the assumption of round-robin striping [CGS95]. The general method is depicted in Figure 25(a) which shows our example 4-ary news story object with the second video clip (stream X_4) *upslided* by x time units. In this example, the server needs to use an extra $x \cdot T \cdot B \cdot r(X_4)$ bits of memory in order to support the object playout as required by the user (Figure 24(a)). Since it is possible for the amount of upsliding to exceed the actual length of the stream (i.e., $x > l(X_4)$), the general expression for the amount of memory required to upslide X_4 by x is $\min\{x, l(X_4)\} \cdot T \cdot B \cdot r(X_4)$. This expression says that if $x > l(X_4)$, then we only need enough memory to buffer the entire stream X_4. (Note that multiplying by the server bandwidth B is necessary, since $r()$ is normalized using B.)

Similarly, the server may also choose to *downslide* a stream, which means that the retrieval of the stream starts *after* its designated playout time in the presentation (Figure 25(b)). Downsliding introduces latency in the composite object presentation, since it is clearly impossible to start the playout of a stream before starting its retrieval. Thus, in Figure 25(b), the entire presentation must be delayed by y time units. This also means that once a stream is downslided, *all the other streams* must be buffered unless they are also downslided by the same amount.

[2] Assuming round-based retrieval of streams with a round length of T, each stream X_{i_j} requires a minimum buffering of $2 \cdot T \cdot B \cdot r(X_{i_j})$ during its retrieval [ÖBRS94, ÖRS95a].

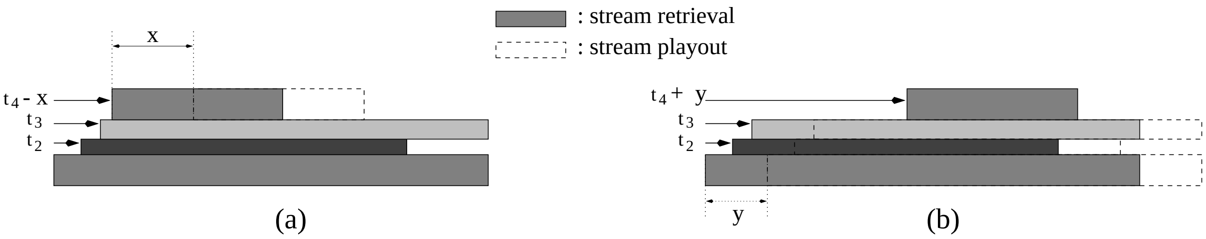


Figure 25: (a) Upsliding stream X_4. (b) Downsliding stream X_4.

Because of these two problems upsliding is preferable to downsliding, whenever both options are available [CGS95, SGC95].

6.1.3 Our Scheduling Problem: Sequence Packing

Given a collection of composite objects to be scheduled using the server's resources, a *schedule* is an assignment of start times to these objects so that, at any point in time, the bandwidth and memory requirements of concurrent presentations (to, perhaps, different users) do not violate the resource capacities of the server. In this chapter, we concentrate on the problem of *off-line makespan (i.e., response time) minimization*, in which the objective is to minimize the overall schedule length for a given collection of objects (i.e., tasks) [GG75, Gra69]. Prior scheduling theory research has shown how to employ solutions to this problem for both *on-line* response time minimization (where tasks arrive dynamically over time) and on-line average response time minimization [CPS$^+$96b, HSSW97, SWW95].

Ignoring the flexibilities allowed in stream synchronization through additional memory buffering (i.e., sliding), the bandwidth requirements of each object to be scheduled are completely specified by its resource demand sequence (Figure 24(b)). Thus, assuming sliding is not an option, our scheduling problem can be abstractly defined as follows.

- **Given:** A collection of (normalized) composite object sequences $\{C_i\}$ over $[0, 1]$.
- **Find:** A start time slot $s(C_i)$ for each i, such that for each time slot t

$$\sum_{\{C_i : s(C_i) \leq t < s(C_i) + l(C_i)\}} r(C_i, t - s(C_i)) \leq 1,$$

  and $\max_i\{s(C_i) + l(C_i)\}$ is minimized.

Conceptually, this corresponds to a *sequence packing* problem, a non-trivial generalization of traditional $\mathcal{NP}$-hard optimization problems like bin packing and multiprocessor scheduling that, to the best of our knowledge, has not been previously studied in the combinatorial optimization literature [CGJ84, CGJ96, Gra69, GG75]. In bin packing terminology, we are given a set of items (normalized object sequences) that we want to pack within unit-capacity bins (server bandwidth) so that the total number of bins (makespan, used time slots) is minimized. Our sequence packing problem also generalizes *multi-dimensional* bin packing models known to be intractable, like *orthogonal rectangle packing* [BCR80, BS83, CGJT80] (a rectangle is a trivial, constant sequence) and *vector packing* [GGJY76, KM77] (vectors are fixed length sequences with start times restricted to bin boundaries). Note that rectangle packing algorithms are directly applicable when the MBR simplification is adopted. However, it is clear that this simplification can result in wasting large fractions of server bandwidth when the object densities are low. Given the inadequacy of the MBR simplification and the intractability of the general sequence packing formulation, we propose novel efficient heuristics for scheduling composite object sequences using a combination of techniques from pattern matching and multiprocessor scheduling.

Sliding further complicates things, since it implies that the scheduler is able to *modify* the composite object sequences at the cost of extra memory. Given a set of object sequences and a finite amount of memory available at the server, the problem is how to utilize memory resources for sliding various object streams around (i.e., modifying the object sequences) so that the scheduler can effectively minimize some scheduling performance metric such as schedule length or average response time. This is obviously a very complex problem that, in many ways, generalizes recently proposed *malleable* multiprocessor scheduling problems [Lud95, TLW$^+$94, TWY92]. To the best of our knowledge, the general sliding problem, as outlined above, has yet to be addressed in the scheduling or multimedia literature.

Our results for the sequence packing problem indicate that simple, greedy scheduling algorithms based on Graham's list-scheduling method [Gra69] can guarantee *provably* near-optimal solutions, as long as the sequences are monotonic. On the other hand, we show that list-scheduling can perform poorly when the monotonicity assumption is violated. Based on this result, we examine the problem of exploiting extra memory and sliding to make object sequences monotonic. Although this problem is itself $\mathcal{NP}$-hard, we propose a polynomial-time approximate solution.

6.2 Algorithms for the Sequence Packing Problem

In this section, we present heuristic algorithms for the object sequence packing problem identified in Section 6.1. Our approach is based on the observation that the result of packing a subset of the given object sequences is itself an object sequence. Thus, we begin by presenting a "basic step" algorithm for obtaining a valid packing of two sequences. We then show how this method can be employed within two different scheduling heuristics to obtain a playout schedule for multiple composite objects.

6.2.1 The Basic Step: Packing Two Sequences

Our basic algorithmic step problem can be abstractly described as follows. We are given two (normalized) object sequences C_o (a new object to be scheduled) and C_p (the partial schedule constructed so far) over $[0, 1]$. We want to determine a valid packing of the two sequences, that is, a way to superimpose C_o over C_p that respects the unit capacity constraint (i.e., all elements of the combined sequence are less than or equal to 1). Given that our overall scheduling objective is to minimize the length of the resulting composite sequence, the presentation of this section assumes a "greedy" basic step that searches for the *first point* of C_p at which C_o can be started without causing capacity constraints to be violated. Since, as we argued in Section 6.1.1, the full-length representation of the object sequences is very inefficient we assume that both object sequences are given in their run-length compressed form. That is, $C_o =< (l_{o_1}, r_{o_1}), \ldots, (l_{o_{k_o}}, r_{o_{k_o}}) >$ and $C_p =< (l_{p_1}, r_{p_1}), \ldots, (l_{p_{k_p}}, r_{p_{k_p}}) >$. In Figure 26, we present an algorithm, termed FINDMIN, for performing the basic sequence packing step outlined above. FINDMIN is essentially a "brute-force" algorithm that runs in time $O(k_o \cdot k_p)$, where k_o, k_p are the lengths of the compressed object sequences.

Since our composite obect sequences can be seen as *patterns* over the alphabet $[0, 1]$, it is natural to ask whether or not ideas from the area of *pattern matching* can be used to make our basic algorithmic step more efficient. As in most pattern matching problems [AG97], the requirement that all "characters" of both patterns must be examined imposes a linear, i.e., $O(k_o + k_p)$ (since we are dealing with the compressed representations), lower bound on the running time of any algorithm. The question is whether or not this lower bound is attainable by some strategy. An equivalent formulation of our basic step packing problem comes from considering the *complementary sequence* $\overline{C_p}$ of the partial schedule C_p, which informally, corresponds to the sequence of free server bandwidth over time. More formally, $\overline{C_p}$ is determined by defining $r(\overline{C_p}, t) = 1 - r(C_p, t)$, for each time slot t. Thus, our problem is equivalent to finding the earliest time slot at which C_o is completely "covered"/dominated by $\overline{C_p}$ (Figure 27(a)). This is a problem that has received some attention from the pattern matching research community. Amir and Farach define the above problem for *uncompressed* patterns as the *"smaller matching"* problem and give an algorithm that runs in time $O\left(l(C_p) \cdot \sqrt{l(C_o)} \cdot \log l(C_o)\right)$, where $l()$ denotes uncompressed sequence lengths (Table 8) [AF91]. Muthukrishnan and Palem show that this is in fact a lower bound on the time complexity of the problem in a special, yet powerful

Algorithm FINDMIN(C_o, C_p)

Input: Sequences
C_o, C_p over $[0,1]$ in run-length compressed form (i.e., $C_o = < (l_{o_1}, r_{o_1}), \ldots, (l_{o_{k_o}}, r_{o_{k_o}}) >$ and $C_p = < (l_{p_1}, r_{p_1}), \ldots, (l_{p_{k_p}}, r_{p_{k_p}}) >$).

Output: The least $0 \leq k \leq l(C_p)$ such that C_o can be validly superimposed over C_p starting at time slot k.

1. For each constant bandwidth block $< l_{o_j}, r_{o_j} >$ of C_o, determine the set of feasible starting points S_{o_j} for $< l_{o_j}, r_{o_j} >$ over C_p. For a given j, this can be done in time $O(k_p)$ and the result is a union of $m_j = O(k_p)$ disjoint temporal intervals:
$$S_{o_j} = [a_{j_1}, b_{j_1}) \cup \ldots \cup [a_{j_{m_j}}, b_{j_{m_j}}),$$
where $b_{j_{m_j}} = \infty$ (at the end of the current partial schedule C_p).
2. Let $I_1 = S_{o_1}$. For $j = 2$ to k_o do
$$\begin{aligned} I_j = I_{j-1} \cap (\quad & [a_{j_1} - (l_{o_1} + \ldots + l_{o_{j-1}}), b_{j_1} - (l_{o_1} + \ldots + l_{o_{j-1}})) \cup \ldots \\ & \cup [a_{j_{m_j}} - (l_{o_1} + \ldots + l_{o_{j-1}}), b_{j_{m_j}} - (l_{o_1} + \ldots + l_{o_{j-1}})) \quad) \end{aligned}$$
Let $|S|$ denote the number of intervals in S. Since the intervals in the unions S_{o_j} are disjoint and in sorted order, each intersection I_j can be computed in time $O(|I_{j-1}| + |S_{o_j}|)$ using a MERGE-like algorithm, and a simple argument can establish that $|I_j| \leq \min\{|I_{j-1}|, |S_{o_j}|\} = O(k_p)$.
3. At this point, I_{k_o} contains the entire set of feasible starting time slots for C_o. Return the earliest slot in I_{k_o}.

Figure 26: Algorithm FINDMIN

convolution-based model; their result implies that faster algorithms for the smaller matching problem would imply a faster method than the Fast Fourier Transform for certain generalizations of convolutions [MP94]. None of these papers addressed the problem when the *run-length compressed* forms of the patterns are used. Furthermore, it is not clear whether the rather complicated algorithm of Amir and Farach will outperform a straightforward $O(l(C_p) \cdot l(C_p))$ solution in a practical implementation [Far97]. Finally, even if the "optimal" method of Amir and Farach could be extended to run-length compressed patterns, the asymptotic performance improvement with respect to FINDMIN would only be $O(\frac{\sqrt{k_o}}{\log k_o})$, a small gain since the number of streams in a single composite object k_o is typically bounded by a small constant.

Thus, prior results from the pattern matching community assure us that our "brute-force" FINDMIN algorithm is a reasonably good strategy for the basic sequence packing step for general

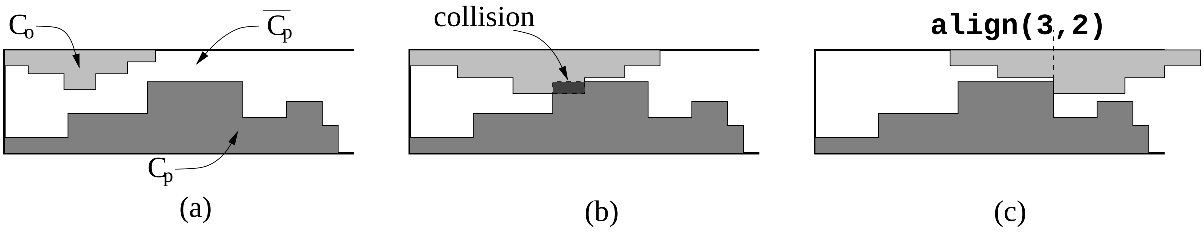


Figure 27: (a)The "smaller matching" analogy. (b) A collision with a bitonic C_o. (c) Resolving the collision by `align`-ing (Algorithm Bitonic-FindMin).

object sequences. However, for special cases of object sequences (C_o), we may still be able to come up with faster algorithms. We now present such an algorithm for the special case of *bitonic object sequences.* Informally, a sequence is bitonic if it can be partitioned into a monotonically increasing prefix followed by a monotonically decreasing prefix. This means that no new component streams can be initiated after the end of a stream in the presentation. Although one can argue that bitonic objects are rather common in multimedia practice (e.g., our example news story composite object shown in Figure 24 is bitonic), our method can also be used within more complex basic step algorithms for general sequences. The idea is to partition objects into a (small) number of bitonic components and schedule each component using our improved strategy for bitonic objects. We will not pursue this idea further in this thesis.

The basic operation of our improved algorithm for bitonic objects is similar to that of the Knuth-Morris-Pratt string matching algorithm, in that it shifts the C_o pattern over C_p until a "fit" is discovered [KMP77]. The crucial observation is that, for bitonic C_o patterns, we can perform this shifting in an efficient manner, without ever having to backtrack on C_p. This is done as follows. Consider a particular alignment of C_o and C_p and let (l_{p_j}, r_{p_j}) be the first block of C_p at which a "collision" (i.e., a capacity violation) occurs. Then the earliest possible positioning of C_o that should be attempted (without losing possible intermediate positions) is that which aligns the right endpoint of block (l_{p_j}, r_{p_j}) with that of block $(l_{o_{i_j}}, r_{o_{i_j}})$, where i_j is the index of the latest block in the increasing segment of C_o such that $r_{p_j} + r_{o_{i_j}} \leq 1$. We denote this alignment operation by $\texttt{align}(j, i_j)$. An example is depicted in Figure 27(b,c). To make the presentation uniform, we assume the existence of a zero block for both sequences, with $l_{p_0} = l_{o_0} = 0$. Our improved basic step algorithm for bitonic C_o, termed Bitonic-FindMin, is depicted in Figure 28.

The time complexity of algorithm Bitonic-FindMin is $O(k_o + k_p \cdot \log k_o)$. The first term comes from the preprocessing step for C_o. The second term is based on the observation that both the partial sums vector s_j and the bandwidth requirements vector for the increasing segment

Algorithm BITONIC-FINDMIN(C_o, C_p)

Input: Sequences C_o, C_p over $[0,1]$ in run-length compressed form (i.e., $C_o =< (l_{o_1}, r_{o_1}), \ldots, (l_{o_{k_o}}, r_{o_{k_o}}) >$ and $C_p =< (l_{p_1}, r_{p_1}), \ldots, (l_{p_{k_p}}, r_{p_{k_p}}) >$). Sequence C_o is assumed to be *bitonic*.

Output: The least $0 \leq k \leq l(C_p)$ such that C_o can be validly superimposed over C_p starting at time slot k.

1. Preprocess C_o to obtain a "partial sums" vector $s_j = \sum_{m=1}^{j} l_{o_m}$ for $j = 1, \ldots, k_o$.
2. Initialize: `align`$(0,0)$, $j = i = 0$, $l_{cov} = 0$.
3. while ($l_{cov} < l(C_o)$) do
   3.1. Find the least i_j such that $s_{i_j} - l_{cov} \geq l_{p_j}$.
   3.2. To check for a collision at block (l_{p_j}, r_{p_j}) we distinguish two cases.
   - *C_o is decreasing after l_{cov}.* (We just need to check the first block of C_o placed over (l_{p_j}, r_{p_j}).) If $r_{p_j} + r_{o_{i_j}} \leq 1$ (i.e., no collision) then set $l_{cov} = l_{cov} + l_{p_j}$, $j = j+1$, $i = i_j$. Else, find the largest block index m in the increasing part of C_o such that $r_{o_m} \leq 1 - r_{p_j}$, shift C_o to `align`(j, m), and set $i = m$, $j = j+1$, $l_{cov} = s_m$.
   - *C_o is increasing after l_{cov}.* If the block index i_j is in the decreasing part of C_o then check if $r_{p_j} + r_{max}(C_o) \leq 1$, otherwise check if $r_{p_j} + r_{o_{i_j}} \leq 1$. If the condition holds set $l_{cov} = l_{cov} + l_{p_j}$, $j = j+1$, $i = i_j$. Else, find the largest block index m in the increasing part of C_o such that $r_{o_m} \leq 1 - r_{p_j}$, shift C_o to `align`(j, m), and set $i = m$, $j = j+1$, $l_{cov} = s_m$.
4. Return the starting time slot for the current placement of C_o.

Figure 28: Algorithm BITONIC-FINDMIN

of C_o are *sorted*, which means that the "find the least/largest index s.t. <condition>" steps of BITONIC-FINDMIN can be performed in time $O(\log k_o)$, using binary search. More elaborate search mechanisms (e.g., based on Thorup's priority queue structures [Tho96]) can be employed to reduce the asymptotic complexity[3] of BITONIC-FINDMIN to $O(k_o + k_p \cdot \log\log k_o)$.

6.2.2 A List-Scheduling Algorithm for Sequence Packing

We present a heuristic algorithm that uses the basic packing step described in the previous section in the manner prescribed by Graham's greedy list-scheduling strategy for multiprocessor scheduling [Gra69]. The operation of our heuristic, termed $\mathcal{LS}$, is as follows. Let L be a list of the composite object sequences to be scheduled. At each step, the $\mathcal{LS}$ algorithm takes the next

[3]Note, however, that the practicality of such search structures for the small domains encountered in this chapter is questionable [Mut98].

object from L and (using FINDMIN or BITONIC-FINDMIN) places it at the earliest possible start point based on the current schedule. Note that this rule is identical to Graham's list-scheduling rule for an m-processor system, when all objects have a constant bandwidth requirement of $1/m$ throughout their duration.

Unfortunately, this simple list-scheduling rule does not offer a good guaranteed worst-case bound on the suboptimality of the obtained schedule. Even for the special case of bitonic objects, it is easy to construct examples where the makespan of the list schedule is $\Omega(\min\{l, |L|\})$ times the optimal makespan, where $|L|$ is the number of objects and l is the length (in rounds) of an object sequence. One such bad example for $\mathcal{LS}$ is depicted in Figure 29. Note that as the example object sequences become more "peaked", the behavior of $\mathcal{LS}$ compared to the optimal schedule becomes even worse. However, even for this bad example, $\mathcal{LS}$ will behave significantly better than MBR scheduling, which would not allow *any* overlap between consecutive "columns" in the schedule.

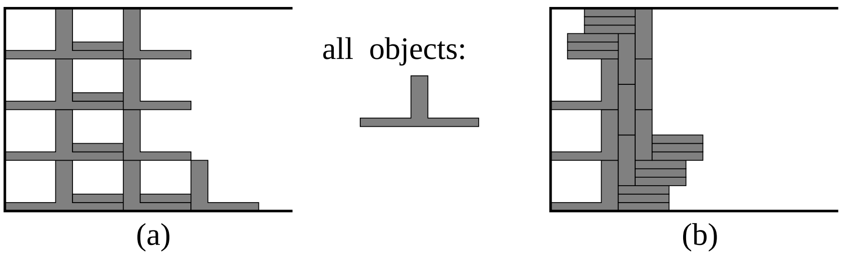


Figure 29: A "bad" example for $\mathcal{LS}$: (a) Schedule produced by $\mathcal{LS}$. (b) Optimal schedule.

Thus, in the worst case, the behavior of $\mathcal{LS}$ can be arbitrarily bad compared to the optimal schedule. Furthermore, note that since all the objects in the example of Figure 29 are *identical*, ordering the list L in any particular order (e.g., by decreasing object "height") will not help worst-case behavior. Assuming the maximum resource requirements of objects (i.e., $r_{max}(C_i)$'s) to be bounded by some small constant (a reasonable assumption for large-scale CM servers) also does not help, as long as the objects are sufficiently "peaky". However, as the following theorem shows, the situation is much better when the object sequences in L are appropriately constrained.

Theorem 6.2.1 Let L be a list of *monotonically non-increasing* composite object sequences C_i and assume that $r_{max}(C_i) \leq \lambda < 1$, for each i. Also, let $l_{max}(L) = \max_i\{l(C_i)\}$ and $V(L) = \sum_i V(C_i)$ (i.e., the total volume in L), and let $T_{OPT}(L)$ be the makespan of the optimal schedule for L. Then, the makespan returned by $\mathcal{LS}$, $T_{\mathcal{LS}}(L)$, satisfies the inequality:

$$T_{\mathcal{LS}}(L) \leq \frac{V(L)}{1-\lambda} + l_{max}(L) \leq \left(1 + \frac{1}{1-\lambda}\right) \cdot T_{OPT}(L).$$

□

It is easy to see that a slightly modified version of $\mathcal{LS}$ can guarantee the same worst-case performance bound for any list of monotonically *non-decreasing* objects, as well. The basic idea is to "time-reverse" each object sequence and schedule these reversed (non-increasing) sequences using $\mathcal{LS}$. Of course, the schedule obtained is then played out in reverse (i.e., from end to start). We can combine these observations with algorithm $\mathcal{LS}$ to obtain the following simple strategy, termed Monotonic $\mathcal{LS}$ ($\mathcal{MLS}$), for scheduling *monotonic* objects. First, schedule all non-increasing sequences using $\mathcal{LS}$. Second, reverse all remaining objects and schedule them using $\mathcal{LS}$. Third, concatenate the two schedules. It is easy to prove the following corollary.

Corollary 6.2.1 For any list L of *monotonic* composite objects C_i with $r_{max}(C_i) \leq \lambda < 1$, for each i, algorithm $\mathcal{MLS}$ guarantees a makespan $T_{\mathcal{MLS}}(L)$ such that:

$$T_{\mathcal{MLS}}(L) \;\leq\; \frac{V(L)}{1-\lambda} \;+\; 2 \cdot l_{max}(L) \leq \left(2 + \frac{1}{1-\lambda}\right) \cdot T_{OPT}(L).$$

□

With respect to the time complexity of our list-scheduling algorithms, it is easy to see that the decisive factor is the complexity of the basic packing step discussed in Section 6.2.1. Using the FINDMIN algorithm for general object sequences implies an overall time complexity of $O(N^2)$, where $N = \sum_{C_i \in L} n_i$ is the total number of streams used in L. If all the sequences are bitonic or monotonic, BITONIC-FINDMIN can be used giving an overall time complexity of $O(N \cdot |L| \cdot \log \frac{N}{|L|})$. (Note that that the number of composite objects $|L|$ is smaller than the number of streams N and the average number of streams per object $\frac{N}{|L|}$ is typically a small constant.)

6.2.3 Improving over the MBR Assumption: Monotonic Covers

Informally, we define a *monotonic cover* of a composite object C_i as another composite object $\hat{C}_i$ that (a) is monotonic, and (b) completely "covers" C_i at any point in time. More formally:

Definition 6.2.1 A *monotonic cover* for $C_i =< (l_{i_1}, r_{i_1}), \ldots, (l_{i_{k_i}}, r_{i_{k_i}}) >$ is an object $\hat{C}_i =< (l_{i_1}, \hat{r}_{i_1}), \ldots, (l_{i_{k_i}}, \hat{r}_{i_{k_i}}) >$ such that $\hat{r}_{i_j} \geq r_{i_j}$ for each j and $\hat{r}_{i_j} \geq \hat{r}_{i_{j+1}}$ for $j = 1, \ldots, k_i - 1$ *or* $\hat{r}_{i_j} \leq \hat{r}_{i_{j+1}}$ for $j = 1, \ldots, k_i - 1$. □

Corollary 6.2.1 suggests an immediate improvement over the simplistic MBR assumption for scheduling composite multimedia objects: *Instead of using the MBR of an object C_i, use* *a minimal monotonic cover of C_i, that is, a monotonic cover that minimizes the extra volume.* List-scheduling the monotonic covers of composite objects (using $\mathcal{MLS}$) is an attractive option because of the following two reasons. First, compared to scheduling the object sequences unchanged, using the covers implies a simpler placement algorithm (a special case of BITONIC-FINDMIN), with reduced time complexity. Second, compared to using MBRs, minimal monotonic covers can significantly reduce the amount of wasted volume in the cover. (Note that a MBR is itself a trivial monotonic cover.) For example, in the case of a bitonic object, a minimal monotonic cover wastes at most half the volume that would be wasted in an MBR cover. Also note that the minimal monotonic cover of a compressed composite object sequence can be easily computed in linear time.

6.2.4 Utilizing Server Memory: Stream Sliding

A different way of employing our near-optimality results for list-scheduling in the case of monotonic objects is through the use of the *stream sliding* techniques described in Section 6.1.2. The idea is to use extra server memory to turn non-monotonic object sequences into monotonic ones. In this section, we attack the problem of efficiently utilizing server memory to make composite object sequences monotonic. More specifically, we concentrate on the use of stream *upsliding* to convert an object sequence to a non-increasing sequence *with minimal extra memory*. Our techniques are also applicable to the dual problem (i.e., making sequences non-decreasing by stream downsliding). However, as we discussed in Section 6.1.2, downsliding introduces latencies into the schedule and should therefore be avoided. Possible techniques that combine both upsliding and downsliding are left as an open problem for future work.

A straightforward way of making an object sequence non-increasing is to simply upslide all streams with a non-zero lag to the beginning of the composite object (Figure 30(a,b)). Given an object C_i, this naive approach will obviously require a total memory of $B \cdot \sum_{j=2}^{n_i} lt_{i_j} \cdot r_{i_j}$, where we define $lt_{i_j} = \min\{l(X_{i_j}), t_{i_j}\}$ (Section 6.1.2). However, we might be able to do significantly better than that. The idea is that, depending on the object's structure, it may be possible for some stream to "shield" the starting edge of another stream, without requiring the second stream to be upslided all the way to the beginning of the object. This is depicted graphically in Figure 30(c). Note that the use of such clever upsliding methods not only reduces the amount of memory required for making the object monotonic, but it also reduces the maximum bandwidth requirement of the object (i.e., $r_{max}(C_i)$). This is obviously important since it implies smaller λ's in Theorem 6.2.1 and, consequently, better worst-case performance guarantees for $\mathcal{LS}$.

Unfortunately, as the following theorem shows, this problem of optimal (i.e., minimum memory) stream upsliding is $\mathcal{NP}$-hard. This result can be proved using the observation that

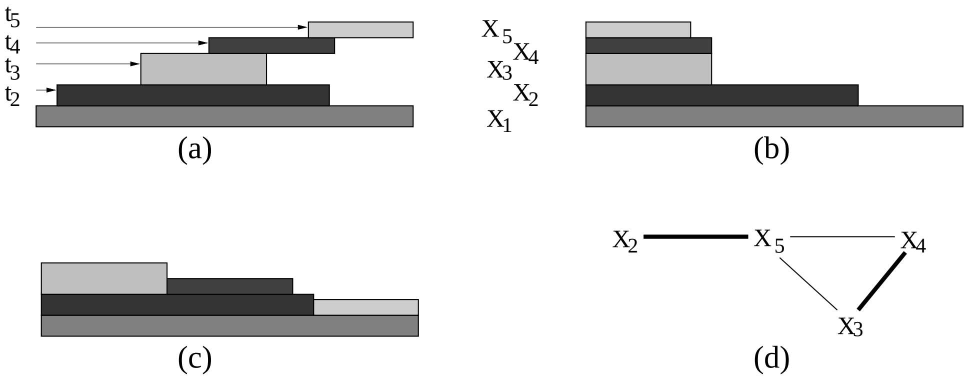


Figure 30: (a)A 5-ary composite object C. (b) "Naive" stream upsliding. (c) "Clever" stream upsliding. (d) The *shield graph* of C. (Thick lines indicate the matching used in (c).)

each stream can "shield" multiple streams to give a reduction from **PARTITION** [GJ79].

Theorem 6.2.2 Given a composite object sequence C_i, determining an *optimal (i.e., minimum memory)* sequence of stream upslides to make C_i non-increasing is $\mathcal{NP}$-hard. $\square$

Given the above intractability result, we now propose a simple heuristic strategy for improving upon the naive "upslide everything" method. Our solution is based on the following definition that simply states the above observations in a more formal manner.

Definition 6.2.2 Consider a composite object C_i. We say that stream X_{i_j} can *shield* stream X_{i_k} $(k \neq j)$ if and only if $l(X_{i_j}) \leq t_{i_k}$ and $r(X_{i_j}) \geq r(X_{i_k})$. The *benefit* of shielding X_{i_k} by X_{i_j} is defined as:

$$b(j,k) = \begin{cases} B \cdot r(X_{i_k}) \cdot \Big(lt_{i_k} - \min\{l(X_{i_k}), t_{i_k} - l(X_{i_j})\}\Big) & , \quad \text{if } X_{i_j} \text{ can shield } X_{i_k} \\ 0 & , \quad \text{otherwise} \end{cases}$$

$\square$

The intuition behind Definition 6.2.2 can be seen from Figure 30. The benefit $b(j,k)$ is exactly the gain in server memory (compared to the naive "upslide everything" solution) by using X_{i_j} to shield X_{i_k}. Our heuristic strategy for stream upsliding uses a *edge-weighted graph* representation of the "can shield" relationships in a composite object C_i. Specifically, we define the *shield graph of* C_i, $SG(C_i)$, as an undirected graph with nodes corresponding to the streams X_{i_j} of C_i and an edge e_{jk} between node X_{i_j} and X_{i_k} if and only if X_{i_j} can shield X_{i_k} *or* X_{i_k} can shield X_{i_j}. The *weight* $w()$ of an edge is defined as the maximum gain in memory (compared to the naive approach) that can result by utilizing that edge. More formally, we define $w(e_{jk}) = \max\{\ b(j,k)\ ,\ b(k,j)\ \}$.

Our heuristic method for stream upsliding builds the shield graph $SG(C_I)$ for object C_i and determines a *maximum weighted matching* M on $SG(C_I)$. Essentially, this matching M is a collection of node-disjoint edges of $SG(C_I)$ with a maximum total weight $w(M) = \sum_{e_{jk} \in M} w((e_{jk})$. The maximum weighted matching problem for $SG(C_I)$ can be solved in time $O(n_i^3)$, where n_i is the number of streams in C_i [PS82]. The edges in M determine the set of "stream shieldings" that will be used in our approximate upsliding solution. Furthermore, by our edge weight definitions, it is easy to see that the the total amount of memory that will be used equals exactly $B \cdot \left(\sum_{j=2}^{n_i} lt_{i_j} \cdot r_{i_j}\right) - w(M)$, that is, the memory required by the naive approach minus the weight of the matching.

6.2.5 Local Improvements to $\mathcal{LS}$: List-Scheduling with Backtracking ($\mathcal{LSB}$)

The problem with the stream sliding approach outlined in the previous section is that it may often require large amounts of memory per object that the server simply cannot afford. In such cases, we are still faced with the general sequence packing problem. The results of Section 6.2.2 indicate that using a simple list-scheduling approach for general object sequences can result in arbitrarily bad sequence packings, leading to severe underutilization of the server's bandwidth resources. The main problem of $\mathcal{LS}$ is that by making greedy (i.e., earliest start time) decisions on the placement of objects at each step, it may end up with very "sparse" sequence packings (i.e., schedules with very poor density). This is clearly indicated in the example of Figure 29. Thus, it appears that a better scheduling rule would be, instead of trying to minimize the start time of the new object in the current schedule, try to *maximize the density* of the final, combined sequence. However, maximizing density alone also does not suffice. Returning to the example of Figure 29, it is fairly easy to see that the placement that maximizes the density of the final object is one that simply juxtaposes all the object peaks (i.e., never places one peak on top of another). Since our goal is to minimize the overall schedule length, this is not satisfactory. Instead of trying to maximize density alone, the scheduler should also make sure that the *entire* bandwidth capacity of the server (i.e., height of the bin) is utilized. (Note that using the server's bandwidth capacity instead of r_{max} to define object density does not help; since the bandwidth is fixed, placing objects to maximize this new density is equivalent to trying to maximize their overlap with the current schedule, i.e., the $\mathcal{LS}$ rule.)

In this section, we propose a novel family of scheduling heuristics for sequence packing, termed *list-scheduling with k-object backtracking* ($\mathcal{LSB}(k)$). Informally, these new algorithms try to improve upon simple $\mathcal{LS}$ by occasional local improvements to the schedule. More specifically, the operation of $\mathcal{LSB}(k)$ is as follows. The algorithm schedules objects using simple $\mathcal{LS}$, as long as the incoming objects can be placed in the current schedule without causing the length of the schedule to increase. When placing a new object results in an increase in the makespan, $\mathcal{LSB}(k)$ tries to locally improve the density of the schedule and check if this results in a better schedule. This is done in four steps. First, the last k objects scheduled are removed from the current schedule. Second, these k object sequences are combined into a single sequence in a manner that tries to maximize the density of of the resulting sequence. Third, the "combined" sequence is placed in the schedule using simple $\mathcal{LS}$. Finally, the length of this new schedule is compared to that of the original schedule, and the shorter of the two is retained. The second step in the above procedure can be performed using a slightly modified version of FindMin (Figure 26). The main idea is to maintain some additional state with each interval of candidate start times (step 1) and update this state accordingly when taking interval intersections (step 2). More specifically, for each candidate interval $[a_{j_k}, b_{j_k})$ in S_{o_j} (step 1), we maintain a "height" h_{j_k} storing the bandwidth demand at that interval after placing $< l_{o_j}, r_{o_j} >$ to start there. (Note that the total number of intervals remains $O(k_p)$.) Then, in step 2, when taking the intersection of two intervals $[a_{j-1_l}, b_{j-1_l})$ and $[a_{j_k}, b_{j_k})$ (with the appropriate displacement), we associate a height of $\max\{h_{j_k}, h_{j-1_l}\}$ with the intersection (assuming it is non-empty). Finally, in step 3, we select the interval $[a_l, b_l)$ from I_{k_o} that minimizes the product $\max\{h(C_p), h_l\} \cdot \max\{l(C_p), a_l + l(C_o)\}$, and schedule C_o to start at a_l. We term the resulting basic step algorithm FindMin-D. Similar modifications can also be defined for the Bitonic-FindMin algorithm for bitonic object sequences C_o. The complete $\mathcal{LSB}(k)$ algorithm is depicted in Figure 31.

Figure 32 shows the operation of the $\mathcal{LSB}(3)$ algorithm on our "bad" example for $\mathcal{LS}$ (Figure 29). More specifically, Figure 32(a) shows the schedule after the placement of the 5th object, at which point the algorithm is first forced to backtrack, trying to locally improve schedule density. The result of the improved density packing (after using FindMin-D to combine the last three objects) is depicted in Figure 32(b). Since this schedule is shorter than the one in Figure 32(a), it is retained and $\mathcal{LSB}(3)$ goes back to using $\mathcal{LS}$. Figure 32(c) shows the schedule after $\mathcal{LS}$ places the 6th and 7th objects. Finally, Figure 32(d) shows the final schedule obtained by $\mathcal{LSB}(3)$, which is in fact the *optimal* schedule. Further, note that $\mathcal{LSB}(3)$ had to backtrack only once during the whole scheduling process.

The extra effort involved in backtracking to improve schedule density translates to increased time complexity for $\mathcal{LSB}(k)$ compared to simple $\mathcal{LS}$. Specifically, the complexity of of $\mathcal{LSB}(k)$ can be shown to be $O(|L| \cdot N^2)$, where $|L|$ is the number of objects to be scheduled and N is the total number of component streams. This, of course, assuming general object sequences that cannot employ the more efficient basic step algorithms.

Algorithm $\mathcal{LSB}$(k , L)

Input: A list of composite object sequences $L =< C_1, \ldots, C_n >$ and a backtracking parameter k.

Output: A valid packing of the object sequences in L.

1. Set $T_{curr} = 0$, $C_p = \emptyset$.
2. For $i = 1$ to n do

   2.1. Schedule C_i at time slot FINDMIN(C_i, C_p). Let T'_{curr} be the length of the resulting schedule.

   2.2. If $T'_{curr} = T_{curr}$ continue. Else, do the following.

   2.2.1. Remove C_i and the last $k-1$ objects from C_p. (If k exceeds the number of objects in C_p, remove all objects from C_p.)

   2.2.2. Set $C = \emptyset$ and schedule each object C_j removed from C_p using FINDMIN $-$ D(C_j, C).

   2.2.3. Schedule C over the remaider of C_p using FINDMIN(C_i, C_p). Let T''_{curr} be the length of the resulting schedule.

   2.2.4. If $T'_{curr} \leq T''_{curr}$, set $T_{curr} = T'_{curr}$, restore the original C_p, and continue with the next C_i. Else, leave C_p as is, set $T_{curr} = T''_{curr}$, record that C is now a *single* object, and continue with the next C_i.

Figure 31: Algorithm $\mathcal{LSB}$

6.3 Experimental Study

In this section, we present the results of several experiments we have conducted in order to compare the average-case performance of our composite object scheduling algorithms with that of (a) schedulers based on the MBR simplification; and, (b) the optimal schedule. Given the increased complexity of our algorithms compared to simple MBR packing, another interesting issue is the cost/benefit tradeoff involved in choosing a more elaborate scheduler. We start by presenting our experimental testbed and methodology.

6.3.1 Experimental Testbed

We have experimented with the following algorithms:

- $\mathcal{LS}$: Greedy, list-based scheduling of composite multimedia objects.
- $\mathcal{MBR}(FFDH)$: Level-based scheduling of composite multimedia objects using the MBR simplification and the First-Fit-Decreasing-Height (FFDH) rectangle packing method of Coffman et al. [CGJT80].

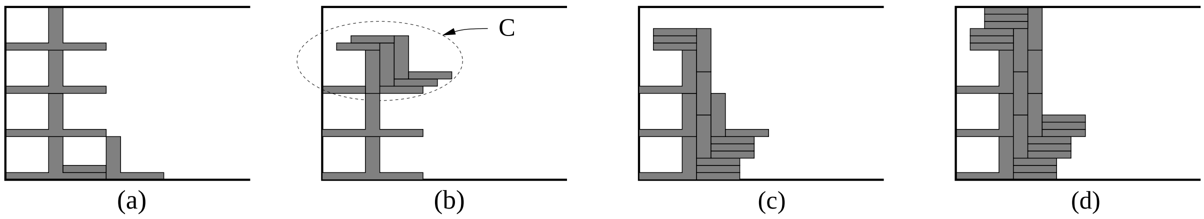


Figure 32: $\mathcal{LSB}(3)$ in action: (a)The point of the first backtracking. (b) The locally improved schedule. (c) Placing the next two objects. (d) The final (optimal) schedule.

We selected the level-based FFDH rectangle packing algorithm since it is known to be one the best-performing rectangle packing methods, both in theory and in practice [CGJT80, CGJ84, BS83]. The average performance of $\mathcal{LS}$ was compared to that of $\mathcal{MBR}(FFDH)$ in order to understand the potential performance benefits of using more clever scheduling techniques to avoid the MBR simplification. We also compared the performance of the two algorithms to a *lower bound* on the response time of the optimal execution schedule. This lower bound ($LBOUND$) was estimated using the formula $LBOUND = \max\{ l_{max}(L), V(L) \}$, where L is the list of objects to be scheduled, $l_{max}(L)$ is the maximum object length in L, and $V(L)$ is the total volume of all objects in L.

We experimented with randomly generated composite objects, obtained with the following procedure. First, the length and bandwidth requirement of the first (i.e., with zero lag) stream of the object was selected randomly from a set of possible lengths and rates (see Table 9). Second, the number of additional component streams was chosen randomly between 0 and 7. Third, for each of the additional streams a length and bandwidth demand was again randomly selected, and a starting point (i.e., lag) was randomly chosen across the length of the first stream. This scheme ensures that our objects are *continuous*, i.e., they have no gaps in the presentation. Note that the MBR assumption is *particularly bad* for objects with gaps, whereas our $\mathcal{LS}$ algorithm can handle these bandwidth gaps and use them effectively to schedule other objects. Thus, we expect $\mathcal{LS}$ to outperform MBR-schemes by an even larger margin when non-continuous objects are allowed. Other than the continuity restriction, note that the "shape" of the object sequences obtained with the above procedure is completely general; that is, it is not constrained to be bitonic, monotonic, etc.

The number of objects to be scheduled varied between 400 and 1400, and the server bandwidth ranged between 40 and 400 Megabits per second (Mbps). For each choice of the number of objects, ten different object lists L were generated randomly using the procedure described above for each object. We used two performance metrics in our study of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$:

(1) the *average response time* of the schedules produced by the two algorithms over all lists of the same size; and, (2) the *average performance ratio* defined as the response time of the schedules produced by the algorithms divided by the corresponding lower bound and averaged over all lists of the same size. (The results presented in the next section are indicative of the results obtained for all values of the number of objects and server bandwidth.)

In all experiments, stream bandwidth demands were chosen from a discrete set of choices, ranging from 62.5 Kbps (e.g., low-quality audio) to 5 Mbps (e.g., MPEG-2 quality video). Similarly, the choice of stream lengths ranged between 10 min (e.g., a short audio clip) and 5 hrs (e.g., a long documentary). Table 9 summarizes the parameter settings used in our experiments.

Experimental Parameter	Value
Aggregate Server Disk Bandwidth (in Megabits per sec – Mbps)	40Mbps – 400Mbps
Number of Composite Objects	400 — 1400
Number of CM Streams per Object	1 – 8
Set of Possible Stream Lengths (in min)	{ 10 , 20 , 30, 60, 90, 120 , 180, 240, 300 }
Set of Possible Stream Bandwidth Requirements (in Mbps)	{ 0.0625 , 0.125 , 1 , 1.5 , 2 , 3 , 4 , 5 }

Table 9: Experimental Parameter Settings

6.3.2 Experimental Results

Figure 33(a) depicts the average response time (in minutes) of the schedules produced by $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for 1000 random composite objects. Our numbers show that $\mathcal{LS}$ consistently outperformed $\mathcal{MBR}(FFDH)$ over the entire range of available server bandwidth, offering relative improvements in the range of 50%–55%. That is, $\mathcal{LS}$ managed to cut down the schedule response time to less than half of that obtained by $\mathcal{MBR}(FFDH)$. The average density of the composite objects created during this experimental run was 0.461448, i.e., on the average, more than half of an object's MBR was "empty". Thus, although $\mathcal{MBR}(FFDH)$ does a very good job of packing the given rectangles, it is still bounded by the inherent inefficiency of the MBR simplification. On the other hand, $\mathcal{LS}$ takes advantage of the object shapes and irregularities to achieve better densities in the final schedule and, consequently, improved schedule response times.

The average performance ratios of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ obtained over the same experimental run (i.e., with 1000 composite objects) are shown in Figure 33(b). Figure 34(a) shows

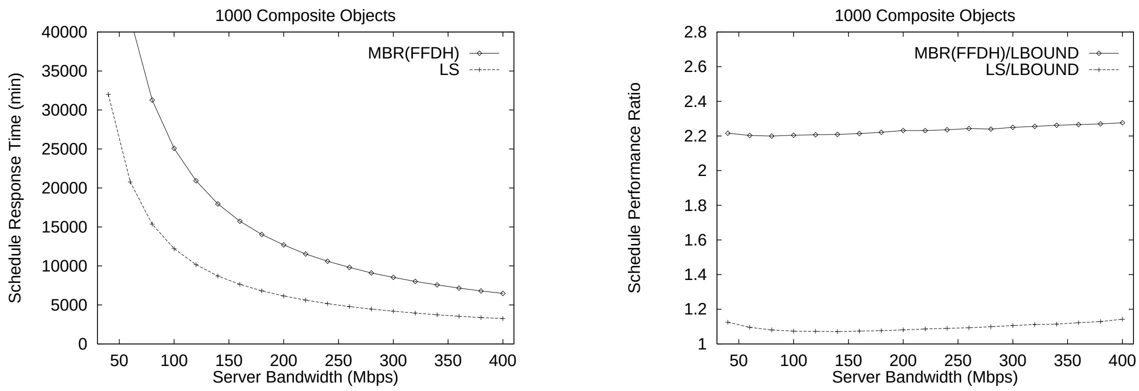
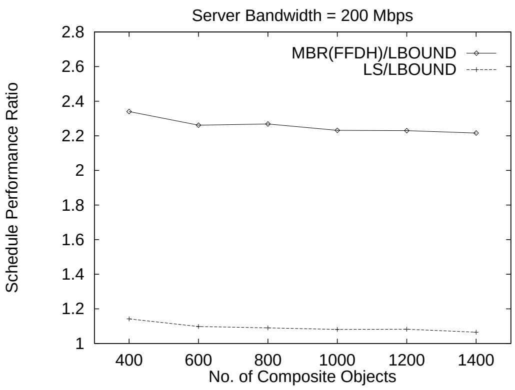


Figure 33: (a) Average schedule response times obtained by $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for 1000 objects. (b) Average performance ratios of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for 1000 objects.

the average performance ratios of the two algorithms as a function of the number of objects for a fixed amount of server bandwidth (200Mbps). Note that the performance of $\mathcal{LS}$ is consistently within less than 15% of the lower bound on the optimal response time. Thus, even though we have shown that simple $\mathcal{LS}$ can be arbitrarily bad under certain worst-case scenarios, our results show that it offers excellent average-case behavior (for randomly generated object sequences). Furthermore, since $\mathcal{LS}$ is so close to optimal, the margin of possible improvement becomes very limited. Even if more complicated schemes like $\mathcal{LSB}$ could offer some improvement on the average over $\mathcal{LS}$, the potential benefit certainly does not seem to warrant the extra complexity. It is still possible, however, that small local perturbations on the greedy $\mathcal{LS}$ schedule, like the ones performed by $\mathcal{LSB}(k)$ with a *small* backtracking parameter k, or stream sliding methods could prove useful to avoid "bad" scenarios. Such scenarios could occur, for example, when object shapes are not completely random. We intend to investigate this issue in our future experimental work.

Finally, the table in Figure 34(b) shows the running times of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for scheduling a list of 1000 composite objects. These times were recorded on a 100Mhz SUN SPARCstation. As expected, $\mathcal{MBR}(FFDH)$ is significantly faster than $\mathcal{LS}$, since its complexity is only $O(|L| \cdot \log |L|)$, where $|L|$ is the number of composite objects to be scheduled. On the other hand, $\mathcal{LS}$ is still fast enough for all practical purposes (the average scheduling time per object is a few milliseconds) and, as our results have shown, offers dramatically improved schedules compared to $\mathcal{MBR}(FFDH)$.



Time to schedule 1000 objects
(in sec)

Bandwidth	MBR(FFDH)	LS
100	0.016	9.041
200	0.012	6.233
300	0.010	4.709
400	0.009	3.760

Figure 34: (a) Average performance ratios of $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$ for 200 Mbps of server bandwidth. (b) Running times for $\mathcal{LS}$ and $\mathcal{MBR}(FFDH)$.

6.4 Ongoing Work: Stream Sharing

Until now, we have implicitly assumed that the composite multimedia objects to be scheduled are *disjoint*, in the sense that their component streams correspond to different data objects in the underlying repository. However, it is quite possible for distinct composite objects to have one or more components in common. Examples of such non-disjoint composite objects range from simple movie presentations, where the same video needs to be displayed with different soundtracks (e.g., in different languages), to complex news stories authored by different users, that can share a number of daily event clips.

In the presence of such common components, it is possible that exploiting *stream sharing* can lead to better schedules. The basic idea is that by appropriately scheduling non-disjoint composite objects, the streams delivering their common component(s) can be shared by all the composite object presentations. Clearly, stream sharing can reduce the aggregate resource requirements of a set of non-disjoint objects and it is easy to construct examples for which exploiting stream sharing can drastically improve the response time of a presentation schedule. On the other hand, stream sharing possibilities also increase the complexity of the relevant scheduling problems. Even simple cases of the problem (e.g., when all streams and composite objects are of unit length) give rise to hard scheduling problems that, to the best of our knowledge, have not been addressed in the scheduling literature [Cof98]. The problem becomes even more challenging when extra memory is available, since stream sliding and caching techniques can be used to increase the possibilities for stream sharing across composite objects. Finally, note that our stream sharing problem also appears to be related to the problem of exploiting common subexpressions during the simultaneous optimization of multiple queries [Sel88]. However, the use of a schedule makespan optimization metric (instead of total plan cost) makes our problem significantly harder to formulate and solve.

6.5 Conclusions

Effective resource scheduling for composite multimedia objects is a crucial requirement for next generation multimedia database systems. Despite the importance of the problem, the complexity of the relevant task scheduling models has limited prior research to very specific subproblems. Furthermore, today's systems typically employ worst-case (i.e., MBR) assumptions that can lead to severe wastage of precious server resources. In this chapter, we have presented a novel *sequence packing* formulation of the composite object scheduling problem and we have proposed novel efficient algorithms drawing on techniques from pattern matching and multiprocessor scheduling. More specifically, we have developed efficient "basic step" methods for combining two object sequences into a single, combined sequence and we have incorporated these methods within: (1) a simple, greedy scheduler base on Graham's list-scheduling paradigm ($\mathcal{LS}$); and, (2) a more complex scheduler ($\mathcal{LSB}$) that tries to improve upon simple $\mathcal{LS}$ by occasional local backtracking. We have shown that although simple list-scheduling schemes are provably near-optimal for monotonic object sequences, they exhibit poor worst-case performance for general object sequences. It is exactly this worst-case behavior that $\mathcal{LSB}$ has been designed to avoid. On the other hand, our experimental results with randomly generated objects have shown that simple $\mathcal{LS}$ offers excellent average-case performance compared to both an MBR-based approach and the optimal solution.

Chapter 7

Throughput-Competitive Admission Control for Continuous Media Databases

In this chapter[1], we explore the implications of the on-line nature of the admission control problem for CM databases which has, for the most part, been ignored in the multimedia literature. Our performance metric for admission control strategies is the *total server throughput* over a sequence of requests and our methodology is based on the competitive analysis framework for on-line algorithms [ST85]. The basic quality metric in this framework is the *competitive ratio* of an on-line algorithm, which is defined to be the maximum (over all possible request sequences) value of the ratio of the performance of the optimal off-line algorithm for a request sequence to the performance of the on-line algorithm for the same request sequence. Note that, by definition, competitive analysis is tantamount to a worst-case analysis in the off-line case. An algorithm with a low competitive ratio is one that performs close to optimal in all situations. Since no assumptions are made about the sequence of requests offered to the server, the competitive ratio provides a very *robust* measure of performance.

We assume a centralized database server where incoming playback requests require some fraction of the server's bandwidth for some period of time. For example, a request to view a half-hour MPEG-1 video clip requires 1.5 Megabits per second (Mbps) of the server's bandwidth for the 30 minutes of playback. We consider two different cases of the problem. In the first case, we assume that all requests require the same fraction of the server's bandwidth (e.g., all clips are MPEG-1 encoded videos); thus, the server can be viewed as a of collection of available playback channels. In the second, more general case, different fractions of the server's bandwidth can be reserved. We show that the conventional Work-Conserving ($\mathcal{WC}$) policy where an incoming request is always admitted if there is sufficient bandwidth to accommodate it, can behave poorly in an on-line setting. More specifically, we show that the competitive ratio of $\mathcal{WC}$ is

[1] Parts of this chapter have appeared in the *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98)* [GIÖS98].

$1+\Delta$ for the case of identical bandwidth requests and $\frac{1+\Delta}{1-\rho}$ for the case of variable bandwidth requests, where Δ is the ratio of maximum to minimum request length and ρ is the maximum fraction of the server's bandwidth that a request can demand. We introduce novel admission control strategies based on the idea of *prepartitioning* the bandwidth capacity of the server among requests of different length and prove that, for sufficiently large server bandwidth, these strategies are $O(\log\Delta)$-competitive. We also show an $\Omega(\log\Delta)$ (resp. $\Omega(\frac{\log\Delta}{1-\rho})$) lower bound on the competitive ratio of any deterministic *or* randomized algorithm for the identical (resp. variable) bandwidth case, thereby establishing the near-optimality of our on-line algorithms. Based on the above results, we propose a bandwidth prepartitioning scheme that makes use of clip popularities to ensure good average-case as well as worst-case performance. The results of our experimental study verify the benefits of our scheme as compared to $\mathcal{WC}$. More specifically, both algorithms are shown to perform adequately well when the server is underutilized or persistently overloaded. However, we expect that a well designed system has undergone effective capacity planning and, therefore, will not be overloaded persistently but only at short time intervals. We capture such short term overloads in our experiments, and demonstrate that our admission control scheme outperforms $\mathcal{WC}$ substantially under these workloads.

7.1 Problem Formulation

We view a CM database server as a "black box" capable of offering a sustained bandwidth capacity of B. The input sequence consists of a collection of *requests* $\overline{\sigma} = \sigma_1, \sigma_2, \ldots, \sigma_N$. The i^{th} request is represented by the tuple $\sigma_i = (t_i, l_i, r_i)$, where l_i, r_i denote the length and bandwidth requirement (respectively) of the requested CM clip and t_i is the arrival time of σ_i. Given a collection of different requests that are handled by a server (based, for example, on the clips available at the server or the server's usage patterns), we use l_{max} (l_{min}) to denote the length of the longest (shortest) request. (r_{max} and r_{min} are defined similarly.) Finally, we define $\Delta = \frac{l_{max}}{l_{min}}$ and $\rho = \frac{r_{max}}{B}$.

We use *competitive analysis* [ST85] to measure the performance of different admission control strategies. Our optimization metric is the *total throughput*; that is, the bandwidth-time product over a given sequence of requests. More formally, given an on-line scheduling policy A and an input sequence $\overline{\sigma}$, we define the *benefit* of A on $\overline{\sigma}$ as $V_A(\overline{\sigma}) = \sum_{S_A} l_i \cdot r_i$, where $S_A \subseteq \overline{\sigma}$ is the set of requests scheduled by A. The *competitive ratio* of an on-line algorithm A is defined as the maximum value $\kappa(A)$ over all possible request sequences of the ratio of the throughput achieved by the optimal off-line algorithm for a request sequence to the throughput achieved by A for the same sequence. If A is a *randomized algorithm*, then the throughput achieved by

A for a request sequence is averaged over all possible "coin flips" of A [MR95]. More formally,

$$\kappa(A) = \begin{cases} \sup_{A^*,\overline{\sigma}} \frac{V_{A^*}(\overline{\sigma})}{V_A(\overline{\sigma})} & , \text{ if } A \text{ is deterministic} \\ \\ \sup_{A^*,\overline{\sigma}} \frac{V_{A^*}(\overline{\sigma})}{E_A[V_A(\overline{\sigma})]} & , \text{ if } A \text{ is randomized} \end{cases}$$

where $\overline{\sigma}$ ranges over all possible request sequences, A^* ranges over all off-line (i.e, *clairvoyant*) algorithms, and the expectation $E_A[\cdot]$ is taken over the random choices of A. Thus, an algorithm with a *small* competitive ratio is guaranteed to perform close to optimal in all situations. We say that algorithm A is k-competitive if $\kappa(A) \leq k$.

It is conventional in the analysis of on-line methods to describe things in terms of a game between a player (the on-line algorithm) and an *adversary* (the off-line algorithm), whose goal is to produce a request sequence that would force the player to perform poorly. For randomized algorithms, different models of adversaries have been proposed depending on the adversary's knowledge of the player's random choices [MR95]. The lower bounds presented in this chapter assume the "weakest" model of an *oblivious* adversary; that is, an adversary that is oblivious to the random choices made by the on-line algorithm.

7.2 Competitive Analysis of Admission Control

7.2.1 The Greedy/Work-Conserving Policy

The scheduling strategy used as a starting point in our study is the basic *Work-Conserving* ($\mathcal{WC}$) scheme traditionally used for admission control in CM servers. $\mathcal{WC}$ is based on the following greedy rule: *"schedule request σ_i immediately if the server has at least r_i bandwidth available at time t_i; otherwise, reject σ_i".* As our results show, $\mathcal{WC}$ offers rather poor performance guarantees in an on-line setting.

First, consider a restricted version of the admission control problem in which all requests require a constant fraction of the server's bandwidth B. That is, $r_i = r$ for all i. Let $c = \left\lfloor \frac{B}{r} \right\rfloor \geq 1$ denote the number of playback *channels* available at the server[2]. The following theorem establishes the competitiveness of $\mathcal{WC}$ in this setting.

Theorem 7.2.1 $\mathcal{WC}$ is $(1+\Delta)$-competitive for scheduling requests with identical bandwidth requirements on-line; that is, $\kappa(\mathcal{WC}) \leq 1 + \Delta$. Furthermore, this bound is tight. □

In general, a playback request requires an arbitrary portion of the server's bandwidth B. This bandwidth requirement depends, for example, on the data encoding method used (e.g.,

[2]Note that, in this case, maximizing throughput is equivalent to maximizing total scheduled request length [LT94].

MPEG-1, MPEG-2) or the Quality of Service (QoS) specified by the client. The following theorem shows the effect of this more general model on the competitive factor of the $\mathcal{WC}$ policy.

Theorem 7.2.2 $\mathcal{WC}$ is $\frac{B\cdot(1+\Delta)}{\max\{B-r_{max},r_{min}\}}$-competitive for scheduling requests with different bandwidth requirements on-line. □

Thus, allowing variability along the second dimension (i.e., bandwidth) multiplies the competitiveness of $\mathcal{WC}$ by a factor that depends on B, r_{max}, and r_{min}. Intuitively, this term captures the effects of the worst-case bandwidth loss due to fragmentation. If $B \geq r_{max} + r_{min}$, as will usually be the case for CM servers and requests, then the following corollary applies.

Corollary 7.2.1 If $B \geq r_{max} + r_{min}$ then $\mathcal{WC}$ is $\frac{1+\Delta}{1-\rho}$-competitive, where $\rho = \frac{r_{max}}{B}$. □

Note that the competitiveness bound of $\frac{1+\Delta}{1-\rho}$ is, in fact, valid regardless of the relative sizes of B and $r_{max} + r_{min}$. Theorem 7.2.2 just gives a tighter bound when $B < r_{max} + r_{min}$. Also, note that for typical CM numbers the fraction ρ is much smaller than unity. For example, even for the relatively high MPEG-2 rate requirements of 6-8 Mbps, a CM server with a low-end RAID can sustain 40-60 concurrent streams [ÖRS95b, WLDH96]. The denominator in our competitiveness bound agrees with the bounds given by Bar-Noy et al. [BNCK$^+$95] for the *preemptive* version of the problem. (They avoid dependence on Δ through clever use of the preemption mechanism.)

7.2.2 Lower Bounds

In this section, we prove lower bounds on the competitive ratio of any *deterministic or randomized* algorithm for on-line admission control. Our results demonstrate the existence of an exponential separation between the competitive ratio of $\mathcal{WC}$ and the lower bound on the competitive ratio of any deterministic or randomized algorithm. This clearly suggests the possibility for improvement by using non-greedy schemes. We propose such schemes with near-optimal competitiveness in Section 7.2.3.

Once again, let us start with the identical bandwidth case (i.e., $r_i = r$ for all i). A simple adversary argument shows that for the case of a single bandwidth channel (i.e., the OIS problem [LT94]), there is a lower bound of $1 + \Delta$ on the competitiveness of any deterministic scheduler. This argument fails when the number of channels is increased. However, as the following theorem shows, no deterministic *or* randomized admission control scheme can be better than $\Omega(\log \Delta)$ competitive.

Theorem 7.2.3 Any deterministic *or* randomized on-line admission control algorithm for CM requests with identical bandwidth requirements has a competitive ratio of $\Omega(\log \Delta)$. $\square$

Similar lower bounds on the competitive ratio hold for the variable bandwidth case. Again, the effect of bandwidth fragmentation introduces a multiplicative factor of $\frac{1}{1-\rho}$.

Theorem 7.2.4 Consider a sequence of CM requests with variable bandwidth requirements. Then:

(1) Any *deterministic* on-line admission control algorithm has a competitive ratio of (a) $\Omega(\frac{\rho\cdot(1+\Delta)}{1-\rho})$, if $\rho > \frac{1}{2}$; and, (b) $\Omega(\frac{\log \Delta}{1-\rho})$, otherwise.

(2) Any *randomized* on-line admission control algorithm has a competitive ratio of $\Omega(\frac{\log \Delta}{1-\rho})$, if $\rho \leq \frac{1}{2 \log \Delta}$.

$\square$

We should note that Awerbuch et al. [AAP93] also proved an $\Omega(\log \Delta)$ lower bound for *deterministic* on-line circuit routing in the case of requests with identical bandwidth requirements. However, our lower bounds for the more general variable bandwidth case also demonstrate the effect of the maximum bandwidth demand (ρ) which was not factored into their results. Furthermore, we have shown that the logarithmic lower bounds cannot be improved upon through the use of randomization.

7.2.3 Bandwidth Prepartitioning Policies

We now propose novel deterministic admission control policies that guarantee near-optimal competitive ratios for reasonably large bandwidth capacities. Our policies are based on *prepartitioning* the bandwidth capacity of a CM server among requests of different length. Roughly speaking, the basic idea of the bandwidth prepartitioning schemes is to isolate requests with large differences in length, thus ensuring that short requests cannot "steal" the entire server bandwidth from longer (and, more profitable) requests[3].

The first policy we introduce is termed *Simple Bandwidth Prepartitioning* ($\mathcal{SBP}$) and is depicted in Figure 35(a)[4]. The $\mathcal{SBP}$ algorithm exploits the server's knowledge of the Δ ratio by classifying requests to channel groups based on their length and then using a $\mathcal{WC}$ policy within each group. The idea is that by classifying the requests into different partitions according to their length range, we are ensuring that the maximum to minimum length ratio is bounded by a constant *within each partition*.

[3] Note that, if $\Delta = 1$ (i.e., all requests have identical lengths) then simple $\mathcal{WC}$ offers optimal competitiveness. Thus, we will assume that $\Delta > 1$ or, equivalently, $\log \Delta > 0$ in the remainder of this chapter.

[4] We describe our policies in terms of the more general variable bandwidth case. The restriction to identical bandwidth requests should be straightforward.

Simple Bandwidth Prepartitioning
1. Divide the available bandwidth B into $\lceil \log \Delta \rceil$ partitions $B_1, \ldots, B_{\lceil \log \Delta \rceil}$, where the size of the i^{th} partition is $\lvert B_i \rvert = \frac{B}{\lceil \log \Delta \rceil}$. 2. For each arriving request $\sigma_j = (t_j, l_j, r_j)$ 2.1. Let $i \in \{1, \ldots, \lceil \log \Delta \rceil\}$ be such that: $2^{i-1} \cdot l_{min} \leq l_j < 2^i \cdot l_{min}$ (allowing for $l_j = 2^i \cdot l_{min}$ if $i = \lceil \log \Delta \rceil$). 2.2. If the amount of free bandwidth in partition B_i is less than r_j, then reject σ_j; Otherwise, schedule σ_j in partition B_i;

Down-shift Bandw. Prepartitioning
1. Divide the available bandwidth B into $\lceil \log \Delta \rceil$ partitions $B_1, \ldots, B_{\lceil \log \Delta \rceil}$, where the size of the i^{th} partition is $\lvert B_i \rvert = \frac{B}{\lceil \log \Delta \rceil}$. 2. For each arriving request $\sigma_j = (t_j, l_j, r_j)$ 2.1. Let $i \in \{1, \ldots, \lceil \log \Delta \rceil\}$ be such that: $2^{i-1} \cdot l_{min} \leq l_j < 2^i \cdot l_{min}$ (allowing for $l_j = 2^i \cdot l_{min}$ if $i = \lceil \log \Delta \rceil$). 2.2. If the total amount of free bandwidth in $B_1 \cup \ldots \cup B_i$ is less than r_j, then reject σ_j; Otherwise, schedule σ_j using available bandwidth from partitions $B_i, B_{i-1}, \ldots, B_1$ *in that order.*

Figure 35: (a) Algorithm $\mathcal{SBP}$. (b) Algorithm $\mathcal{DBP}$.

The following theorem shows that this simple prepartitioning scheme results in a significant improvement in the competitive ratio for CM servers with bandwidth B larger than $r_{max} \cdot \lceil \log \Delta \rceil$. This requirement is typically satisfied by today's servers, even for large values of r_{max} and Δ. For example, if $r_{max} = 8$ Mbps and $l_{max} = 120 \cdot l_{min}$, then $r_{max} \cdot \lceil \log \Delta \rceil = 56$ Mbps, i.e., less than the transfer rate of a *single* high-end magnetic disk [ÖRS96c].

Theorem 7.2.5 Assume $\rho = \frac{r_{max}}{B} < \frac{1}{\lceil \log \Delta \rceil}$ (or, equivalently, $c > \lceil \log \Delta \rceil$ for the identical bandwidth case). Then the $\mathcal{SBP}$ admission control policy is:

(1) $3 \cdot \lceil \log \Delta \rceil$-competitive for the identical bandwidth case; and,

(2) $\frac{3 \cdot \lceil \log \Delta \rceil}{1 - \rho \cdot \lceil \log \Delta \rceil}$-competitive for the variable bandwidth case.

□

Thus, by merely isolating different length ranges, the $\mathcal{SBP}$ admission control policy improves the competitiveness of $\mathcal{WC}$ from linear to logarithmic in Δ, at least for the identical bandwidth case. The main idea behind $\mathcal{SBP}$ is that in order to be competitive under a worst-case scenario, the scheduler should not allow short duration requests to monopolize the server's bandwidth. However, $\mathcal{SBP}$ can also suffer from bandwidth fragmentation in the variable bandwidth case. In the worst case, bandwidth approximately equal to r_{max} is lost *in each partition*, leading to a total bandwidth loss of $r_{max} \cdot \lceil \log \Delta \rceil$ in the server. Intuitively, we would like to be able to "combine" these bandwidth fragments to allow for incoming requests to be scheduled across partitions, especially if these requests are long since this implies more guaranteed profit.

The *Down-shift Bandwidth Prepartitioning* ($\mathcal{DBP}$) policy depicted in Figure 35(b) is based exactly on these observations. As in $\mathcal{SBP}$, the $\mathcal{DBP}$ algorithm also prohibits short requests from monopolizing the server, *but* it also allows longer (and thus, more profitable requests) to be "down-shifted" to lower groups and steal unused bandwidth that would otherwise be dedicated to shorter requests. Theorem 7.2.6 shows that incorporating this change does not compromise logarithmic competitiveness.

Theorem 7.2.6 Assume $\rho = \frac{r_{max}}{B} < \frac{1}{\lceil \log \Delta \rceil}$ (or, equivalently, $c > \lceil \log \Delta \rceil$ for the identical bandwidth case). Then the $\mathcal{DBP}$ admission control policy is:

(1) $(1 + 8 \cdot \lceil \log \Delta \rceil)$-competitive for the identical bandwidth case; and,

(2) $\left(1 + \frac{8 \cdot \lceil \log \Delta \rceil}{1 - \rho \cdot \lceil \log \Delta \rceil}\right)$-competitive for the variable bandwidth case.

□

Corollary 7.2.2 follows directly from Theorems 7.2.5 and 7.2.6. Combined with the lower bounds in Section 7.2.2, Corollary 7.2.2 establishes the *near-optimality* of the $\mathcal{SBP}$ and $\mathcal{DBP}$ policies for the variable bandwidth case, assuming that the server bandwidth B is larger than $2 \cdot r_{max} \cdot \lceil \log \Delta \rceil$. Again, this is a requirement that is typically satisfied by today's CM servers and applications. (See the discussion before Theorem 7.2.5.) Note that even smaller competitive ratios can be obtained if $B > k \cdot r_{max} \cdot \lceil \log \Delta \rceil$, where $k > 2$.

Corollary 7.2.2 Assume $\rho = \frac{r_{max}}{B} < \frac{1}{2 \cdot \lceil \log \Delta \rceil}$. Then:

1. The $\mathcal{SBP}$ admission control policy is $6 \cdot \lceil \log \Delta \rceil$-competitive for the variable bandwidth case; and,
2. The $\mathcal{DBP}$ admission control policy is $(1 + 16 \cdot \lceil \log \Delta \rceil)$-competitive for the variable bandwidth case.

□

Although the constants in the competitiveness bounds we have shown for $\mathcal{DBP}$ are larger than those of $\mathcal{SBP}$, we conjecture that they can be improved. To support our conjecture, note that for the identical bandwidth case when a request of length $l_j \in [2^{i-1} \cdot l_{min}, 2^i \cdot l_{min})$ is rejected in $\mathcal{SBP}$, the scheduler can guarantee that the benefit of running requests is at least $\frac{c}{\lceil \log \Delta \rceil} \cdot 2^{i-1} \cdot l_{min}$, whereas with $\mathcal{DBP}$ the corresponding guaranteed benefit is at least:

$$\frac{c}{\lceil \log \Delta \rceil} \cdot \sum_{k=1}^{i} 2^{k-1} \cdot l_{min} = \frac{c}{\lceil \log \Delta \rceil} \cdot (2^i - 1) \cdot l_{min},$$

that is, nearly double the benefit for $\mathcal{SBP}$. Of course, the main advantage of $\mathcal{DBP}$ over $\mathcal{SBP}$ is that, by "down-shifting", it can significantly reduce the effects of bandwidth fragmentation in the variable bandwidth case. A formal proof of improved competitive ratios for $\mathcal{DBP}$ is left as an open problem for future research.

Even though $\mathcal{SBP}$ and $\mathcal{DBP}$ guarantee logarithmic competitiveness under a worst-case scenario, they may also severely underutilize the server in average cases. For example, when all the requests address the shortest group of clips residing on the server, both schemes will end up utilizing only $\frac{1}{\lceil \log \Delta \rceil}$ of the available bandwidth. This is clearly undesirable. We now propose a novel on-line admission control policy that employs the intuition of prepartitioning schemes (to avoid worst-case scenarios for $\mathcal{WC}$) within a framework that also allows for good average-case performance. Roughly speaking, the idea is to use the methodology of $\mathcal{DBP}$ but define the sizes of the bandwidth partitions B_i as a function of the popularities and/or the lengths of all requests in the length range $[2^{i-1} \cdot l_{min}, 2^i \cdot l_{min})$. The resulting admission control scheme, termed *Popularity-based Bandwidth Prepartitioning* ($\mathcal{PBP}$), is depicted in Figure 36. Note that $\mathcal{PBP}$ is given in parameterized form with the parameter f being the specific function of popularities and lengths used to define the partition sizes. In Section 7.3, we describe two specific choices for f used in our preliminary experimental study. The $\mathcal{PBP}$ admission control scheme relies on the assumption that request (i.e., clip) popularities can be estimated with reasonable accuracy (e.g., using a "moving window" prediction method [LV95]). Clearly, taking popularities into account is necessary to avoid worst-case scenarios for $\mathcal{DBP}$ (i.e., when the most frequent requests are also the shortest). In fact, assuming that requests are independent and the given popularities are accurate, we can give simple arguments based on *Chernoff bounds* [MR95] to show that the probability of a worst-case "loss" for $\mathcal{PBP}$ (with specific choices for f) is exponentially small.

7.3 Experimental Study

In this section, we describe the results of a preliminary set of experiments we have conducted with the $\mathcal{WC}$ and $\mathcal{PBP}$ strategies for on-line admission control. Since our competitiveness results clearly demonstrate the superiority of prepartitioning schemes with respect to worst-case scenarios, our goal was to ensure that the worst-case guarantees did not impair *average-case performance.*

7.3.1 Experimental Testbed

To examine the average-case behavior of the $\mathcal{WC}$ and $\mathcal{PBP}$ schemes, we have experimented with three distinct random arrival patterns:

Popularity-based Bandwidth Prepartitioning[f: *function*]

1. Let p_j denote the probability that the length of an incoming request is l_j (i.e., the popularity of l_j). Let PL_i denote the set of (popularity, length) pairs with lengths in the i^{th} range; that is,

$$PL_i \;=\; \{\; (p_j, l_j) \mid l_j \in [2^{i-1} \cdot l_{min}, 2^i \cdot l_{min}) \;\}.$$

2. Divide the available bandwidth B into $\lceil \log \Delta \rceil$ partitions $B_1, \ldots, B_{\lceil \log \Delta \rceil}$, with

$$|B_i| \;=\; \frac{f(PL_i)}{\sum_i f(PL_i)} \cdot B.$$

3. For each arriving request $\sigma_j = (t_j, l_j, r_j)$

   3.1. Let $i \in \{1, \ldots, \lceil \log \Delta \rceil\}$ be such that: $2^{i-1} \cdot l_{min} \leq l_j < 2^i \cdot l_{min}$ (allowing for $l_j = 2^i \cdot l_{min}$ if $i = \lceil \log \Delta \rceil$).

   3.2. If the total amount of free bandwidth in $B_1 \cup \ldots \cup B_i$ is less than r_j, then reject σ_j;
   Otherwise, schedule σ_j using available bandwidth from partitions $B_i, B_{i-1}, \ldots, B_1$ *in that order.*

Figure 36: Algorithm $\mathcal{PBP}$

- Poisson Arrivals. Requests of different lengths arrive at the server according to a Poisson process model with an arrival rate of λ. This is a plausible probabilistic model for servers with a reasonably steady traffic flow (e.g., video servers in scientific research labs serving clips of recorded experiments to scientists around the globe).

- Bursty Arrivals. Requests of different lengths arrive at the server in bursts at regular intervals of time (termed *burst separation*). Each such burst itself consists of a sequence of *request batches*, where each batch consists of requests of identical length arriving during a very short period of time. The batch arrivals are again modeled as a Poisson process with an arrival rate of λ. This workload is intended to model "rush-hour traffic" situations in CM servers.

- Poisson + Short Burst Arrivals. Long requests arrive at the server according to a Poisson process model with an arrival rate of λ_{long}. At the same time, bursts of short requests arrive based on a Poisson process with an arrival rate of λ_{short}. This workload model combines some features of the previous two models. It is intended to represent situations where servers operating under a relatively steady flow of long requests (e.g., movies or sports events), occasionally have to handle bursts of short requests (e.g., the 6 o'clock news).

In most of our experiments, the request lengths were sampled from a discrete set of values between 5 and 150 minutes, with sampling probabilities (i.e., popularities) taken from a Zipfian distribution with skew parameter z [Zip49]. We varied this skew parameter from 0.0 (uniform) to 2.0 (very skewed). Results were obtained for three different models of correlation between request lengths and popularities:

- Positive. Larger popularities are assigned to longer requests.
- Negative. Larger popularities are assigned to shorter requests.
- Random. No length/popularity correlation exists; that is, the values of the Zipfian probability vector are assigned to the different request lengths in a random manner.

We also experimented with two different choices for the f function parameter of the $\mathcal{PBP}$ scheme. The first choice f_1 captured the cumulative popularity of a length range, that is $f_1(PL_i) = \sum_{(p,l)\in PL_i} p$. The second choice f_2 was the total popularity-length product of a range, that is $f_2(PL_i) = \sum_{(p,l)\in PL_i} p \cdot l$.

In our experiments for the identical bandwidth case, we assumed a server with 100 available channels. For the variable bandwidth case, we varied the server's sustained bandwidth capacity between 100 and 250 Megabits per second (Mbps) and selected the rate requirement of a request randomly between 500 Kbps and 8 Mbps. The parameter values are summarized in Table 10.

System Parameter	**Value**
Server Bandwidth Capacity	100 channels / 100–250 Mbps
Request Lengths	5 minutes – 150 minutes
Request Rates (Variable Bandwidth Case)	500 Kbps - 8 Mbps
Zipfian Popularity Skew (z)	0.0 - 2.0

Table 10: Experimental Parameter Settings

For each different combination of input parameters, we modeled the system behavior under each scheduling policy for 20,000 minutes of simulated time and 10 randomly generated request sequences. The results presented here represent the averages over these 10 runs of the system. In all cases, the comparison metric was the *fraction of the server capacity effectively utilized*; that is, the ratio $\frac{V_A(\overline{\sigma})}{(\text{server bandwidth})\times(\text{simulation time})}$, for each scheduler A and request sequence $\overline{\sigma}$.

7.3.2 Experimental Results

We present an overview of our experimental comparison of the $\mathcal{WC}$ and $\mathcal{PBP}$ schemes for the (general) variable bandwidth case. Similar results were obtained for identical bandwidth requests. For the numbers presented here, the request lengths were sampled (based on z and the model of correlation) from the collection $\{5, 10, 15, 90, 120, 150\}$ (in minutes), the request rates were selected (uniformly) from the set $\{0.5, 1.5, 3.0, 4.5, 6.0, 8.0\}$ (in Mbps), and the server bandwidth capacity was 250 Mbps. The plots shown in this section are indicative of the results obtained for other values of request and server parameters.

We focus our discussion on $\mathcal{PBP}[f_2]$; that is, $\mathcal{PBP}$ using the "total popularity×length" partitioning criterion, since it exhibited uniformly better performance than $\mathcal{PBP}[f_1]$ in our experiments. We should stress, however, that even with "cumulative popularity" partitioning, $\mathcal{PBP}$ outperformed $\mathcal{WC}$ by a significant margin for our "bursty" workloads.

The first set of experiments studied the relative effectiveness of the $\mathcal{WC}$ and $\mathcal{PBP}$ schemes under Poisson arrivals for different values of the Zipfian skew parameter z and different length/popularity correlations. Figure 37(a) shows the performance of the schemes as a function of the Poisson arrival rate λ for $z = 0.6$ and random length/popularity correlation. Our basic finding is that, by exploiting its knowledge of clip popularities $\mathcal{PBP}$ is able to do at least as good as $\mathcal{WC}$ in all cases. We should mention that we also experimented with different models of the arrival process (e.g., using uniformly rather than exponentially distributed inter-arrival times) that also led to the same conclusions regarding the relative performance of the strategies under random arrivals.

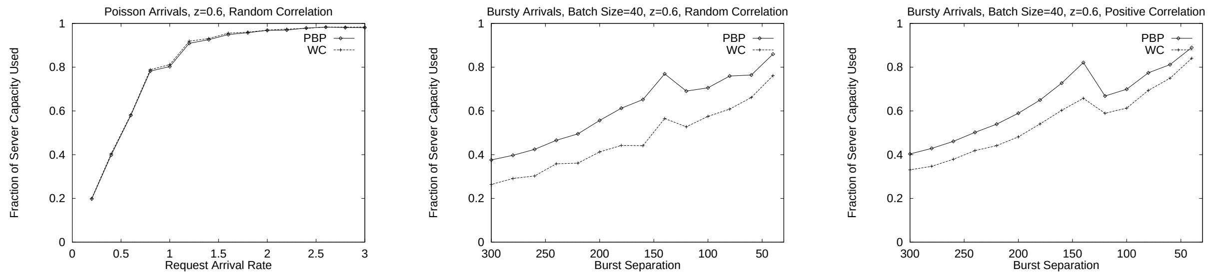


Figure 37: (a) Server throughput under Poisson arrivals. (b) Server throughput under bursty arrivals (random correlation). (c) Server throughput under bursty arrivals (positive correlation).

The second set of experiments concentrated on the relative performance of the algorithms under the Bursty Arrival model described in the previous section. We studied the server utilization as a function of the length of the burst separation interval as well as the size of a batch of arrivals for different values of the z and λ parameters, the size (i.e., number of batches) of a burst, and different modes of correlation. Figure 37(b) shows the server utilization as a function of the burst separation interval for batch size equal to 40, $z = 0.6$, batch arrival rate $\lambda = 0.8$, burst size equal to 10, and random length/popularity correlation. (We show burst separations decreasing from left to right as this reflects increasing load, as in Figure 37(a).) Our results show that under such conditions, $\mathcal{PBP}$ outperforms $\mathcal{WC}$ by an average margin of 15% - 40%. Note that the "jump" observed in the curves as the burst separation approaches 150 minutes is caused by our specific choice of request lengths and our model of "bursty" arrivals. The numbers from the same experiment but for *positive* length/popularity correlation (i.e., longer requests are more popular) are depicted in Figure 37(c). $\mathcal{WC}$ clearly performs better under the positive correlation assumption, since it is able to allocate more of its channels to the more popular (and, more profitable) long requests. Still, $\mathcal{PBP}$ continues to outperform $\mathcal{WC}$ by up to 25%. Figure 38(a) shows the results of the same experiment but for *negative* length/popularity correlation (i.e., shorter requests are more popular). Under such scenarios, our results show that the relative improvement offered by $\mathcal{PBP}$ over $\mathcal{WC}$ can reach 50% - 60%. A different perspective is depicted in Figure 38(b), where server utilization (for the same parameter values and negative correlation) is given as a function of the batch size for a fixed burst separation of 180 minutes. Note that as the batch size increases, the Bursty Arrival model gives rise to worst-case scenarios for $\mathcal{WC}$, where a large batch of short requests can flood the server leaving no capacity for a following batch of larger requests. On the other hand, our $\mathcal{PBP}$ scheme is capable of maintaining a reasonable level of utilization under all circumstances.

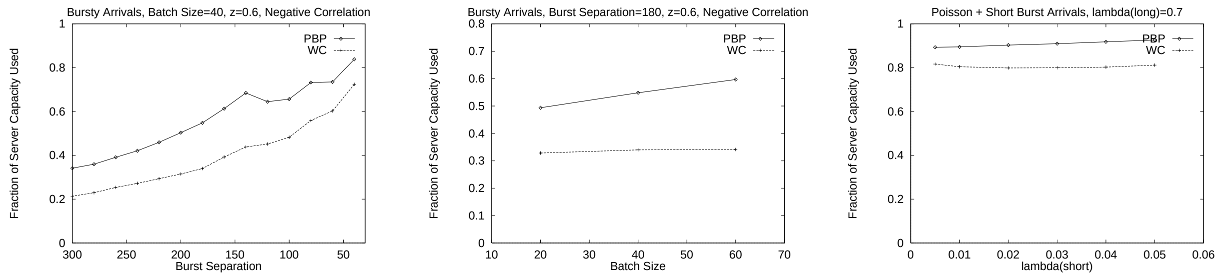


Figure 38: (a) Server throughput under bursty arrivals (negative correlation). (b) Server throughput under bursty arrivals as a function of batch size (negative correlation). (c) Server throughput under Poisson+Short Bursts.

The final set of experiments studied the behavior of the algorithms under the combined Poisson + Short Bursts arrival process. We concentrated on a particular scenario which, we believe, is common in Video-On-Demand environments. Specifically, we assumed that the server is working close to capacity serving requests for long (i.e., $\{90, 120, 150\}$ minutes) movies but occasionally has to handle bursts of short (i.e., $\{5, 10, 15\}$ minutes) requests. That is, λ_{long} was selected large enough to ensure high system utilization and we studied the server utilization as a function of λ_{short}. All length popularities were assumed uniform for this experiment. The results depicted in Figure 38(c) show that, under this scenario, $\mathcal{PBP}$ can offer a 10% - 15% performance improvement over $\mathcal{WC}$, even at high levels of system utilization.

7.4 Conclusions

In this chapter, we have addressed the admission control problem associated with CM database servers from a novel, on-line perspective. Using server throughput as our optimization metric, we showed that the traditionally used Work-Conserving policy has a competitive ratio of $\frac{1+\Delta}{1-\rho}$, where Δ is the ratio of the maximum to minimum request length and ρ is the maximum fraction of the server's bandwidth that a request can demand. We developed novel admission control strategies based on the simple idea of preprepartitioning the bandwidth capacity of the server among requests of different length and proved that that our strategies are $O(\log \Delta)$-competitive for sufficiently large server bandwidth. We also showed an $\Omega(\frac{\log \Delta}{1-\rho})$ lower bound on the competitive ratio of any deterministic *or* randomized algorithm for the problem, thus establishing that our bandwidth prepartitioning algorithms are within a multiplicative constant of the optimal on-line strategy. Based on the intuition gained from our competitiveness results, we proposed prepartitioning schemes that make use of request popularities to ensure good average-case as well as robust worst-case performance, and experimentally verified their effectiveness against the Work-Conserving policy.

We believe that the analytical and experimental results presented in this chapter offer new insights to other optimization problems that arise in CM data management. For example, consider the problem of data placement and static load balancing in distributed CM servers. Briefly, the problem can be described as follows: Given a collection of continuous media clips with lengths (l_i), rates (r_i), and expected popularity (or, probability of access, p_i), determine a "good quality" mapping of these clips to a collection of servers, where each server is characterized by a bandwidth capacity (B_j) and a storage capacity (S_j) and a clip can be mapped to more than one servers (i.e., replication of clips is allowed).

Traditionally, the goal of data placement schemes in this setting is to balance the expected bandwidth load (according to the popularities $\{p_i\}$) across the available servers under the given server storage constraints [LV95, DS95, WYS95]. This model of "popularity-based data placement" aims at achieving good system utilization and balanced system load in an average sense. On the other hand, our competitiveness results indicate that to ensure *robust* system performance, a placement strategy should also try to achieve some secondary "goals". One such goal, for example, would be to place clips with large bandwidth requirements on servers with large bandwidth capacities to guarantee small ρ fractions for each server. As another example, the placement policy should try to replicate short clips across many servers, so that there are many possibilities of dynamically re-assigning (e.g., using a baton passing primitive [WYS95]) streams delivering these clips to different servers. This obviously reduces the probability of a short request causing the rejection of a long request from the system, which is the worst-case scenario in all our competitiveness results. Achieving such secondary data placement goals is especially important in order to ensure good system utilization under short-term fluctuations of the load away from the averages $\{p_i\}$ or overload situations where some client requests simply must be rejected. A detailed investigation of the problem is left for future research.

Chapter 8

Periodic Resource Scheduling for Continuous Media Databases

In this chapter[1], we address the resource scheduling problems associated with supporting EPPV service in multimedia database systems in their most general form. We present a scheduling framework that handles Continuous Media (CM) data with (possibly) different display rates (depending on the media type and/or the compression scheme), different periods (depending on the popularity of a clip), and arbitrary lengths. Given a hardware configuration and a collection of CM clips to be scheduled, we present schemes for determining a schedulable subset of clips under different assumptions about data layout:

- **Clustering.** Each disk is viewed as an independent storage unit; that is, the data of each clip is stored on a single disk and multiple clips can be clustered on each disk. Despite its conceptual simplicity, clustered placement can suffer from disk bandwidth and storage fragmentation, leading to underutilization of available resources.

- **Striping.** Each clip's data is declustered over all available disks. Striping schemes eliminate disk storage fragmentation. On the other hand, as we will see, striping significantly increases the complexity of scalable and cost-effective EPPV services.

In each of the above two cases, our main objective is to maximize the amount of disk bandwidth that is effectively scheduled under the given layout and storage constraints. This is typically the situation facing large-scale CM servers that periodically need to re-schedule their offerings to adapt to a changing audience, content, and popularity profile[2][LV95, PRE]. For the Clustering scheme, we formulate these optimization problems as generalized variants of the 0/1 knapsack problem [IK75, Law79, Sah75]. Since the problems are $\mathcal{NP}$-hard, we present provably near-optimal heuristics with low polynomial-time complexity. We then present two alternative

[1]Parts of this chapter have appeared in the *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)* [GÖS97] and in the *Proceedings of the Eighth International Workshop on Research Issues in Data Engineering – Continuous-Media Databases and Applications (RIDE'98)* [GÖS98].

[2]A related problem is *capacity planning:* Given a fixed user profile, determine the minimum system configuration that can accommodate it. This problem is essentially dual to ours, and we believe that most of our techniques are applicable. We will not address capacity planning further in this thesis.

schemes for striping clips. *Fine-grained Striping (FGS)* views the entire disk array as a single large disk in a manner similar to the RAID-3 data organization [ÖRS95b, PGK88]. This scheme is conceptually simple and significantly reduces the complexity of the relevant resource scheduling problems. However, FGS suffers from increased disk latency overheads that render it impractical, especially for large disk arrays. *Coarse-grained Striping (CGS)* is based on a round-robin distribution of clip data across the disks and has the potential of offering much better scalability and disk utilization than Fine-grained Striping. This, however, comes at the cost of the more sophisticated scheduling methods required to support periodic stream retrieval. Specifically, we demonstrate that the scheduling problems involved in supporting the EPPV service model under the CGS data layout are non-trivial generalizations of the Periodic Maintenance Scheduling Problem (PMSP) [WL83]. Given that PMSP is known to be $\mathcal{NP}$-complete in the strong sense [BRTV90], we propose novel heuristic algorithms for scheduling the periodic retrieval of Coarse-grained striped clips. We follow a two-step approach. First, we introduce the novel concept of a *scheduling tree* structure and demonstrate its use in obtaining collision-free schedules for Periodic Maintenance. Next, we extend our definitions and algorithms to handle the more complex problems introduced by the periodic retrieval under CGS. Thus, our work also contributes to the area of hard real-time scheduling theory by proposing the scheduling tree structure and algorithms as a new approach to Periodic Maintenance. We also present a scheme for packing multiple scheduling trees to effectively utilize disk bandwidth and storage, and show that all the scheduling problems examined in this chapter can be seen as special cases of this general packing formulation. Finally, we present experimental results that compare the average performance of the schemes proposed in this chapter and confirm the superiority of the "Coarse-grained Striping + Scheduling Trees" combination under different workloads. We believe that our proposed scheduling tree framework is a powerful tool, applicable to any scenario involving regular, periodic use of a shared resource.

8.1 Notation and System Model

In this section we present a brief overview of our CM server model and the notation that will be used throughout this chapter. First, we present a model of *round-based retrieval* for continuous media. Our model follows the conventions used in most earlier work on multimedia storage servers [CKY93, GVK$^+$95, ÖRS95a, RV91, RV93]. Next, we present the three different multi-disk data organization schemes considered in this chapter (Clustering, Fine-grained and Coarse-grained Striping) and describe how the round-based retrieval model adapts to each different organization scenario. Finally, we describe the *matrix-based allocation scheme* [ÖRS96b, ÖRS96c] used to optimize the data layout of a clip based on the knowledge of its retrieval period. Table 11 summarizes the notation used in this chapter with a brief description of its semantics. Additional notation will be introduced when necessary.

Param.	**Semantics**
C_i	Continuous media clip ($i = 1, \ldots, N$) (also, task of retrieving C_i)
r_i	Display rate for clip C_i (in Mbps)
T_i	Retrieval period for clip C_i (in sec)
T	Time unit of clip retrieval (round length)
l_i	Length of clip C_i (in sec)
n_i	Retrieval period of C_i in rounds
c_i	Number of columns in the matrix of clip C_i
d_i	Maximum column size (in bits)

Param.	**Semantics**
n_{disk}	Number of disks in CM server
r_{disk}	Disk transfer rate
c_{disk}	Disk storage capacity
t_{seek}	Disk seek time
t_{lat}	Disk latency

Table 11: Clip and Disk Parameters

8.1.1 Retrieving Continuous Media Data

Consider a single magnetic disk storing a collection of continuous media clips. We assume that the disk has a transfer rate of r_{disk}, a storage capacity of c_{disk}, a worst-case seek time of t_{seek}, and a worst-case latency of t_{lat} (which consists of rotational delay and settle time). A clip C_i is characterized by a display rate r_i (the rate at which data for C_i must be transmitted to clients) and a length l_i (in units of time) We refer to the transmission of a clip starting at a given time as a *stream*. Data for streams is retrieved from the disk in *rounds of length* T. For a stream displaying clip C_i (denoted by $stream(C_i)$), a circular buffer of size $2 \cdot T \cdot r_i$ is reserved in the server's buffer cache. In each round, while the stream is consuming $T \cdot r_i$ bits of data from its buffer, the $T \cdot r_i$ bits that the stream will consume in the next round are retrieved from the disk into the buffer. This ensures that each stream will have sufficient data to display the corresponding clip continuously. The quantity $T \cdot r_i$ is termed the *retrieval unit* of clip C_i.

During a round, for streams $stream(C_1), \ldots, stream(C_k)$ for which data is to be retrieved from disk, $T \cdot r_1, ..., T \cdot r_k$ bits are read using the C-SCAN disk head scheduling algorithm [SG94]. C-SCAN scheduling ensures that the disk heads move in a single direction when servicing streams during a round. As a result, random seeks to arbitrary locations are eliminated and the total seek overhead during a round is bounded by $2 \cdot t_{seek}$. Furthermore, retrieval of each non-contiguously stored piece of data can incur a disk latency overhead of at most t_{lat} during a round. To ensure that no stream starves during a round, the sum of the total disk transfer time for all data retrieved and the overall latency and seek time overhead cannot exceed the length T of the round [GVK+95, ÖRS95b, RV91, RV93]. More formally, we require the following inequality to hold[3]:

$$\sum_{\{stream(C_i)\}} \left(\frac{T \cdot r_i}{r_{disk}} + t_{lat}\right) \leq T - 2 \cdot t_{seek}. \tag{5}$$

8.1.2 Multi-Disk Data Organization Schemes

Clustering

In Clustering, each disk of a multi-disk system is viewed as an autonomous unit – entire clips are stored on and retrieved from a single disk and multiple clips can be clustered on each disk. For our round-based model of data retrieval, this means that Inequality (5) must be satisfied on each disk, where the summation is taken over all streams retrieving clips C_i stored on that disk. Note that by the definition of EPPV service, the number of concurrent streams retrieving clip C_i is exactly $\left\lceil \frac{l_i}{T_i} \right\rceil$. Furthermore, each disk has a limited storage capacity that cannot be exceeded by the set of stored clips. Thus, if we let $\{C_i\}$ denote the set of clips clustered on any single disk in the server, we require the following conditions to be satisfied:

$$\left\lceil \frac{l_i}{T_i} \right\rceil \cdot \left[\sum_{\{C_i\}} \left(\frac{T \cdot r_i}{r_{disk}} + t_{lat}\right) \right] \leq T - 2 \cdot t_{seek} \quad \text{and} \quad \sum_{\{C_i\}} l_i \cdot r_i \leq c_{disk}. \tag{6}$$

Fine-grained Striping

A major deficiency of clustered data organization for large scale continuous media services is that it can lead to bandwidth *and* storage fragmentation, and, consequently, underutilization of server resources. Striping schemes eliminate storage fragmentation by declustering a clip's data across all available disks. This essentially means that we no longer need to satisfy a storage capacity constraint on each disk, as long as the total storage requirements of the clips to be scheduled do not exceed the storage capacity of the disk array.

In Fine-grained Striping (FGS) [ÖRS96a], each retrieval unit of a clip is striped across all n_{disk} disks of the server. Consequently, every stream retrieval involves all the n_{disk} disk heads working in parallel, with each disk being responsible for fetching $\frac{T \cdot r_i}{n_{disk}}$ bits of C_i in each round. This striping strategy is employed in the RAID-3 data distribution scheme [PGK88]. For EPPV service, the number of concurrent streams retrieving clip C_i from the array is exactly

[3] Although our schemes can be extended to handle disk calibration and multi-zone disks [ÖRS95a], these issues are not addressed in this thesis.

$\left\lceil \frac{l_i}{T_i} \right\rceil$. Thus, to ensure continuous delivery, the following condition must be satisfied:

$$\left\lceil \frac{l_i}{T_i} \right\rceil \cdot \left[\sum_{i=1}^{N} \frac{T \cdot r_i}{r_{disk} \cdot n_{disk}} + N \cdot t_{lat} \right] \leq T - 2 \cdot t_{seek}, \tag{7}$$

where N is the total number of clips on the server (see Table 11). Despite its conceptual simplicity, Fine-grained Striping can lead to underutilization of available disk bandwidth due to increased latency overheads [ÖRS96a]. This is clearly demonstrated in the above condition for continuity, which shows that, during each round, all disks incur a penalty of t_{lat} for each clip stored *in the entire server*. These latency penalties limit the scalability of an EPPV server based on Fine-grained Striping since the problem is obviously exacerbated as the size of the disk array grows.

Coarse-grained Striping

In the Coarse-grained Striping (CGS) scheme [ÖRS96a], the retrieval units of each clip are mapped to individual disks in a round-robin manner. Consequently, the retrieval of data for a stream on clip C_i proceeds in a round-robin fashion along the disk array. During each round, a single disk is used to fetch a retrieval unit of C_i and consecutive rounds employ consecutive disks[4]. This striping strategy is employed in the RAID-5 data distribution scheme [PGK88].

Coarse-grained Striping avoids the large latency overheads of the Fine-grained scheme and, consequently, can offer much better scalability and bandwidth utilization [ÖRS96a]. On the other hand, supporting periodic stream retrieval requires much more sophisticated scheduling methods than either Clustering or Fine-grained Striping. This is because, unlike the two previous data organization schemes, Coarse-grained Striping does not impose a steady load on each disk *during each round*. Consider the retrieval of clip C_i from a particular disk in the array. By virtue of the round-robin placement, each stream retrieving data of C_i needs to fetch a retrieval unit of $T \cdot r_i$ bits from that disk periodically, at intervals of n_{disk} rounds. Furthermore, to support EPPV service, the streams retrieving clip C_i are themselves periodic with a period $T_i = n_i \cdot T$. Thus, supporting continuous, periodic service under Coarse-grained Striping gives rise to complex *periodic real-time task scheduling problems* [LL73] that cannot be reduced to simple algebraic conditions like Inequalities (6) and (7). A concise description of these problems and our proposed solution will be given later in this chapter.

[4]We assume that a disk has sufficient bandwidth to support the retrieval of one or more clips. If this does not hold, one or more disks can be viewed as a single composite disk.

8.1.3 Reducing Disk Latencies: Matrix-Based Allocation

By definition, the EPPV service model associates with each clip C_i a retrieval period T_i. We assume that retrieval periods are multiples of the round length T. This is a reasonable assumption, since retrieval periods will typically be multiples of minutes or even hours and the length of a round (usually bounded by buffering constraints) will not be more than a few seconds. The *matrix-based allocation scheme* [ÖBRS94, ÖRS96c], increases the number of clients that can be serviced under the EPPV model by laying out data based on the knowledge of retrieval periods. The basic idea is to distribute, for each clip C_i, the starting points for the $\lceil \frac{l_i}{T_i} \rceil$ concurrent *display phases* of C_i uniformly across its length. Each such display phase corresponds to a different stream servicing (possibly) multiple clients. Conceptually, each clip C_i is viewed as a matrix consisting of elements of length T (Figure 39(a)).

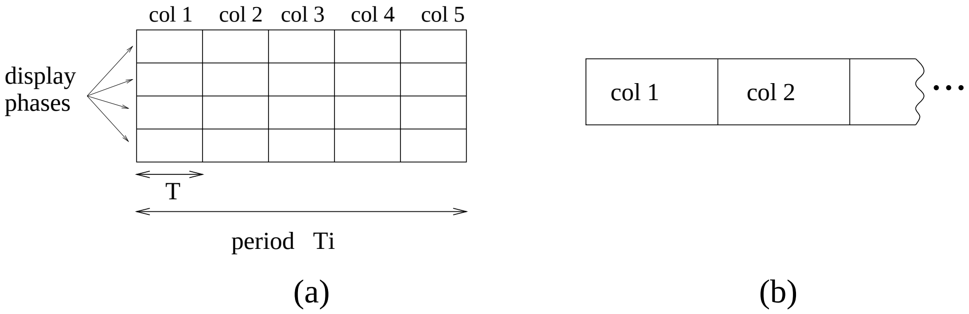


Figure 39: (a) A clip matrix. (b) Its layout on disk.

We define $n_i = \frac{T_i}{T}$ (i.e., the length of the retrieval period of C_i in rounds). The matrix for C_i consists of $c_i = \min\{n_i, \lceil \frac{l_i}{T} \rceil\}$ columns and $\lceil \frac{l_i}{T_i} \rceil$ rows (corresponding to the clip's display phases). Note that we can have $c_i < n_i$ when the retrieval period of the clip exceeds its length (i.e., $l_i < T_i$). Finally, we let d_i denote the amount of data in a column of C_i's matrix, that is $d_i = \lceil \frac{l_i}{T_i} \rceil \cdot T \cdot r_i$. (Although some columns may actually contain less data than d_i [ÖRS96c], in this chapter, we are ignoring possible optimizations for smaller columns.)

To support periodic retrieval, a clip matrix is stored in column-major form (i.e., data in each column is stored contiguously on disk) and its retrieval is performed *in columns* (i.e., one column per round) with each element handed to a different display phase (Figure 39(b)). Matrix-based allocation reduces the overhead of disk latency per stream since, in each round, it incurs a total overhead of only t_{lat} for $\lceil \frac{l_i}{T_i} \rceil$ streams of C_i, rather than $\lceil \frac{l_i}{T_i} \rceil \cdot t_{lat}$ (using Inequalities (5), (6), and (7)). This means that a single disk using the matrix-based scheme can support the periodic retrieval of $C_1, \ldots, C_k$ provided that the following inequality holds:

$$\sum_{\{C_i\}} (\frac{d_i}{r_{disk}} + t_{lat}) + 2 \cdot t_{seek} \leq T. \qquad (8)$$

The disk bandwidth *effectively utilized* by a clip during a round is the amount of raw disk bandwidth consumed by the clip without accounting for the latency overhead. For C_i, this is exactly $\frac{d_i}{T}$, or, equivalently, $\lceil \frac{l_i}{T_i} \rceil \cdot r_i$.

The matrix-based allocation scheme benefits all three multi-disk data organization strategies considered in this chapter by significantly reducing the latency overhead in each round. For example, in Inequalities (6) and (7) for Clustering and Fine-grained Striping the term t_{lat} is replaced by $t_{lat} \cdot \lceil \frac{l_i}{T_i} \rceil^{-1}$. Thus, we present our results assuming that matrix-based allocation is used. For Fine-grained (Coarse-grained) Striping, this means that the retrieval unit striped (resp. distributed) across the disks of the server is an entire *column* of C_i rather than a single matrix element. However, we should stress that the schemes presented in this chapter are equally applicable to the original data organization methods without the matrix-based scheme optimization.

8.2 EPPV under Clustering

Under a Clustered data organization, the EPPV resource scheduling problem reduces to effectively mapping clip matrices onto the server's disks so that the bandwidth and storage requirements of each matrix are satisfied. That is, Inequalities (6) need to hold for each disk. We address this scheduling problem in two stages. First, we present a solution that considers only the bandwidth requirements of clips (essentially, assuming that each disk has infinite storage capacity). Next, we extend our approach to handle disk storage limitations. We present the first case separately since our results for this case will also prove useful for Fine-grained Striping.

8.2.1 Bandwidth Constraint

We associate two key parameters with each clip:

- A *size*: $\texttt{size}(C_i) = \frac{\frac{d_i}{r_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}}$, that captures the normalized contribution of C_i to the length of a round, or, equivalently, its (normalized) disk bandwidth consumption (see Inequality (8)); and,
- A *value*: $\texttt{value}(C_i) = \lceil \frac{l_i}{T_i} \rceil \cdot r_i$, that corresponds to the bandwidth *effectively utilized* by C_i during a round.

Using these definitions, the problem of maximizing the effectively scheduled disk bandwidth can be formally stated as follows: *Given a collection of clips $C = \{C_1, \ldots, C_N\}$, determine a subset C' of C and a packing of $\{\texttt{size}(C_i) : C_i \in C'\}$ in n_{disk} unit capacity bins such that the total value $\sum_{C_i \in C'} \texttt{value}(C_i)$ is maximized.* This problem is a generalization of the traditional 0/1 knapsack optimization problem [IK75, Law79, Sah75]. Thus, it is clearly $\mathcal{NP}$-hard [5]. Given the intractability of the problem, we present a fast heuristic algorithm (termed PackClips) that combines the value density heuristic rule for the classical knapsack problem [GJ79] with a First-Fit packing rule. Briefly, the main idea is to define the value density of clip C_i as the ratio $p_i = \frac{\texttt{value}(C_i)}{\texttt{size}(C_i)}$ and pack the clips in decreasing order of density into unit capacity bins using a First-Fit rule. The schedulable subset (and the corresponding schedule) is determined by selecting the n_{disk} "most valuable" bins from the final packing. Algorithm PackClips is depicted in Figure 40.

Algorithm PackClips**(C, n_{disk})**

Input: A collection of CM clips $C = \{C_1, \ldots, C_N\}$ and a number of disks n_{disk}.

Output: $C' \subseteq C$ and a packing of C' in n_{disk} unit capacity bins.(Goal: Maximize $\sum_{C_i \in C'} \texttt{value}(C_i)$.)

1. Sort the clips in C in non-increasing order of value density to obtain a list $L =< C_1, \ldots, C_N >$ where $p_i \geq p_{i+1}$. Initialize $\mathsf{load}(B_j) = \texttt{value}(B_j) = 0$, $B_j = \emptyset$, for each bin (i.e., disk) B_j, $j = 1, \ldots, N$.
2. For each clip C_i in L (in that order)
   - 2.1. Let B_j be the first bin (i.e., disk) such that $\mathsf{load}(B_j) + \texttt{size}(C_i) \leq 1$.
   - 2.2. Set $\mathsf{load}(B_j) = \mathsf{load}(B_j) + \texttt{size}(C_i)$, $\texttt{value}(B_j) = \texttt{value}(B_j) + \texttt{value}(C_i)$, $B_j = B_j \cup \{C_i\}$, and $L = L - \{C_i\}$.
3. Let $B_{<i>}$, $i = 1, \ldots, n_{disk}$ be the bins corresponding to the n_{disk} largest value's in the final packing.
   Return $C' = \cup_{i=1}^{n_{disk}} B_{<i>}$. (The packing of C' is defined by the $B_{<i>}$'s.)

Figure 40: Algorithm PackClips

The following theorem provides an upper bound on the worst-case performance ratio of our heuristic.

Theorem 8.2.1 Algorithm PackClips runs in time $O(N(\log N + n_{disk}))$ and is 1/2-approximate;

[5] Note that the traditional knapsack problem remains $\mathcal{NP}$-hard even if the size of each item is equal to its value (i.e., the SUBSET SUM problem [GJ79]) Thus, the fact that $\texttt{size}_1(C_i)$ and $\texttt{value}(C_i)$ are correlated does not affect the hardness of the problem.

that is, if V_{OPT} is the value of the optimal schedulable subset and V_H is the value of the subset returned by PACKCLIPS then $\frac{V_H}{V_{OPT}} \geq \frac{1}{2}$. $\square$

8.2.2 Bandwidth and Storage Constraints

We now extend the PACKCLIPS algorithm to handle the storage capacity constraints imposed by the disks. The idea is to define the size of a clip C_i as a *2-dimensional size vector* $\mathbf{s}_i = [\mathtt{size}_1(C_i), \mathtt{size}_2(C_i)]$ where the first component is the normalized bandwidth consumption of the clip (as defined in the previous section) and the second component is the normalized storage capacity requirement of the clip. More formally,

$$\mathtt{size}_1(C_i) = \frac{\frac{d_i}{r_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}} \qquad \text{and} \qquad \mathtt{size}_2(C_i) = \frac{l_i \cdot r_i}{c_{disk}}.$$

Let $l(\mathbf{v})$ denote the maximum component of a vector $\mathbf{v}$ (i.e., its *length*). The 2-dimensional extension of the PACKCLIPS algorithm is based on defining the value density of a clip as the ratio $p_i = \frac{\mathtt{value}(C_i)}{l(\mathbf{s}_i)}$. The $\mathtt{load}$ of a disk is also a 2-dimensional vector equal to the vector sum of sizes of all clips clustered on that disk, and the condition in step 2.1 of PACKCLIPS becomes: $l(\mathsf{load}(B_j) + \mathbf{s}_i) \leq 1$. That is, we require that *both* the bandwidth and storage load on each disk do not exceed the disk's capacities. For our worst-case analysis of the 2-dimensional PACKCLIPS algorithm we also assume that the storage requirements of a clip never exceed one half of a disk's storage capacity, that is, $\mathtt{size}_2(C_i) \leq \frac{1}{2}$. This is a reasonable assumption since current disk storage capacities are in the order of several gigabytes. The following theorem shows that the extra dimension degrades the worst-case performance guarantee of our heuristic by a factor of two.

Theorem 8.2.2 Assuming that the storage requirements of any clip are always less than or equal to one half of a disk's storage capacity, the 2-dimensional PACKCLIPS heuristic is 1/4-approximate; that is, if V_{OPT} is the value of the optimal schedulable subset and V_H is the value of the subset returned by PACKCLIPS then $\frac{V_H}{V_{OPT}} \geq \frac{1}{4}$. $\square$

We have already noted that Clustering can lead to disk storage and bandwidth fragmentation, and this is clearly demonstrated in the rather discouraging worst-case bound of Theorem 8.2.2 – for "bad" lists of clips, PACKCLIPS may be able to utilize only as little as one fourth of the raw server capacity. Since storage fragmentation is not an issue when striping is used, we can effectively ignore storage constraints by assuming that the aggregate storage requirements of the clips to be scheduled do not exceed the storage capacity of the server; that is, we assume that $\sum_i l_i \cdot r_i \leq n_{disk} \cdot c_{disk}$ in the description of our striping-based schemes.

8.3 EPPV under Fine-grained Striping

In the Fine-grained Striping (FGS) scheme, each column of the clip matrix is declustered across all n_{disk} disks of the server (Figure 41(a)). This implies that each clip being retrieved imposes a constant load per round on *all* disks in the server, since each disk is responsible for retrieving $\frac{1}{n_{disk}}$ of the clip's column in each round. Thus, the following condition must be satisfied on each disk:

$$\sum_{i=1}^{N} \frac{d_i}{r_{disk} \cdot n_{disk}} + N \cdot t_{lat} \leq T - 2 \cdot t_{seek}. \qquad (9)$$

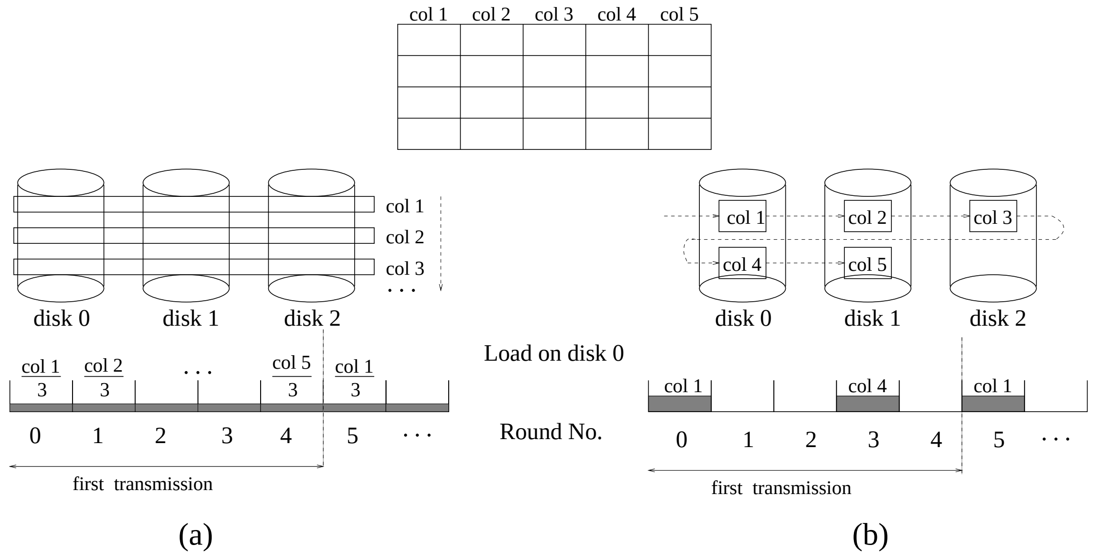


Figure 41: (a) Fine-grained Striping. (b) Coarse-grained Striping.

To ensure continuous retrieval, all disks in the system must satisfy the same condition (namely, Inequality (9)). Consequently, the problem of maximizing the effectively scheduled bandwidth clips under Fine-grained Striping corresponds to a traditional, single-bin, 0/1 knapsack problem with (one-dimensional) clip sizes $\texttt{size}(C_i) = \frac{\frac{d_i}{r_{disk} \cdot n_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}}$ (from Ineq. 9), and values $\texttt{value}(C_i) = \left\lceil \frac{l_i}{T_i} \right\rceil \cdot r_i$ (as in Section 8.2). This is clearly a much simpler version of the knapsack model developed for Clustering and traditional knapsack heuristics can be used to provide near-optimal solutions [IK75, Law79, Sah75]. In fact, our PACKCLIPS algorithm (with number of bins/disks equal to 1) readily provides a 1/2-approximate heuristic for the problem.

We should once again stress that, despite its conceptual and algorithmic simplicity, Fine-grained Striping suffers from excessive disk latency overheads that severely limit the scalability of the scheme. This fact is analytically shown in Inequality (9)) and clearly indicated in our experimental results.

8.4 EPPV under Coarse-grained Striping

In the Coarse-grained Striping (CGS) scheme, the columns of a clip matrix are mapped to (and, retrieved from) individual disks in a round-robin manner (Figure 41(b)). Consider the retrieval of a clip matrix C_i from a particular disk in the array. By virtue of the round-robin placement, during each transmission of C_i, a column of C_i must be retrieved from that disk periodically, at intervals of n_{disk} rounds. From Formula (8), each such retrieval requires a fraction $\frac{\frac{d_i}{r_{disk}}+t_{lat}}{T-2\cdot t_{seek}}$ of the disk's bandwidth. Furthermore, to support EPPV service, the transmissions of C_i are themselves periodic with a period $T_i = n_i \cdot T$.

Thus, the retrieval of a clip matrix C_i from a specific disk in the array can be seen as a collection of *periodic real-time tasks* [LL73] with period T_i (i.e., the clip's transmissions), where each task consists of a collection of *subtasks* that are $n_{disk} \cdot T$ time units apart (i.e., column retrievals within a transmission). Moreover, the computation time of each such subtask is $\frac{d_i}{r_{disk}} + t_{lat}$. An example of such a task is shown in Figure 41(b). Note that the maximum number of subtasks mapped to a disk by C_i equals $\left\lceil \frac{c_i}{n_{disk}} \right\rceil$. ($c_i$ is the number of columns in C_i.) This number may actually be smaller for some disks in the array. However, in order to provide deterministic service guarantees for all disks, we consider only this worst-case number of subtasks in our scheduling formulation.

We say that two (or more) clip retrievals *collide* during a round if they are all reading data off the same disk. Collisions play a crucial role in our scheduling problem. Our algorithms need to ensure that whenever multiple retrievals collide during a round, their total bandwidth requirements do not exceed the capacity of the disk. Before addressing the scheduling problems associated with this general model of periodic tasks, we briefly review two special cases that essentially correspond to the best and worst case workloads for Coarse-grained Striping.

- $\underline{c_i = n_i = \textbf{multiple}(n_{disk})}$ **for all** $\underline{i}$**.** In this case, the retrieval of C_i from a particular disk corresponds to a periodic real-time task with period $n_{disk} \cdot T$ and computation time $\frac{d_i}{r_{disk}} + t_{lat}$. Maximizing the effectively scheduled bandwidth can again be formulated as a generalized knapsack problem which in fact is identical to the problem defined in Section 8.2.1, but with a slightly different interpretation of terms: the n_{disk} unit-capacity bins now correspond to rounds of length T and the items correspond to retrievals of clip columns. The main advantage of CGS over Clustering in this case, is its ability to equally distribute the bandwidth and storage load across all disks. Compared to FGS, the main advantage of CGS is the reduced latency penalty for each clip, which implies better scalability. In fact, a simple analysis shows that, for this special case, CGS is guaranteed to outperform FGS for large disk arrays.

It is important to note that it is not always feasible to reduce the general scheduling problem to this special case, e.g., by "padding" clip lengths or periods so that the equality $c_i = n_i$ =multiple(n_{disk}) is satisfied. For example, if the length of a clip is significantly smaller than its period then $c_i << n_i$ and padding the clip to the length of its period is clearly not an effective solution. For example, consider a clip with $r_i = 1.5$Mbps, $l_i =$ 5min, $T_i =$ 100min, and a system with $T =$ 1sec and $n_{disk} = 10$. Padding the clip's length to reach its period implies that 1GB of storage is wasted per clip.

- **$\gcd(n_i, n_j) = 1$ for all $i \neq j$ and $c_i \geq n_{disk}$ for all i.**[6] In this case, using the *Chinese Remainder Theorem* [Knu81], we can prove the following lemma.

  Lemma 8.4.1 Assume that CGS is used for a collection of clips such that $\gcd(n_i, n_j) = 1$ for all $i \neq j$ and $c_i \geq n_{disk}$ for all i. Then, during any time interval of length $n_1 \cdots n_N \cdot T$ there exists a round at which the retrievals for *all* clips collide. □

  Thus, retrieval periods that are pairwise relatively prime correspond to a worst-case scenario for CGS. That is, there exists a round in which $\frac{d_i}{r_{disk}} + t_{lat}$ units of time need to be allocated for each clip C_i. The existence of such worst-case collisions means that CGS has to be overly conservative and is typically outperformed by FGS.

Consider the case of arbitrary retrieval periods. Using the *Generalized* Chinese Remainder Theorem [Knu81], we can show the following result.

Lemma 8.4.2 Consider two clips C_1 and C_2, and let $\alpha_i = \min\{ \left\lceil \frac{c_i}{n_{disk}} \right\rceil , \frac{\gcd(n_1,n_2)}{\gcd(n_1,n_2,n_{disk})} \}$, $i = 1, 2$. The retrieval of C_1 and C_2 can be scheduled without collisions *if and only if* $\alpha_1 + \alpha_2 \leq \gcd(n_1, n_2)$. □

Lemma 8.4.2 identifies a necessary and sufficient condition for the *collision-free* scheduling (or, *mergeability* [YSB+89]) of two clip retrieval patterns. Our result extends the result of Yu et al. [YSB+89] on merging two simple periodic patterns to the case of periodic tasks consisting of equidistant subtasks. Furthermore, Lemma 8.4.2 can be generalized to any number of clips if their periods can be expressed as $n_i = k \cdot m_i$ for all i, where m_i and m_j are relatively prime for all $i \neq j$. (Note that for two clips this condition is obviously true with $k = \gcd(n_1, n_2)$.)

Lemma 8.4.3 Consider a collection of clips $C = \{C_1, \ldots, C_N\}$, with retrieval periods $n_i = k \cdot m_i$, for all i, where $\gcd(m_i, m_j) = 1$ for $i \neq j$. Let $\alpha_i = \min\{ \left\lceil \frac{c_i}{n_{disk}} \right\rceil , \frac{k}{\gcd(k,n_{disk})} \}$. The retrieval of C can be scheduled without collisions *if and only if* $\sum_{i=1}^{N} \alpha_i \leq k$. □

[6] The gcd() function returns the *greatest common divisor* of a set of integers.

Unfortunately, Lemma 8.4.2 cannot be extended to the general case of multiple clips with arbitrary periods. In fact, in Section 8.5, we will show that deciding the existence of a collision-free schedule for the general case is $\mathcal{NP}$-complete in the strong sense. Thus, no efficient necessary and sufficient conditions are likely to exist. The condition described in Lemma 8.4.2 can easily be shown to be sufficient for no collisions in the general case. However, it is not necessary, as the following example indicates.

Example 1: Consider three clips with periods $n_1 = 4$, $n_2 = 6$, $n_3 = 8$ and let $n_{disk} = 4$. This set can be scheduled with no collisions, by initiating the retrieval of C_1, C_2, C_3 at rounds 0, 1, and 2, respectively. However, the inequality in Lemma 8.4.2 (extended for three clips) fails to hold, since $\gcd(n_1, n_2, n_3) = 2 < \sum_{i=1}^{3} \alpha_i = 3$.

8.5 The Scheduling Tree Structure

In the previous section, we identified the scheduling problem that arises when supporting EPPV service under CGS and examined some special cases. In this section we address the general problem. We first consider a model of simple periodic real-time tasks and show that deciding the existence of a collision-free schedule is equivalent to *Periodic Maintenance* [BRTV90, WL83], a problem known to be intractable. Motivated from this result, we define the novel concept of a *scheduling tree* and discuss its application in a heuristic algorithm for Periodic Maintenance. We then show how the scheduling tree structure can handle the more complex model of periodic tasks identified in Section 8.4.

8.5.1 Periodic Maintenance Scheduling

The *k-server Periodic Maintenance Scheduling Problem (k-PMSP)* [BRTV90] is a special case of the problem of scheduling simple periodic tasks in a hard real-time environment. Briefly, the k-PMSP decision problem can be stated as follows: *Let $C = \{C_1, \ldots, C_N\}$ be a set of periodic tasks with corresponding periods $P = \{n_1, \ldots, n_N\}$, where each n_i is a positive integer. Is there a mapping of the the tasks in C to positive integer time slots such that successive occurrences of C_i are* **exactly** *n_i time slots apart and no more than k tasks ever collide in a slot?* Note that if u_i is the index of the first occurrence of C_i in a schedule for P then the (multi)set of starting time slots $\{u_1, \ldots, u_N\}$ uniquely determines the schedule, since C_i occurs at all slots $u_i + j \cdot n_i$, $j \geq 0$.

Baruah et al. [BRTV90] have shown that for any fixed integer $k \geq 1$, k-PMSP is $\mathcal{NP}$-complete in the strong sense. Consequently, given a collection of simple periodic tasks with periods P, determining the existence of a collision-free schedule is intractable (i.e., it is equivalent to 1-PMSP). The existence of a *scheduling tree* structure (as described below) that contains all the periods in P, guarantees the existence of a collision-free schedule. Furthermore, the starting time slot for each task can be determined from the scheduling tree[7].

Definition 8.5.1 A *scheduling tree* is a tree structure consisting of nodes and edges with integer weights, where:

1. Each internal node of weight w can have *at most* w outgoing edges, each of which has a distinct weight in $\{0, 1, \ldots, w-1\}$; and,

2. Each leaf node represents a period n_i such that n_i is equal to the product of weights of the leaf's ancestor nodes.

□

We define the *level of a node (*or, *edge)* as the number of its proper ancestor nodes. Thus, the level of the tree's root is 0 and the level of all edges emanating from the root is 1. For any node n, let $w(n)$ and $e(n)$ denote the weight and the number of edges of n, respectively. Also, let $\texttt{ancestor\_node}_j(n)$ represent the weight of the ancestor node of n at level j, and let $\texttt{ancestor\_edge}_j(n)$ denote the weight of the ancestor edge of n at level j, where $j \leq level(n)$. Finally, define $\pi_j(n) = \prod_{i=0}^{j} \texttt{ancestor\_node}_i(n)$ for $0 \leq j \leq level(n)$.

Consider a leaf node for period n_i located at level l. The first slot u_i in which the corresponding task is scheduled is defined from the scheduling tree structure as follows:

$$u_i = \texttt{ancestor\_edge}_1(n_i) + \sum_{j=2}^{l} \texttt{ancestor\_edge}_j(n_i) \cdot \pi_{j-2}(n_i). \qquad (10)$$

Some intuition for the scheduling tree structure and the above formula is provided in Figure 42. The basic idea is that all tasks in a subtree rooted at some edge emanating from node n at level l will utilize time slot numbers that are congruent to $i \pmod{\pi_l(n)}$, where i is a unique number between 0 and $\pi_l(n) - 1$. Satisfying this invariant recursively at every internal node ensures the avoidance of collisions.

Note that the existence of a scheduling tree for a set of periods P is only a *sufficient condition* for the existence of a collision-free schedule. For example, the periods 6, 10, and 15 are schedulable using start times of 0, 1, and 2, respectively, although no scheduling tree can be built (since $\gcd(\{6, 10, 15\}) = 1$). However, using the *Generalized Chinese Remainder Theorem* it is straightforward to show that the existence of a *scheduling forest*, as defined below, is both necessary and sufficient for the existence a collision-free schedule.

[7]To the best of our knowledge, no similar notion of tree structure for periodic task scheduling has been proposed in the real-time scheduling literature [SR93].

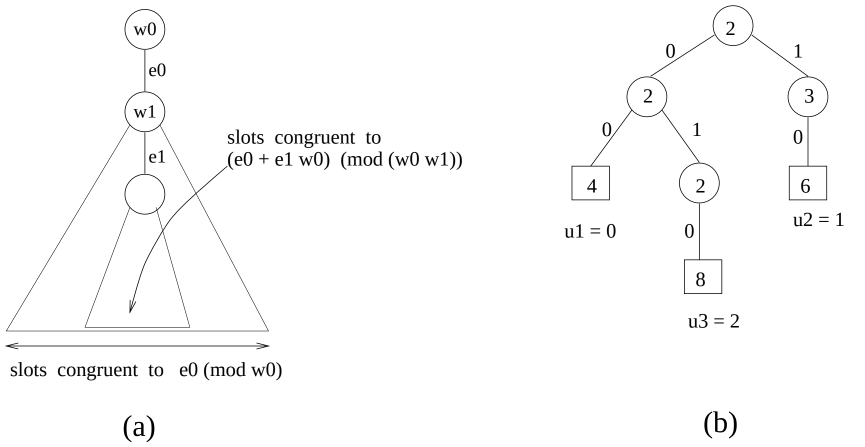


Figure 42: (a) The scheduling tree structure. (b) A tree for the set of tasks in Example 1.

Definition 8.5.2 Let $\Gamma_1, \ldots, \Gamma_k$ be scheduling trees for $P_1, \ldots, P_k$, where $P_1, \ldots, P_k$ is a partitioning of of the periods in P. The trees Γ_i and Γ_j are *consistent* if and only if for each $n_m \in P_i$ and $n_l \in P_j$ we have $u_m \not\equiv u_l \pmod{\gcd(n_m, n_l)}$. A *scheduling forest* for P is a collection of pairwise consistent scheduling trees for some partition of P. □

Corollary 8.5.1 Determining whether there exists a scheduling forest for P is equivalent to 1-PMSP, and, thus, it is $\mathcal{NP}$-complete in the strong sense. □

Given the above intractability result, we present a heuristic algorithm for constructing scheduling trees for a given (multi)set of periods. Our algorithm is based on identifying and incrementally maintaining *candidate nodes* for scheduling incoming periods.

Definition 8.5.3 An *internal* node n at level l is *candidate for period* n_i if and only if $\pi_{l-1}(n) | n_i$ and $\gcd(w(n), \frac{n_i}{\pi_{l-1}(n)}) \geq \frac{w(n)}{w(n)-e(n)}$. □

A period n_i can be scheduled under any candidate node n in a scheduling tree. There are two possible cases:

- **If** $\underline{\pi_l(n) | n_i}$ then the condition in Definition 8.5.3 guarantees that n has at least one free edge at which n_i can be placed (Figure 43(a)).
- **If** $\underline{\pi_l(n) \not| n_i}$ then, in order to accommodate n_i under node n, n must be *split* so that the defining properties of the scheduling tree structure are kept intact.

  This is done as follows. Let $d = \gcd(w(n), \frac{n_i}{\pi_{l-1}(n)})$. Node n is split into a parent node with weight d and child nodes with weight $\frac{w(n)}{d}$, with the original children of n divided

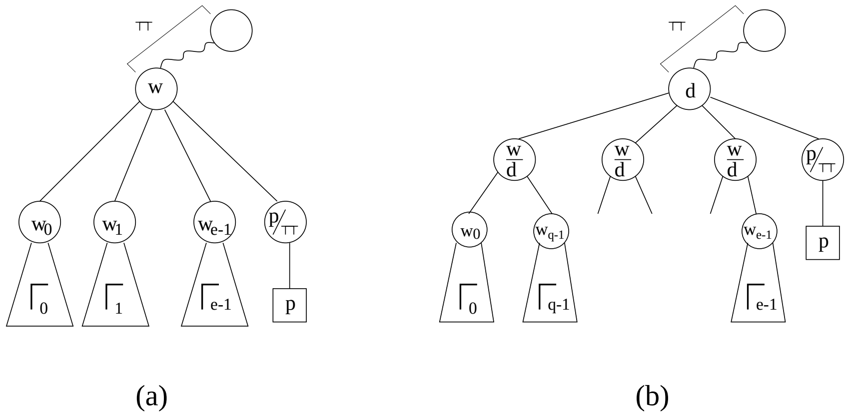


Figure 43: (a) Placing a period p under a scheduling tree node without splitting. (b) Period placement when the node is split.

among the new child nodes, as shown in Figure 43(b); that is, the first batch of $\frac{w(n)}{d}$ children of n are placed under the first child node, and so on. It is easy to see that this splitting maintains the properties of the structure. Furthermore, the condition in Definition 8.5.3 guarantees that the newly created parent node will have at least one free edge for scheduling n_i.

The set of candidate nodes for each period to be scheduled can be maintained efficiently, in an incremental manner. The observation here is that when a new period n_i is scheduled, all remaining periods only have to check a maximum of three nodes, namely the two closest ancestors of the leaf for n_i and, if a split occurred, the last child node created in the split, for possible inclusion or exclusion from their candidate sets.

As in Section 8.2, we assume each task is associated with a value and we aim to maximize the cumulative value of a schedule. The basic idea of our heuristic (termed BUILDTREE) is to build the scheduling tree incrementally in a greedy fashion, scanning the tasks in non-increasing order of value and placing each period n_i in that candidate node M that implies the minimum value loss among all possible candidates. This loss is calculated as the total value of all periods whose candidate sets become empty after the placement of n_i under M. Ties are always broken in favor of those candidate nodes that are located at higher levels (i.e., closer to the leaves), while ties at the same level are broken using the postorder node numbers (i.e., left-to-right order). When a period is scheduled in Γ, the candidate node sets for all remaining periods are updated (in an incremental fashion) and the algorithm continues with the next task/period (with at least one candidate in Γ). Algorithm BUILDTREE is depicted in Figure 44.

Let N be the number of tasks in C. The number of internal nodes in a scheduling tree is always going to be $O(N)$. To see this, note that an internal node will always have at least two

Algorithm BuildTree(**C,** `value`)

Input: A set of simple periodic tasks $C = \{C_1, \ldots, C_N\}$ with corresponding periods $P = \{n_1, \ldots, n_N\}$, and a `value`() function assigning a value to each C_i.

Output: A scheduling tree Γ for a subset C' of C. (Goal: Maximize $\sum_{C_i \in C'} \texttt{value}(C_i)$.)

1. Sort the tasks in C in non-increasing order of `value` to obtain a list $L =< C_1, C_2, \ldots, C_N >$, where $\texttt{value}(C_i) \geq \texttt{value}(C_{i+1})$. Initially, Γ consists of a root node with a weight equal to n_1.
2. For each periodic task C_i in L (in that order)

   2.1. Let $cand(n_i, \Gamma)$ be the set of candidate nodes for n_i in Γ. (Note that this set is maintained incrementally as the tree is built.)

   2.2. For each $n \in cand(n_i, \Gamma)$, let $\Gamma \cup \{n_i\}_n$ denote the tree that results when n_i is placed under node n in Γ. Let $\texttt{loss}(n) = \{C_j \in L - \{C_i\} \mid cand(\Gamma \cup \{n_i\}_n) = \emptyset\}$ and $\texttt{value}(\texttt{loss}(n)) = \sum_{C_j \in \texttt{loss}(n)} \texttt{value}(C_j)$.

   2.3. Place n_i under the candidate node M such that $\texttt{value}(\texttt{loss}(M)) = \min_{n \in cand(n_i, \Gamma)}\{\texttt{value}(\texttt{loss}(n))\}$. (Ties are broken in favor of nodes at higher levels.) If necessary, node M is split.

   2.4. Set $\Gamma = \Gamma \cup \{n_i\}_M$, $L = L - \texttt{loss}(M)$.

   2.5. For each task $C_j \in L$, update the candidate node set $cand(n_i, \Gamma)$.

Figure 44: Algorithm BuildTree

children, with the only possible exception being the rightmost one or two new nodes created during the insertion of a new period (depending on whether splitting was used, see Figure 43). Since the number of insertions is at most N, it follows that the number of internal nodes is $O(N)$. Based on this fact, it is easy to show that BuildTree runs in time $O(N^3)$.

Example 2: Consider the list of periods $< n_1 = 2,\ n_2 = 12,\ n_3 = 30 >$ (sorted in non-increasing order of `value`). Figure 45 illustrates the step-by-step construction of the scheduling tree using BuildTree. Note that period n_3 splits the node with weight 6 into two nodes with weights 3 and 2.

8.5.2 Scheduling Equidistant Subtasks

In Section 8.4, we identified a clip retrieval under Coarse-grained Striping as a periodic real-time task C_i with period $n_i = \frac{T_i}{T}$ (in rounds) that consists of a collection of $\left\lceil \frac{c_i}{n_{disk}} \right\rceil$ subtasks that need to be scheduled n_{disk} rounds apart. The basic observation here is that all the subtasks of C_i are themselves periodic with period n_i, so the techniques of the previous section can be used for each individual subtask. However, the scheduling algorithm also needs to ensure that *all*

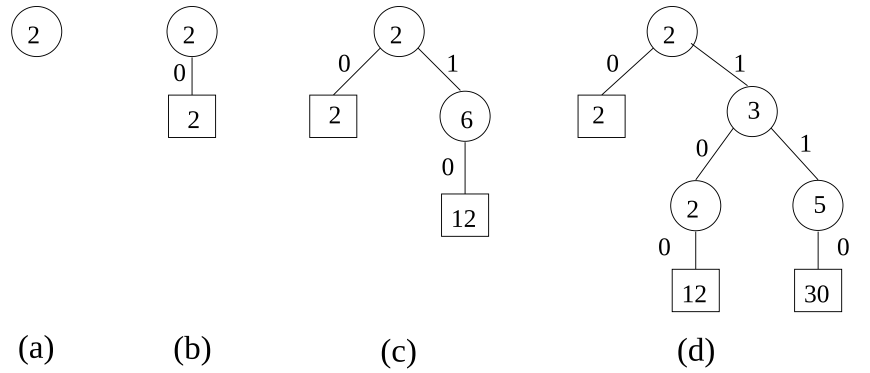


Figure 45: Construction of a scheduling tree for the set of tasks in Example 2.

the subtasks are scheduled together, using time slots (i.e., rounds) placed regularly at intervals of n_{disk}. In this section, we propose heuristic methods for building a scheduling tree in this generalized setting.

An important requirement of this more general task model is that the insertion of new periods cannot be allowed to distort the relative placement of subtasks already in the tree. The splitting mechanism described in the previous section for simple periodic tasks does not satisfy this requirement, since it can alter the starting time slots for all subtasks located under the split node. We describe a new rule for splitting nodes without modifying the retrieval schedule for subtasks already in the tree. The idea is to use a different method for "batching" the children of the node being split, so that the starting time slots for all leaf nodes (as specified by Equation (10)) remain unchanged. This new splitting rule is as follows: *If the node n is split to give a new parent node with weight d, then place at edge i of the new node ($i = 0, \ldots, d-1$) all the children of the old node n whose parent edge weight was congruent to $i \pmod{d}$.* Our claim that retrieval schedules are kept intact under this rule is a direct consequence of Equation (10).

Example 3: Figure 46(a) illustrates a scheduling tree with two tasks with periods $n_1 = 6$, $n_2 = 6$ assigned to slots 0 and 1. Figure 46(b) depicts the scheduling tree after a third task with period $n_3 = 15$ is inserted. Although there is enough capacity for both n_1 and n_2 in the subtree connected to the root with edge 0, the new split forces n_2 to be placed in the subtree connected to the root with edge 1.

In this setting, the notion of a candidate node is defined as follows.

Definition 8.5.4 An internal node n at level l is *candidate for period n_i* if and only if $\pi_{l-1}(n) | n_i$ and there exists an $i \in \{0, \ldots, d-1\}$ such that all edges of n with weights congruent to $i \pmod{d}$ are free, where $d = \gcd(w(n), \frac{n_i}{\pi_{l-1}(n)})$. □

However, under our generalized model of periodic tasks, a candidate node for n_i can only

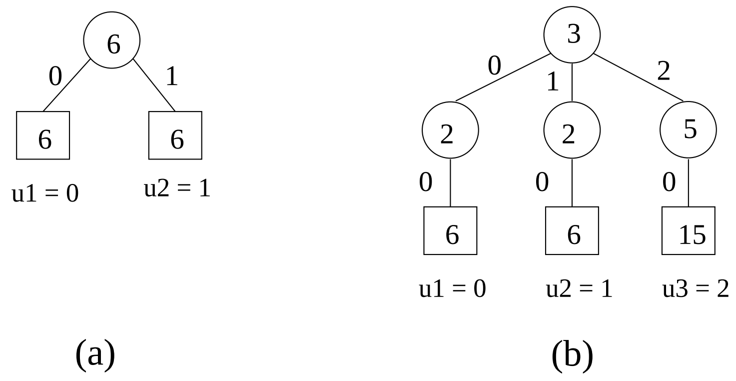


Figure 46: Illustration of the new splitting rule.

accommodate a subtask of C_i. This is clearly not sufficient for the entire task. The temporal dependency among the subtasks of C_i means that our scheduling tree scheme must make sure that *all* the subtasks of C_i are placed in the tree at distances of n_{disk}.

One way to deal with this situation is to maintain candidate nodes for subtasks based on Definition 8.5.4, and use a simple predicate based on Equation (10), for checking the availability of specific time slots in the scheduling tree. The scheduling of C_i can then be handled as follows. Select a candidate node for n_i and a time slot u_i for n_i under this candidate. Place the first subtask of C_i in u_i and call the predicate repeatedly to check if n_i can be scheduled in slot $u_i + j \cdot n_{disk}$, for $j = 1, \ldots, \left\lceil \frac{c_i}{n_{disk}} \right\rceil$. If the predicate succeeds for all j, then C_i is scheduled starting at u_i. Otherwise, the algorithm can try another potential starting slot u_i. Algorithm FreeSlot, depicted in Figure 47, is a predicate for checking the availability of a specific time slot u under a candidate node n for period n_i. The correctness of the algorithm follows from the fact that Equation (10) can be rewritten in nested form as follows:

$$\begin{aligned} u_i = \texttt{ancestor\_edge}_1(n_i) \quad &+ \quad \texttt{ancestor\_node}_0(n_i) \cdot (\texttt{ancestor\_edge}_2(n_i) + \texttt{ancestor\_node}_1(n_i) \cdot (\ldots \\ &\ldots \quad (\texttt{ancestor\_edge}_{l-1}(n_i) + \texttt{ancestor\_node}_{l-2}(n_i) \cdot \texttt{ancestor\_edge}_l(n_i)) \ldots). \end{aligned}$$

A problem with the approach outline above is that even if the number of starting slots tried for C_i is restricted to a constant, scheduling each subtask individually yields pseudo-polynomial time complexity. This is because the number of scheduling operations in a trial will be $O(\frac{c_i}{n_{disk}})$, where $c_i = \min\{n_i, \frac{l_i}{T}\}$ is part of the problem input.

We propose a polynomial time heuristic algorithm for the problem. To simplify the presentation, we assume that every period n_i is a multiple of n_{disk}. Although it is possible to extend our heuristic to handle general periods, we believe that this assumption is not very restrictive in practice. This is because we typically expect round lengths T to be in the area

Algorithm FreeSlot(n, n_i, u)

Input: A scheduling tree node n, a period n_i, and a specific time slot u.

Output: TRUE iff n_i can be scheduled in slot u under node n; FALSE otherwise.

1. If n is *not* a candidate for n_i then return(FALSE).
2. Let $l = level(n)$ and let $t = \texttt{ancestor\_edge}_1(n) + \sum_{j=2}^{l} \texttt{ancestor\_edge}_j(n) \cdot \pi_{j-2}(n)$.
3. If $\pi_{l-1}(n)$ does not divide $u - t$ then return(FALSE);
   Else let $u = \frac{u-t}{\pi_{l-1}(n)}$.
4. Let $d = \gcd(n_i, w(n))$ and $e = u \bmod d$.
5. If all edges of n labeled $k \cdot d + e$, for $k = 0, \ldots, \frac{w(n)}{d} - 1$ are free then return(TRUE);
   Else return(FALSE).

Figure 47: Algorithm FreeSlot

of a few seconds and periods T_i to be multiples of some number of minutes (e.g., 5, 10, 30, or 60 minutes). Therefore, it is realistic to assume the smallest period in the system can be selected to be a multiple of n_{disk}. Our goal is to devise a method that ensures that if the *first subtask* of a task C_i does not collide with the first subtask of any other task in the tree, then no other combination of subtasks can cause a collision to occur. This means that once the first subtask of C_i is placed in the scheduling tree there is no need to check the rest of C_i's subtasks individually.

Our algorithm sets the weight of the root of the scheduling tree to n_{disk}. (This is possible since the n_i's are multiples of n_{disk}.) By Equation (10), this implies that consecutive subtasks of a task will require consecutive edges emanating from nodes at the first level (i.e., the direct descendants of the root). The basic idea of our method is to make sure that when the first subtask of a task is placed at a leaf node, a number of consecutive edges of the first-level ancestor node of that leaf are *disabled*, so that the slots under those edges cannot be used by the first subtask of any future task. By our previous observation, $s_i - 1 = \lceil \frac{c_i}{n_{disk}} \rceil - 1$ consecutive edges of the first-level ancestor of the leaf for n_i must be disabled, starting with the right neighbor of the edge under which that leaf resides. (s_i is the number of subtasks of C_i.) This "edge disabling" is implemented by maintaining an integer *distance* for each edge e emanating from a first-level node that is equal to the number of consecutive neighbors of e that have been disabled. Our placement algorithm has to maintain two invariants. First, the distance of an edge e of a first-level node is always equal to $\max_{C_i}\{s_i\} - 1$, where the max is taken over all tasks placed under e in the tree. Second, the sum of the weight of an edge e of a first-level node n and its distance is always less than the weight of n (so that the defining properties of the tree are maintained). Based on the above scheme, we can define the notion of a candidate node as follows.

Definition 8.5.5 Let n be an internal node at level l. Let n_i be a period and define $d = \gcd(w(n), \frac{n_i}{\pi_{l-1}(n)})$. Node n is *candidate for period* n_i if and only if $\pi_{l-1}(n)|n_i$ and the following conditions hold:

1. If n is the root node, n has a free edge.
2. If $level(n) = 1$, there exists an $i \in \{0, \ldots, d-1\}$ such that all (non-disabled) edges of n whose sum of weight plus distance is congruent to $(i + j) \pmod{d}$, for $0 \leq j < s_i$, are free.
3. If $level(n) \geq 2$,

   3.1. there exists an $i \in \{0, \ldots, d-1\}$ such that all edges of n with weight congruent to $i \pmod{d}$ are free; and,

   3.2. $s_i - 1 + \texttt{ancestor\_edge}_2(n) < \texttt{ancestor\_node}_1(n)$ and $s_i + \texttt{ancestor\_edge}_2(n)$ is less than or equal to the weight of the (non-disabled) edge following $\texttt{ancestor\_edge}_2(n)$, if there is such an edge.

□

Note that clause 2 ensures that edge distances are maintained when first-level nodes are split. Based on the above definition of candidate nodes, Algorithm BuildEquidTree (shown in Figure 48) can be used to construct a scheduling tree in polynomial time.

Example 4: Consider three tasks C_1, C_2 and C_3 with $s_1, s_2, s_3 = 2, 1, 3$, $n_1, n_2, n_3 = 12, 18, 10$, and $n_{disk} = 2$. Figure 49 illustrates the three states of the scheduling tree after placing tasks C_1, C_2, and C_3 respectively.

8.5.3 Handling Slots with Multi-Task Capacities

An interesting property of the scheduling tree formulation is that it can easily be extended to handle time slots that can fit more than one subtask (i.e., can allow for some tasks to collide). As we saw in Section 8.4, this is exactly the case for the rounds of EPPV retrieval under CGS. Using the notation of Section 8.2, we can think of the subtasks of C_i as items of size $\texttt{size}(C_i) \leq 1$ (i.e., the fraction of disk bandwidth required for retrieving one column

Algorithm BUILDEQUIDTREE

Input: A set of periodic tasks $C = \{C_1, \ldots, C_N\}$ with corresponding periods $P = \{n_1, \ldots, n_N\}$ and a `value`() function assigning a value to each C_i. Each task consists of subtasks placed at intervals of n_{disk}.

Output: A scheduling tree Γ for a subset C' of C. (Goal: Maximize $\sum_{C_i \in C'} \texttt{value}(C_i)$.)

1. Sort the tasks in C in non-increasing order of `value` to obtain a list $L = < C_1, C_2, \ldots, C_N >$, where $\texttt{value}(C_i) \geq \texttt{value}(C_{i+1})$. Initially, Γ consists of a root node with a weight equal to n_{disk}.
2. For each task C_i in L (in that order)
   - 2.1. Select a candidate node n for n_i in Γ. (Ties are broken in favor of nodes at higher levels).
   - 2.2. If $w(n) \not| n_i$, split n.
   - 2.3. Schedule the first subtask of C_i under n. (Ties are broken in favor of edges with smaller weights.)
   - 2.4. Let d be the distance of the ancestor edge at the first level of the leaf corresponding to n_i. Set the distance of this edge to $\max\{d, s_i - 1\}$.

Figure 48: Algorithm BUILDEQUIDTREE

of clip C_i) that are placed in unit capacity time slots. In this more general setting, a time slot can accommodate multiple tasks as long as their total `size` does not exceed one. Note that this problem is a generalization of the k-server Periodic Maintenance Scheduling Problem (k-PMSP), where all items are assumed to be of the same size (i.e., $\frac{1}{k}$th of the capacity).

The problem can be visualized as a collection of unit capacity bins (i.e., time slots) located at the leaves of a scheduling tree, whose structure determines the eligible bins for each task's subtasks (based on their period). With respect to our previous model of tasks, the main difference is that since slots can now accommodate multiple retrievals it is possible for a leaf node that is already occupied to be a candidate for a period. Hence, the basic idea for extending our schemes to this case is to keep track of the available slot space at each leaf node and allow leaf nodes to be shared by tasks. Thus, our notion of candidate nodes can simply be extended as follows.

Definition 8.5.6 Let n be a leaf node for of a scheduling tree Γ corresponding to period p. Also, let $S(n)$ denote the collection of tasks (with period p) mapped to n. The *load of leaf* n is defined as: $\texttt{load}(n) = \sum_{C_i \in S(n)} \texttt{size}(C_i)$. □

Definition 8.5.7 A node n at level l is *candidate for a task of* C_i (with period n_i) if and only if:

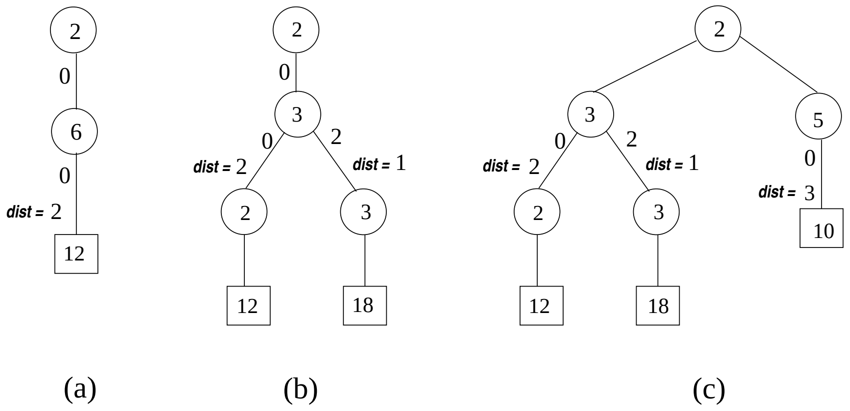


Figure 49: Scheduling equidistant subtasks with edge disabling.

1. n is internal, conditions in Definition 8.5.4 hold, or

2. n is external (leaf node) corresponding to n_i (i.e., $\pi_l(n) = n_i$), and $\texttt{load}(n)+\texttt{size}(C_i) \leq 1$.

□

With these extensions, it is easy to see that our BuildEquidTree algorithm can be used without modification to produce a scheduling tree for the multi-task capacity case.

8.6 Combining Multiple Scheduling Trees

To construct forests of multiple non-colliding scheduling trees, trees already built can be used to restrict task placement in the tree under construction. By the *Generalized Chinese Remainder Theorem*, the scheduling algorithm needs to ensure that each subtask of task C_i is assigned a slot u_i such that $u_i \not\equiv u_j \pmod{\gcd(n_i, n_j)}$ for any subtask of any task C_j that is scheduled in slot u_j in a previous tree within the same forest. This obviously is a very expensive method and efficient heuristics for constructing scheduling forests still elude our efforts. In this section, however, we provide a general packing-based scheme that can be used for combining independently built scheduling forests. Of course, for our purposes, a forest can always consist of a single tree. Our goal is to improve the utilization of scheduling slots that can accommodate multiple tasks.

Given a collection of tasks, scheduling forests are constructed until each task is assigned a time slot. We know that no pair of tasks within a forest will collide at any slot except for tasks with the same period that are assigned to the same leaf node as described in Section 8.5.3. A simple conservative approach is to assume a worst-case collision across forests. That is, we define the size of a forest as $\texttt{size}(F_i) = \max_{n_j \in F_i}\{\texttt{load}(n_j)\}$ where n_j is any leaf node in F_i, and the load of a leaf node is as in Definition 8.5.6. Further, a forest F_i has a value: $\texttt{value}(F_i) = \sum_{C_j \in F_i} \texttt{value}(C_j)$. Thus, under the assumption of a worst-case collision, the problem of maximizing the total scheduled value for a collection of forests is a traditional 0/1 knapsack optimization problem. A packing-based heuristic like PackClips can be used to provide an approximate solution.

In some cases, the worst-case collision assumption across forests may be unnecessarily restrictive. For example, consider two scheduling trees Γ_1 and Γ_2 that are constructed independently. Let e_1 be an edge emanating from the root node n_1 of Γ_1 and e_2 be an edge emanating from the root node n_2 of Γ_2. If $e_1 \bmod (\gcd(n_1, n_2)) \neq e_2 \bmod (\gcd(n_1, n_2))$ holds, then the tasks scheduled in the subtrees rooted at e_1 and e_2 can never collide. Using such observations, we can devise more clever packing-based strategies for combining forests. As an example, consider the following strategy. Assume a collection of independently built scheduling forests $\{F_i\}$. Let d be the gcd of all the root nodes of all the trees in every forest. Let $F_i(j)$, $0 \leq j < d$, denote the collection of all subtrees rooted at a first level node (i.e., a child of the root) in each tree within forest F_i, such that the weight of the edge connecting the subtree to the root is congruent to $j \pmod d$. As previously, we define the size and value of $F_i(j)$ as $\texttt{size}(F_i(j)) = \max_{n_k \in F_i(j)}\{\texttt{load}(n_k)\}$ where n_k is any leaf node in $F_i(j)$, and $\texttt{value}(F_i(j)) = \sum_{C_k \in F_i(j)} \texttt{value}(C_k)$. Finally, let $F(j)$ denote the collection of all $F_i(j)$'s for a fixed value of j. We consider three different cases of the scheduling problem.

1. <u>No subtasks.</u> In this case, each task is a simple periodic task. We are then faced with a packing problem that can be described as: *Given d collections of objects $F = \{F(0), \ldots, F(d-1)\}$, for each collection $F(j)$, $0 \leq j < d$, determine a subset $F'(j)$ of $F(j)$ and a packing of $F'(j)$ in a unit capacity bin such that the total value* $\sum_{F_i(j) \in F'[j]} \texttt{value}(F_i(j)$ *is maximized.* Since each collection an be treated independently, our problem corresponds to a traditional 0/1 knapsack optimization problem. Thus, knapsack heuristics (e.g., algorithm PackClips$(F(j), 1))$ can be used for each collection $F(j)$ of objects. (The set of scheduled tasks is defined by the set of subtrees selected in the packing.)

2. <u>$\gcd(n_{disk}, d) > 1$:</u> In this case, if we set d to be equal to $\gcd(n_{disk}, d)$, then the optimization problem is the same as in case (1). In other words, in spite of the subtasks, we can pack subtrees of different forests, rather than packing the entire forest. This is because, if $\gcd(n_{disk}, d) > 1$ holds, then all the subtasks of a task reside in subtrees rooted at edges (emanating from the root) with weights that are congruent to $j \pmod{\gcd(d, n_{disk})}$ for some j $(0 \leq j < d)$.

3. <u>Otherwise.</u> With each forest F_i, we associate a d-dimensional size vector (with the j^{th}

component equal to $\texttt{size}(F_i(j))$) and a value $\texttt{value}(F_i) = \sum_{j=1}^{d} \texttt{value}(F_i(j))$. We are then faced with a d-dimensional variant of our original (i.e., worst-case collision) packing problem, in which forests are packed into a d-dimensional unit capacity bin with the objective of maximizing the accumulated value. Again, heuristics (like PACKCLIPS) based on d-dimensional vector packing [CGJ84] can provide approximate solutions.

8.7 Experimental Performance Evaluation

In this section, we describe the results of a preliminary performance evaluation comparing the average performance of the schemes presented in this chapter for supporting EPPV service.

8.7.1 Experimental Testbed

For our experiments, we used two basic workload components, modeling typical scenarios encountered in today's pay-per-view video servers.

- **Workload 1** consisted of relatively long MPEG-1 compressed videos with a duration between 90 and 120 minutes (e.g., movie features). The display rate for all these videos was equal to $r_i = 1.5$ Mbps. To model differences in video popularity, our workload comprised two distinct regions: a "hot region" with retrieval periods between 40 and 60 minutes and a "cold region" with periods between 150 and 180 minutes.

- **Workload 2** consisted of small video clips with lengths between 2 and 10 minutes (e.g., commercials or music video clips). The display rates for these videos varied between 2 and 4 Mbps (i.e., MPEG-1 and 2 compression). Again, clips were divided between a "hot region" with periods between 20 and 30 minutes and a "cold region" with periods between 40 and 60 minutes.

We experimented with each component executing in isolation and with mixed workloads consisting of mixtures of type 1 and type 2 workloads. We concentrated on scaleup experiments in which the total expected storage requirements of the offered workload were approximately equal to the total storage capacity of the server. This allowed us to effectively ignore the storage capacity constraint for the striping-based schemes. For Clustering, storage capacities were accounted for by using the 2-dimensional version of PACKCLIPS (Section 8.2.2). Our basic performance metric was the *effectively scheduled disk bandwidth* (in Mbps) for each of the resource scheduling schemes presented in this chapter. (The graphs presented in the next section are indicative of the results obtained over the ranges of the workload parameters.)

The results discussed in this chapter were obtained assuming a bandwidth capacity of r_{disk} = 80 Mbps and a storage capacity of c_{disk} = 4 GBytes for each disk in the server. The (worst-case) disk seek time and latency were set at t_{seek} = 24 ms and t_{lat} = 9.3 ms, respectively, and the round length was T = 1 sec. As part of our future work, we plan to examine the effect of these parameters on the performance of our scheduling schemes. Table 12 summarizes our experimental parameter settings.

Disk Params.	
r_{disk}	80 Mbps
c_{disk}	4 GBytes
t_{seek}	24 ms
t_{lat}	9.3 ms
T	1 sec

Workload 1	
l_i	90 – 120 min
T_i	40-60 , 150-180 min
r_i	1.5 Mbps
No. clips	20 – 200
Hot clips	5% – 50%

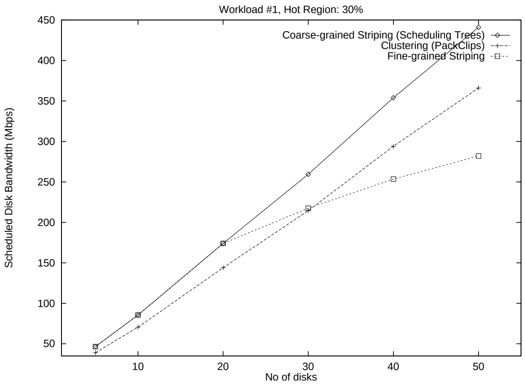
Workload 2	
l_i	2 – 10 min
T_i	20-30, 40-60 min
r_i	2 – 4 Mbps
No. clips	80 – 200
Hot clips	5% – 50%

Table 12: Experimental Parameter Settings

8.7.2 Experimental Results

The results of our experiments with type 1 workloads with hot regions of 30% and 10% are shown in Figures 50(a) and 50(b), respectively. Clearly, the CGS-based scheme outperforms both Clustering and FGS over the entire range of values for the number of disks. Observe that for type 1 workloads and for the parameter values given in Table 12, the maximum number of clips that can be scheduled is limited by the aggregate disk storage. Specifically, it is easy to see that the maximum number of clips that can fit in a disk is 3.95 and the average number of concurrent streams for a clip is $(0.3 \cdot 3 + 0.7 \cdot 1) = 1.6$. Thus, the maximum bandwidth that can be utilized on a single disk for this mix of accesses is $1.6 \cdot 3.95 \cdot 1.5 = 9.48$Mbps. This explains the low scheduled bandwidth output shown in Figure 50. We should note that in most cases our scheduling tree heuristics were able to schedule the entire offered workload of clips. On the other hand, the performance of FGS schemes quickly deteriorates as the size of the disk array increases. This confirms our remarks on the limited scalability of FGS in Section 8.3. The performance of our Clustering scheme under Workload 1 suffers from the disk storage fragmentation due to the large clip sizes. We also observe a deterioration in the performance of Clustering as the access skew increases (i.e., the size of the hot region becomes smaller). This can be explained as follows: PACKCLIPS first tries to pack the clips that give the highest profit (i.e., the hot clips). Thus when the hot region becomes smaller the relative value of the scheduled subset (as compared to the total workload value) decreases.

The relative performance of the three schemes for a type 2 workload with a 50% hot region is



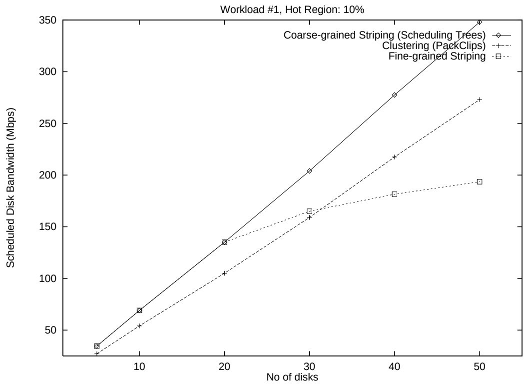


Figure 50: (a) Workload 1, 30% hot. (b) Workload 1, 10% hot.

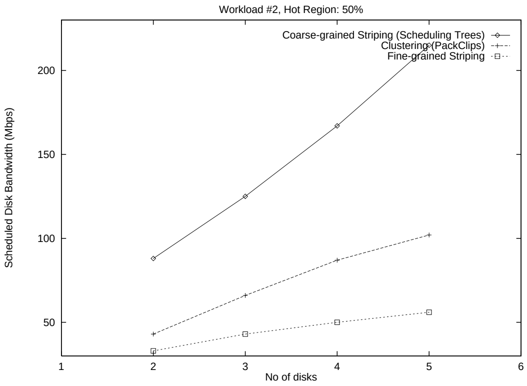
depicted in Figure 51(a). Again, the CGS-based scheme outperforms both Clustering and FGS over the entire range of n_{disk}. Note that, compared to type 1 workloads, the relative performance of Clustering and FGS schemes under this workload of short clips is significantly worse. This is because both these schemes, being unaware of the periodic nature of clip retrieval, reserve a specific amount of bandwidth for every clip C_i during every round of length T. However, for clips whose length is relatively small compared to their period this bandwidth will actually be needed only for small fraction of rounds. Figure 51(a) clearly demonstrates the devastating effects of this bandwidth wastage and the need for periodic scheduling algorithms.

Finally, Figure 51(b) depicts the results obtained for a mixed workload consisting of 30% type 1 clips and 70% type 2 clips. CGS is once again consistently better than FGS and Clustering over the entire range of disk array sizes. Compared to pure type 1 or 2 workloads, the Clustering-based scheme is able to exploit the non-uniformities in the mixed workload to produce much better packings. This gives Clustering a clear win over FGS. Still, its wastefulness of disk bandwidth for short clips does not allow it to perform at the level of CGS.

8.8 Extensions

8.8.1 Periods greater than Length

In general, the period T_i of a clip C_i may be greater than its length l_i. The algorithms presented in Sections 8.2 and 8.3 for Clustering and Fine-grained Striping (FGS) can be used to schedule such clips, however, they may be unnecessarily restrictive. This is because for Clustering and FGS, PackClips reserves disk time equal to $\frac{d_i}{r_{disk}} + t_{lat}$ and $\frac{d_i}{n_{disk} \cdot r_{disk}} + t_{lat}$, respectively, every T units of time for clip C_i. However, if the length the clip is much smaller than its period, then



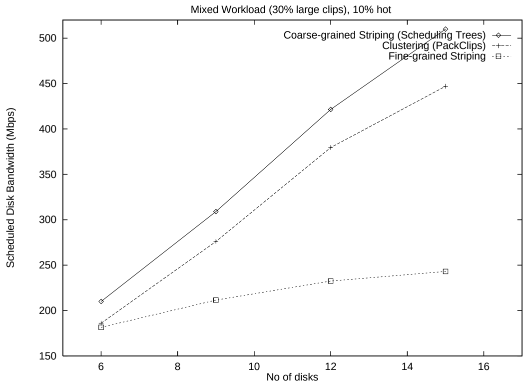


Figure 51: (a) Workload 2, 50% hot. (b) Mixed Workload (30%-70%), 10% hot.

in every T_i time slots the reserved disk capacity in $T_i - l_i$ time slots is wasted. The effects of this bandwidth wastage and the need for periodic scheduling techniques are also demonstrated in our experimental results in Section 8.7.2.

Under Clustering and FGS, the retrieval of a clip C_i can be modeled as a collection of periodic real-time tasks with period $T_i = n_i \cdot T$, where each task consists of a collection of c_i subtasks that are T time units apart and have a computation time equal to the column retrieval time. (c_i is the number of columns in C_i.) Note that the only difference between this task model and the one defined in Section 8.4 is that the distance between consecutive subtasks is only one time slot (rather than n_{disk}). Our scheduling tree algorithms and the packing-based schemes for combining forests and trees can easily be modified to deal with this case.

Obviously, to deal with the storage dimension for Clustering, the packing-based scheme presented in Section 8.6 needs to become two-dimensional. That is, each forest F_i is characterized by a two-dimensional size vector $s_i = [\mathtt{size}_1(F_i), \mathtt{size}_2(F_i)]$ (similar to the one described in Section 8.2.2), where $\mathtt{size}_1(F_i)$ is the maximum disk bandwidth requirement of any task scheduled within F_i and $\mathtt{size}_2(F_i)$ is the total storage requirement of all the tasks scheduled within F_i. More formally,

$$\mathtt{size}_1(F_i) = \max_{C_j \in F_i} \{ \frac{\frac{d_j}{r_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}} \} \quad \text{and} \quad \mathtt{size}_2(F_i) = \sum_{C_j \in F_i} \frac{l_j \cdot r_j}{c_{disk}}.$$

Using these definitions, the two-dimensional PackClips algorithm can provide an approximate solution to the value maximization problem for a given collection of forests. Note that if the number of subtasks of a task C_i is equal to $n_i = \frac{T_i}{T}$, then C_i will occupy all available time slots in a scheduling tree. Thus, if $T_i > l_i$ holds for each clip C_i, our scheme reduces exactly to the one in Section 8.2.

FGS can be handled in a similar manner. In this case, each forest F_i is characterized by a scalar (i.e.,one-dimensional) size $\texttt{size}_1(F_i) = \max_{C_j \in F_i}\{\frac{\frac{d_j}{n_{disk} \cdot r_{disk}} + t_{lat}}{T - 2 \cdot t_{seek}}\}$, and forests can be packed using one unit capacity bin.

8.8.2 Conventional Data Layout

Previously in this chapter, we assumed that clips are stored on disks using the matrix-based layout scheme. That is, each column of a clip matrix is stored contiguously. A column is nothing more than the total amount of data that needs to be retrieved in a round for all concurrent display phases. Thus, the matrix-based layout provides the nice property of reducing the disk latency overhead within a round for all the concurrent phases to a single t_{lat}. On the other hand, our scheduling and packing algorithms can also handle conventional data layout schemes that do not exploit the knowledge of retrieval periods during data layout.

Assume the conventional data layout scheme that stores the clip data contiguously on disk (i.e., stores the clip matrix in row-major order). This scheme has been the basis of most work on continuous media. Let b_i denote the disk bandwidth overhead for supporting the periodic retrieval of clip C_i. If $T_i \leq l_i$, then b_i is the same under both the conventional and the matrix-based scheme. However, if $T_i > l_i$, then Clustering and Coarse-grained Striping require $b_i = \left\lceil \frac{l_i}{T_i} \right\rceil (\frac{T \cdot r_i}{r_{disk}} + t_{lat})$ under conventional layout, and only $b_i = \left\lceil \frac{l_i}{T_i} \right\rceil \frac{T \cdot r_i}{r_{disk}} + t_{lat}$ under matrix-based layout. Similarly, if $T_i > l_i$, then Fine-grained Striping requires $b_i = \left\lceil \frac{l_i}{T_i} \right\rceil (\frac{T \cdot r_i}{n_{disk} r_{disk}} + t_{lat})$ under conventional layout, and only $b_i = \left\lceil \frac{l_i}{T_i} \right\rceil \frac{T \cdot r_i}{n_{disk} r_{disk}} + t_{lat}$ under matrix-based layout. Thus, our packing and scheduling algorithms remain the same once b_i and, thus, $\texttt{size}_1(C_i) = \frac{b_i}{T - 2 \cdot t_{seek}}$ are appropriately redefined.

8.8.3 Random Access Service Model

In addition to supporting EPPV service, our tree-based scheduling algorithms can also offer support for *Random Access* service models where resource reservations have to be placed to allocate an independent channel for each admitted client.

Most CM storage systems are built using the round-based disk scheduling/buffer management algorithm described in Section 8.1. That is, data for CM streams is retrieved into a buffer cache from disks in rounds of length T. For each stream C_i, a buffer of size $2 \cdot T \cdot r_i$ is reserved for the duration of the stream and a disk time of length $\frac{T \cdot r_i}{r_{disk}} + t_{lat}$ is reserved in every round. In each round, while the stream is consuming $T \cdot r_i$ bits of data from the buffer cache, the next $T \cdot r_i$ bits of data that the stream will consume in the next round are retrieved from disk into the buffer cache. The optimum value of T that maximizes the throughput depends on the available buffer space, disk bandwidth, latency and rates of the incoming stream requests. In order to maximize the throughput under this basic round-based approach, the value of the round length T needs to be changed *dynamically* depending on the incoming requests. However, adjusting the value of T dynamically complicates the system design.

An alternative strategy for supporting continuous retrieval of CM data is to prefetch a constant amount, say d bits, for each stream independent of its rate and maintain a fixed round size. The consumption time of d bits depends on the rate r_i of a stream C_i. More specifically, if we denote this time by P_i, then P_i can be estimated as $P_i = \frac{d}{r_i}$. Hence, for each stream C_i, instead of prefetching $T \cdot r_i$ bits in every round, d bits can be prefetched in every $p_i = \frac{P_i}{T}$ rounds. Thus, in every p_i rounds, $\frac{d}{r_{disk}} + t_{lat}$ time must be reserved for a stream C_i. (If each clip is round-robin striped over n_{disk} disks, then d bits need to be retrieved from a disk in every $n_{disk} \cdot \frac{P_i}{T}$ rounds, in which case $p_i = n_{disk} \cdot \frac{P_i}{T}$.) Prefetching d bits in every p_i rounds ensures that stream C_i will have sufficient data to display the corresponding clip continuously. This length is independent of a stream's rate and constant for each stream. Moreover, in order to reduce the buffer space requirement of a stream from $2 \cdot d$ to $d + T \cdot r_i$ one needs to schedule each retrieval for C_i exactly p_i units apart. Thus, we once again need to deal with scheduling a collection of simple periodic tasks.

Our methodology can be applied to the above problem setup as follows. For each disk (there are n_{disk} disks in the case of Coarse-grained Striping or Clustering and one "big" disk in case of Fine-grained Striping striping), $\frac{T - 2 \cdot t_{seek}}{\frac{d}{r_{disk}} + t_{lat}}$ scheduling trees can be maintained. When a new stream C_i arrives, the admission controller can check whether C_i can be inserted into any of the scheduling trees and whether there is $d + T \cdot r_i$ bits of buffer space to reserve for C_i, and if this is the case, stream C_i can be admitted. The scheduling tree guarantees that data retrievals scheduled within the same tree will never collide. On the other hand, collisions can occur across different scheduling trees. However, the number of data retrievals that collide in a round will never exceed $\frac{T - 2 \cdot t_{seek}}{\frac{d}{r_{disk}} + t_{lat}}$, since only so many trees are maintained for each disk. Thus, our approach ensures that each stream C_i will always have sufficient data to display the corresponding clip continuously while requiring only $d + T \cdot r_i$ buffer space.

Example 5: Consider a single disk system with $r_{disk} = 40$ Mbps, $t_{lat} = 9.3$ ms and $t_{seek} = 14$ ms. Suppose that the value of T is set to 1 second in both approaches. Let d be 1.5 Mb. Suppose that for a while all the incoming requests have a rate of $r_1 = 1.5$ Mbps. Both schemes will support approximately 21 requests. Now, let us assume after a while all the incoming requests have a rate of $r_2 = 28.8$ Kbps. If the value of T is not changed in the basic round-based scheme, this scheme can support approximately 99 requests with rate r_2. On the other hand, the scheduling-tree based approach can support 1092 requests with rate r_2.

8.9 Conclusions

In this chapter, we have addressed the resource scheduling and data organization problems associated with supporting EPPV service in their most general form; that is, for clips with possibly different display rates, periods, lengths. We studied three different approaches to utilizing multiple disks: Clustering, Fine-grained Striping (FGS), and Coarse-grained Striping (CGS). In each case, the periodic nature of the EPPV service model raises a host of interesting resource scheduling problems. For Clustering and FGS, we presented a knapsack formulation that allowed us to obtain a provably near-optimal heuristic with low polynomial time complexity. However, both these data layout schemes have serious drawbacks: Clustering can suffer from severe storage and bandwidth fragmentation, and FGS incurs high disk latency overheads that limit its scalability. CGS, on the other hand, avoids these problems but requires sophisticated hard real-time scheduling methods to support periodic retrieval. Specifically, we showed the EPPV scheduling problem for CGS to be a generalization of the Periodic Maintenance Scheduling Problem [WL83] and developed a number of novel concepts and algorithmic solutions to address the issues involved. We also presented a preliminary set of experimental results that verified our expectations about the average performance of the three schemes: Clustering can lead to fragmentation and underutilization of resources and the performance of FGS does not scale linearly in the number of disks due to increased latencies. Our tree-based algorithm for CGS emerged as the clear winner under a variety of randomly generated workloads. Finally, we demonstrated that the novel resource scheduling framework developed in this chapter is powerful enough to handle a number of different multimedia-related scheduling problems.

Chapter 9

Conclusions and Future Research

9.1 Thesis Summary

Effective resource management support for parallelism and multimedia data is an important mandate for next-generation database systems. In this thesis, we have addressed a number of resource scheduling problems arising in the context of query processing and optimization in parallel and multimedia database systems.

Our contributions to the area of parallel query processing and optimization can be summarized as folows. We have proposed a framework and provably near-optimal scheduling algorithms for multi-dimensional *time-shared* resource scheduling in hierarchical parallel systems. Even though our original algorithms assume given degrees of parallelism as determined by a coarse granularity condition, we have shown that they can also be extended to the more general *malleable* scheduling problem in which the scheduler is free to select the parallelization of operators without compromising the worst-case bound. We then go on to extend our problem formulation to address the co-existance of *time-shared* (e.g., CPU) and *space-shared* (e.g., memory) resources at the sites of a hierarchical parallel database system. We have shown that the inclusion of both types of resources has given rise to interesting tradeoffs with respect to the degree of partitioned parallelism, which are nicely exposed within our analytical models and results, and for which we have provided some effective resolutions. We have provided efficient, near-optimal heuristic algorithms for query scheduling in such parallel environments, paying special attention to various constraints that arise from the existence of space-shared resources. As a side-effect of our effort, we have identified an important parameter that captures one aspect of parallel query execution cost, which should play an important role in obtaining realistic cost models for parallel query optimization. Finally, we have presented a set of experimental results from the implementation of our scheduling algorithms on top of a detailed simulation environment for hierarchical database systems based on the Gamma parallel database machine. These results have verified the effectiveness of our scheduling algorithms in a realistic system setting.

Moving to the area of query scheduling for multimedia database systems, our research contributions have focused on three important problems for effective multimedia data management. First, we have formulated the resource scheduling problems for composite multimedia objects and we have developed novel efficient scheduling algorithms, drawing on a number of techniques from pattern matching and multiprocessor scheduling. More specifically, we have presented an algorithm based on Graham's list scheduling method that is provably near-optimal for monotonic object sequences and we have proposed a number of optimizations on the base list scheduling scheme. We have also given experimental results with randomly generated composite objects confirm the effectiveness of our approach. Second, we have explored the implications of the on-line nature of the admission control problem for multimedia databases which have, for the most part, been ignored in the multimedia literature. We have proposed novel admission control schemes based on the idea of *bandwidth prepartitioning* and we have employed competitive analysis techniques to prove the near-optimality of our schemes in an on-line setting. We have also proposed prepartitioning schemes that make use of request popularities to ensure good average-case as well as robust worst-case performance, and experimentally verified their effectiveness. Third, we have presented the first comprehensive study of the resource scheduling problems associated with supporting periodic (i.e., "Pay-Per-View"-style) service for CM clips with (possibly) different display rates, frequencies, and lengths. Given the intractability of the problems, we have proposed novel heuristic solutions with polynomial-time complexity and we have studied their performance both experimentally and analytically. In the process, we have introduced a robust scheduling framework that, we believe, can provide solutions for a variety of realistic "Pay-Per-View" resource scheduling scenarios, as well as any scheduling problem involving regular, periodic use of a shared resource.

Table 13 is an expanded version of Table 1 summarizing the principal dimensions of the query scheduling problem for parallel and multimedia databases together with our contributions to the regions of the search space explored in this thesis.

9.2 Future Research

The search space classification in Table 13 provides a useful guide for identifying directions for future research. In this closing section, we argue about the "research potential" of some of these search space regions and offer some thoughts on possible extensions to the work presented in this thesis.

We believe that our work on parallel query scheduling has addressed the issue of task *multi-dimensionality* within a fairly general and complete framework. On the other hand, our work on multimedia databases has only touched on the issue of multi-dimensionality (i.e., the sliding

	Parallel DB Systems	Multimedia DB Systems
Multi-dimensionality	• Near-optimal algorithms for parallel query scheduling with multiple time- and space-shared resources. • Experimental validation using simulation	• Disk bandwidth and memory allocation for composite objects using sliding
Malleability	• Selecting the degrees of operator parallelism to tradeoff response time vs. total work.	
On-line scheduling	• Handling on-line arrivals of parallel query tasks. Near-optimality results can be extended using the results of [SWW95].	• Competitive admission control with on-line arrivals. • Near-optimality results for monotonic composite object sequences can be extended to on-line arrivals using the results of [SWW95].
Time-varying resource demands	• Resource scheduling algorithms for composite multimedia objects	
Admission control		• Throughput-competitive admission control algorithms based on bandwidth prepartitioning.
Periodic service		• Strategies for scheduling periodic retrievals of CM clips under different layouts, using Scheduling Trees and/or knapsack-type heuristics.
Impact on Query Optimization	• Identified a simple 3-dimensional cost model for query optimization that captures all the important parameters of parallel query execution.	

Table 13: Dimensions of the Query Scheduling Problem and Thesis Contributions

techniques of Chapter 6 for memory and bandwidth allocation). Note that, by its definition, a CM stream requires certain portions of resources throughout its duration in order to ensure a certain level of service. Thus, for such CM streams, essentially *all* system resources (including disk and CPU bandwidth) are SS resources. The framework we have already developed for parallel queries is clearly general enough to handle simple CM streams with multi-dimensional needs. In fact, the problem can be mapped directly to the traditional resource constrained scheduling model of Garey and Graham [GG75, GI95]. However, the problem becomes much more challenging when composite objects and synchronization constraints enter the picture. (Essentially, the multi-dimensional version of our sequence packing problem in Chapter 6.)

Malleability is a problem dimension that introduces very interesting and hard problems in both parallel and multimedia database systems. In parallel database systems, our work has only addressed the issue of how the degree of operator parallelism should be selected to effectively tradeoff the overhead of parallelism to response time (Chapter 3). That result, however, did not consider the effect of SS resources. In the presence of SS resources, a different type of malleability comes from the dependence of the operator's work on its memory allotment. For example, the number of pages reserved for a hash-join operator can be anywhere within a range of possible allocations with smaller allocations typically implying more disk I/O. Scheduling multiple such malleable operators is a very challenging and interesting problem. In multimedia database systems, malleability typically arises from flexible *Quality of Service (QoS)* specifications for user queries. For example, if some loss in video quality is acceptable, the server can use an on-the-fly lossy compression scheme to reduce the consumption of transmission bandwidth while increasing the demand on the CPU resource. Furthermore, even a specific level of QoS can be satisfied by different presentation algorithms with largely different resource requirements. For example, a lossless compression scheme can be used to tradeoff network bandwidth for extra CPU cycles without reducing the quality of delivered video. Again, this is a challenging *malleable multi-resource* scheduling problem, especially when stream synchronization is also taken into account.

Issues of scheduling tasks with *time-varying resource demands* and *periodic service requirements* appear to be specific to multimedia database systems, since they are tightly coupled to the temporal dimension of CM data. Although there has been some prior work on policies for dynamically modifying the resource allocation of standard query operators (e.g., the memory-adaptive join algorithms of Pang et al. [PCL93]), such policies typically try to deal with unexpected fluctuations in the system workload which cannot be predicted in advance. Since these fluctuations are not known when a query is scheduled, it is not clear how clever query scheduling algorithms can help in such scenarios.

Although the role of *admission conrol* in multimedia databases is to provide rate guarantees for CM data, many similar issues arise for query processing over conventional data when the input queries have specific SS (e.g., memory) requirements. In fact, the schedulability conditions that we present for our level-based algorithm in Chapter 4 can be seen as a form of "admission control" for parallel query plans. Similar problems have also been explored in recent work under the guise of "controlling the MultiProgramming Level (MPL)" for multi-user database systems (e.g., [Bro95, Dav95, DG95]). Even though this thesis did not explicitly address admission policy issues for parallel database systems, it is a very interesting direction for future work.

Finally, as we already mentioned in Section 1.3.5, query optimization for multimedia database systems is a widely open problem for which no concrete solutions exist. However, developing appropriate optimization techniques is necessary in order to support high-level, declarative queries over multimedia databases. The major issue here is that the querying environment and therefore the resulting optimization questions differ in many ways from the traditional SQL-style querying [Cha94]. An important difference comes from the fact that answers to multimedia queries will be ranked and will not necessarily be exact matches. This problem was addressed in a recent paper by Chaudhuri and Gravano that presents a "fuzzy" querying model and techniques for optimizing selections over multimedia repositories [CG96]. Two additional issues that arise in the optimization of multimedia queries are intra/inter-media synchronization and QoS. Ignoring synchronization constraints during optimization can lead to excessive buffer requirements and underutilization of resources at run-time or unacceptable flaws in the presentation (e.g., glitches in the video, out-of-sync audio). QoS requirements are significant for optimization since they impact the space of execution alternatives as well as the metric of optimization. For example, a query generated by a fraud detection application needs to be evaluated speedily with quality of video being of secondary importance. The QoS specification in such a case could be:

{`delay < 2sec, compression = OK`},

and the optimization metric could also be an ordered list of parameters, such as:

`[reponse time, compression ratio].`

To the best of our knowledge, these issues have not been addressed in the literature except for the position paper by Chaudhuri that only suggests a very general methodology [Cha94].

Bibliography

[AA96] Kevin C. Almeroth and Mostafa H. Ammar. “On the Use of Multicast Delivery to Provide a Scalable and Interactive Video-on-Demand Service”. *IEEE Journal on Selected Areas in Communications*, August 1996.

[AAP93] Baruch Awerbuch, Yossi Azar, and Serge Plotkin. “Throughput-Competitive On-Line Routing”. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 32–40, Palo Alto, California, November 1993.

[ABFR94] Baruch Awerbuch, Yair Bartal, Amos Fiat, and Adi Rosén. “Competitive Non-Preemptive Call Control”. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 312–320, Arlington, Virginia, January 1994.

[AF91] Amihood Amir and Martin Farach. “Efficient 2-dimensional Approximate Matching of Non-rectangular Figures”. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 212–223, San Francisco, California, January 1991.

[AG97] Alberto Apostolico and Zvi Galil, editors. *“Pattern Matching Algorithms”*. Oxford University Press, 1997.

[AGH95] Sudhanshu Aggarwal, Juan A. Garay, and Amir Herzberg. “Adaptive Video On Demand”. In *Proceedings of the 3rd Annual European Symposium on Algorithms (ESA'95)*, Corfu, Greece, September 1995.

[All83] J.F. Allen. “Maintaining Knowledge About Temporal Intervals”. *Communications of the ACM*, 26(11):832–843, November 1983.

[APS97] Emmanuel L. Abram-Profeta and Kang G. Shin. “Scheduling Video Programs in Near Video-on-Demand Systems”. In *Proceedings of ACM Multimedia '97*, pages 359–369, Seattle, Washington, November 1997.

[AWY96a] Charu C. Aggarwal, Joel L. Wolf, and Philip S. Yu. “On Optimal Batching Policies for Video-on-Demand Storage Servers”. In *Proceedings of the 1996 International Conference on Multimedia Computing and Systems*, pages 253–258, Hiroshima, Japan, June 1996.

[AWY96b] Charu C. Aggarwal, Joel L. Wolf, and Philip S. Yu. “The Maximum Factor Queue Length Batching Scheme for Video-on-Demand Systems”. Technical Report RC 20621 (11/11/96), IBM Research Division, 1996.

[BB90] Krishna P. Belkhale and Prithviraj Banerjee. “Approximate Algorithms for the Partitionable Independent Task Scheduling Problem”. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I72–I75, August 1990.

[BB91] Krishna P. Belkhale and Prithviraj Banerjee. “A Scheduling Algorithm for Parallelizable Dependent Tasks”. In *Proceedings of the Fifth International Parallel Processing Symposium*, pages 500–506, 1991.

[BCR80] Brenda S. Baker, E.G. Coffman, Jr., and Ronald L. Rivest. "Orthogonal Packings in Two Dimensions". *SIAM Journal on Computing*, 9(4):846–855, November 1980.

[BCSW86] Jacek Blazewicz, Wojciech Cellary, Roman Slowinski, and Jan Weglarz. "Scheduling under Resource Constraints - Deterministic Models". *Annals of Operations Research*, 7, 1986. (Book Edition).

[BFG+95] C.K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G.P. Copeland, and W.G. Wilson. "DB2 Parallel Edition". *IBM Systems Journal*, 34(2):292–322, 1995.

[BFV96] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. "Dynamic Load Balancing in Hierarchical Parallel Database Systems". In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 436–447, Mumbai(Bombay), India, September 1996.

[BGMJ94] Steven Berson, Shahram Ghandeharizadeh, Richard Muntz, and Xiangyu Ju. "Staggered Striping in Multimedia Information Systems". In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 79–90, Minneapolis, Minnesota, May 1994.

[BKM+91] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. "On the Competitiveness of On-Line Real-Time Task Scheduling". In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, pages 106–115, San Antonio, Texas, December 1991.

[BLRK83] J. Blazewicz, J.K. Lenstra, and A.H.G. Rinnooy Kan. "Scheduling Subject to Resource Constraints: Classification and Complexity". *Discrete Applied Mathematics*, 5:11–24, 1983.

[BNCK+95] Amotz Bar-Noy, Ran Canetti, Shay Kutten, Yishay Mansour, and Baruch Schieber. "Bandwidth Allocation with Preemption". In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, pages 616–625, Las Vegas, Nevada, May 1995.

[BNGHA96] Amotz Bar-Noy, Juan A. Garay, Amir Herzberg, and Sudhanshu Aggarwal. "Sharing Video on Demand". Unpublished Manuscript (available from `http://www.eng.tau.ac.il/~ amotz/publications.html`), 1996.

[Bro94] Kurt Brown. "PRPL: A Database Workload Specification Language (Version 1.4)". Unpublished manuscript, March 1994.

[Bro95] Kurt Patrick Brown. *"Goal-Oriented Memory Allocation in Database Management Systems"*. PhD thesis, University of Wisconsin-Madison, 1995.

[BRTV90] Sanjoy Baruah, Louis Rosier, Igor Tulchinsky, and Donald Varvel. "The Complexity of Periodic Maintenance". In *Proceedings of the 1990 International Computer Symposium*, pages 315–320, Taiwan, 1990.

[BS83] Brenda S. Baker and Jerald S. Schwarz. "Shelf Algorithms for Two-Dimensional Packing Problems". *SIAM Journal on Computing*, 12(3):508–525, August 1983.

[CG96] Surajit Chaudhuri and Luis Gravano. "Optimizing Queries over Multimedia Repositories". In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 91–102, Montreal, Canada, June 1996.

[CGJ84] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson. "Approximation Algorithms for Bin-Packing – An Updated Survey". In *"Algorithm Design for Computing System Design"*, pages 49–106. Springer-Verlag, New York, 1984.

[CGJ96] E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson. "Approximation Algorithms for Bin-Packing: A Survey". In *"Approximation Algorithms for NP-Hard Problems", D. Hochbaum (Ed.)*, pages 46–93. PWS Publishing, Boston, 1996.

[CGJT80] E.G. Coffman, Jr., M.R. Garey, D.S. Johnson, and R.E. Tarjan. "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms". *SIAM Journal on Computing*, 9(4):808–826, November 1980.

[CGS95] Surajit Chaudhuri, Shahram Ghandeharizadeh, and Cyrus Shahabi. "Avoiding Retrieval Contention for Composite Multimedia Objects". In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 287–298, Zurich, Switzerland, September 1995.

[Cha94] Surajit Chaudhuri. "On Optimization of Multimedia Queries". In *Proceedings of the ACM Multimedia '94 Conference Workshop on Multimedia Database Management Systems*, San Francisco, California, 1994.

[CHM95] Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. "Scheduling Problems in Parallel Query Optimization". In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–265, San Jose, California, May 1995.

[CKY93] Mon-Song Chen, Dilip D. Kandlur, and Philip S. Yu. "Optimization of the Grouped Sweeping Scheduling (GSS) with Heterogeneous Multimedia Streams". In *Proceedings of ACM Multimedia '93*, pages 235–242, Anaheim, California, August 1993.

[CLYY92] Ming-Syan Chen, Ming-Ling Lo, Philip S. Yu, and Honesty C. Young. "Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins". In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 15–26, Vancouver, Canada, August 1992.

[CM96] Soumen Chakrabarti and S. Muthukrishnan. "Resource Scheduling for Parallel Database and Scientific Applications". In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 329–335, Padua, Italy, June 1996.

[Cof98] E.G. Coffman. Personal Communication, February 1998.

[CPS96a] K. Selçuk Candan, B. Prabhakaran, and V.S. Subrahmanian. "Retrieval Schedules Based on Resource Availibility and Flexible Presentation Specifications". Technical Report CS-TR-3616, University of Maryland, College Park, 1996.

[CPS+96b] Soumen Chakrabarti, Cynthia A. Phillips, Andreas S. Schulz, David B. Shmoys, Cliff Stein, and Joel Wein. "Improved Scheduling Algorithms for Minsum Criteria". In *Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming (ICALP'96)*, pages 646–657, Paderborn, Germany, July 1996.

[CYW92] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. "Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries". In *Proceedings of the Eighth International Conference on Data Engineering*, pages 58–67, Phoenix, Arizona, February 1992.

[Dav95] Diane Leslie Davison. *"Dynamic Resource Allocation for Multi-User Query Execution"*. PhD thesis, University of Colorado, 1995.

[DG92] David J. DeWitt and Jim Gray. "Parallel Database Systems: The Future of High Performance Database Database Systems". *Communications of the ACM*, 35(6):85–98, June 1992.

[DG95] Diane L. Davison and Goetz Graefe. "Dynamic Resource Brokering for Multi-User Query Execution". In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 281–292, May 1995.

[DGS+90] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. "The Gamma Database Machine Project". *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[DL89] Jianzhong Du and Joseph Y-T. Leung. "Complexity of Scheduling Parallel Task Systems". *SIAM Journal on Discrete Mathematics*, 2(4):473–487, November 1989.

[DS95] Asit Dan and Dinkar Sitaram. "An Online Video Placement Policy based on Bandwidth to Space Ratio (BSR)". In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 1995.

[DSS94] Asit Dan, Dinkar Sitaram, and Perwez Shahabuddin. "Scheduling Policies for an On-Demand Video Server with Batching". In *Proceedings of ACM Multimedia '94*, pages 15–23, San Francisco, California, October 1994.

[Far97] Martin Farach. Personal Communication, December 1997.

[FGK96] U. Faigle, R. Garbe, and W. Kern. "Randomized Online Algorithms for Maximizing Busy Time Interval Scheduling". *Computing*, 56:95–104, 1996.

[FN95] Ulrich Faigle and Willem M. Nawijn. "Note on Scheduling Intervals On-line". *Discrete Applied Mathematics*, 58:13–17, 1995.

[Gaw95] Rainer Gawlick. *"Admission Control and Routing: Theory and Practice"*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1995. (Technical Report MIT/LCS/TR-679).

[GC92] Jim Gemmell and Stavros Christodoulakis. "Principles of Delay-Sensitive Multimedia Data Storage and Retrieval". *ACM Transactions on Office Information Systems*, 10(1):51–90, January 1992.

[Gem95] D. James Gemmell. *"Support for Continuous Media in File Servers"*. PhD thesis, Simon Fraser University, April 1995.

[GG75] M.R. Garey and R.L. Graham. "Bounds for Multiprocessor Scheduling with Resource Constraints". *SIAM Journal on Computing*, 4(2):187–200, June 1975.

[GGJY76] M.R. Garey, R.L. Graham, D.S. Johnson, and Andrew Chi-Chih Yao. "Resource Constrained Scheduling as Generalized Bin Packing". *Journal of Combinatorial Theory (A)*, 21:257–298, 1976.

[GGK+97] Juan A. Garay, Inder S. Gopal, Shay Kutten, Yishay Mansour, and Moti Yung. "Efficient On-Line Call Control Algorithms". *Journal of Algorithms*, 23:180–194, 1997.

[GGS96] Sumit Ganguly, Akshay Goel, and Avi Silberschatz. "Efficient and Accurate Cost Models for Parallel Query Optimization". In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Montreal, Canada, June 1996.

[GGW95] Sumit Ganguly, Apostolos Gerasoulis, and Weining Wang. "Partitioning Pipelines with Communication Costs". In *Proceedings of the 6th International Conference on Information Systems and Data Management (CISMOD'95)*, pages 302–320, Bombay, India, November 1995.

[GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. "Query Optimization for Parallel Execution". In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 9–18, San Diego, California, June 1992.

[GI95] Minos N. Garofalakis and Yannis E. Ioannidis. "Scheduling Issues in Multimedia Query Optimization". *ACM Computing Surveys*, 27(4):590–592, December 1995. (Symposium on Multimedia).

[GI96] Minos N. Garofalakis and Yannis E. Ioannidis. "Multi-dimensional Resource Scheduling for Parallel Queries". In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 365–376, Montreal, Canada, June 1996.

[GI97] Minos N. Garofalakis and Yannis E. Ioannidis. "Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources". In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 296–305, Athens, Greece, August 1997.

[GIÖ98] Minos N. Garofalakis, Yannis E. Ioannidis, and Banu Özden. "Resource Scheduling for Composite Multimedia Objects". In *Proceedings of the 24th International Conference on Very Large Data Bases*, New York City, U.S.A., August 1998.

[GIÖS98] Minos N. Garofalakis, Yannis E. Ioannidis, Banu Özden, and Avi Silberschatz. "Throughput-Competitive Admission Control for Continuous Media Databases". In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 79–88, Seattle, Washington, June 1998.

[GJ79] M.R. Garey and D.S. Johnson. *"Computers and Intractability: A Guide to the Theory of NP-Completeness"*. W.H. Freeman, 1979.

[GLLRK79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey". *Annals of Discrete Mathematics*, 5:287–326, 1979.

[GLM96] Leana Golubchik, John C.S. Lui, and Richard R. Muntz. "Adaptive Piggybacking: A Novel Technique for Data Sharing in Video-on-Demand Storage Servers". *ACM Multimedia Systems*, 4(3):140–155, 1996.

[GMSY93] Shahram Ghandeharizadeh, Robert R. Meyer, Gary L. Schultz, and Jonathan Yackel. "Optimal Balanced Assignments and a Parallel Database Application". *ORSA Journal on Computing*, 5(2):151–167, Spring 1993.

[GÖS97] Minos N. Garofalakis, Banu Özden, and Avi Silberschatz. "Resource Scheduling in Enhanced Pay-Per-View Continuous Media Databases". In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 516–525, Athens, Greece, August 1997.

[GÖS98] Minos N. Garofalakis, Banu Özden, and Avi Silberschatz. "On Periodic Resource Scheduling for Continuous Media Databases". In *Proceedings of the Eighth International Workshop on Research Issues in Data Engineering – Continuous-Media Databases and Applications (RIDE'98)*, pages 111–120, Orlando, Florida, February 1998.

[Gra66] R.L. Graham. "Bounds for Certain Multiprocessing Anomalies". *The Bell System Technical Journal*, 45:1563–1581, November 1966.

[Gra69] R.L. Graham. "Bounds on Multiprocessing Timing Anomalies". *SIAM Journal on Computing*, 17(2):416–429, March 1969.

[Gra93] Goetz Graefe. "Query Evaluation Techniques for Large Databases". *ACM Computing Surveys*, 25(2):73–170, June 1993.

[GVK$^+$95] D. James Gemmell, Harrick M. Vin, Dilip D. Kandlur, P. Venkat Rangan, and Lawrence A. Rowe. "Multimedia Storage Servers: A Tutorial". *IEEE Computer*, 28(5):40–49, May 1995.

[GW93] Sumit Ganguly and Weining Wang. "Optimizing Queries for Coarse Grain Parallelism". Technical Report LCSR-TR-218, Department of Computer Sciences, Rutgers University, October 1993.

[GY93] Apostolos Gerasoulis and Tao Yang. "On the Granularity and Clustering of Directed Acyclic Task Graphs". *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[HCL$^+$90] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene J. Shekita. "Starburst Mid-Flight: As the Dust Clears". *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[HCY94] Hui-I Hsiao, Ming-Syan Chen, and Philip S. Yu. "On Parallel Execution of Multiple Pipelined Hash Joins". In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 185–196, Minneapolis, Minnesota, May 1994.

[HFV96] Waqar Hasan, Daniela Florescu, and Patrick Valduriez. "Open Issues in Parallel Query Optimization". *ACM SIGMOD Record*, 25(3):28–33, September 1996.

[HLvdV97] J.A. Hoogeveen, J.K. Lenstra, and S.L. van de Velde. "Sequencing and Scheduling: An Annotated Bibliography". Memorandum COSOR 97-02, Eindhoven University of Technology, 1997.

[HM94] Waqar Hasan and Rajeev Motwani. "Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism". In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 36–47, Santiago, Chile, August 1994.

[Hon92] Wei Hong. "Exploiting Inter-Operation Parallelism in XPRS". In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 19–28, San Diego, California, June 1992.

[HS91] Wei Hong and Michael Stonebraker. "Optimization of Parallel Query Execution Plans in XPRS". In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1991.

[HSSW97] Leslie A. Hall, Andreas S. Schulz, David B. Shmoys, and Joel Wein. "Scheduling to Minimize Average Completion Time: Off-line and On-line Approximation Algorithms". *Mathematics of Operations Research*, 22:513–544, 1997.

[IK75] Oscar H. Ibarra and Chul E. Kim. "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems". *Journal of the ACM*, 22(4):463–468, October 1975.

[JPS93] Anant Jhingran, Sriram Padmanabhan, and Ambuj Shatdal. "Join Query Optimization in Parallel Database Systems". In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, 1993.

[KLMS84] Richard M. Karp, Michael Luby, and A. Marchetti-Spaccamela. "A Probabilistic Analysis of Multidimensional Bin Packing Problems". In *Proceedings of the Annual ACM Symposium on the Theory of Computing*, pages 289–298, 1984.

[KM77] L.T. Kou and G. Markowsky. "Multidimensional Bin Packing Algorithms". *IBM Journal of Research and Development*, pages 443–448, September 1977.

[KM92] Ramesh Krishnamurti and Eva Ma. "An Approximation Algorithm for Scheduling Tasks on Varying Partition Sizes in Partitionable Multiprocessor Systems". *IEEE Transactions on Computers*, 41(12):1572–1579, December 1992.

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. "Fast Pattern Matching in Strings". *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[Knu81] Donald E. Knuth. *"The Art of Computer Programming (Vol. 2 / Seminumerical Algorithms)"*. Reading, Mass. : Addison-Wesley Pub. Co., 1981.

[KRT95] Mohan Kamath, Krithi Ramamritham, and Don Towsley. "Continuous Media Sharing in Multimedia Database Systems". In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA'95)*, pages 79–86, Singapore, April 1995.

[KS91] Gilad Koren and Dennis Shasha. "An Optimal Scheduling Algorithm with a Competitive Factor for Real-Time Systems". Technical Report TR 572, Department of Computer Science, New York University, July 1991.

[KSW97] David Karger, Cliff Stein, and Joel Wein. "Scheduling Algorithms". In *"Algorithms and Theory of Computation Handbook", M.J. Atallah (Ed.)*. CRC Press, 1997.

[Law79] Eugene L. Lawler. "Fast Approximation Algorithms for Knapsack Problems". *Mathematics of Operations Research*, 4(4):339–356, November 1979.

[LCRY93] Ming-Ling Lo, Ming-Syan Chen, C.V. Ravishankar, and Philip S. Yu. "On Optimal Processor Allocation to Support Pipelined Hash Joins". In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 69–78, Washington, D.C., June 1993.

[LG90] Thomas D.C. Little and Arif Ghafoor. "Sunchronization and Storage Models for Multimedia Objects". *IEEE Journal on Selected Areas in Communications*, 8(3):413–427, April 1990.

[LG93] Thomas D.C. Little and Arif Ghafoor. "Interval-Based Conceptual Models for Time-Dependent Multimedia Data". *IEEE Transactions on Knowledge and Data Engineering*, 5(4):551–563, August 1993.

[LL73] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". *Journal of the ACM*, 20(1):46–61, January 1973.

[LLG97] Mary Y.Y. Leung, John C.S. Lui, and Leana Golubchik. "Buffer and I/O Resource Pre-allocation for Implementing Batching and Buffering Techniques for Video-on-Demand Systems". In *Proceedings of the Thirteenth International Conference on Data Engineering*, Birmingham, U.K., April 1997.

[LT93] Darrell D.E. Long and Madhukar N. Thakur. "Scheduling Real-Time Disk Transfers for Continuous Media Applications". In *Proceedings of the Twelfth IEEE Symposium on Mass Storage Systems*, pages 227–232, 1993.

[LT94] Richard J. Lipton and Andrew Tomkins. "Online Interval Scheduling". In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 302–311, Arlington, Virginia, January 1994.

[Lud95] Walter T. Ludwig. *"Algorithms for Scheduling Malleable and Nonmalleable Parallel Tasks"*. PhD thesis, University of Wisconsin-Madison, August 1995. (Computer Sciences Technical Report #1279).

[LV95] T.D.C. Little and D. Venkatesh. "Popularity-Based Assignment of Movies to Storage Devices in a Video-on-Demand System". *ACM Multimedia Systems*, 2:280–287, 1995.

[LVZ93] Rosana S.G. Lanzelotte, Patrick Valduriez, and Mohamed Zaït. "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces". In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 493–504, Dublin, Ireland, August 1993.

[MD95] Manish Mehta and David J. DeWitt. "Managing Intra-operator Parallelism in Parallel Database Systems". In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 382–394, Zurich, Switzerland, September 1995.

[MD97] Manish Mehta and David J. DeWitt. "Data Placement in Shared-Nothing Parallel Database Systems". *The VLDB Journal*, 6(1):53–72, January 1997.

[Meh94] Manish Mehta. *"Resource Allocation for Parallel Shared-Nothing Database Systems"*. PhD thesis, University of Wisconsin-Madison, 1994.

[MNÖ+96] Cliff Martin, P.S. Narayanan, Banu Özden, Rajeev Rastogi, and Avi Silberschatz. "The *Fellini* Multimedia Storage Server". In *"Multimedia Information Storage and Management", S.M. Chung (Ed.)*. Kluwer Academic Publishers, 1996.

[MP94] S. Muthukrishnan and K. Palem. "Non-standard Stringology: Algorithms and Complexity". In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, pages 770–779, Montreal, Quebec, Canada, May 1994.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *"Randomized Algorithms"*. Cambridge University Press, 1995.

[Mut98] S. Muthukrishnan. Personal Communication, January 1998.

[NKN91] Steven R. Newcomb, Neill A. Kipp, and Victoria T. Newcomb. "The `HyTime` Hypermedia/Time-based Document Structuring Language". *Communications of the ACM*, 34(11), November 1991.

[NMW97] Guido Nerjes, Peter Muth, and Gerhard Weikum. "Stochastic Service Guarantees for Continuous Media Data on Multi-Zone Disks". In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 154–160, Tucson, Arizona, May 1997.

[NSHL95] Thomas M. Niccum, Jaideep Srivastava, Bhaskar Himatsingka, and Jianzhong Li. "Query Optimization and Processing in Parallel Databases". *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 22:259–287, 1995.

[NTB96] Kingsley C. Nwosu, Bhavani Thuraisingham, and P. Bruce Berra, editors. *"Multimedia Database Systems: Design and Implementation Strategies"*. Kluwer Academic Publishers, 1996.

[NZT96] Michael G. Norman, Thomas Zurek, and Peter Thanisch. "Much Ado About Shared-Nothing". *ACM SIGMOD Record*, 25(3):16–21, September 1996.

[ÖBRS94] Banu Özden, Alexandros Biliris, Rajeev Rastogi, and Avi Silberschatz. "A Low-Cost Storage Server for Movie on Demand Databases". In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 594–605, Santiago, Chile, September 1994.

[ÖBRS95] Banu Özden, Alexandros Biliris, Rajeev Rastogi, and Avi Silberschatz. "A Disk-Based Storage Architecture for Movie on Demand Servers". *Information Systems*, 20(6):465–482, 1995.

[ÖRS95a] Banu Özden, Rajeev Rastogi, and Avi Silberschatz. "A Framework for the Storage and Retrieval of Continuous Media Data". In *Proceedings of the 1995 International Conference on Multimedia Computing and Systems*, pages 2–13, Washington, D.C., May 1995.

[ÖRS95b] Banu Özden, Rajeev Rastogi, and Avi Silberschatz. "Disk Striping in Video Server Environments". *IEEE Data Engineering Bulletin*, 18(4):4–16, December 1995. (Special Issue on Multimedia Information Systems).

[ÖRS96a] Banu Özden, Rajeev Rastogi, and Avi Silberschatz. "Disk Striping in Video Server Environments". In *Proceedings of the 1996 International Conference on Multimedia Computing and Systems*, Hiroshima, Japan, June 1996.

[ÖRS96b] Banu Özden, Rajeev Rastogi, and Avi Silberschatz. "On the Design of a Low-Cost Video-on-Demand Storage System". *ACM Multimedia Systems*, 4(1):40–54, 1996.

[ÖRS96c] Banu Özden, Rajeev Rastogi, and Avi Silberschatz. "The Storage and Retrieval of Continuous Media Data". In *"Multimedia Database Systems: Issues and Research Directions", V.S. Subrahmanian and Sushil Jajodia (Eds.)*, pages 237–261. Springer-Verlag, 1996.

[ÖRS97] Banu Özden, Rajeev Rastogi, and Avi Silberschatz. "Periodic Retrieval of Videos from Disk Arrays". In *Proceedings of the Thirteenth International Conference on Data Engineering*, Birmingham, U.K., April 1997.

[ÖRSM95] Banu Özden, Rajeev Rastogi, Avi Silberschatz, and Cliff Martin. "Demand Paging for Movie-on-Demand Servers". In *Proceedings of the 1995 International Conference on Multimedia Computing and Systems*, Washington, D.C., May 1995.

[PCL93] HweeHwa Pang, Michael J. Carey, and Miron Livny. "Partially Preemptible Hash Joins". In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, D.C., June 1993.

[PGK88] David A. Patterson, Garth A. Gibson, and Randy H. Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, June 1988.

[PI96] Viswanath Poosala and Yannis E. Ioannidis. "Estimation of Query-Result Distribution and its Application in Parallel-Join Load Balancing". In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 448–459, Mumbai(Bombay), India, September 1996.

[Plo95] Serge Plotkin. "Competitive Routing of Virtual Circuits in ATM Networks". *IEEE Journal on Selected Areas in Communications*, pages 1128–1136, August 1995. (Invited Paper).

[PMC+90] Hamid Pirahesh, C. Mohan, Josephine Cheng, T.S. Liu, and Pat Selinger. "Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches". In *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*, pages 4–29, Dublin, Ireland, July 1990.

[PRE] PRECEPT Software, Inc. IP/TV Datasheets. (`http://www.precept.com/datasheets/html/iptvds1.htm`).

[Pri93] R. Price. "MHEG: An Introduction to the Future International Standard for Hypermedia Object Interchange". In *Proceedings of ACM Multimedia '93*, pages 121–128, Anaheim, California, August 1993.

[PS82] Christos Papadimitriou and Kenneth Steiglitz. *"Combinatorial Optimization: Algorithms and Complexity"*. Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1982.

[RM95] Erhard Rahm and Robert Marek. "Dynamic Multi-Resource Load Balancing in Parallel Database Systems". In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 395–406, Zurich, Switzerland, September 1995.

[RV91] P. Venkat Rangan and Harrick M. Vin. "Designing File Systems for Digital Video and Audio". In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 81–94, Monterey, California, October 1991.

[RV93] P. Venkat Rangan and Harrick M. Vin. "Efficient Storage Techniques for Digital Continuous Multimedia". *IEEE Transactions on Knowledge and Data Engineering*, 5(4):564–573, August 1993.

[SAC+79] P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. "Access Path Selection in a Relational Database Management System". In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston, Massachusetts, June 1979.

[Sah75] Sartaj Sahni. "Approximate Algorithms for the 0/1 Knapsack Problem". *Journal of the ACM*, 22(1):115–124, January 1975.

[Sch90a] Donovan A. Schneider. *"Complex Query Processing in Multiprocessor Database Machines"*. PhD thesis, University of Wisconsin-Madison, September 1990.

[Sch90b] Herb Schwetman. "CSIM User's Guide". Technical Report ACT-126-90, MCC, Austin, Texas, March 1990.

[SE93] Jaideep Srivastava and Gary Elsesser. "Optimizing Multi-Join Queries in Parallel Relational Databases". In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 84–92, San Diego, California, January 1993.

[Sel88] Timos K. Sellis. "Multiple-Query Optimization". *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[SG94] Abraham Silberschatz and Peter Galvin. *"Operating System Concepts"*. Addison-Wesley Publishing Company, 1994.

[SG95] Cyrus Shahabi and Shahram Ghandeharizadeh. "Continuous Display of Presentations Sharing Clips". *ACM Multimedia Systems*, 3(2), May 1995.

[SG97] Weifeng Shi and Shahram Ghandeharizadeh. "Buffer Sharing in Video-On-Demand Servers". *ACM SIGMETRICS Bulletin*, 25(2):13–20, September 1997.

[SG98] Abraham Silberschatz and Peter Galvin. *"Operating System Concepts"*. Addison-Wesley Publishing Company, 1998. (Fifth Edition).

[SGC95] Cyrus Shahabi, Shahram Ghandeharizadeh, and Surajit Chaudhuri. "On Scheduling Atomic and Composite Multimedia Objects". Technical Report USC-CS-95-622, University of Southern California, 1995. (To appear in *IEEE Transactions on Knowledge and Data Engineering.*).

[Sha86] Leonard D. Shapiro. "Join Processing in Database Systems with Large Main Memories". *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[SR93] John A. Stankovic and Krithi Ramamritham, editors. *"Advances in Real-Time Systems"*. IEEE Computer Society Press, Los Alamitos, California, 1993.

[ST85] Daniel D. Sleator and Robert E. Tarjan. "Amortized Efficiency of List Update and Paging Rules". *Communications of the ACM*, 28(2):202–208, February 1985.

[ST94] Hadas Shachnai and John J. Turek. "Multiresource Malleable Task Scheduling". Unpublished Manuscript, July 1994.

[Sto87] Harold S. Stone. *"High-performance Computer Architecture"*. Reading, Mass. : Addison-Wesley Pub. Co., 1987.

[SWW95] David B. Shmoys, Joel Wein, and David P. Williamson. "Scheduling Parallel Machines On-line". *SIAM Journal on Computing*, 24(6):1313–1331, December 1995.

[SY95] Hadas Shachnai and Philip S. Yu. "The Role of Wait Tolerance in Effective Batching: A Paradigm for Multimedia Scheduling Schemes". Technical Report RC 20038 (88607), IBM Research Division, April 1995.

[Tho96] Mikkel Thorup. "On RAM Priority Queues". In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, Atlanta, Georgia, January 1996.

[TK96] Heiko Thimm and Wolfgang Klas. "δ-Sets for Optimized Reactive Adaptive Playout Management in Distributed Multimedia Database Systems". In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 584–592, New Orleans, Louisiana, February 1996.

[TL93] Kian-Lee Tan and Hongjun Lu. "On Resource Scheduling of Multi-join Queries in Parallel Database Systems". *Information Processing Letters*, 48:189–195, 1993.

[TLW+94] John Turek, Walter Ludwig, Joel L. Wolf, Lisa Fleischer, Prasoon Tiwari, Jason Glasgow, Uwe Schwiegelshohn, and Philip S. Yu. "Scheduling Parallelizable Tasks to Minimize Average Response Time". In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 200–209, Cape May, New Jersey, June 1994.

[TWPY92] John Turek, Joel L. Wolf, Krishna R. Pattipati, and Philip S. Yu. "Scheduling Parallelizable Tasks: Putting it All on the Shelf". In *Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement & Modeling of Computer Systems*, pages 225–236, Newport, Rhode Island, June 1992.

[TWY92] John Turek, Joel L. Wolf, and Philip S. Yu. "Approximate Algorithms for Scheduling Parallelizable Tasks". In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, San Diego, California, June 1992.

[Val93] Patrick Valduriez. "Parallel Database Systems: Open Problems and New Issues". *Distributed and Parallel Databases*, 1:137–165, 1993.

[VGGG94] Harrick M. Vin, Pawan Goyal, Alok Goyal, and Anshuman Goyal. "A Statistical Admission Control Algorithm for Multimedia Servers". In *Proceedings of ACM Multimedia '94*, pages 33–40, San Francisco, California, October 1994.

[WC92] Qingzhou Wang and Kam Hoi Cheng. "A Heuristic of Scheduling Parallel Tasks and its Analysis". *SIAM Journal on Computing*, 21(2):281–294, April 1992.

[WFA92] Annita N. Wilschut, Jan Flokstra, and Peter M.G. Apers. "Parallelism in a Main-Memory DBMS: The Performance of PRISMA/DB". In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 521–532, Vancouver, Canada, August 1992.

[WFA95] Annita N. Wilschut, Jan Flokstra, and Peter M.G. Apers. "Parallel Evaluation of Multi-join Queries". In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 115–126, San Jose, California, May 1995.

[WL83] W.D. Wei and C.L. Liu. "On a Periodic Maintenance Problem". *Operations Research Letters*, 2(2):90–93, June 1983.

[WLDH96] Yuewei Wang, Jonathan C.L. Liu, David H.C. Du, and Jenwei Hsieh. "Video File Allocation over Disk Arrays for Video-on-Demand". In *Proceedings of the 1996 International Conference on Multimedia Computing and Systems*, pages 160–163, Hiroshima, Japan, June 1996.

[Woe94] Gerhard J. Woeginger. "On-line Scheduling of Jobs with Fixed Start and End Times". *Theoretical Computer Science*, 130:5–16, 1994.

[WTCY94] Joel L. Wolf, John Turek, Ming-Syan Chen, and Philip S. Yu. "Scheduling Multiple Queries on a Parallel Machine". In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement & Modeling of Computer Systems*, pages 45–55, Nashville, Tennessee, May 1994.

[WYS95] Joel L. Wolf, Philip S. Yu, and Hadas Shachnai. "DASD Dancing: A Disk Load Balancing Optimization Scheme for Video-on-Demand Computer Systems". In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement & Modeling of Computer Systems*, Ottawa, Canada, May 1995.

[YSB+89] Clement Yu, Wei Sun, Dina Bitton, Qi Yang, Richard Bruno, and John Tullis. "Efficient Placement of Audio Data on Optical Disks for Real-Time Applications". *Communications of the ACM*, 32(7):862–871, July 1989.

[Zip49] George Kingsley Zipf. *"Human Behavior and the Principle of Least Effort – An Introduction to Human Ecology"*. Addison-Wesley Press, Inc., Cambridge 42, Massachusetts, 1949.

Appendix A

Proofs of Theoretical Results

A.1 Proofs for Chapter 3

PROOF OF THEOREM 3.3.1 We start by presenting two ancillary lemmas that are used in the proof. Lemma A.1.1 establishes an important property of the length function. In Lemma A.1.2 we derive a lower bound on the optimal bin capacity for vector-packing.

Lemma A.1.1 *Let S be a set of d-dimensional vectors, and let $\pi = \{S_1, \ldots, S_k\}$ be* **any** *partition of S. Then,*

$$\frac{\sum_{i=1}^{k} l(S_i)}{d} \leq l(S) \leq \sum_{i=1}^{k} l(S_i)$$

Proof: The inequality $l(S) \leq \sum_{i=1}^{k} l(S_i)$ is obvious by the definition of $l()$. For the other side of the inequality, let $\overline{v}_i = \sum_{\overline{w} \in S_i} \overline{w}$ for each $i = 1, \ldots, k$. Also, define the function $m(\overline{w}) = \min_{1 \leq j \leq d}\{j : l(\overline{w}) = w[j]\}$, for each d-dimensional vector $\overline{w}$.
Rearranging the sum $\sum_{i=1}^{k} l(S_i)$, we obtain

$$\begin{aligned}
\sum_{i=1}^{k} l(S_i) &= \sum_{i=1}^{k} \max_{1 \leq j \leq d}\{v_i[j]\} \\
&= \sum_{\overline{v}_i : m(\overline{v}_i) = 1} \max_{1 \leq j \leq d}\{v_i[j]\} + \ldots + \sum_{\overline{v}_i : m(\overline{v}_i) = d} \max_{1 \leq j \leq d}\{v_i[j]\}
\end{aligned}$$

Observe that, by the definition of $m()$, the $\sum$ and max functions can be legally interchanged in each of the above d terms. Therefore,

$$\begin{aligned}
\sum_{i=1}^{k} l(S_i) &= \max_{1 \leq j \leq d}\{\sum_{\overline{v}_i : m(\overline{v}_i) = 1} v_i[j]\} + \ldots + \max_{1 \leq j \leq d}\{\sum_{\overline{v}_i : m(\overline{v}_i) = d} v_i[j]\} \\
&\leq d \max_{1 \leq j \leq d}\{\sum_{i=1}^{k} v_i[j]\} = d \max_{1 \leq j \leq d}\{\sum_{i=1}^{N} w_i[j]\} = dl(S)
\end{aligned}$$

Which, of course, implies $l(S) \geq \frac{\sum_{i=1}^{k} l(S_i)}{d}$. □

Lemma A.1.2 *Let $OPT[S,P]$ denote the optimal (i.e., minimum) common bin capacity among* **all possible** *packings of a set of vectors $S = \{\overline{W}_1, \ldots, \overline{W}_N\}$ into P bins. Then,*

$$OPT[S,P] \geq \max\{\ \frac{1}{P} l(S)\ ,\ \max_{1 \leq i \leq N}\{l(\overline{W}_i)\}\ \}$$

Proof: Obviously, $OPT[S,P] \geq \max_i\{l(\overline{W}_i)\}$ since each $\overline{W}_i$ must be placed in a bin. Observe that an optimal packing of S into P bins is essentially a partition of S into P sets $\{S_1, \ldots, S_P\}$, and $OPT[S,P] = \max_{1 \leq j \leq P}\{l(S_j)\}$. Applying Lemma A.1.1 gives

$$l(S) \leq \sum_{j=1}^{P} l(S_j) \leq P\, OPT[S,P]$$

which completes the proof. □

We now proceed with the proof of Theorem 3.3.1. Let R, F denote the set of all rooted and floating operators, respectively. Let SCHED denote the schedule produced by OPERATORSCHED and let OPT_1 be the optimal execution schedule for the operators in $R \cup F$ for the given degree of intra-operator parallelism (N_i for each $\mathtt{op}_i \in R \cup F$). We say that a schedule is *R-valid* if it respects the placement constraints for rooted operators. We use $S_R = \{\overline{r}_1, \ldots, \overline{r}_P\}$ to describe the collection of vectors describing the total work placed at each resource site by the set of rooted operators. Note that since both SCHED and OPT_1 are R-valid schedules, they have identical parallelizations for the operators in R. Finally, let $S_F = \{\overline{w}_1, \ldots, \overline{w}_N\}$, where $N = \sum_{\mathtt{op}_i \in F} N_i$, denote the collection of work vectors for all the floating operator clones, and define $S = S_R \cup S_F$.

By Lemma A.1.2 and the fact that each individual operator must execute on its alloted number of sites, we have the following inequality

$$T^{par}(\mathrm{OPT}_1, P) \geq \max\{\ \max_{\mathtt{op}_i \in R \cup F}\{T^{max}(\mathtt{op}_i, N_i)\}\ ,\ \frac{1}{P} l(S)\ \} \tag{11}$$

Note that we dropped the term $\max_{\overline{w} \in S}\{l(\overline{w})\}$ from Lemma A.1.2, since it is clearly less than $\max_{\mathtt{op}_i \in R \cup F}\{T^{max}(\mathtt{op}_i, N_i)\}$.

Now, let S_i denote the set of floating operator work vectors placed at site i by SCHED ($i = 1, \ldots, P$). Using equation (3), we have

$$T^{par}(\mathrm{SCHED}, P) = \max\{\ \max_{\mathtt{op}_i \in R \cup F}\{T^{max}(\mathtt{op}_i, N_i)\}\ ,\ \max_{1 \leq i \leq P}\{l(S_i \cup \{\overline{r}_i\})\ \}\ \} \tag{12}$$

The following proposition establishes a property of any R-valid packing of S into P d-dimensional bins.

Proposition A.1.1 *There exists an index* $j \in \{1, \ldots, P\}$ *such that*

$$l(S_j \cup \{\overline{r}_j\}) \leq \sum_{\overline{w} \in S_j \cup \{\overline{r}_j\}} l(\overline{w}) \leq \frac{d\, l(S)}{P}$$

Proof: The first inequality holds for all j and follows by direct application of Lemma A.1.1. For the second inequality, let us assume that, to the contrary, we have

$$\sum_{\overline{w} \in S_j \cup \{\overline{r}_j\}} l(\overline{w}) > \frac{d\, l(S)}{P}, \text{ for all } j = 1, \ldots, P.$$

Summing over all j, this gives

$$\sum_{\overline{w} \in S} l(\overline{w}) > d\, l(S)$$

which contradicts Lemma A.1.1. □

Consider the work vector packing produced by SCHED, where, without loss of generality, we assume that the sites have been renumbered in non-increasing order of bin capacity; that is, $l(S_1 \cup \{\overline{r}_1\}) \geq l(S_2 \cup \{\overline{r}_2\}) \geq \ldots \geq l(S_P \cup \{\overline{r}_P\})$ (Figure (52)).

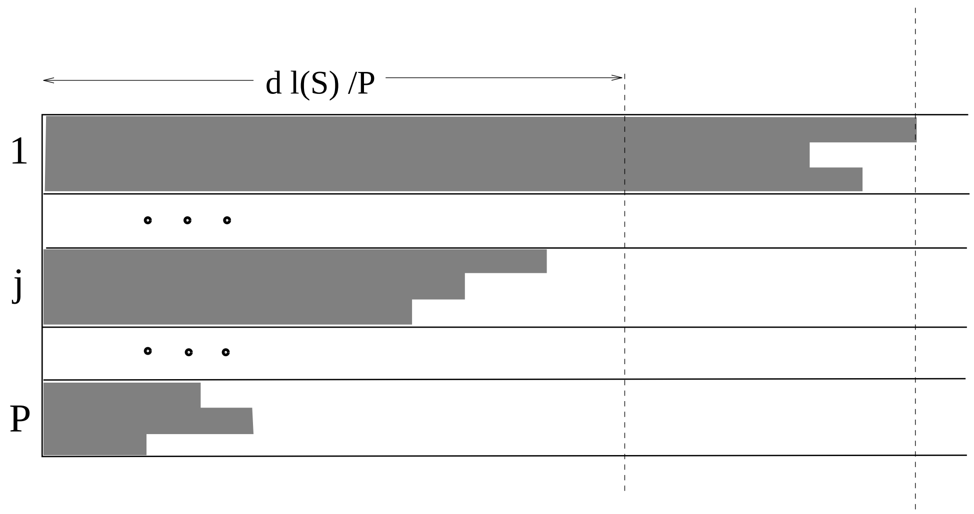


Figure 52: A packing of S's work vectors in P sites.

We distinguish the following two cases.

(a) $l(S_1 \cup \{\overline{r}_1\}) \leq \frac{dl(S)}{P}$

In this case, combining Equations (11) and (12) gives

$$T^{par}(\text{SCHED}, P) \leq d\, T^{max}(\text{OPT}_1, P)$$

and the theorem obviously holds.

(b) $l(S_1 \cup \{\overline{r}_1\}) > \frac{dl(S)}{P}$

Let $\overline{w}_{i_0}$ be the first work vector to "push" the bin capacity of site 1 over $\frac{dl(S)}{P}$. Also let $V = \{\overline{w}_{i_0}, \ldots, \overline{w}_{i_m}\}$, $m \geq 0$, be the set of vectors packed at site 1 after that moment (including $\overline{w}_{i_0}$). By Proposition A.1.1, we know that in the packing produced by SCHED there exists a site j such that $l(S_j \cup \{\overline{r}_j\}) \leq \frac{dl(S)}{P}$. The only reason that OPERATORSCHED could not pack the vectors $\overline{w}_{i_k}$, $k = 1, \ldots, m$ at site j is that site j already contained a work vector from the same floating operator. Since the work vectors for each operator are considered in non-increasing order of $l()$, this means that for each vector $\overline{w}_{i_k}$, $k = 1, \ldots, m$, there exists a vector $\overline{w}$ packed at site j such that $l(\overline{w}) \geq l((\overline{w}_{i_k})$. Thus, we have

$$\begin{aligned} l(S_1 \cup \{\overline{r}_1\}) - \frac{dl(s)}{P} \leq l(V) &\leq l(\overline{w}_{i_0}) + \sum_{k=1}^{m} l(\overline{w}_{i_k}) \\ &\leq \max_{\overline{w} \in S}\{l(\overline{w})\} + \sum_{\overline{w} \in S_j} l(\overline{w}) \\ &\leq \max_{\overline{w} \in S}\{l(\overline{w})\} + \frac{d\,l(S)}{P} \quad \text{(by Proposition A.1.1 and the choice of } S_j) \end{aligned}$$

This inequality implies

$$\begin{aligned} l(S_1 \cup \{\overline{r}_1\}) = \max_{1 \leq i \leq P}\{\, l(S_i \cup \{\overline{r}_i\})\,\} &\leq 2d\frac{l(S)}{P} + \max_{\overline{w} \in S}\{l(\overline{w})\} \\ &\leq (2d+1)\ \max\{\, \frac{l(S)}{P}\,,\ \max_{\overline{w} \in S}\{l(\overline{w})\}\,\} \end{aligned}$$

Combining Equations (11) and (12), we obtain

$$T^{par}(\text{SCHED}, P) \leq (2d+1)\, T^{par}(\text{OPT}_1, P)$$

This completes the proof of the first part of Theorem 3.3.1.

For the second part of the theorem, let OPT_2 be the optimal CG_f schedule for $R \cup F$ and let S'_F denote the collection of work vectors for all floating operators *excluding all communication cost components*. That is, we have

$$\sum_{\overline{w} \in S'_F} \sum_{j=1}^{d} w[j] = \sum_{\texttt{op} \in F} W_p(\texttt{op})$$

where $W_p(\texttt{op})$ is the processing area of operator $\texttt{op}$ (section 3.2).

Since the processing area of any operator is fixed and independent of its parallelism, we can use Lemma A.1.2 to show that the response time of *any parallel execution schedule* for $R \cup F$

will be at least $\frac{l(S'_F \cup S_R)}{P}$. Furthermore, by the estimation of N_i in OperatorSched we know that it represents the maximum allowable degree of intra-operator parallelism for a CG_f execution (assuming a fixed system size P). Therefore, by assumption (A4) we have the following inequality

$$T^{par}(\mathrm{OPT}_2, P) \geq \max\{ \max_{\mathtt{op}_i \in R \cup F} \{T^{max}(\mathtt{op}_i, N_i)\} \,,\, \frac{1}{P} l(S'_F \cup S_R) \} \tag{13}$$

By the definition of the length function and our coarse granularity criterion we have

$$\begin{aligned} l(S) &\leq l(S'_F \cup S_R) + \sum_{\mathtt{op}_i \in F} W_c(\mathtt{op}_i, N_i) \\ &\leq l(S'_F \cup S_R) + f \sum_{\mathtt{op}_i \in F} W_p(\mathtt{op}_i) \end{aligned}$$

Obviously,

$$\sum_{\mathtt{op}_i \in F} W_p(\mathtt{op}_i) \leq d l(S'_F) \leq d l(S'_F \cup S_R)$$

Thus combining the two inequalities we have

$$l(S) \leq (fd + 1)\, l(S'_F \cup S_R) \tag{14}$$

In the first part of the proof we showed that

$$T^{par}(\mathrm{SCHED}, P) \leq 2d\frac{l(S)}{P} + \max_{\mathtt{op}_i \in R \cup F} \{T^{max}(\mathtt{op}_i, N_i)\}$$

Combining this with (13) and (14), we obtain

$$T^{par}(\mathrm{SCHED}, P) \leq [2d(fd+1)+1]\, T^{par}(\mathrm{OPT}_2, P)$$

This completes the proof of Theorem 3.3.1. □

PROOF OF LEMMA 3.5.1 (Sketch) The proof follows immediately from the inequalities:

$$\begin{aligned} T^{par}(\mathrm{SCHED}, P, \underline{N}) &\leq (2d+1)\, LB(\underline{N}) \\ LB(\underline{N}^*) &\leq T^{par}(\mathrm{OPT}, P, \underline{N}^*) \end{aligned}$$

□

PROOF OF LEMMA 3.5.2 (Sketch) Note that the second property is clearly satisfied at the beginning of the algorithm. The first time an operator is alloted N_i sites such that

- $T^{max}(\mathtt{op}_i, N_i) \leq h(\underline{N}^*)$, and
- $\overline{W}_{\mathtt{op}_i}(N_i) \leq_d \overline{W}_{\mathtt{op}_i}(N_i^*)$,

it will not be alloted more sites by the algorithm until all other operators have been alloted a number of sites obeying the same conditions. The lemma follows trivially from these observations. □

A.2 Proofs for Chapter 4

PROOF OF LEMMA 4.3.1 We only need to establish the result concerning the new term in the max function (see Lemma A.1.2). Let T_{ij}, $\overline{W}_{ij}$, and $\overline{V}_{ij}$ denote, respectively, the stand-alone time, work vector, and demand vector of the j-th clone of $\texttt{op}_i$ ($j = 1, \ldots, N_i$). Note that the j-th clone of $\texttt{op}_i$ will require resource fractions $\overline{V}_{ij}$ for *at least* T_{ij} time. (It may be longer if the clone experiences contention on some preemptable resource(s).) Thus, for *any* schedule SCHED, the total resource-time product for the parallel execution must exceed these requirements for all non-preemptable resources. That is, the parallel execution time of SCHED on P sites should satisfy the following (componentwise) inequality:

$$T^{par}(\text{SCHED}, P)\; P\, \overline{1} \;\geq_s\; \sum_{i,j} T_{ij} \overline{V}_{ij},$$

where $\overline{1}$ denotes an s-dimensional vector of 1's. Taking this inequality for the maximum component of the right hand side gives the desired result. □

PROOF OF THEOREM 4.3.1 Let n_i denote the number of compatible clone subsets in site B_i and let S_{ij} be the j-th such subset. Define S^V_{ij}, S^W_{ij}, S^{TV}_{ij}, and $T^{max}(S_{ij})$ in the usual manner (see Sections 3.3.1 and 4.3.1). Consider any two consecutive compatible subsets S_{ij} and $S_{i,j+1}$ in site B_i. Let $c_m = (T_m, \overline{W}_m, \overline{V}_m)$ be the first clone placed in $S_{i,j+1}$. By the operation of the algorithm we know that $l(S^V_{ij}) + l(\overline{V}_m) > 1$. Further, by the order of clone placement we know that for all clones c_k placed in S_{ij} we will have $T_k \geq T_m = T^{max}(S_{i,j+1})$. So, considering the total volume packed on the two shelves we have:

$$l(S^{TV}_{ij}) + l(S^{TV}_{i,j+1}) \geq T^{max}(S_{i,j+1}) \cdot l(S^V_{ij}) + T^{max}(S_{i,j+1}) \cdot l(\overline{V}_m) > T^{max}(S_{i,j+1}).$$

Thus, taking the sum of T^{max}'s over all compatible subsets in B_i we have:

$$\sum_{j=1}^{n_i} T^{max}(S_{ij}) < T^{max}(S_{i,1}) + \sum_{j=2}^{n_i} l(S^{TV}_{i,j-1}) + \sum_{j=2}^{n_i} l(S^{TV}_{ij}) \leq T^{max}(S) + 2 \cdot \sum_{j=1}^{n_i} l(S^{TV}_{ij}).$$

Consider the placement of the final clone $c_N = (T_N, \overline{W}_N, \overline{V}_N)$. Without loss of generality, assume that the clone is placed in B_1. By the operation of our list scheduling heuristic we know that all the other bins must be of height larger than or equal to the height of B_1. Thus, using the notation of Section 4.3.2, we have

$$P \cdot T^{site}(B_1) \leq \sum_{i,j} T(S_{ij}) = \sum_{i,j} \max\{T^{max}(S_{ij}), l(S^W_{ij})\} \leq \sum_{i,j} T^{max}(S_{ij}) \;+\; \sum_{i,j} l(S^W_{ij}).$$

And using our previous inequality, we have:

$$P \cdot T^{site}(B_1) < P \cdot T^{max} + 2 \cdot \sum_{i,j} S_{ij}^{TV} + \sum_{i,j} l(S_{ij}^{W}).$$

We can now use the fundamental property of $l()$ (Lemma A.1.1) to get:

$$T^{site}(B_1) < T^{max}(S) + 2s \cdot \frac{l(S^{TV})}{P} + d \cdot \frac{l(S^{W})}{P}.$$

Finally, note that the makespan of the schedule SCHED obtained by our heuristic will certainly satisfy $T^{par}(\text{SCHED}, P) \leq T^{site}(B_1) + T_N \leq T^{site}(B_1) + T^{max}(S)$. Combining this with the above inequality for $T^{site}(B_1)$ gives the result. □

PROOF OF LEMMA 4.3.2 Assume the claim is false. This implies that there exists at least one SS vector $\overline{v}$ in S^V that cannot fit in any of the $\frac{l(S^V)\cdot s}{1-\lambda}$ sites used thus far. Since $l(\overline{v}) \leq \lambda$, this means that the SS "length" $l(B_i^V)$ of all these bins B_i will be $l(B_i^V) > 1 - \lambda$. Summing over all bins this gives:

$$\sum_i l(B_i^V) > (1-\lambda) \cdot \frac{l(S^V) \cdot s}{1-\lambda} = l(S^V) \cdot s$$

Since $\sum_i l(B_i^V) \leq s \cdot l(S^V - \{\overline{v}\})$ (by the fundamental property of $l()$), the above inequality gives: $l(S^V - \{\overline{v}\}) > l(S^V)$, which is impossible. This completes the proof. □

PROOF OF THEOREM 4.3.2 First, note that by Lemma 4.3.2 PIPESCHED will be able to pack C in P_C. It is easy to demonstrate the following proposition based on the fundamental property of $l()$ (Lemma A.1.1).

Proposition A.2.1 There exists an index $j \in \{1, \ldots, P_C\}$ such that

$$\sum_{\overline{w} \in B_j^W} l(\overline{w}) \leq d \cdot \frac{l(S^W)}{P_C}$$

Consider the work vector packing produced by PIPESCHED, where, without loss of generality, we assume that the sites have been renumbered in non-increasing order of total work; that is, $l(B_1^W) \geq l(B_2^W) \geq \ldots \geq l(B_{P_C}^W)$ (Figure (52)).

If $l(B_1^W) \leq d \cdot \frac{l(S^W)}{P_C}$, the theorem obviously holds. Assume $l(B_1^W) > d \cdot \frac{l(S^W)}{P_C}$ Let $\overline{w}_{i_0}$ be the first work vector to "push" the total work of site B_1 over $d \cdot \frac{l(S^W)}{P_C}$. Also let $W_e =< \overline{w}_{i_0}, \ldots, \overline{w}_{i_M} >$, $m \geq 0$, be the time-ordered list of vectors packed at site B_1 after that moment (including $\overline{w}_{i_0}$).

By Proposition A.2.1, we know that in the packing produced by PipeSched there exists a site B_j such that $\sum_{\overline{w} \in B_j^W} l(\overline{w}) \leq d \cdot \frac{l(S^W)}{P_C}$. By the logic of PipeSched we know that site B_j was not allowed to pack any of the vectors $\overline{w}_{i_k}$, $k = 1, \ldots, M$ only because of ss resource constraints. This implies that when the packing of $\overline{w}_{i_1}$ was taking place we had:

$$\sum_{\overline{v} \in B_j^V} l(\overline{v}) \geq l(B_j^V) > 1 - \lambda.$$

Also, by the order of packing we know that at the time of packing $\overline{w}_{i_1}$ we had:

$$\frac{l(\overline{w}_{i_1})}{l(\overline{v}_{i_1})} \leq \frac{\sum_{\overline{w} \in B_j^V} l(\overline{w})}{\sum_{\overline{v} \in B_j^V} l(\overline{v})},$$

which by the above inequality and Proposition A.2.1 gives:

$$\frac{l(\overline{w}_{i_1})}{l(\overline{v}_{i_1})} \leq \frac{d \cdot l(S^W)}{P_C \cdot (1 - \lambda)}.$$

Again, by the order of packing we know that $\frac{l(\overline{w}_{i_1})}{l(\overline{v}_{i_1})} \geq \frac{l(\overline{w}_{i_2})}{l(\overline{v}_{i_2})} \geq \ldots \geq \frac{l(\overline{w}_{i_M})}{l(\overline{v}_{i_M})}$, which in turn implies that:

$$\frac{l(\overline{w}_{i_1})}{l(\overline{v}_{i_1})} \geq \frac{\sum_{k=1}^{M} l(\overline{w}_{i_k})}{\sum_{k=1}^{M} l(\overline{v}_{i_k})}.$$

Combining the last two inequalities and using the fact that $\sum_{k=1}^{M} l(\overline{v}_{i_k}) \leq s \cdot l(\sum_{k=1}^{M} \overline{v}_{i_k}) \leq s$ (since all these ss vectors "fit" in one site), we have:

$$\frac{\sum_{k=1}^{M} l(\overline{w}_{i_k})}{s} \leq \frac{d \cdot l(S^W)}{P_C \cdot (1 - \lambda)},$$

or,

$$\sum_{k=1}^{M} l(\overline{w}_{i_k}) \leq \frac{d \cdot s}{1 - \lambda} \cdot \frac{l(S^W)}{P_C}.$$

Thus, the makespan of PipeSched can be bounded as follows (see Figure(52)):

$$T_H \leq d \cdot \frac{l(S^W)}{P_C} + \sum_{k=1}^{M} l(\overline{w}_{i_k}) + l(\overline{w}_{i_0}) \leq d \cdot (1 + \frac{s}{1 - \lambda}) \cdot \frac{l(S^W)}{P_C} + T^{max}.$$

This completes the proof. □

PROOF OF THEOREM 4.3.3 Let $S_j^W = \cup_{C \in L_j} S_C^W$ for all $j = 1, \ldots, k$, with S_j^V, S_j^{TV} defined similarly. Also, define $T_j^{max} = \max_{C \in L_j} T_C^{max}$. Finally, let H_j denote the parallel execution time of the j^{th} layer (i.e., the clones in L_j) as determined by PipeSched. From

Theorem 4.3.2 we know that $H_j \le d(1+\frac{s}{1-\lambda}) \cdot \frac{l(S_j^W)}{P} + T_j^{max}$, for all $j = 1, \ldots, k$. Thus, for the overall execution time we have:

$$T_H = \sum_{j=1}^{k} H_j \;\le\; d(1+\frac{s}{1-\lambda}) \cdot \sum_{j=1}^{k} \frac{l(S_j^W)}{P} \;+\; \sum_{j=1}^{k} T_j^{max}. \tag{15}$$

By the ordering of the pipelines in L we know that the total volume packed in layer j is $l(S_j^{TV}) \ge T_{j+1}^{max} \cdot l(S_j^V)$ for all $j = 1, \ldots, k-1$. Furthermore, by the condition used in the layering of the pipes (Step 2 of LEVELSCHED), we have:

$$l(S_j^{TV}) + l(S_{j+1}^{TV}) \;\ge\; T_{j+1}^{max} \cdot [l(S_j^V) + l(S_{C_{i_j+1}}^V)] \;>\; T_{j+1}^{max} \cdot \frac{P(1-\lambda)}{s}$$

for all $j = 1, \ldots, k-1$. Summing over all j this gives

$$2 \cdot \sum_{1}^{k} l(S_j^{TV}) \;>\; \sum_{1}^{k-1} [l(S_j^{TV}) + l(S_{j+1}^{TV})] \;>\; \frac{P(1-\lambda)}{s} \sum_{2}^{k} T_j^{max}.$$

Combining this with Inequality 15 we get:

$$T_H \;<\; d(1+\frac{s}{1-\lambda}) \cdot \sum_{j=1}^{k} \frac{l(S_j^W)}{P} \;+\; \frac{2s}{1-\lambda} \cdot \sum_{j=1}^{k} \frac{l(S_j^{TV})}{P} \;+\; T_1^{max}.$$

Note that by the ordering of L, $T_1^{max} = T^{max}$ (the overall maximum). Using the fundamental propery of $l()$ for the two summations in the above inequality we have:

$$T_H \;<\; d^2(1+\frac{s}{1-\lambda}) \cdot \frac{l(S^W)}{P} \;+\; \frac{2s^2}{1-\lambda} \cdot \frac{l(S^{TV})}{P} \;+\; T^{max}.$$

□

A.3 Proofs for Chapter 6

PROOF OF THEOREM 6.2.1 Our proof is based on the following claim.

Claim: Let s_k denote the start time of object C_k determined by our $\mathcal{LS}$ algorithm. For every slot $j \in \{0, 1, \ldots, s_k - 1\}$, the resource utilization (i.e., the percentage of resource bandwidth used) during time slot j is larger than $1 - \lambda$.

We prove this claim using induction on the number of objects packed. The claim is vacuously true for the first object scheduled that is assigned to slot 0 by our algorithm. Assume that it is true after packing objects $C_1, \ldots, C_{k-1}$. We will show that that it remains true after packing object C_k. Let s_m denote the *maximum start time* among all objects $C_1, \ldots, C_{k-1}$. We distinguish two cases for the start time s_k of C_k.

(1) If $s_m \ge s_k$, then the claim obviously holds by the inductive hypothesis.

(2) If $s_m < s_k$, then by the inductive hypothesis all slots in $[0, s_m - 1]$ have utilization larger than $1 - \lambda$. Assume that there exists some slot j in $[s_m, s_k - 1]$ with utilization less than or equal to $1 - \lambda$. Since all objects are *non-increasing* and (by our choice of s_m) no new objects can start in $[s_m + 1, s_k - 1]$, this means that *every slot* $n \in [j, s_k - 1]$ must have utilization less than or equal to $1 - \lambda$. This, however, means that $\mathcal{LS}$ could have placed object C_k to start at slot $j < s_k$, which contradicts the operation of our algorithm. Thus, all slots in $[s_m, s_k - 1]$ must also have utilization greater than $1 - \lambda$. This completes the proof of the claim.

Let C_i be the last composite object to complete in the schedule determined by $\mathcal{LS}$ (i.e., the object determining the makespan $T_{\mathcal{LS}}(L)$), and let s_i denote its assigned start time. If we let V_i denote the total volume "packed" in slots $[0, s_{i-1}]$ then, by our claim, $V(L) \geq V_i > (1 - \lambda) \cdot s_i$ or , equivalently, $s_i < \frac{V(L)}{1-\lambda}$. Thus:

$$T_{\mathcal{LS}}(L) = s_i + l(C_i) < \frac{V(L)}{1 - \lambda} + l_{max}(L) \leq \left(1 + \frac{1}{1 - \lambda}\right) \cdot T_{OPT}(L).$$

$\square$

A.4 Proofs for Chapter 7

PROOF OF THEOREM 7.2.1 Let $V_{\mathcal{WC}}(\overline{\sigma})$ and $V_{OPT}(\overline{\sigma})$ denote the total throughput achieved over a sequence of requests $\overline{\sigma}$ by the Work-Conserving policy and the optimal off-line scheduler, respectively. Let $A_{\mathcal{WC}}$ denote the set of requests in $\overline{\sigma}$ that are accepted by $\mathcal{WC}$.

Observe that by the operation of the $\mathcal{WC}$ policy, a request $(t_i, l_i) \in A_{\mathcal{WC}}$ can cause a later request (t_j, l_j) to be rejected only if the time interval $[t_i, t_i + l_i)$ contains the starting point of l_j (i.e., t_j). (This condition is not sufficient since there may be a free channel to accommodate the later request.) This implies that the maximum possible total length that was rejected because of the selection of $(t_i, l_i) \in A_{\mathcal{WC}}$ is $l_i + l_{max}$ (that is, when scheduling (l_i, t_i) causes the rejection $(l_i - \epsilon, t_i + \frac{\epsilon}{2})$ and $(l_{max}, t_i + l_i - \frac{\epsilon}{2})$, $\epsilon > 0$). This is obviously an upper bound, since scheduling such a total length could conflict with other scheduled intervals. Thus, an upper bound on the maximum total scheduled length for any (off-line) algorithm that does not violate the server's bandwidth constraint is given by the expression:

$$\sum_{(t_i,l_i) \in A_{\mathcal{WC}}} (l_i + l_{max}) = \sum_{(t_i,l_i) \in A_{\mathcal{WC}}} l_i + |A_{\mathcal{WC}}| \cdot l_{max}.$$

So, we have:

$$\frac{V_{OPT}(\overline{\sigma})}{V_{\mathcal{WC}}(\overline{\sigma})} \leq \frac{\sum_{(t_i,l_i) \in A_{\mathcal{WC}}} l_i + |A_{\mathcal{WC}}| \cdot l_{max}}{\sum_{(t_i,l_i) \in A_{\mathcal{WC}}} l_i} \leq 1 + \frac{|A_{\mathcal{WC}}| \cdot l_{max}}{|A_{\mathcal{WC}}| \cdot l_{min}} = 1 + \Delta.$$

We now describe a problem instance to show that this $(1+\Delta)$ competitive factor is tight for $\mathcal{WC}$. Consider a single-channel system (i.e., $c = 1$) and assume the sequence of requests: $(0, l_{min} + \epsilon)$, $(\frac{\epsilon}{2}, l_{min})$, and $(l_{min} + \frac{\epsilon}{2}, l_{max})$. It is then easy to see that the competitive ratio of $\mathcal{WC}$ for this instance will be:

$$\frac{l_{min} + l_{max}}{l_{min} + \epsilon} \longrightarrow^{\epsilon \to 0} 1 + \Delta.$$

This completes the proof. $\square$

PROOF OF THEOREM 7.2.2 Consider a sequence of requests $\sigma_1, \ldots, \sigma_N$ arriving at the server, where $\sigma_i = (t_i, l_i, r_i)$ for each i. (t_i is the time of arrival of σ_i.) We visualize the actions of the $\mathcal{WC}$ policy using a bipartite *rejection graph* $G = (V, E)$, $V = A \cup R$ where $A(R)$ is the set of requests accepted (rejected) by $\mathcal{WC}$, and the edge set E is defined by connecting each rejected request σ_i to the set of accepted requests that caused σ_i to be rejected (i.e., the requests executing at time t_i).

For each $\sigma_i \in R$ let $A(\sigma_i)$ denote the set of (accepted) neighbor nodes of σ_i. Define the *acceptance region* of $R_i \subseteq R$ as $A(R_i) = \bigcup_{\sigma_i \in R_i} A(\sigma_i)$. Our proof considers two different cases for such regions.

• *Fully-Overlapped Acceptance Regions.* In this case we assume that a set of arriving requests R_i is rejected by a set of running requests A (or, a subset of A) *with no new request(s) accepted between rejections*. This situation is depicted in Figure 53(a).

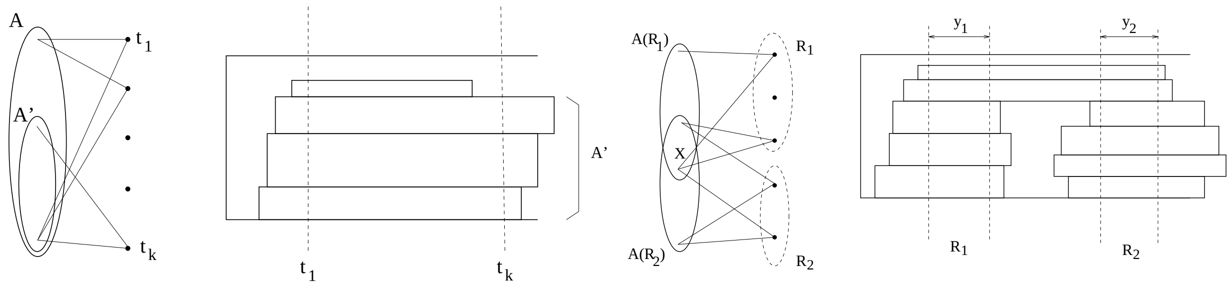


Figure 53: (a) Fully-Overlapped Case. (b) Partially-Overlapped Case.

Let t_1 and t_k denote the arrival times of the (chronologically) first and last request in R_i, respectively. Also let $A' \subseteq A$ be the subset of requests in A executing at time t_k. By the operation of $\mathcal{WC}$ we know that $\sum_{A'} r_i > \max\{B - r_{max}, r_{min}\}$ (otherwise $\mathcal{WC}$ would have scheduled the request arriving at t_k. Furthermore, our assumptions imply that all requests in A' start before t_1 and complete after t_k. Thus, $t_k - t_1 < \min_{A'}\{l_i\}$. Consequently, the benefit obtained by $\mathcal{WC}$ from A is:

$$V_{\mathcal{WC}} = \sum_A l_i r_i \geq \sum_{A'} l_i r_i > \min_{A'}\{l_i\} \cdot \max\{B - r_{max}, r_{min}\ \}.$$

Whereas the loss incurred because of the rejections is:

$$\begin{aligned} L_{\mathcal{WC}} &\leq \underbrace{(t_k - t_1) \cdot \min\{r_{max}, B - r_{min}\ \}}_{\text{Loss in } [t_1, t_k]} + \underbrace{B \cdot l_{max}}_{\text{Loss due to rejections at } t_k} \\ &< \min_{A'}\{l_i\} \cdot \min\{r_{max}, B - r_{min}\ \} + B \cdot l_{max}. \end{aligned}$$

Thus, if V_{OPT} is the benefit obtained by the optimal off-line scheduler, then:

$$\begin{aligned} \frac{V_{OPT}}{V_{\mathcal{WC}}} \leq \frac{V_{\mathcal{WC}} + L_{\mathcal{WC}}}{V_{\mathcal{WC}}} &< 1 + \frac{\min_{A'}\{l_i\} \cdot \min\{r_{max}, B - r_{min}\ \} + B \cdot l_{max}}{\min_{A'}\{l_i\} \cdot \max\{B - r_{max}, r_{min}\ \}} \\ &\leq 1 + \frac{\min\{r_{max}, B - r_{min}\ \}}{\max\{B - r_{max}, r_{min}\ \}} + \frac{B \cdot l_{max}}{\max\{B - r_{max}, r_{min}\ \} \cdot l_{min}}. \end{aligned}$$

And, using the identity $\max\{a, b\} + \min\{-a, -b\} = 0$, we obtain $\frac{V_{OPT}}{V_{\mathcal{WC}}} < (1 + \frac{l_{max}}{l_{min}}) \frac{B}{\max\{B - r_{max}, r_{min}\}}$.

•*Partially-Overlapped Acceptance Regions.* In this case we assume that the acceptance region for a set of rejections can be broken into a collection of $n \geq 2$ consecutive acceptance sub-regions $A(R_1), \ldots, A(R_n)$ where each sub-region rejects some requests with no intermediate arrivals, *but requests scheduled in a sub-region can also extend to future sub-region(s).* That is, we are allowing new requests to be accepted between the last rejection in R_i and the first rejection of R_{i+1}, and requests in $A(R_i)$ can also "participate" in $A(R_{i+k})$, $k \geq 1$ (i.e., the acceptance regions are allowed to partially overlap). This situation is depicted in Figure 53(b). Note that if no such overlapping occurs then we would have multiple independent instances of the Fully-Overlapped case.

We will first prove the competitiveness bound for the case $n = 2$ and then extend our proof to cover larger n. Let $X \subseteq A(R_1)$ denote the requests in A_1 that extend into $A(R_2)$. Let $l_X = \min_X\{l_i\}$ (the length of the shortest request in X) and let $r_X = \sum_X r_i$ (the total bandwidth requirement of X). Note that the length covered by *all* requests in X is at most l_X and the benefit of X is at least $l_X \cdot r_X$. Also, $l_{min} \leq l_X \leq l_{max}$ and $r_X \geq r_{min}$. Finally, let y_1, y_2 be the length of X's overlap with $A(R_1)$ and $A(R_2)$, respectively. (See Figure 53(b).)

Using Figure 53(b) it is easy to see that the benefit obtained by $\mathcal{WC}$ is:

$$\begin{aligned} V_{\mathcal{WC}} &> (l_{min} - y_1) \cdot \max\{B - r_{max} - r_X, r_{min}\} + (l_{min} - y_1) \cdot \max\{B - r_{max} - r_X, r_{min}\} + \\ &\quad (y_1 + y_2) \cdot \max\{B - r_{max}, r_{min}\} + (l_X - y_1 - y_2) \cdot r_X, \end{aligned}$$

or, after some arithmetic:

$$V_{\mathcal{WC}} > 2 \cdot l_{min} \cdot \max\{B - r_{max} - r_X, r_{min}\} + l_X \cdot r_X \geq 2 \cdot l_{min} \cdot \max\{B - r_{max}, 2r_{min}\} + (l_X - 2l_{min}) \cdot r_X. \tag{16}$$

Similarly, the loss incurred by the rejections of $\mathcal{WC}$ is:

$$\begin{aligned} L_{\mathcal{WC}} &< \underbrace{y_1 \cdot \min\{r_{max}, b - r_{min} - r_X\} + y_2 \cdot \min\{r_{max}, B - r_{min} - r_X\}}_{\text{Loss during } y_1,\, y_2} + \underbrace{(l_X - y_1 - y_2) \cdot (B - r_X)}_{\text{Loss between } y_1 \text{ and } y_2} + \\ & \quad \underbrace{B \cdot l_{max}}_{\text{Loss after } y_2} - \underbrace{(l_{min} - y_2) \cdot \max\{B - r_{max} - r_X, r_{min}\}}_{\text{Min. benefit of requests in } A(R_2) - X \text{ outside } y_2} \end{aligned}$$

which after some arithmetic manipulation gives:

$$\begin{aligned} L_{\mathcal{WC}} &< B \cdot (l_X + l_{max}) - l_X \cdot r_X - l_{min} \cdot \max\{B - r_{max} - r_X, r_{min}\} \\ &\le B \cdot (l_X + l_{max}) - (l_X - l_{min}) \cdot r_X - l_{min} \cdot \max\{B - r_{max}, 2r_{min}\}. \end{aligned} \tag{17}$$

We now consider the following two cases for l_X:

1. $\underline{l_X > 2 \cdot l_{min}.}$ In this case, Inequalities 16 and 17 give:

$$V_{\mathcal{WC}} > 2 \cdot l_{min} \cdot \max\{B - r_{max}, 2r_{min}\} \quad \text{and} \quad L_{\mathcal{WC}} < 2 \cdot B l_{max}.$$

   Thus: $\frac{V_{OPT}}{V_{\mathcal{WC}}} < 1 + \frac{B}{\max\{B - r_{max}, 2r_{min}\}} \cdot \frac{l_{max}}{l_{min}}$, and the bound clearly holds.

2. $\underline{2 \cdot l_{min} \ge l_X \ge l_{min}.}$ In this case, let $l_X = \alpha \cdot l_{min}$, where $\alpha \in [1, 2]$. Using Inequality 16 we have:

$$\begin{aligned} V_{\mathcal{WC}} &> \alpha \cdot l_{min} \cdot \max\{B - r_{max} - r_X, r_{min}\} + \alpha \cdot l_{min} \cdot r_X + (2 - \alpha) \cdot \max\{B - r_{max} - r_X, r_{min}\} \\ &\ge \alpha \cdot l_{min} \cdot \max\{B - r_{max}, 2r_{min}\}. \end{aligned}$$

   And, combining this with Inequality 17:

$$\frac{V_{OPT}}{V_{\mathcal{WC}}} < 1 + \frac{B}{\max\{B - r_{max}, 2r_{min}\}} + \underbrace{\frac{B \cdot l_{max} - (\alpha - 1) l_{min} \cdot r_{min} - l_{min} \cdot \max\{B - r_{max}, 2r_{min}\}}{\alpha \cdot l_{min} \cdot \max\{B - r_{max}, 2r_{min}\}}}_{f(\alpha)}.$$

   Differentiating $f(\alpha)$, it is easy to see that $\frac{df(\alpha)}{da} < 0$. Thus, $f(\alpha)$ is monotonically decreasing in $\alpha \in [1, 2]$, which implies that:

$$\begin{aligned} \frac{V_{OPT}}{V_{\mathcal{WC}}} &< 1 + \frac{B}{\max\{B - r_{max}, 2r_{min}\}} + f(1) = \\ &= 1 + \frac{B}{\max\{B - r_{max}, 2r_{min}\}} + \frac{B}{\max\{B - r_{max}, 2r_{min}\}} \cdot \frac{l_{max}}{l_{min}} - 1 \\ &\le \frac{B}{\max\{B - r_{max}, r_{min}\}} \cdot (1 + \frac{l_{max}}{l_{min}}). \end{aligned}$$

This completes the proof for the case of $n = 2$ Partially-Overlapped accepting regions. Now consider the case $n > 2$. Let X_i denote the overlap of $A(R_i)$ and $A(R_{i+1})$, for $i = 1, \ldots, n-1$. Similar to our previous notation, let $r_{X_i} = \sum_{X_i} r_j$ and $l_{X_i} = \min_{X_i}\{l_j\} = \alpha_i \cdot l_{min}$, where $\alpha_i \geq 1$. It is not hard to see that Inequality 16 can be extended as follows:

$$V_{\mathcal{WC}} > l_{min} \cdot \sum_{i=1}^{n-1} \max\{B - r_{max} - r_{X_i}, r_{min}\} + l_{min} \cdot \sum_{i=1}^{n-1} \alpha_i \cdot r_{X_i} + l_{min} \cdot \max\{B - r_{max} - r_{X_{n-1}}, r_{min}\},$$

where the last term in the sum captures the (minimum possible) contribution of the last subregion. After some manipulation the above inequality gives:

$$V_{\mathcal{WC}} > n \cdot l_{min} \max\{B - r_{max}, 2r_{min}\} + l_{min} \cdot \sum_{i=1}^{n-1} (\alpha_i - 1) \cdot r_{X_i} - l_{min} \cdot r_{X_{n-1}}. \tag{18}$$

And, using a method similar to that used for the case $n = 2$ we can derive the following inequality for the loss incurred by $\mathcal{WC}$ (extension of Inequality 17):

$$L_{\mathcal{WC}} < B \cdot l_{max} + B \cdot l_{min} \cdot \sum_{i=1}^{n-1} \alpha_i - r_{min} \cdot l_{min} \cdot \sum_{i=1}^{n-1} (\alpha_i - 1) - (n-1) \cdot l_{min} \max\{B - r_{max}, 2r_{min}\}. \tag{19}$$

Again, we consider two cases depending on the value of α_{n-1} (i.e., the length $l_{X_{n-1}}$).

1. $\underline{\alpha_{n-1} > 2.}$ Then, Inequalities 18 and 19 give:

$$V_{\mathcal{WC}} > n \cdot l_{min} \cdot \max\{B - r_{max}, 2r_{min}\} \quad \text{and} \quad L_{\mathcal{WC}} < n \cdot B l_{max}.$$

   Thus: $\frac{V_{OPT}}{V_{\mathcal{WC}}} < 1 + \frac{B}{\max\{B - r_{max}, 2r_{min}\}} \cdot \frac{l_{max}}{l_{min}}$, and the bound clearly holds.

2. $\underline{2 \geq \alpha_{n-1} \geq 1.}$ Then, Inequality 18 gives:

$$V_{\mathcal{WC}} > (n-1) \cdot l_{min} \cdot \max\{B - r_{max}, 2r_{min}\} + l_{min} \cdot r_{min} \cdot \sum_{i=1}^{n-1} (\alpha_i - 1).$$

   Combining this with Inequality 19 gives:

$$\begin{aligned} \frac{V_{OPT}}{V_{\mathcal{WC}}} &< \frac{B \cdot l_{max} + B \cdot l_{min} \cdot \sum_{i=1}^{n-1} \alpha_i}{(n-1) \cdot l_{min} \cdot \max\{B - r_{max}, 2r_{min}\} + lmin \cdot r_{min} \cdot \sum_{i=1}^{n-1} (\alpha_i - 1)} \\ &\leq \frac{(n-1) \cdot B \cdot l_{max} + \alpha_{n-1} \cdot B l_{min}}{(n-1) \cdot l_{min} \cdot \max\{B - r_{max}, 2r_{min}\}} \end{aligned}$$

   or, equivalently:

$$\begin{aligned} \frac{V_{OPT}}{V_{\mathcal{WC}}} &< \frac{B}{\max\{B - r_{max}, 2r_{min}\}} \cdot \frac{l_{max}}{l_{min}} + \frac{B}{\max\{B - r_{max}, 2r_{min}\}} \cdot \frac{\alpha_{n-1}}{n-1} \\ &\leq \frac{B}{\max\{B - r_{max}, r_{min}\}} \cdot (1 + \frac{l_{max}}{l_{min}}). \end{aligned}$$

   Since $n - 1 \geq 2 \geq \alpha_{n-1}$ for each $n > 2$.

This completes the proof for the case of multiple Partially-Overlapped acceptance regions.

For the general bound, observe that the behavior of $\mathcal{WC}$ over any incoming sequence of requests can be seen as a sequence of independent (i.e., non-overlapping) execution segments, where each such segment consists of either Fully-Overlapped or Partially-Overlapped acceptance regions. Thus, for each such execution segment s we have shown that the ratio of the maximum possible loss to the benefit obtained by $\mathcal{WC}$ is $\frac{L^S_{\mathcal{WC}}}{V^S_{\mathcal{WC}}} < \frac{B}{\max\{B-r_{max},r_{min}\}} \cdot (1+\frac{l_{max}}{l_{min}}) - 1$. Thus, for the entire sequence of segments the ratio of loss to benefit is:

$$\frac{L_{\mathcal{WC}}}{V_{\mathcal{WC}}} \le \frac{\sum_s L^S_{\mathcal{WC}}}{\sum_s V^S_{\mathcal{WC}}} \le \max_s\{\frac{L^S_{\mathcal{WC}}}{V^S_{\mathcal{WC}}}\} < \frac{B}{\max\{B-r_{max},r_{min}\}} \cdot (1+\frac{l_{max}}{l_{min}}) - 1.$$

The result follows directly from this last inequality. This completes the proof. $\square$

PROOF OF THEOREM 7.2.3

(1) Deterministic Lower Bound. Let A be any (deterministic) scheduling algorithm and let x denote the competitive ratio of A. We will prove that $x \ge O(\log \Delta)$ using an adversary argument. The basic idea in the proof is that the adversary presents A with a sequence of requests that forces A to fill up its channels with "low profit" (i.e., small duration) requests in order to maintain its competitive ratio.

More specifically, the adversary presents A with a sequence of Δ "request batches" $B_1, \ldots, B_\Delta$. (To simplify the presentation we assume that Δ is an integer.) Batch B_i consists of c (= number of channels) requests of length $i \cdot l_{min}$ arriving at time $(i-1)\cdot\epsilon$, where ϵ is some arbitrarily small interval of time. Note that, since all these requests are pairwise overlapping, only c requests can be scheduled. Clearly the optimal (off-line) strategy is to schedule the requests in B_Δ, accumulating a total profit of $c \cdot \Delta \cdot l_{min} = c \cdot l_{max}$.

Consider the on-line operation of A. Let n_i denote the number of requests in B_i scheduled by A. In order to maintain its competitive ratio of x, A must schedule *at least* $\frac{c}{x}$ requests from B_1. (Otherwise, the adversary could stop the request sequence after B_1 and force A to have a competitive ratio worse than x.) Thus $n_1 \ge \frac{c}{x}$. Similarly, in order to maintain a competitiveness of x after B_2, the following inequality must be satisfied:

$$x \cdot (n_1 \cdot l_{min} + n_2 \cdot 2 \cdot l_{min}) \ge c \cdot 2 \cdot l_{min}.$$

It is easy to see that for A to satisfy the above equation (subject to the constraint $n_1 \ge \frac{c}{x}$) and, at the same time, minimize the total number of channels used $n_1 + n_2$ (thus allowing more profit from future larger requests), n_1 has to take its minimum value, i.e., $n_1 = \frac{c}{x}$. Substituting gives:

$$c \cdot l_{min} + x \cdot n_2 \cdot 2 \cdot l_{min} \ge c \cdot 2 \cdot l_{min},$$

or, equivalently: $n_2 \geq \frac{c}{2 \cdot x}$. A simple inductive argument along the same lines can show the following claim.

<u>Claim:</u> *In order to maintain a competitive ratio of x and maximize its total profit, algorithm A must schedule $n_i = \frac{c}{i \cdot x}$ requests from batch B_i for each $i = 1, \ldots, \Delta$.*

A can then use its remaining channels to schedule requests from the final (and, most profitable) batch. Let $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$ (the n^{th}-Harmonic number). Note that if, at any point during this "game", A exhausts its channels then $V_A < \sum_{i=1}^{\Delta} \frac{c}{i \cdot x} \cdot i \cdot l_{min} = \frac{c}{x} \cdot \Delta \cdot l_{min}$, and thus:

$$\frac{V_{OPT}}{V_A} > \frac{c \cdot \Delta \cdot l_{min}}{\frac{c}{x} \cdot \Delta \cdot l_{min}} = x,$$

which is impossible since A has a competitive ratio of x. Thus, we must have:

$$\frac{c}{x} \cdot \sum_{i=1}^{\Delta} \frac{1}{i} \leq c,, \quad \text{or, equivalently,} \quad x \geq H_\Delta = \ln \Delta + O(1) = O(\log \Delta).$$

This completes the proof for the deterministic lower bound.

(2) (Oblivious) Randomized Lower Bound. Our proof is based on the application of *Yao's result* to competitive analysis [MR95, Gaw95]. Briefly, this result states that the lower bound on the oblivious competitive ratio for a given problem is greater than the lower bound on the competitive ratio of *deterministic* on-line algorithms, when the request sequences for the problem are restricted to a distribution.

Our methodology is as follows. We construct a probability distribution $D_{\overline{\sigma}}$ over the request sequences $\overline{\sigma}$, and based on that randomized input sequence we provide:

1. A lower bound (L) for $E_{D_{\overline{\sigma}}}[V_{OPT}(\overline{\sigma})]$, the expected benefit accepted by the optimal off-line algorithm; and,
2. An upper bound (U) for $E_{D_{\overline{\sigma}}}[V_A(\overline{\sigma})]$, the expected benefit accrued by *any* deterministic on-line algorithm for the problem.

If κ^b denotes the oblivious competitive ratio of *any* randomized algorithm, then, by Yao's result [MR95, Gaw95]: $\kappa^b \geq \frac{L}{U}$.

First, we define the probability distribution $D_{\overline{\sigma}}$ over request sequences. Let c denote the number of playback channels at the server and, without loss of generality, assume $\log \Delta$ is integer. We assume that requests arrive in $\log \Delta + 1$ batches of c requests $B_0, \ldots, B_{\log \Delta}$ with the time separating the arrivals being a very small $\epsilon > 0$. Furthermore, all requests in batch B_i have length equal to $2^i \cdot l_{min}$ and batches arrive according to the following probabilities:

$$P[B_0 \text{ arrives}] = 1 \qquad \text{and} \qquad P[B_i \text{ arrives} | B_{i-1} \text{ arrives}] = \frac{1}{2}.$$

Thus, for $i \in \{0, \ldots, \log \Delta\}$, the probability of the arrival sequence $B_0 \cdots B_i$ is 2^{-i}.

Next, we provide a lower bound (L) for $E_{D_{\overline{\sigma}}}[V_{OPT}(\overline{\sigma})]$. Consider an off-line strategy that always accepts the requests in the *last* arriving batch. The expected benefit of that strategy is:

$$\sum_{i=0}^{\log \Delta} P[B_0 \cdots B_i] \cdot c \cdot 2^i \cdot l_{min} = c \cdot l_{min} \cdot (\log \Delta + 1).$$

Thus $E_{D_{\overline{\sigma}}}[V_{OPT}(\overline{\sigma})] \geq c \cdot l_{min} \cdot (\log \Delta + 1) \equiv L$.

Finally, we provide an upper bound (U) for $E_{D_{\overline{\sigma}}}[V_A(\overline{\sigma})]$ for all deterministic on-line algorithms A. Let $B(i, k)$ denote the *maximum expected benefit* from batches $B_i, \ldots, B_{\log \Delta}$, where the maximization is taken over all possible ways to accept requests from $B_0, \ldots, B_{i-1}$ so that *at most* $k \leq c$ channels are free. We can bound $B(i, k)$ with the following recurrence relation, where the first term in the max{}, represents the benefit from requests in B_i (with l denoting the number of requests accepted from batch B_i) and the second term represents the maximum expected benefit from requests in batches $B_{i+1}, \ldots, B_{\log \Delta}$, given that $k - l$ channels are available:

$$B(i, k) \;\leq\; \max_{l \leq k} \left\{ \; l \cdot l_{min} \cdot 2^i \; , \; \frac{1}{2} B(i+1, k-l) \; \right\},$$

with the initial condition: $B(\log \Delta, k) \;\leq\; k \cdot 2^{\log \Delta} \cdot l_{min} = k \cdot l_{max}$. Note that the factor $\frac{1}{2}$ in front of the second max{} term comes from the fact that the probability of B_{i+1} arriving given that B_i has arrived is $\frac{1}{2}$.

A simple induction on $j = \log \Delta - i$ shows that for each $i \in \{0, \ldots, \log \Delta\}$, $B(i, k) \;\leq\; k \cdot 2^i \cdot l_{min}$. Thus, for any deterministic algorithm A:

$$E_{D_{\overline{\sigma}}}[V_A(\overline{\sigma})] \;\leq\; B(0, c) \;\leq\; c \cdot l_{min} \equiv U.$$

Consequently, by Yao's result, the oblivious competitive ratio of any randomized on-line algorithm is $\kappa^b \geq \frac{L}{U} = \log \Delta + 1$. This completes the proof for the randomized lower bound. $\square$

<u>**PROOF OF THEOREM 7.2.4**</u>

(1) Deterministic Lower Bound. Case (a) can be shown by a simple construction. For Case (b), assume that $\frac{1}{k} \geq \rho > \frac{1}{k+1}$ where $k \geq 2$ is an integer. The adversary constructs the sequence of requests in a manner that is very similar to the proof of the deterministic lower bound in Theorem 7.2.3, but it also exploits the bandwidth variability to ensure that the on-line algorithm will end up using only a fraction $\frac{k}{k+1+\delta} \approx 1 - \rho$ of the available bandwidth B, where $\delta > 0$ is arbitrarily small. Similar to the one-dimensional case, we can show that the on-line algorithm must have a competitive ratio of at least H_Δ *within that portion of the* *server's bandwidth*, i.e.,

$$V_{OPT}^{partial} \geq H_\Delta \cdot V_A.$$

But the optimal off-line scheduler can avoid this bandwidth fragmentation and schedule the entire bandwidth B, thus $V_{OPT} \geq \frac{k+1+\delta}{k}$, and the competitive factor must be at least $H_\Delta \cdot \frac{k+1+\delta}{k} \approx \frac{\log \Delta}{1-\rho}$.

(2) (Oblivious) Randomized Lower Bound. We follow the same methodology as in the proof of the randomized lower bound for Theorem 7.2.3 but taking into account the worst-case bandwidth fragmentation due to variable bandwidth requests. We assume that the bandwidth B of the server is $B = k \cdot (r_{max} - \delta)$, where $\delta > 0$ is an arbitrarily small positive constant, and we consider the following request batches, which (as in the proof of Theorem 7.2.3) arrive with very small separation in time:

- For $i \in \{0, \ldots, \log \Delta\}$, batch B_i consists of $k-1$ requests with bandwidth requirement r_{max} and length $2^i \cdot l_{min}$; and,
- Batch $B_{\log \Delta + 1}$ consists of k requests with bandwidth requirement $r_{max} - \delta$ and length $2^{\log \Delta} \cdot l_{min} = l_{max}$.

Also, as in the proof of Theorem 7.2.3: $P[B_0 \text{ arrives}] = 1$ and $P[B_i \text{ arrives} | B_{i-1} \text{ arrives}] = \frac{1}{2}$. Thus, for $i \in \{0, \ldots, \log \Delta + 1\}$, the probability of the arrival sequence $B_0 \cdots B_i$ is 2^{-i}.

For the lower bound (L) on $E_{D_{\overline{\sigma}}}[V_{OPT}(\overline{\sigma})]$, observe that the off-line strategy that always accepts the last batch will have an expected benefit:

$$\begin{aligned}
\sum_{i=0}^{\log \Delta} P[B_0 \cdots B_i] \quad &\cdot \quad (k-1) \cdot r_{max} \cdot l_{min} \cdot 2^i + k \cdot (r_{max} - \delta) \cdot 2^{-\log \Delta - 1} \cdot l_{min} \cdot 2^{\log \Delta} \\
&= \quad (k-1) \cdot l_{min} \cdot r_{max} \cdot (\log \Delta + 1) + \frac{1}{2} \cdot k \cdot (r_{max} - \delta) \cdot l_{min} \equiv L.
\end{aligned}$$

We now provide an upper bound (U) on $E_{D_{\overline{\sigma}}}[V_A(\overline{\sigma})]$ for all deterministic on-line algorithms A. As in the proof of Theorem 7.2.3, let $B(i, m)$ denote the *maximum expected benefit* from batches $B_i, \ldots, B_{\log \Delta + 1}$ where m now denotes the number of "channels" of bandwidth r_{max} that are left in the server after batches $B_0, \ldots, B_{i-1}$. Note that $m \leq k-1$ by our definitions. We can then bound $B(i, m)$ using the recurrence:

$$B(i,m) \leq \begin{cases} \max_{l \leq m} \left\{ l \cdot r_{max} \cdot l_{min} \cdot 2^i + \frac{1}{2} B(i+1, m-l) \right\} & , \quad \text{if } i \leq \log \Delta \\ m \cdot (r_{max} - \delta) \cdot l_{min} \cdot 2^{\log \Delta} & , \quad \text{if } i = \log \Delta + 1 \text{ and } m < k-1 \\ k \cdot (r_{max} - \delta) \cdot l_{min} \cdot 2^{\log \Delta} & , \quad \text{if } i = \log \Delta + 1 \text{ and } m = k-1 \end{cases}$$

Note that the last two clauses in the above expression follow from the observation that once a single request with bandwidth requirement r_{max} is scheduled, the total number of requests that can be scheduled is at most $k-1$ (whereas the entire last batch of k requests can be scheduled otherwise).

Using induction on $j = \log\Delta + 1 - i$, it is easy to show that $B(i,m) \leq m \cdot r_{max} \cdot l_{min} \cdot 2^i$ for all $i \in \{0, \ldots, \log\Delta + 1\}$. Thus, for any deterministic algorithm A, $E_{D_{\overline{\sigma}}}[V_A(\overline{\sigma})] \leq B(0, k-1) \leq (k-1) \cdot r_{max} \cdot l_{min} \equiv U$.

By Yao's result, the oblivious competitive ratio of any randomized on-line algorithm is

$$\kappa^b \geq \frac{L}{U} = \frac{(k-1) \cdot r_{max} \cdot (\log\Delta + 1) + \frac{1}{2}B}{(k-1) \cdot r_{max}}.$$

By our assumption $B > 2 \cdot r_{max} \cdot \log\Delta$, so the above inequality gives:

$$\kappa^b \geq \frac{k}{k-1} \cdot \log\Delta \approx \frac{\log\Delta}{1-\rho},$$

since $k \approx \frac{1}{\rho}$ by our choice of parameters. This completes the proof. $\square$

PROOF OF THEOREM 7.2.5

(1) Identical Bandwidth Case. We first give a definition of *strong competitiveness* as defined by Bar-Noy et al. [BNCK+95]. Given an input $\overline{\sigma} = \sigma_1, \sigma_2, \ldots, \sigma_N$ where $\sigma_i = (t_i, l_i, r_i)$, it is easy to see that the bandwidth required by $\overline{\sigma}$ at time t is $B_{\overline{\sigma}}(t) = \sum_{\sigma_i : t \in [t_i, t_i + l_i)} r_i$. We say that $\overline{\sigma}$ is *feasible* if $B_{\overline{\sigma}}(t) \leq B$ for all times t. We define the *cover* of a sequence $\overline{\sigma}$ as

$$V(\overline{\sigma}) = \int_t \min\{B_{\overline{\sigma}}(t), B\} dt.$$

Note that if $\overline{\sigma}$ is feasible then $V(\overline{\sigma})$ is exactly the total throughput (i.e., length-rate product) in $\overline{\sigma}$. We say that an on-line algorithm A is *strongly* k-competitive if

$$\sup_{\overline{\sigma}} \frac{V(\overline{\sigma})}{V_A(\overline{\sigma})} \leq k.$$

Clearly, any strongly k-competitive algorithm is also k-competitive (Section 7.1).

Observe that the analysis in the proof of Theorem 7.2.1 actually establishes that $\mathcal{WC}$ is *strongly* $(1+\Delta)$-competitive. Consider an input sequence $\overline{\sigma}$. Fix a particular group of channels C_i ($|C_i| = \frac{B}{\lceil \log\Delta \rceil}$) and let $\overline{\sigma}_i$ denote the subsequence of $\overline{\sigma}$ with lengths in the range $2^{i-1} \cdot l_{min} \leq l_j < 2^i \cdot l_{min}$, for any $i \in \{1, \ldots, \lceil \log\Delta \rceil\}$. Since the operation of $\mathcal{SBP}$ on $\overline{\sigma}_i$ is identical to $\mathcal{WC}$ using only the channels in C_i we have:

$$(1 + \left(\frac{l_{max}}{l_{min}}\right)_{\overline{\sigma}_i}) \cdot V_{\mathcal{SBP}}(\overline{\sigma}_i) \geq \int_t \min\{B_{\overline{\sigma}_i}(t), \frac{B}{\lceil \log\Delta \rceil}\} dt.$$

Observe that for all requests in $\overline{\sigma}_i$ we have $\left(\frac{l_{max}}{l_{min}}\right)_{\overline{\sigma}_i} \leq 2$. Thus the above inequality gives:

$$3 \cdot \lceil \log\Delta \rceil \cdot V_{\mathcal{SBP}}(\overline{\sigma}_i) \geq \int_t \min\{B_{\overline{\sigma}_i}(t), B\} dt.$$

It is easy to see that the left-hand side of the above inequality is an upper bound on the benefit that *any* scheduler can obtain from the requests in $\overline{\sigma}_i$ (even when using all available bandwidth). Thus, we have shown that $3 \cdot \lceil \log \Delta \rceil \cdot V_{\mathcal{SBP}}(\overline{\sigma}_i) \geq V_{OPT}(\overline{\sigma}_i)$, where OPT is the optimal clairvoyant scheduler using the entire server. Clearly, $V_{\mathcal{SBP}}(\overline{\sigma}) = \sum_i V_{\mathcal{SBP}}(\overline{\sigma}_i)$ and $V_{OPT}(\overline{\sigma}) \leq \sum_i V_{OPT}(\overline{\sigma}_i)$. Thus, $3 \cdot \lceil \log \Delta \rceil \cdot V_{\mathcal{SBP}}(\overline{\sigma}) \geq V_{OPT}(\overline{\sigma})$ for any sequence $\overline{\sigma}$. The result follows.

(2) Variable Bandwidth Case. Again, the proof for the case of variable bandwidth requests is based on the observation that the proof procedure for Theorem 7.2.2 actually establishes that $\mathcal{WC}$ is *strongly* $\frac{1+\Delta}{1-\rho}$-competitive, where $\rho = \frac{r_{max}}{B}$. The proof then proceeds along the same lines as the proof for the identical bandwidth case. □

PROOF OF THEOREM 7.2.6

(1) Identical Bandwidth Case. We partition the schedule derived by $\mathcal{DBP}$ along the time axis into consecutive intervals $I_1, \ldots, I_M$ of length l_{max} (or, more accurately $2^{\lceil \log \Delta \rceil} \cdot l_{min}$) each. Let A_j denote the set of requests accepted by $\mathcal{DBP}$ inside interval I_j (i.e., accepted requests whose starting point is in I_j), and let R_j be the set of requests rejected by $\mathcal{DBP}$ *and* accepted by the optimal scheduler in I_j. We use s_j to denote the *saturation level of* I_j: the largest i such that for some point in time $t \in I_j$ *all* the channels in $C_1 \cup \ldots C_i$ are busy at time t. Note that by the operation of $\mathcal{DBP}$, only requests of length less than $2^{s_j} \cdot l_{min}$ will be rejected in I_j. Finally, let $V(S)$ denote the total benefit (i.e., rate-length product) of a set of requests S.

Fix a specific interval I_j and let $k = \lceil \log \Delta \rceil$. Partition the set R_j into $R_j^1 \cup \ldots \cup R_j^{s_j}$, where R_j^i is the set of requests in R_j with lengths in the range $[2^{i-1} \cdot l_{min}, 2^i \cdot l_{min})$. Fix a specific $i \in \{1, \ldots, s_j\}$ and slice the interval I_j into 2^{k-i} sub-intervals of length $2^i \cdot l_{min}$ each. Consider the requests in R_j^i rejected in such a sub-interval. Clearly, the maximum benefit that any scheduler could obtain from these requests is $c \cdot 2^i \cdot l_{min}$, by allowing a "batch" of c requests of maximum length $(2^i \cdot l_{min})$. But, by the operation of $\mathcal{DBP}$, since these requests were rejected $\mathcal{DBP}$ must have already accepted a benefit of

$$\frac{c}{\lceil \log \Delta \rceil} \cdot \sum_{j=1}^{i} 2^{j-1} \cdot l_{min} = \frac{c}{\lceil \log \Delta \rceil} \cdot (2^i - 1) \cdot l_{min}.$$

Furthermore, by the operation of the algorithm, this benefit will be distinct for each "batch" of rejections within different sub-intervals. Thus, the *maximum loss-to-benefit ratio* within each such sub-interval is:

$$\frac{c \cdot 2^i \cdot l_{min}}{\frac{c}{\lceil \log \Delta \rceil} \cdot (2^i - 1) \cdot l_{min}} = \lceil \log \Delta \rceil \cdot \frac{2^i}{2^i - 1} \leq 2 \cdot \lceil \log \Delta \rceil, \quad \textit{independent of } i.$$

Thus, if we let L_j denote the loss that the $\mathcal{DBP}$ scheme incurs within the interval I_j, then:

$$V(A_j) + V(A_{j-1}) \geq \max\{\frac{c}{\lceil \log \Delta \rceil} \cdot (2^{s_j} - 1) \cdot l_{min}, \frac{L_j}{2 \cdot \lceil \log \Delta \rceil}\},$$

where the first term in the max{} follows from the definition of the saturation level of I_j. Also, it is easy to see that:

$$V(R_j) \leq L_j + c \cdot 2^{s_j} \cdot l_{min},$$

where the second term captures the maximum possible loss due to rejections at the end of I_j. Combining the last two inequalities, we have:

$$V(A_j)+V(A_{j-1}) \geq \max\{\frac{c}{2 \cdot \lceil \log \Delta \rceil} \cdot 2^{s_j} \cdot l_{min}, \frac{L_j}{2 \cdot \lceil \log \Delta \rceil}\} \geq \frac{1}{2 \cdot \lceil \log \Delta \rceil} \cdot \frac{c \cdot 2^{s_j} \cdot l_{min} + L_j}{2} \geq \frac{V(R_j)}{4 \cdot \lceil \log \Delta \rceil}.$$

Thus, summing over all intervals I_j we have $V_{\mathcal{DBP}}(\overline{\sigma}) \geq \frac{\sum_j V(R_j)}{8 \cdot \lceil \log \Delta \rceil}$.

Observe that for the optimal clairvoyant scheduler A^*, $V_{A^*}(\overline{\sigma}) \leq V_{\mathcal{DBP}}(\overline{\sigma}) + \sum_j V(R_j)$, or, using the above inequality, $V_{A^*}(\overline{\sigma}) \leq V_{\mathcal{DBP}} \cdot (1 + 8 \cdot \lceil \log \Delta \rceil)$. This completes the proof for the identical bandwidth case.

(2) Variable Bandwidth Requests. The proof proceeds along the same lines as for the identical bandwidth case. We now define the *saturation level* s_j of interval I_j as the largest i such that for some point in time $t \in I_j$ the *total available bandwidth* in partitions $B_1 \cup \ldots B_i$ is less than r_{max}. Note that by the operation of $\mathcal{DBP}$, only requests of length less than $2^{s_j} \cdot l_{min}$ can be rejected in I_j.

Fix a specific interval I_j and let $k = \lceil \log \Delta \rceil$. Partition the set R_j into $R_j^1 \cup \ldots \cup R_j^{s_j}$, where R_j^i is the set of requests in R_j with lengths in the range $[2^{i-1} \cdot l_{min}, 2^i \cdot l_{min})$. Fix a specific $i \in \{1, \ldots, s_j\}$ and slice the interval I_j into 2^{k-i} sub-intervals of length $2^i \cdot l_{min}$ each. Consider the requests in R_j^i rejected in such a sub-interval. Clearly, the maximum benefit that any scheduler could obtain from these requests is $B \cdot 2^i \cdot l_{min}$. ut, by the operation of $\mathcal{DBP}$, since these requests were rejected $\mathcal{DBP}$ must have already accepted a benefit of *at least*

$$\frac{B}{\lceil \log \Delta \rceil} \cdot \sum_{j=1}^{i-1} 2^{j-1} \cdot l_{min} + (\frac{B}{\lceil \log \Delta \rceil} - r_{max}) \cdot 2^{i-1} \cdot l_{min} = \frac{c}{\lceil \log \Delta \rceil} \cdot (2^i - 1) \cdot l_{min} - r_{max} \cdot (2^{i-1}) \cdot l_{min}.$$

Furthermore, by the operation of the algorithm, this benefit will be distinct for each "batch" of rejections within different sub-intervals. Thus, the *maximum loss-to-benefit ratio* within each such sub-interval is:

$$\begin{aligned} \frac{B \cdot 2^i \cdot l_{min}}{\frac{B}{\lceil \log \Delta \rceil} \cdot (2^i - 1) \cdot l_{min} - r_{max} \cdot (2^{i-1}) \cdot l_{min}} &= \lceil \log \Delta \rceil \cdot \frac{2^i}{(2^i - 1) - \rho \cdot 2^{i-1} \cdot \lceil \log \Delta \rceil} \\ &\leq \lceil \log \Delta \rceil \cdot \frac{2}{1 - \rho \cdot \lceil \log \Delta \rceil}, \quad \textit{independent of } i. \end{aligned}$$

Thus, if we let L_j denote the loss that the $\mathcal{DBP}$ scheme incurs within the interval I_j, then:

$$V(A_j) + V(A_{j-1}) \geq \max\{\frac{B}{\lceil \log \Delta \rceil} \cdot (2^{s_j} - 1) \cdot l_{min} - r_{max} \cdot (2^{s_j - 1}) \cdot l_{min} \, , \, \frac{L_j \cdot (1 - \rho \cdot \lceil \log \Delta \rceil)}{2 \cdot \lceil \log \Delta \rceil}\},$$

where the first term in the max{} follows from the definition of the saturation level of I_j. Also:

$$V(R_j) \leq L_j + B \cdot 2^{s_j} \cdot l_{min},$$

where the second term captures the maximum possible loss due to rejections at the end of I_j. Combining the last two inequalities, we have:

$$\begin{aligned} V(A_j) + V(A_{j-1}) &\geq \max\{\frac{B \cdot (1 - \rho \cdot \lceil \log \Delta \rceil)}{2 \cdot \lceil \log \Delta \rceil} \cdot 2^{s_j} \cdot l_{min}, \frac{L_j \cdot (1 - \rho \cdot \lceil \log \Delta \rceil)}{2 \cdot \lceil \log \Delta \rceil}\} \\ &\geq \frac{1 - \rho \cdot \lceil \log \Delta \rceil}{2 \cdot \lceil \log \Delta \rceil} \cdot \frac{B \cdot 2^{s_j} \cdot l_{min} + L_j}{2} \\ &\geq \frac{V(R_j) \cdot (1 - \rho \cdot \lceil \log \Delta \rceil)}{4 \cdot \lceil \log \Delta \rceil}. \end{aligned}$$

And, summing over all intervals I_j we have $V_{\mathcal{DBP}}(\overline{\sigma}) \geq \frac{(1-\rho \cdot \lceil \log \Delta \rceil) \cdot \sum_j V(R_j)}{8 \cdot \lceil \log \Delta \rceil}$.

Observe that for the optimal clairvoyant scheduler OPT, $V_{OPT}(\overline{\sigma}) \leq V_{\mathcal{DBP}}(\overline{\sigma}) + \sum_j V(R_j)$, or, using the above inequality, $V_{OPT}(\overline{\sigma}) \leq V_{\mathcal{DBP}} \cdot (1 + \frac{8 \cdot \lceil \log \Delta \rceil)}{(1-\rho \cdot \lceil \log \Delta \rceil)}$. This completes the proof for the variable bandwidth case. □

A.5 Proofs for Chapter 8

PROOF OF THEOREM 8.2.1 First observe that, by the specific form of clip values and sizes, the inequality $p_i \geq p_j$ is equivalent to $\texttt{value}(C_i) \geq \texttt{value}(C_j)$, and it is also equivalent to $\texttt{size}(C_i) \geq \texttt{size}(C_j)$.

Consider the heuristic H_1 that always selects the *first* n_{disk} bins from the First-Fit packing described in PackClips. Obviously, PackClips will always do at least as good as H_1; that is $V_H \geq V_{H_1}$. Consider the clips in order of decreasing value density (which is also decreasing `size`) and let C_m be the first clip placed in the $n_{disk} + 1$th bin by H_1. At that point in time, the following observations can be made:

1. Each of the first n_{disk} bins is *at least half-full* (by virtue of First-Fit and decreasing clip `siześ`).

2. The bin capacity filled up to now by H_1 was filled at a value density that is greater than or equal to the density that the optimal algorithm would use to fill the corresponding part of the bin(s) (by the order of clip placement).

Let $V_{H_1}(m)$ denote the total value of the first $m-1$ clips in the list (already packed in the first n_{disk} bins). By observations (1) and (2) it is clear that the remaining capacity at that point cannot contribute more than $V_{H_1}(m)$ to the optimal solution. From this and observation (2) we conclude that:

$$V_{OPT} \leq V_{H_1}(m) + V_{H_1}(m) \leq 2 \cdot V_{H_1} \leq 2 \cdot V_H$$

□

PROOF OF THEOREM 8.2.2 Following the notation of Chapter 3, we extend the length function $l()$ (defined in Section 8.2.2) to sets of d-dimensional vectors S as follows: Let $\mathbf{w} = \sum_{\mathbf{v} \in S} \mathbf{v}$, then $l(S) = \max_{1 \leq i \leq d}\{\mathbf{w}[i]\}$ (i.e., the maximum component of the vector sum of all elements of S).

Consider the items in order of decreasing value density and let C_m be the first item that is placed in the $n_{disk}+1$th bin by PACKCLIPS. Let B_j, S_j denote the fraction of bandwidth and storage capacity (respectively) of the j-th disk ($1 \leq j \leq n_{disk}$) that is used by clips mapped onto that disk. We also use $\mathbf{d}_j$ to describe the vector with components B_j and S_j.

The First-Fit rule used by our heuristic ensures that when C_m is pushed to the extra disk we have:

$$\max\{B_j + \mathbf{s}_m[1], S_j + \mathbf{s}_m[2]\} > 1$$

for every disk $j = 1, \ldots, n_{disk}$.

Since decreasing value density implies decreasing values for $\mathbf{s}_i[1]$ and $\mathbf{s}_i[2] \leq \frac{1}{2}$, the above condition implies that:

$$B_j > 1 - \frac{1}{2} \qquad \text{or} \qquad S_j > 1 - \frac{1}{2}$$

or, equivalently: $l(\mathbf{d}_j) > \frac{1}{2}$ for every disk $j = 1, \ldots, n_{disk}$.

Let S_H denote the sets of clips packed in the first n_{disk} bins by our heuristic. Let S_{OPT} be the set of clips scheduled by an optimal policy. Consider the partitions of these two sets $S_H^1, \ldots, S_H^{n_{disk}}$ and $S_{OPT}^1, \ldots, S_{OPT}^{n_{disk}}$ produced by our heuristic and the optimal policy, respectively. We also use these symbols to represent the corresponding collection of size vectors (i.e., S_H^j is also the collection of size vectors packed in the j-th disk). Using Lemma A.1.1 and the analysis outlined in the previous paragraph it is easy to see that:

$$n_{disk} \geq \sum_{j=1}^{n_{disk}} l(S_{OPT}^j) \;\geq\; \frac{1}{2} \cdot \sum_{C_i \in S_{OPT}} l(\mathbf{s}_i), \qquad \text{and}$$

$$\frac{n_{disk}}{2} < \sum_{j=1}^{n_{disk}} l(S_H^j) \;\leq\; \sum_{C_i \in S_H} l(\mathbf{s}_i)$$

which implies:

$$\sum_{C_i \in S_H} l(\mathbf{s}_i) > \frac{1}{4} \cdot \sum_{C_i \in S_{OPT}} l(\mathbf{s}_i).$$

Thus, if we consider the optimal solution as a "bin" with total capacity of $\sum_{C_i \in S_{OPT}} l(\mathbf{s}_i)$ the above inequality guarantees that the PACKCLIPS heuristic will fill up more than 1/4-th of that capacity (before placing a clip in disk $n_{disk}+1$). Further, by the order of clip placement, we know that that 1/4-th was filled at the *maximum possible density* for the given set of clips. Thus, the total "capacity" of the optimal solution cannot possibly contribute more than $4 \cdot value(S_H)$ to the total value. This obviously implies the result.

□

PROOF OF LEMMA 8.4.1 Assume that the retrieval of C_i from a particular disk is initiated at time slot t_i. Then, the slots (partially) occupied by the retrieval of C_i from that disk are given by the equation $t = t_i + k \cdot n_i + j \cdot n_{disk}$, where $k \geq 0$ and $0 \leq j < \left\lceil \frac{c_i}{n_{disk}} \right\rceil$.

Thus, given t_i, the slots for clip C_i are characterized by a set of ≥ 1 moduli: $u_{ij} = t \bmod n_i = (t_i + jn_{disk}) \bmod n_i$ for $0 \leq j < \left\lceil \frac{c_i}{n_{disk}} \right\rceil$. The Chinese Remainder Theorem can now be directly used to obtain the result.

□

PROOF OF LEMMA 8.4.2 Our proof uses the following ancillary result, which is a direct consequence of the *Generalized Chinese Remainder Theorem* [Knu81]. It provides necessary and sufficient conditions under which collisions will occur in a fixed schedule of periodic clip retrieval tasks.

Lemma A.5.1 Consider a specific disk and let t_i denote the start time for the retrieval of C_i and $n = \mathrm{lcm}(n_1, \ldots, n_N)$ (i.e., the least common multiple of $n_1, \ldots, n_N$). Also, let $u_{ij} = (t_i + j \cdot n_{disk}) \bmod n_i$ for $0 \leq j < \left\lceil \frac{c_i}{n_{disk}} \right\rceil$. Then, in any interval of length n (after all retrievals are initiated) there exists exactly one time slot where all retrievals collide *if and only if* for all $1 \leq i < k \leq N$, $u_{ij} \equiv u_{kl} \pmod{\gcd(n_i, n_k)}$ for *some* combination of $0 \leq j < \left\lceil \frac{c_i}{n_{disk}} \right\rceil$ and $0 \leq l < \left\lceil \frac{c_k}{n_{disk}} \right\rceil$. □

Observe that, by its definition, α_i is exactly the number of *distinct* moduli $(\bmod \gcd(n_1, n_2))$ that are "occupied" by the retrieval of clip C_i. To see this, assume that the retrieval of C_i is initiated at some time slot x. Then the pattern of the time slots occupied by C_i $(\bmod \gcd(n_1, n_2))$ will start repeating itself as soon as the condition $x + k \cdot n_{disk} \equiv x \pmod{\gcd(n_1, n_2)}$ is satisfied, or equivalently $k \cdot n_{disk} = \mathrm{multiple}(\gcd(n_1, n_2))$. This equality will be satisfied for the first time when the right hand side equals $\mathrm{lcm}(n_{disk}, \mathrm{gcd}(n_1, n_2))$, i.e., the least common multiple of n_{disk} and $gcd(n_1, n_2)$. Thus, the number of distinct moduli $(\mathrm{mod}\, \mathrm{gcd}(n_1, n_2))$ occupied by C_i is either $k = \frac{\mathrm{lcm}(n_{disk}, \mathrm{gcd}(n_1,n_2))}{n_{disk}} = \frac{\mathrm{gcd}(n_i,n_j)}{\mathrm{gcd}(n_i,n_j,n_{disk})}$ (using the identity $\mathrm{lcm}(x,y) = \frac{x \cdot y}{\mathrm{gcd}(x,y)}$) or $\left\lceil \frac{n_i}{n_{disk}} \right\rceil$, whichever is smaller. Also observe that by the form of the retrieval patterns, these α_i moduli occupied by C_i are regularly spaced $(n_{disk}(\mathrm{mod}\, \mathrm{gcd}(n_1, n_2))$ time slots apart).

Thus it is easy to see that if the condition $\alpha_1 + \alpha_2 \leq \mathrm{gcd}(n_1, n_2)$ is satisfied then it is always possible to "shift" one of the patterns so that no collisions occur. On the other hand, if $\alpha_1 + \alpha_2 > \mathrm{gcd}(n_1, n_2)$ then a simple pigeonhole argument shows that the moduli equality in Lemma A.5.1 will always be satisfied (for some combination of l and j). Hence, the retrievals of C_1 and C_2 will collide. This completes the proof.

□

PROOF OF LEMMA 8.4.3 The proof proceeds in a manner exactly similar to the proof of Lemma 8.4.2 (using the observation that $\mathrm{gcd}(n_i, n_j) = k$ for all $i \neq j$).

□

PROOF OF CORRECTNESS OF ALGORITHM BUILDEQUIDTREE The correctness of BUILDEQUIDTREE relies on the following claim. Given a set of tasks, there is a collision-free schedule, if the tasks can be partitioned into n_{disk} partitions such that the following conditions hold:

1. For each partition, there is an integer d, which is a multiple of n_{disk}, that divides the periods of all the tasks in that partition.

2. Each partition can be further partitioned into subpartitions such that the following hold:

   2.1 The first subtasks of any pair of tasks C_i and C_j in each subpartition can be scheduled collision-free in time slots u_i and u_j such that $u_i \bmod \frac{d}{n_{disk}} = u_j \bmod \frac{d}{n_{disk}}$ holds.

   2.2 $\sum_{j=0}^{\frac{d}{n_{disk}}-1} s^j_{max} \leq \frac{d}{n_{disk}}$, where s^j_{max} is the maximum of the number of subtasks of any task in the subpartition.

We will now prove that if tasks can be partitioned such that if Conditions 1 and 2 hold, one can generate a collision-free schedule by modifying each u_i in each partition j, $0 \leq j < n_{disk}$, by $u'_i = j + u_i \cdot n_{disk}$. Obviously, the subtasks of tasks within subpartitions of different partitions cannot collide. This is because for any pair of subtasks C_i and C_j in a partition, $u'_i \bmod n_{disk} = u'_j \bmod n_{disk}$ holds, where u'_i and u'_j are the time slots in which subtasks C_i

and C_j are scheduled, respectively. Since the distance between two consecutive subtasks of a task is n_{disk}, all the subtasks of a task will be in the same partition (i.e., $u'_i \bmod n_{disk} = (u'_i + j \cdot n_{disk}) \bmod n_{disk}$, $0 \leq j < s_i$). Due to Condition 2.1, the first subtasks within the same subpartition do not collide. Furthermore, Condition 2.1 implies that for any pair of tasks C_i and C_j in a subpartition, $\frac{u'_i \bmod d - u'_i \bmod n_{disk}}{n_{disk}} = \frac{u'_j \bmod d - u'_j \bmod n_{disk}}{n_{disk}}$ holds. Let us enumerate the subpartitions of a partition by the value of $\frac{u'_i \bmod d - u'_i \bmod n_{disk}}{n_{disk}}$ where u'_i the time slot in which the first subtask of a task C_i in that subpartition is scheduled. Since $\frac{u'_i \bmod d - u'_i \bmod n_{disk}}{n_{disk}}$ and $\frac{(u'_i + n_{disk}) \bmod d - (u'_i + n_{disk}) \bmod n_{disk}}{n_{disk}}$ differ only by one, in such a schedule, the subtasks of a task will be in consecutive subpartitions of a partition. Once the first subtask of a task C_i is scheduled in a subpartition, due to Condition 2.2, we can ensure that in the next $s_i - 1$ subpartitions none of the tasks' first subtask is scheduled, we ensure that none of the subtasks of the tasks scheduled within a subpartition will collide with any of the subtasks scheduled within a different subpartition of the same partition. Furthermore, it is straightforward to prove that if the first subtasks of two tasks are scheduled within the same subpartition j without collision, none of their subtasks will collide provided that none of the tasks' first subtask is scheduled in the next $s^j_{max} - 1$ subpartitions.

□