2
2001

This is to certify that the

dissertation entitled

Restricted Cache Scheduling

presented by

Stephen Wagner

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Major professor

Date July 10, 2001

Restricted Cache Scheduling

By

Stephen Wagner

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTORATE

Department of Computer Science and Engineering

2001

Abstract

Restricted Cache Scheduling

By

Stephen Wagner

This dissertation studies the caching problem in a *restricted* cache where each memory item can be placed in only a restricted subset of cache locations. Examples of restricted caches in practice include victim caches, assist caches, and skew caches. Restricted caches differ fundamentally from traditional set-associative caches because the exact location that an item is placed in the cache is important. This difference greatly complicates the scheduling problems for restricted caches.

We first consider the off-line restricted cache scheduling problem. This problem is closely related to restricted interval scheduling. We show that any interesting version of restricted interval scheduling is **NP**-complete and also **APX**-complete, meaning that there is a limit to how well the problem can be approximated unless $\mathbf{P} = \mathbf{NP}$.

We then study the on-line restricted cache scheduling problem. We focus on companion caches, the simplest restricted cache. Companion caches include victim caches and assist caches as special cases. We show that the commonly studied Least Recently Used (LRU) algorithm is not competitive unless cache *reorganization* is allowed

while the performance of the First In First Out (FIFO) algorithm is competitive but not optimal. We present two near optimal algorithms for this problem as well as lower bound arguments. We also extend these results to more complicated restricted caches, such as the skew cache.

Finally we present a model to measure the flexibility of cache designs as a means to compare different cache designs independent of scheduling algorithms and input sequences. The model is based on the concept of working sets, and the probability that working sets of different sizes will fit into a given cache.

To my parents.

ACKNOWLEDGMENTS

I would like to thank all of the people who have helped me throughout the years of graduate school, and who made this dissertation possible.

First I would like to thank my advisor, Dr. Eric Torng, who always had good ideas and even better feedback, and who was more than patient with my sometimes less than diligent manner. I would also like to thank my committee members, Dr. Enbody, Dr. Esfahanian, and Dr. Palmer, all of whom were extremely helpful and always willing to discuss my research with me. I would also like to thank Dr. Esfahanian, who as the Graduate Director, always made sure that I had financial support.

I would like to thank Dr. Stockman for his help over the years. He noticed me as an undergraduate student those many years ago, and it is largely because of him that I was able to return to MSU as a graduate student after those long years in limbo.

I would like to thank my fellow student Mark Brehob for bringing the skew cache problem to my attention, and for all of his help and work on the problem. Finding an interesting problem worth working on is one of the great hurdles in completing a dissertation, so I am very grateful that he helped find a problem for me.

Other students that I would like to thank are Patchrawat Uthaisombut and Ryan McFall. I would like to thank Jay Kusler for hiring me as a systems administrator and for all of the interesting conversations about food and life. I would like to thank Cora Fong for proof reading my dissertation. I would like to thank Linda Moore and

all of the secretaries for helping me find and fill out all the necessary forms needed to make it through graduate school. I especially would like to thank Susanna Tellschow for all of the dances and for providing robots and monsters where needed.

My family has always been my most constant source of support. I would like to thank all of my siblings, Peter, Robert, Paul, Mary and Michael and wish them well in their various pursuits. I would like to thank my father, who's financial and educational support made so much possible, and my mother, may she rest in peace, for caring so much about me. Lastly I would like to thank God, who made this and all things possible.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation: Memory Latency and Caches

Modern processors are becoming increasingly faster. A version of Moore's Law says that processor *speed* doubles every 18 months. However, this exponential increase in processor speed does not guarantee that processor *performance* will actually improve. Most instructions executed by the processor need to access memory. If memory speed is not on par with processor speed, the processor will spend much of its time waiting on memory accesses and the advantages of a fast processor will be lost. This is the *memory latency* problem. As a further complication, new applications have increased memory demands dramatically which means that computers require larger and larger main memories to operate effectively. Unfortunately memory technology has not kept pace with processor technology. Even though large memory access speeds are increasing exponentially, the gap between processor speeds and large memory access

speeds is still increasing exponentially. Note, *small* fast memory devices can be built, but it is not feasible to build large fast memories.

Since small fast memories can be built, a common approach for coping with memory latency is to implement the memory of a computer as a *memory hierarchy* consisting of a large, inexpensive, slow main memory, and a relatively small, expensive, high-speed memory known as a *cache*. In the most basic memory hierarchy, there is a single cache located between the main memory and the processor. When the processor requests a memory word, it first looks for the word in the cache. If the word is in the cache, it is returned immediately to the processor and the processor can continue executing instructions without delay. Ideally nearly every memory reference will be a cache hit, in which case the memory system will behave like a single large fast memory.

Until now, traditional cache designs and scheduling algorithms have sufficiently reduced memory latency so that it has not been the significant bottleneck in processor performance. However, as the gap between processor speed and large memory access speed increases, better cache designs and better cache scheduling algorithms are needed to ensure that increases in processor speed result in corresponding increases in processor performance.

This dissertation studies the scheduling of *restricted* caches, a general class of cache designs that encompasses many of the new, and proposed non-standard cache designs being considered by the architecture community. Until now, the scheduling problems related to this class of cache designs have not been studied by the theory community.

1.2 Overview of Caches

A cache is a small, high speed memory. Whenever the processor requests a word from memory, it first looks for the word in the cache. If the word is in the cache, it is returned immediately to the processor. This is referred to as a *cache hit.* If the word is not in the cache, it is called a *cache miss.* When a cache miss occurs, main memory is accessed and the requested word is brought into the cache. In order to make room for the newly accessed word, a word currently in the cache must be evicted. A *cache miss* may result in the processor stalling until the memory access can be completed. The percentage of memory accesses that result in cache misses is known as the *miss rate.*

The reason caches work is because typical programs exhibit *locality of reference,* which includes both *temporal locality* and *spatial locality.* Temporal locality is the tendency to reuse recently accessed memory words. Spatial locality is the tendency to use memory words that are "physically" close to recently accessed memory words. A cache takes advantage of temporal locality by keeping recently accessed words in the cache. To take advantage of spatial locality, when a cache miss occurs, a block of memory, typically 8 to 16 words, is brought into the cache instead of a single word. All of the words in the block are likely to be accessed in the near future, and because the entire block was brought into the cache, each of these accesses will be a cache hit. Blocks are the smallest unit of memory in a caching scheme. We will henceforth use the term *item* to describe the smallest unit of memory. A result of locality of reference is that programs use only a relatively small number of items for relatively

long periods of time. The set of items currently needed by the program is known as the *working set.* If the working set can be placed in the cache, the program will not generate any cache misses until the working set changes.

The *size* of a cache is defined as the number of items the cache can hold. A cache of size k has k distinct locations in which to place items. Because every memory access involves searching the cache to determine if the item is currently in the cache, the placement of items in the cache is typically restricted. The *cache design* determines where each item can be placed in the cache. In general, each item can only be placed in a subset of the cache locations. We will defer the discussion of cache designs to Section 1.4, and first discuss the cache scheduling problem.

1.3 The Cache Scheduling Problem

The overall goal of a cache is to reduce memory latency. A major component of latency is the miss rate. The goal of a cache scheduling algorithm is to minimize the miss rate. However, a cache scheduling algorithm has no control over which items will be accessed in the future, and has to rely largely on locality of reference to achieve a low miss rate. The only control the algorithm has is deciding where in the cache to place an item, and consequentially which item to evict from the cache. A cache scheduling algorithm is also limited by the cache design. The cache design restricts where in the cache an item can be placed. This in turn limits the choice of item to be evicted when a cache miss occurs. The algorithm can only evict items that are occupying locations in which the newly accessed item can be placed.

We formally define the cache scheduling problems as follows:

Definition 1.3.1. Cache Scheduling:

INSTANCE: We are given a cache of size k, a set of memory items R, and a function $g : R \rightarrow 2^M$, *where M is the set of cache locations. We are also given a sequence of memory requests,* $s \in R^*$.

GOAL: Find a schedule, that is an assignment of items in s to legal cache locations, with the fewest cache misses.

The function g indicates where each item from R can be placed in the cache and is determined by the cache design. A schedule specifies where in the cache each item from s is placed. This is the optimization version of cache scheduling. The problem can also be phrased as a decision problem.

A scheduling algorithm needs to consider both where in the cache to place the newly accessed item, and which item to evict. Intuitively a newly accessed item should be placed in a location where it will not conflict with other needed items, and the evicted item should be one that will not be referenced again soon. Solving this problem optimally requires knowledge of future requests. Optimal cache scheduling is therefore an *off-line* problem. The entire request sequence s must be known before making any scheduling decisions. In a real system, cache scheduling is an *on-line* problem. The decision about which item to evict must be made before the next access occurs. The algorithm can only see a single element of the request sequence s at a time. Both the off-line and on-line versions of cache scheduling are interesting from a theoretical and practical viewpoint.

Cache scheduling can be viewed as a machine scheduling problem if we think of each cache location as an individual machine, and the memory requests as jobs. We can then take advantage of the scheduling terminology used to describe the types of machines available. If the cache locations are *identical*, then any item can be placed in any cache location. In the terms of definition 1.3.2, the cache locations are identical if $\forall x \in M, g(x) = C$. If the cache locations are *restricted*, then each item can only be placed in a subset of the locations. If the cache locations are identical, the decision about where to place an item is no longer important, and we only need to consider which item to evict.

1.3.1 Cache Bypassing and Interval Scheduling

We typically assume that when a cache miss occurs, the memory item accessed must be placed in the cache. If *bypassing* is allowed, the item may be fetched directly from secondary memory and need not be placed in the cache. This still qualifies as a cache miss and will increase memory latency. However, this is advantageous if it is known that the item is not going to be referenced again soon. Consider the case where everything currently in the cache will be referenced again, and the newly accessed item x will not be accessed again. If we place the item X in the cache, we must evict some other item Y, which will result in a second cache miss when the evicted item Y is accessed again. If item X bypasses the cache, there is only a single cache miss. Whether of not bypassing is actually permitted, it is worth studying the bypassing

case, as it gives us an upper bound on possible cache performance for a given cache design.

When bypassing is used, it is clear that in the optimal solution an item should only be put in the cache if it will remain in the cache until its next reference. With this in mind, we can view the problem of finding an optimal cache scheduling algorithm for a cache of size k as an *interval scheduling problem* on k parallel machines. We formally define the problem as follows:

Definition 1.3.2. Interval Scheduling:

INSTANCE: We are given a set M of machines, a set I of intervals, and a mapping $g : I \rightarrow 2^M$ which determines on which machines each interval can be scheduled. Each interval i has a fixed start time s_i and a fixed processing time p_i. Interval i must be assigned to a machine immediately at time s_i, and it must be continuously processed until it is complete, or else it is lost. A machine can only process one interval at a time. In other words, overlapping intervals cannot be scheduled on the same machine.
GOAL: Find a schedule that processes the maximum number of intervals.

The machines correspond to the cache locations. An interval is defined by successive references to the same memory item. Figure 1.1 shows an example request sequence and the corresponding intervals. Placing an item in the cache is only profitable if the item will still be in the cache the next time it is referenced. Likewise, scheduling an interval on a machine is only profitable if the machine is not otherwise busy during that interval. The schedule that loses the fewest intervals corresponds to a cache schedule with the fewest cache misses.

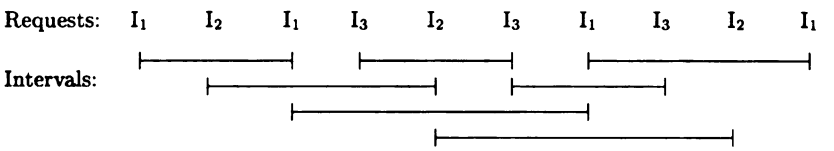

Figure 1.1: A Request Sequence and the Corresponding Intervals

In order to exactly model cache scheduling, we need to add an additional constraint to the interval scheduling problem. Consider item I_1 in the request sequence depicted in Figure 1.1. The first three references to I_1 define two *adjacent* intervals. Intervals are considered adjacent if they share an endpoint and correspond to the same memory item. In a regular interval scheduling problem it would be legal to schedule these intervals on different machines. However, this would be equivalent to item I_1 being in one cache location between its first and second reference, and in a different cache location between its second and third reference. In general items are not allowed to freely move within the cache. To correctly model caching, we can only schedule two adjacent intervals if they are scheduled on the same machine. The general interval scheduling problem though, is still of theoretical and practical interest.

1.4 Traditional Cache Designs

The cache design determines where in the cache specific items can be placed. The key difference between different designs is how freely items can be placed in the cache. Despite the apparent differences, cache scheduling for all traditional cache designs is equivalent to scheduling a cache with identical cache locations.

1.4.1 Set Associative Caches

There are several traditional cache designs, but all of these can be considered under the general classification scheme of *set associative* caches. In an m-way set associative cache of size k, the k locations are divided into k/m disjoint sets. Each memory item can only be placed in one set of cache locations, but the item can be placed in any of the m locations within that set. A simple function such as the modulo operator is used to determine which set of cache locations an item can be placed into. A *fully associative* cache can be thought of as a k-way set associative cache of size k; that is, any item can be placed in any cache location. In a fully associative cache, all of the cache locations are identical. On the other hand, a *direct mapped* cache can be thought of as a one-way set associative cache; that is, any item can be placed in exactly one cache location. Typical values of m in practice are 2, 4, and 8, and m is referred to as the associativity of the cache. We will use the notation SA(m,n) to refer to an m-way set associative cache of size mn. SA(1,n) is a direct mapped cache of size n, and SA(n,1) is a fully associative cache of size n.

We illustrate the effect of different associativities with the following simple example. Suppose we have a cache of size six and that the following six items are repeatedly referenced: I_1, I_2, I_4, I_7, I_8, and I_9. The associativity of the cache design will determine how many of these items can simultaneously fit in the cache. This is illustrated in Figure 1.2. Clearly all of the items can be fit into a fully associative cache as there are 6 items, the cache has 6 locations, and each item can be placed in any cache location. It is worth noting that the items can be arranged in any order in

the cache. The exact location of each item is unimportant. In a direct mapped cache, each item's location in the cache would be determined by the item number modulo 6. Item I_1 and item I_7 share the same location in the cache, so they both cannot simultaneously be in the cache. This is known as a *conflict.* Likewise items I_2 and I_8 cannot both be in the cache because of a conflict. Only 4 of the 6 items can be placed in the cache simultaneously leaving two of the locations in the cache unused. In a two-way set associative cache there would be a total of 3 sets of size two. The set into which each item is placed would be determined by the item number modulo 3. Items I_2 and I_8 would both be placed in set 2, and item I_9 would be placed in set 0. Items I_1, I_4, and I_7 would all have to be placed in set 3. However, each set can hold only two items, so it is impossible to put all three items into the cache simultaneously. Only 5 of the 6 items can be placed in the cache simultaneously leaving one of the cache locations unused.

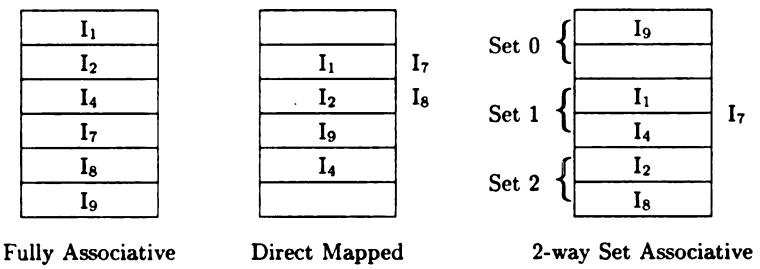

Figure 1.2: Example of Item Placement in Different Cache Designs

With respect to flexibility of placement, the fully associative cache is the ideal cache design. Unfortunately this flexibility is not easily realizable. Finding an item in a fully associative cache requires all cache locations to be quickly searched in

parallel. This search has to be done on every memory access, and must be on par with processor speed, or else the advantage of a cache is lost. It is impossible to build a fully associative cache that is both large enough and fast enough for a modern processor. On the other extreme, we have the direct mapped cache. The advantage of this design is that it is simple and fast. An item's location in the cache can be found simply by looking at the low order bits of the item's location in main memory. The disadvantage of the direct mapped cache is that the inflexibility of placement may result in a higher miss rate. To balance flexibility of placement and simplicity of implementation, most caches in practice are set associative with associativity 2, 4, or 8 rather than direct mapped or fully associative.

1.4.2 Cache Scheduling for Set Associative Caches

In a set associative cache, the locations are not identical. Each item can only be placed in a subset of the cache locations. However, the set associative cache scheduling problem reduces to scheduling a cache with identical cache locations, i.e. a fully associative cache. When discussing cache scheduling for traditional set associative caches, it is important to note that set associative caches satisfy the following decomposition property. For a traditional m-way set associative cache, any sequence of requests can be decomposed into k/m independent subsequences of memory requests. Specifically, for each of the k/m sets of locations in the cache, the subsequence of requests which can be placed into the locations in that set forms one of the independent subsequences. Thus, scheduling an m-way set associative cache reduces to scheduling

k/m fully associative caches of size m, so we can focus on the problem of scheduling fully associative caches.

In general, a cache scheduling algorithm, also known as a replacement policy, needs to consider both the location where an item should be placed, and which item should be evicted. However, as stated in Section 1.3, cache scheduling for fully associative caches is simplified by the fact that the cache locations are identical and the only thing that need be considered is which item should be evicted. Intuitively the optimal choice is to evict the item whose next access is farthest in the future. This is known as Belady's algorithm [8] and is often simply referred to as OPT.

Belady's algorithm is an off-line algorithm. It can only be accurately computed if the entire sequence of memory requests is known at the beginning. A commonly used on-line scheduling algorithm is Least Recently Used (LRU). LRU evicts the item that was least recently accessed. The intuition behind LRU is that when temporal locality is in effect, the near future looks like the recent past. Therefore, LRU is a good approximation of Belady's optimal algorithm. Random replacement policies are also used in practice. The advantage to random is that it is fast and easy to implement. LRU requires storing additional information, and the improved performance may not be worth the extra effort. Another common replacement policy is First In First Out (FIFO). FIFO approximates LRU and can be implemented more easily.

The interval scheduling problem that corresponds to fully associative cache scheduling is also simplified by the fact that location is not important. When the machines are identical, the optimal interval scheduling algorithm is earliest end time first [27]. Also,

the constraint on adjacent intervals is no longer relevant, so in this case the normal interval scheduling problem is an exact model of fully associative cache scheduling.

1.5 Restricted Caches

While set associative caches have been fairly successful in coping with memory latency, new cache designs are needed to bridge the widening gap between processor speeds and memory access speeds. In this section we describe a new and promising class of cache designs which we call restricted caches.

The basic memory hierarchy consists of two levels, the cache and main memory. This hierarchy can be generalized to include several levels of caches. This is known as a *multi-level cache.* Multi-level caches are one approach to improving cache performance. Another approach is a *multi-lateral cache.* A multi-lateral cache design consists of two or more cache structures, of possibly different types, at the same level in the memory hierarchy. A given item may be in any of the component cache structures and each will be searched, in parallel, when a memory reference occurs. This is the class of cache designs that we will call *restricted caches.* We further limit the definition of a restricted cache to *not* include those degenerate cases where the combined cache structures are the equivalent of a set associative cache.

Restricted caches are interesting not only because of their potential to reduce memory latency and improve processor performance, but also because they also introduce interesting and challenging scheduling problems. In an m-way set associative cache, the only relevant factor is which item to evict from the cache: it does not

matter in which of the legal m locations we put the new item. In a restricted cache, not all locations are created equal: one location may see much more conflict than another. Both off-line and on-line algorithms must consider where items are placed in the cache, not just which item is evicted.

Several restricted cache designs have been proposed to cope with the increasing gap between processor and memory speeds. Two of the more interesting designs are the *victim cache* [32] and the *m-way skewed associative cache* [51, 49, 50]. We will study generalized versions of these restricted caches.

1.5.1 The Companion Cache Structure

Direct mapped caches and set associative caches both have the disadvantage that they may not be able to utilize all the locations in the cache due to conflicts. This was illustrated in Figure 1.2. However, the number of conflicts at any one time is likely to be small. This is the intuition behind many restricted cache designs. We generalize this idea as the Companion Cache Structure (CCS(m, n)). A CCS(m, n) consists of a direct mapped cache of size m paired with a fully associative cache of size n. Each item can be stored in one of the locations of the direct mapped cache, and any of the locations in the fully associative cache. We will refer to the small fully associative cache as the *companion buffer*, and to the direct mapped cache as the *main cache*. The companion buffer provides extra locations in which to store conflicting items. Two items which conflict in the direct mapped cache can simultaneously fit in the cache if one of the conflicting items is placed in the companion buffer. The

direct mapped cache can be large and still very fast. The companion buffer must be equally fast, because on every memory request, both caches must be searched for the requested item. Therefore, the companion buffer will be very small, even as small as one item. However, because the number of conflicts at any one time is likely to be small, a small companion buffer can be quite effective in reducing the miss rate.

In a CCS(m, n), location is important. Consider the example in Figure 1.3. The CCS$(6, 2)$ is based on the direct mapped cache in Figure 1.2. Items I_1 and I_7 conflict in the main cache, and items I_2 and I_8 conflict in the main cache. If I_1 or I_7 and I_2 or I_8 are put in the companion buffer, then all of the items will fit into the cache. However, if some other item such as I_9 is put in the companion buffer, then the items will not all fit into the cache. In a traditional set associative cache, a set of items either fits in the cache or it does not, regardless of exactly where each item is placed in the cache.

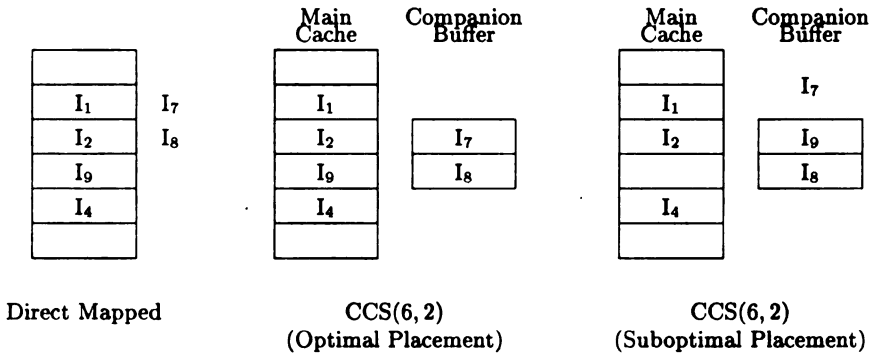

Figure 1.3: Example of Item Placement in a CCS$(6, 2)$

The definition of a CCS(m, n) does not include a replacement policy. We will consider general scheduling algorithms for this particular cache design. Many of the

proposed restricted cache designs are a CCS(m, n) with a specific replacement policy. One example is the *victim cache* [32]. In the victim-caching scheme, a regular direct mapped (or set associative) cache is supplemented with a small fully associative cache called the victim cache, and the following replacement policy is used. When a cache miss occurs, the item is retrieved from the main memory and placed in the direct mapped cache. However, the item which is evicted is not moved into main memory, rather it is placed into the victim cache. This will cause an item to be evicted from the victim cache. Also when a hit occurs in the companion buffer, the item is moved into the direct mapped cache. This is an example of cache reorganization, which will be explained in Section 1.5.3. This replacement policy is identical to the way multi-level caches are managed. The key difference is that the direct mapped cache and the victim cache are searched in parallel. Another example of a CCS(m, n) with a specific replacement policy is the *assist cache* implemented on the PA7200 [13]. In the assist cache, items are first put into the companion buffer, and then moved to the direct mapped cache or evicted altogether.

1.5.2 The Skew Associative Cache

An m-way set associative cache can be thought of as m direct mapped caches of size k/m, where k is the total cache size. The m direct mapped caches are called *banks*. A given memory item can be placed in the same position in each of the m banks. In other words, the same mapping function is used for each bank. Figure 1.4 illustrates this idea for a 2-way set associative cache, and compares it to the 2-way set associative

cache depicted in Figure 1.2. The mapping function f determines where each item goes in both banks. A small modification to this design results in an m-way skewed associative cache, also known as a *skew cache.* Instead of using a single mapping function, a different function is used for each bank. A 2-way skewed associative cache is illustrated in Figure 1.5. In this case two different functions, f_0 and f_1 are used, and a given item may be placed in different locations in different banks.

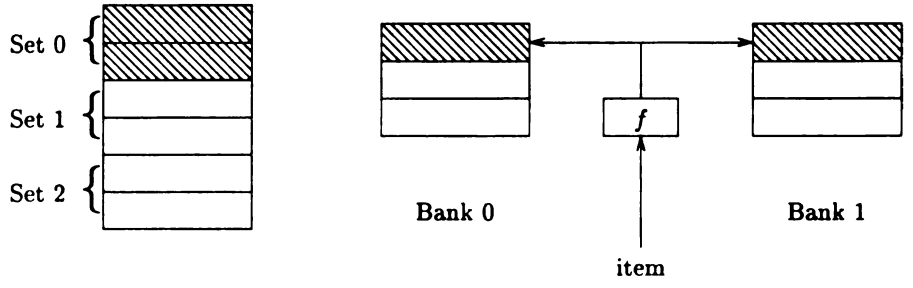

Figure 1.4: Two different views of a 2-way set associative cache.

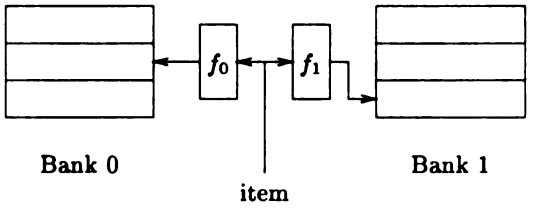

Figure 1.5: A 2-way skew associative cache.

The key difference between a skew cache and a set associative cache is the following: In an m-way set associative cache, items that conflict with each other in one bank will conflict with each other in all banks. In a skew cache, items that conflict in one bank will not necessarily conflict with each other in another bank. This difference is the intuition behind the skew cache design, and is illustrated in Figure 1.6. In the example from Figure 1.2, it is not possible to fit all three items I_1, I_4, and I_7 into a

set associative cache because they all conflict with each other and the sets are of size two. In a skewed associative cache, even though the items conflict with each other in one bank, they may not conflict with each other in another bank. For example, in Figure 1.6, items I_1, I_4, and I_7 all conflict in the first bank, but only items I_1 and I_4 conflict in the second bank because a different mapping function is used.

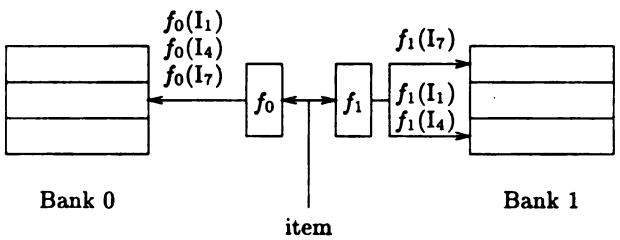

Figure 1.6: Different banks, different conflicts.

The fact that items do not necessarily conflict with each other in all banks in a skew cache means that the issue of placement in a skew cache is relevant. For example, consider Figure 1.6 again. If item I_7 is placed in the first bank, then only two of the three items, I_1, I_4, and I_7 can simultaneously be in the skew cache. However, if item I_7 is placed in the second bank, then all three of the items, I_1, I_4, and I_7 can simultaneously be in the skew cache.

A skew cache requires only slightly more hardware than a set associative cache and will be nearly as fast. The hardware cost of an m-way skewed associative cache is approximately the same as the cost of an m-way set associative cache [49]. Due to the fact that a skew cache is better able to utilize the entire cache, it may perform better than a set associative cache of the same size. Simulation results in [50] show that a

2-way skewed associative cache achieves a miss rate similar to a 4-way set associative cache.

1.5.3 Restricted Cache Scheduling

Restricted cache scheduling is the general cache scheduling problem described in Section 1.3. Restricted Cache Scheduling differs from traditional cache scheduling. This is because a scheduling algorithm for a restricted cache needs to consider both where to place an item and which item to evict. As we will show, this greatly complicates the scheduling problem. For example in Chapter 3, we will show that Belady's algorithm, which is optimal for set associative caches, is not optimal for restricted caches.

Restricted cache scheduling also needs to consider *reorganization.* Because location is important, it may be beneficial to reorganize the cache by moving items from one location to another when a hit occurs. Reorganization must be supported by the cache hardware. The proposed design for the victim cache allows items to be moved between the two components of the cache whenever there is a miss or there is a hit on the companion buffer. Reorganization does not make sense in a traditional cache. We will consider algorithms that allow for reorganization in our analysis of restricted cache scheduling.

Related to the reorganization is the fact that restricted caches have an adaptive nature with regards to working sets [50, 10]. In a sense, the goal of cache scheduling is to fit the current working set into the cache. In order to to this, the items in the working set need to be placed in the cache in such a way that they will not interfere

with other items in the working set. When the working set changes, items may need to be moved within the cache to accommodate the new working set. When the cache locations are identical as they are in a traditional cache, position is irrelevant and moving items within the cache is not beneficial. However, in a restricted cache, a scheduling algorithm may move an item from one cache location to another, by first evicting it and then re-accessing it, in order for adjust to a new working set. Some restricted caches explictly allow reorganization when a hit or miss occurs. This is another feature not seen in traditional cache designs.

1.6 Methodology

In this section we define some of the methods that will be used to analyze cache scheduling problems and algorithms.

1.6.1 Off-line Methodology

Many off-line scheduling problems are **NP**-hard [25]. There are no known efficient solutions for **NP**-hard problems and the running times of the optimal algorithms are typically exponential with respect to the size of the input. For this reason, it is necessary to find polynomial time approximation algorithms for **NP**-hard optimization problems. We say that a polynomial-time algorithm A is an *ρ-approximation algorithm* if for all inputs x we have

$$\max\left(\frac{A(x)}{OPT(x)}, \frac{OPT(x)}{A(x)}\right) \leq \rho$$

where ρ is a constant, $A(x)$ is the value returned by A on input x and OPT(x) is the optimal value for input x. For a minimization problem like cache scheduling, a ρ-approximation algorithm returns solutions that never have more than ρ times the optimal number of page faults. The class **APX** is the class of problems that have ρ-approximation algorithms [31].

An *polynomial-time approximation scheme* (PTAS) is a family of approximation algorithms. If a PTAS exists for a problem P, then for any $\epsilon > 0$, there is an $(1+\epsilon)$-approximation algorithm for P, though the running time may be exponential in $\frac{1}{\epsilon}$. A *fully polynomial-time approximation scheme* (FPTAS) is an approximation scheme where the running times of the algorithms are polynomial with respect to $\frac{1}{\epsilon}$.

Ideally we would like to find a PTAS for any **NP**-hard problem. However, for some problems it is known that a PTAS does not exist unless **P** = **NP**. The class **APX**-complete is a class of optimization problems that do not have polynomial time approximation schemes unless **P** = **NP** [2]. Proving that a problem is **APX**-complete is similar to proving that a problem is **NP**-complete. To prove that a problem P is **APX**-complete, we first need to show that P is in **APX**. We then need to reduce a known **APX**-complete problem Q to P using an approximation preserving reduction, such as an L-reduction [47]. An L-reduction consists of two polynomial time functions R and S with the following properties. For any instance I of Q with optimum cost $\mathcal{OPT}(I)$, $R(I)$ is an instance of P with optimum cost $\mathcal{OPT}(R(I))$, such that

$$\mathcal{OPT}(R(I)) \leq \alpha \cdot \mathcal{OPT}(I),$$

for some positive constant α. Also, for any feasible solution s of $R(I)$, $S(s)$ is a feasible solution of I such that

$$OPT(I) - c(S(s)) \leq \beta \cdot (\mathcal{OPT}(R(I)) - c(s)),$$

for some positive constant β, where $c(S(s))$ and $c(s)$ denote the costs of $S(s)$ and s respectively. Because an L-reduction is an approximation preserving reduction, if a problem P has a PTAS and there is an L-reduction from Q to P, then the problem Q also has a PTAS.

1.6.2 On-line Methodology

An on-line algorithm is not likely to find an optimal solution due to the lack of future knowledge. For example, an on-line cache scheduling algorithm does not know when an item will next be referenced and may inadvertently evict a needed item from the cache. We therefore need some metric to measure the quality of an on-line algorithm. Absolute metrics such as the number of cache misses, however, are not very meaningful. For example, for any cache scheduling algorithm there exists a sequence of memory requests that will result in a cache miss every time.

A standard method of analyzing on-line algorithms is to use *competitive analysis.* Competitive analysis compares how an on-line algorithm performs against the optimal off-line algorithm on all input instances. The *competitive ratio* $c_{\mathcal{A}}$ of an on-line

algorithm $\mathcal{A}$ is defined as

$$c_{\mathcal{A}} = \sup_{I} \frac{\mathcal{A}(I)}{\mathcal{OPT}(I)}$$

where $\mathcal{OPT}$ denotes the optimal off-line algorithm, and sup is the *supremum*. The supremum of a set is the smallest number that is as large or larger than all numbers in the set. An on-line algorithm is within a factor of $c_{\mathcal{A}}$ of optimal on all input instances. A *c-competitive* algorithm has a competitive ratio of c. An algorithm is said to be *strongly competitive* if it has the smallest possible competitive ratio.

Lower bound proofs on the competitive ratio for an on-line problem often make use of an *adversary*. The adversary constructs the input sequence "on-line" in response to the on-line algorithm's decisions in such a way that results in the worst competitive ratio. For example, the adversary in the caching problem could always request the item most recently evicted from the cache. Because an on-line algorithm cannot see the future, there is no difference between an input sequence that is being constructed on-line and one that is determined from the beginning.

1.7 Overview

The focus of this dissertation is Restricted Cache Scheduling. The remainder of this dissertation is organized as follows.

In Chapter 2 we discuss previous work related to cache scheduling and interval scheduling. There has been considerable work done on traditional cache scheduling,

but to the best of our knowledge restricted cache scheduling has not previously been studied. We also discuss previous work on interval scheduling and related scheduling problems.

In Chapter 3 we analyze the offline restricted cache scheduling problem. This includes hardness results, optimal solutions, and approximation algorithms for the different problems. The off-line restricted cache problem is significantly more difficult than identical cache scheduling. Whereas the latter is optimally solvable in $O(n \lg n)$ time using Belady's algorithm, the former is **NP**-complete and **APX**-complete, as are the related interval scheduling problems.

In Chapter 4 we analyze the on-line restricted cache scheduling problem. This includes upper and lower bounds for different on-line algorithms and different cache designs. Again the on-line restricted cache scheduling problem differs greatly from traditional on-line restricted cache scheduling. For the traditional on-line cache scheduling problem, LRU and FIFO are both strongly competitive. We will show that LRU is not competitive for restricted cache scheduling, and that FIFO is competitive but not necessarily optimal.

In Chapter 5 we present a mathematical model to measure the flexibility of a cache design. This is an attempt to directly compare different cache designs, independent of scheduling algorithms and input sequences.

In Chapter 6 we summarize the dissertation and discuss future work.

Chapter 2

Related Work

This chapter describes previous work done in cache scheduling, interval scheduling and other related scheduling problems.

2.1 Cache Scheduling

To the best of our knowledge, restricted cache scheduling has not been studied by the theory community. There has been extensive work done on scheduling for fully-associative caches.

2.1.1 Off-line Cache Scheduling

For traditional cache designs, the off-line cache scheduling problem is well studied and easily solved using Belady's algorithm [8]. Belady's algorithm simply evicts the item in the cache whose next reference is the farthest in the future. This is optimal for both set-associative and fully-associative caches.

2.1.2 On-line Cache Scheduling

Caching is one of the most natural on-line problems and has been extensively studied by the theory community. Much of the work actually considers the *paging problem*, which is closely related to cache scheduling. The memory hierarchy can include secondary storage, in order to create a virtual memory much larger than the actual physical memory. As in cache scheduling, the goal is to keep the information that will needed soon in main memory. The unit of memory that is moved between main memory and secondary storage is known as a *page*, and is typically on the order of 16KB. A *page fault* occurs when the processor attempts to access a page that is currently in secondary storage. The goal of a page scheduling algorithm is to minimize the number of page faults. To a large degree, caching and paging are the same problem from a theoretical point of view.

The Least Recently Used (LRU) replacement policy is k-competitive for a fully associative cache size of k [52, 19, 54]. LRU belongs to a class of algorithms known as *marking algorithms*, which are all k-competitive. A marking algorithms works as follows. The input sequence is divided into *phases*. Each phase contains k distinct items. Initially all pages are unmarked. When an item is accessed it is marked. When the $k + 1^{st}$ distinct item is accessed, all the marks are cleared. This is the end of the phase. A new phase then begins, and the newly accessed item is marked. The key idea behind a marking algorithm is that it never evicts a marked item from the cache. LRU fits this description. If there are both marked and unmarked items in the cache, clearly the marked items were used more recently than the unmarked items,

so LRU will not evict a marked item. If all the items in the cache are marked, and a cache miss occurs, then a new phase begins, all the marks are cleared, and whatever item LRU evicts is not marked. First In, First Out (FIFO), is another example of a marking algorithm.

We briefly outline why marking algorithms are k-competitive. First, during each phase, the optimal off-line algorithm must incur one cache miss. A phase ends after $k+1$ distinct items have been accessed. It is not possible to keep $k+1$ items in the cache, so there must be at least one cache miss. A marking algorithm will incur no more than k cache misses during a phase. Whenever a cache miss occurs, the item newly placed in the cache is marked. Because marked items are never evicted, at most k cache misses can occur during a phase. It follows that marking algorithms are k-competitive. A detailed proof can be found in [54].

It turns out that marking algorithms are optimal for deterministic algorithms. No deterministic on-line algorithm can be better than k-competitive. For any deterministic on-line algorithm A, the adversary can construct a request sequence that always requests the item that A just evicted. Clearly A will miss on every access. The optimal algorithm will only page fault after every k distinct items are accessed. Therefore, no deterministic on-line algorithm can be better than k-competitive. A detailed proof can be found in [52] and [54].

This theoretical analysis of caching does not match observed results very well. First, in practice on-line caching algorithms perform much better than the competitive analysis suggests. Second, algorithms with the same competitive ratio perform differently in real systems. In particular, LRU usually performs much better than

FIFO in a real system, despite the fact that they have the same competitive ratio. Most of these problems stem from the fact that competitive analysis is a worst case analysis, and does not properly consider locality of reference. The whole concept of a memory hierarchy is built on the assumption that we can take advantage of locality of reference. For this reason, researchers have focused on ways to better model locality of reference.

In [54] the author analyzes the performance of marking algorithms on request sequences that exhibit significant locality of reference. This is formally defined as request sequences where the length of the phases are large with respect to k. Real programs tend to display this behavior, accessing only a small set of items for relatively long periods of time. When the request sequences are restricted in this way, marking algorithms have a constant competitive ratio. A different approach to modeling locality of reference is the *access graph* [9, 18, 14, 30]. The vertices of the access graph G represent the memory items. A directed edge between vertices x and y indicates that memory item y can be accessed after memory item x. A legal sequence of accesses is defined by a walk on G. In the most general case, G is a complete graph and any item can be referenced at any time. Using access graphs it is possible to prove that LRU outperforms FIFO, and to find competitive ratios better than k.

Randomized caching algorithms have also been studied [19, 18, 43]. The adversary is not as effective against a randomized algorithm because it does not know the outcome of the random choices. Randomized marking algorithms are $2H_k$-competitive, where H_k is the kth harmonic number. A strongly H_k-competitive randomized caching algorithm is described in [43].

Caching is a restricted version of the k-server problem [42, 36]. In the k-server problem there are k servers and a sequence of requests. The servers and requests both reside in the same metric space. When a request appears, one of the servers must move to the request in order to serve it. The goal is to minimize the total distance moved by the servers. In caching, each memory item defines a point in the metric space, and all points are distance one from each other. The servers are the cache locations, and putting an item in a cache location is equivalent to moving the corresponding server to the corresponding point in the metric space.

2.2 Interval Scheduling

There has been a considerable amount of work done on interval scheduling, also known as fixed job scheduling problems. In the literature there is not a consistent naming convention for these problems; the same problems are given different names and the same name is used to refer to different problems. The common feature in these problems is that the jobs have a fixed starting time, a fixed length, and a job must be processed continually for fixed amount of time, or else it is lost. In some problems there is an interval during which a job must start, but we are primarily interested in the case when the starting times are fixed.

There are two main types of interval scheduling problems. In the first type of problem, there is a fixed number of machines, and the goal is to determine if all of the jobs can be scheduled on the available machines, or to maximize the number of jobs that can be scheduled. This is the problem described in Definition 1.3.2 and is the

interval scheduling problem most closely related to cache scheduling. We will simply call this problem interval scheduling (IS). It may be the case that some jobs have a higher priority than others, in which case the jobs are weighted, and the goal is to maximize the weight of the scheduled jobs. In the second type of problem, there is an unlimited pool of machines and the goal is to find a set of machines with minimal cost that can process all of the jobs. We will call this problem tactical interval scheduling (TIS). TIS typically has different types of machines that have different costs. The unweighted maximization version of IS is the problem most relevant to our research, and unless otherwise noted, this is the problem we are referring to when we use the term interval scheduling.

We will use terms from traditional scheduling to describe the types of machines available. If the machines are *identical*, then any job can be scheduled on any machine, and the weight of the job is independent of the machine on which it is run. It is worth noting that in interval scheduling problems, the concept of machine speed is not relevant. Jobs have a fixed start time and end time which does not depend on the machine on which the job is run. All machines operate at the same speed. In this case the goal of TIS is to find the smallest number of machines necessary to process all jobs. If the machines are *restricted*, then a given job can only be scheduled on a subset of the machines. Again all of the machines have the same speed. We will refer to these problems as the restricted interval scheduling problem (RIS) and the restricted tactical interval scheduling problem (RTIS). Often the jobs and machines are divided up into classes, and jobs of certain classes can only be scheduled on a subset of the machine classes. This is particularly useful in the RTIS as it allows

different costs to be assigned to the different classes of machines. If the machines are restricted and the jobs are weighted, then the weight is independent of which machine processes the job. It is also possible to define uniform and unrelated machines where the weight of a job depends on the machine on which it is scheduled. However, these definitions to do not seem to correspond to real world problems and rarely appear in the literature.

In the following subsections we look at previous work on the interval scheduling problem on identical machines, the restricted interval scheduling problem, the tactical interval scheduling problem, and on-line and other versions of these problems.

2.2.1 Identical Interval Scheduling

If all the machines are identical and the jobs are unweighted, then interval scheduling is easily solvable in polynomial time. Scheduling the jobs by earliest end time will produce an optimal schedule. This problem can also be represented by an *interval graph* [27]. Each interval is represented by a vertex in the graph, and there is an edge between two vertices if the corresponding intervals overlap. Interval scheduling on k identical machines is equivalent to k-coloring the corresponding interval graph. Interval graphs are a subclass of chordal graphs. An interesting property of chordal graphs is that a chordal graph always has a *simplicial* vertex. A vertex is simplicial if its neighbors form a clique. Clearly a simplicial vertex can always be in a maximum independent set. For this reason, independent set and k-coloring problems which are normally NP-complete are easily solvable in polynomial time for chordal graphs.

The weighted version of IS is considered in [1, 37]. In both cases they reformulate the problem as a minimum cost flow problem; however the underlying graphs are remarkably different. In [1] they use a graph based on the maximal cliques of the underlying interval graph. The vertices of the graph correspond to the maximal cliques and the arcs correspond to jobs with capacities of 1 and cost equal to the weight of the job. In the solution to the minimum cost flow problem, the arcs with non zero flow correspond to the intervals that are not in the solution to IS. The complexity of the algorithm is $O(n^2 \log n)$. The construction of the graph used in [37] is more direct. The vertices of the graph represent all starting and finishing times of jobs in chronological order. For each interval create an arc from the vertex representing the interval's start time to the vertex representing the interval's end time. The capacity of this arc is 1, and its cost is the negation of the intervals weight. Additional zero cost, infinite capacity arcs are added between consecutive vertices. In this case, the arcs with non zero flow correspond to the intervals in the solution to IS. This flow problem can be solved in $O(n^2 \log^2 n)$ time.

In [1] they also show that IS is **NP**-complete if precedence constraints are added. The precedence constraints are pairs of jobs (j_1, j_2) such that j_2 can only be scheduled if j_1 is also scheduled.

2.2.2 Restricted Interval Scheduling

Restricted interval scheduling is first considered in [1]. This is the basic IS problem where each interval can be scheduled on an arbitrary subset of the machines. The

authors show that that RIS is **NP**-complete. The reduction is from 3-SAT and is relatively straightforward. Create a machine for both the positive and negative instance of each variable. For each clause create a short interval that can be scheduled on the machines that correspond to the literals in the clause. The clause intervals are arranged so that none of them intersect. For each variable, create a long interval that intersects with all of the short intervals. Each long interval can be scheduled on the machines corresponding to the positive and negative instance of the variable. Scheduling the long intervals defines a truth assignment. Scheduling a long interval on the "negative" machine is equivalent to assigning "true" to the corresponding variable. It will only be possible to schedule all of the short intervals if one of the variables in each clause is true. The authors also show that the weighted problem can be solved optimally in $O(n^{m+1})$ time. Their solution involves constructing a DAG such that paths through the DAG represent possible schedules. It is essentially a dynamic programming solution.

In [34] the authors refine the complexity results of interval scheduling. They consider what they call the Class Scheduling problem, which is the version of IS where the jobs and machines are divided into classes. In the most general case, each machine and job forms its own class. They show that if there are three or more dependent classes of machines, then deciding if all the intervals can be scheduled is **NP**-complete. A set of machine classes is dependent if the classes of jobs they can process overlap. They also show that if there are two or more dependent classes of machines then the optimization problem is **NP**-hard. The proofs rely on the fact that there are only a small number of different ways that job classes can be assigned

to machine classes when there are only two or three dependent classes of machines. For example if there are only two classes of machines, there must be at least two classes of jobs: jobs that run on one machine class and jobs that run on both machine classes; and there can be at most three classes of jobs: jobs that run on the first machine class, jobs that run on the second machine class, and jobs that run on both machine classes. For each case they use a reduction from N3DM (Numerical Three Dimensional Matching) to show that the particular case is **NP**-complete or **NP**-hard. Note that even though there are only two or three classes of machines, there are many machines that belong to each class. This is a necessary part of the hardness results. In [35] the same authors apply the same techniques to a variation of the problem where the machines are only available for specified intervals and show that in general this problem is **NP**-complete.

The general complexity of IS is also discussed in [12] in the context of classroom assignment. This is an interval scheduling problem where classrooms correspond to machines and the classes are jobs. They consider variations where classes meet several times a week but must be scheduled in the same room, in which case sets of intervals must all be scheduled on the same machine. They also consider the problem where classes can be scheduled in any room that is large enough. This is a special case of restricted interval scheduling where the machines can be arranged in a hierarchy. In general these problems remain hard.

The case where the weights of the intervals are dependent on the machine is considered in [45]. This is the unrelated machines case of interval scheduling. It can be solved as a maximum weight m-coloring of an interval graph. For the case

when intervals are restricted to certain machines but the interval weight is machine dependent, they generalize the result in [1] to show that the problem is solvable in $O(k^{m+1})$ time.

Approximation algorithms for RIS are discussed in [24] and [37]. In [24] the authors formulate the interval scheduling problem as an independent set problem. A graph is defined as in Section 3.3.1. The approximation algorithms take advantage of the clique structure of the graphs. They do not do a worst case analysis of the approximations, but instead provide computational results to show that the approximations are good. They also consider the Variable Start Interval Scheduling problem, where jobs have a starting interval instead of a starting point, and show that their algorithms can be adapted for this problem. In [37] they extend the network flow model they used for identical machines to handle multiple classes of machines. For each class of machine, a separate network graph is created. The graphs are all linked to a common source and sink and additional constraints have to be added to the flow problem to ensure that a job is not processed twice. Their approximation algorithm uses a dual cost heuristic. They use a greedy algorithm to find a lower bound, and they relax the constraint on jobs being processed multiple times to find an upper bound. The two bounds are used in an iterative fashion to refine the approximation. Again, there is no worst case analysis of the algorithm but they provide computational results to show that the approximations are good.

2.2.3 Tactical Interval Scheduling

The tactical interval scheduling problem is first discussed in [26]. The authors call the problem the fixed job scheduling problem. They note that the problem is a special case of Dilworth's problem. Dilworth's problem is the following: given a partially ordered set N, find the minimum chain decomposition of N where a chain is a sequence of elements $i_1, i_2, \ldots, i_k$ such that $i_1 < i_2 < \ldots i_k$. In terms of intervals i and j, $i < j$ if j starts after i ends, and the minimum chain decomposition consists of sets of non-overlapping intervals. The result for TIS is that the number of machines necessary to schedule all jobs is simply the maximum overlap of jobs. To assign the jobs they present a straightforward algorithm that essentially assigns a maximal non-overlapping set of jobs to a machine, removes those jobs, then repeats the process. They also consider what they call the variable job schedule problem. This is TIS but the starting time of a job is defined by an interval. They present two approximation algorithms for the problem. Computational results show that the approximations are very good but no worst case analysis is provided. An exact integer programming solution to VSP is also given.

In [35] they consider TIS and using an analysis similar to their analysis of IS in [34], determine under which conditions the problem is hard. If there are only two types of machines, then the problem can be solved in polynomial time using a combination of linear programming and network flow algorithms. The authors of [16] also consider the case where there are two classes of processors and three classes of jobs and show that it is solvable in polynomial time by reducing it to a network flow

problem. In both cases the networks used are similar to the networks used in [37, 11]. Vertices represent points in time, and jobs are represented by arcs from their start time to the end time. In general if there are more than two classes of machines, then TIS is **NP**-complete[35]. The proof reduces IS to TIS.

Two generalizations of TIS are considered in [20, 21, 22], which they call the Fixed Job Schedule Problem with Spread-Time Constraints and the Fixed Job Schedule Problem with Working-Time Constraints. The spread time for machine i is defined as the time between when the first job assigned to i starts and the time the last job assigned to i ends. The Spread-Time constraints impose an upper bound on the spread time for all processors. The working time for machine i is the sum of the processing time of all jobs assigned to i. Working-Time constraints impose an upper bound on the working time for all processors. Clearly extra machines will usually be necessary to satisfy these constraints. In [20, 21] these two problems are shown to be **NP**-hard. In [22] they present approximation algorithms for both problems. For the spread time constraints a simple greedy algorithm that assigns jobs to available machines and adds new machines if no machines are available uses at most twice as many machines as the optimal solution. A more complicated algorithm that uses an exact solution to the preemptive version of TIS as a starting point also has an approximation factor of 2. For the working time constraints a greedy algorithm with an approximation factor between 2 and 3 exists. An algorithm based on preemption has an approximation factor of 2.

The case when jobs may be preempted and moved from one machine to another with no penalty is considered in [15]. Jobs still have to be processed continually for

a fixed amount of time, but they can migrate from one machine to another. They show that when preemption is allowed, then two cases of RTIS become solvable in polynomial time. If there are m classes of processors and $m+1$ classes of jobs, and a job in class k can be done only by a processor of class $k, k = 1, 2, \ldots, m$ and jobs of class 0 can be done by any processor, then the problem can be solved in polynomial time. Likewise if the processor classes are hierarchical, meaning that processors in class k can perform any job that processors in class $j < k$ can, the problem is also solvable in polynomial time. Both of these problems are **NP**-complete when preemption is not allowed. It is also shown that if preemption is allowed, the decision version of TIS, which is equivalent to the decision version of IS, can be solved in polynomial time by transforming it into a set of transportation problems. However, this method cannot be applied to the optimization problem.

The authors of [37] present a similar study of TIS in [38]. They only consider the case where machines have equal cost. They show that the problem can be represented as a network flow problem as in [37]. One additional constraint has to be added to ensure that the flow in arcs corresponding to jobs is one unit. They present upper and lower bound approximation algorithms similar to those in [37]. The main new result of this paper is the fact that TIS remains **NP**-hard when preemptions are allowed.

A hybrid version of TIS is studied in [33]. In this problem, there are two types of machines, fast and slow. Normally machine speed is not considered in interval scheduling problems. Each job has a fixed start time and requires a fixed amount of time on the slow machines, but the fast machines can process the jobs more quickly. Therefore, the completion time of a job is not necessarily fixed. They consider the

case where jobs have to be started at a fixed time and the case where the fast machines can delay the start of a job. They refer to these cases as fixed starting times (FST) and variable starting times (VST). VST is **NP**-hard in the strong sense even, if all release dates are equal. FST is **NP**-hard in general, but can be solved in $O(n)$ time if the release dates are equal. If the fast machines are able to process jobs in one unit of time, then VST can be solved in $O(n \log n)$ time if all release dates are equal and FST can be solved in $O(n \log n)$ time.

2.2.4 On-line Interval Scheduling

An on-line version of IS is studied in [55]. They consider the single machine case. Job lengths and weights are known when they arrive. Jobs can be preempted, but they are lost. The quality of on-line algorithms depends on the relation between the length of the jobs and the weight of the jobs. If the weights of the jobs have no relation to the lengths of the jobs then no on-line algorithm is competitive. An instance of the problem is called $f - related$ if there is a function $f(x)$ that maps lengths to weights. If $f(x)$ is concave or decreasing then there exists a 4-competitive on-line algorithm. The algorithm is straightforward. A job J_k is scheduled if the machine is idle, or if its value is twice the value of the job currently being processed, or if its endtime is before the endtime of the job currently being processed and its value is greater than the value of the job currently being processed. They show that for concave $f - related$ instances, no algorithm has a competitive ratio better than $4 - \epsilon$, so the straightforward algorithm is optimal.

An on-line extension of the algorithm in [11] is presented in [17]. The k-coloring interval algorithm is nearly an on-line algorithm due to its greedy nature. However, it requires the intervals to be sorted by right endpoint, which is impossible in an on-line setting. They show that a greedy algorithm that considers the intervals in order of left endpoints is also optimal. This algorithm can be applied on-line. The new algorithm is essentially Belady's algorithm applied to intervals.

In the above work, the length of the interval is known when it appears. With regards to caching, this is not truly on-line. Knowing the length of an interval is equivalent to knowing when an item will next be referenced. In the fully on-line version of IS the lengths of intervals are not known until the interval ends. This problem is studied in [41]. They consider the 1 machine case. Their basic results are that if there are only intervals of length 1 and length $k \gg 1$ then a strongly 2-competitive algorithm exists. In the general case they present an algorithm with a competitive factor of $O((\log \Delta)^{1+\epsilon})$ where Δ is the ratio of the longest to shortest interval. They also show that no $O(\log \Delta)$-competitive algorithm can exist. For the instance with only two lengths of intervals, the on-line algorithm does the following. If the machine is free and a length k interval arrives, schedule it. If a length 1 interval arrives, flip a coin and schedule it if the result is heads. Otherwise do not schedule it or any other length 1 interval for the next unit of time. Time is not discrete in this problem. This simple algorithm is strongly 2-competitive. They generalize this for the case of arbitrary interval lengths as follows: if we reject an interval I, then schedule no other interval that begins during I unless it is twice as long as I. Because

time is not discrete in this model, it is not very relevant to caching, where something happens at each time step.

2.2.5 Other Interval Scheduling Problems

A variation of IS is discussed in [53]. In this problem, there is a single machine and n k-tuples of intervals. The goal is to schedule as many intervals as possible such that no two intervals overlap and no two intervals from the same k-tuple are scheduled. They call this the Job Interval Selection Problem (JISP). The "jobs" are the k-tuples of intervals. The graph representation of this problem is an interval graph with additional edges added to indicate the k-tuples. The subgraph induced by these additional edges is a collection of cliques of size k. Using this fact they are able to find an L-reduction from B-OCC-MAX-3-SAT. B-OCC-MAX-3-SAT is a version of SAT where each clause contains exactly 3 literals, and each variable appears at most B times [47]. It follows that no polynomial time approximation scheme exists for JISP. They also show that there is a straightforward greedy 2-approximation algorithm for JISP.

RIS is actually a restricted version of JISP, even though JISP only has a single machine. We can transform any instance of RIS into an instance of JISP where all of the intervals in a job have the same length. Let P be an instance of RIS with m machines, let T be the maximum end time of any interval in P, and assume the machines are numbered $M_0, M_1, \ldots, M_{m-1}$. For each interval j and each machine M_i in P, create an interval offset $i \cdot T$ units to the right if interval j can be scheduled

on machine M_i. The result is a set of intervals where the intervals in the range $[i \cdot T, (i+1) \cdot T]$ correspond to the intervals in P that can be scheduled on machine M_i. All of the intervals created for a specific interval j in P define the k-tuples. The restriction that only one interval from a job can be scheduled is therefore equivalent to the restriction that a interval can only be scheduled on a single machine.

2.3 Circuit Routing and Load Balancing

Another family of on-line interval scheduling problems appear in the domain of circuit routing. These problems are studied in [3, 5, 6, 7]. In the basic circuit routing problem there is a network, and a series of connections. Connections have a source and destination, and fixed starting and ending times. Connections have to be assigned a route through the network on-line. The goal is to balance the load on the circuits in the network.

Circuit routing can be viewed as a generalization of load balancing. The circuits correspond to machines, and connections correspond to a series of jobs. The basic load balancing problem is studied in [6]. Two different types of tasks are considered: permanent jobs that have a start time but remain in the system forever, and temporary jobs that have a start time and a fixed length. Jobs have to be assigned to a machine when they appear. The duration of a job may or may not be known when a job first appears. What distinguishes this problem from IS is that overlapping jobs can be scheduled on the same machine. Each job adds a load to the machine, which may be dependent on the machine. They consider the full range of machine types: identical,

uniform, restricted and unrelated. The goal is to minimize the maximum load on any one machine. For permanent jobs, constant competitive ratios are possible if the machines are identical or uniform. Otherwise $\Theta(\log n)$-competitive algorithms are possible. For temporary jobs with unknown durations, constant competitive ratios are possible. If the machines are restricted the competitive ratio is $\Theta(\sqrt{n})$. The best known competitive ratio for unrelated machines is $O(n)$. These problems and results are also discussed in [7] and [3].

A more complicated and realistic version of circuit routing is discussed in [5]. In this case links of the network have a limited bandwidth, so it may not be possible to satisfy all connection requests. In addition to determining how to route a connection, an admission policy to determine if a request will be satisfied is also necessary. Each connection has a load and a profit associated with it. The goal is to maximize the profit of admitted requests without violating the capacity constraint. They present an on-line algorithm with a profit on the interval $[\tau_1 - T, \tau_2 + T]$ that is within a logarithmic factor of the off-line profit on the interval $[\tau_1, \tau_2]$, where T is the maximum duration of a connection. The slight difference in the intervals is necessary in order to be competitive. This problem is a generalization of IS. If we consider the case where all links have capacity one, and all calls have a load of one and cover exactly one link, then circuit routing becomes IS.

2.4 General Scheduling

There is a very large body of work on scheduling problems. A survey of classical scheduling results can be found in [28]. They present a detailed classification of scheduling problems. Problems are classified by the number and type of machines available. Restricted interval scheduling can be classified as the following scheduling problem: $R|p_{ij} = \{d_j - r_j, \infty\}|\sum w_j U_j$. The machines are unrelated. The job characteristics are that a job's processing time on a given machine is exactly the difference between the job's deadline and release time, or it is infinite. The optimality criteria is the unit penalty criteria. $U_j = 0$ if a job completes before its deadline and 1 otherwise. In this case the sum of U_j is the number of intervals that are not successfully scheduled. The unit penalty criteria has not been widely studied. It is known that $1||\sum w_j U_j$, $1|prec, p_j = 1|\sum U_j$, and $1|r_j|\sum U_j$ are all **NP**-hard.

Many interesting and natural scheduling problems are **NP**-hard. Therefore, approximation algorithms for scheduling problems are often studied. General methods of approximation are particularly interesting. If the scheduling problems can be expressed as Integer Programs, then general approximation methods such as Linear Relaxation and Lagrangian Relaxation can be used [40, 23]. In Linear Relaxation, the IP is solved as a linear program. The result is then coerced into a feasible solution. For certain classes of integer programs, the LP relaxation will still be optimal. For preemptive scheduling problems, the LP may be optimal [39].

Chapter 3

Off-line Restricted Cache Scheduling

This chapter describes results for the off-line restricted scheduling cache problem. Cache scheduling is by nature an on-line problem. However, the off-line problem is of interest to researchers because it indicates the best possible performance a new cache design can achieve, and is a benchmark against which the performance of on-line algorithms can be compared. Off-line cache scheduling is closely related to interval scheduling, therefore many of our results focus on variations of interval scheduling problems.

This chapter is organized as follows. We first present hardness results including inapproximability results, for different interval scheduling and off-line cache scheduling problems. We then present some optimal algorithms for certain interval scheduling problems. Finally, we present approximation algorithms for the different problems.

3.1 Hardness Results

In this section we will prove that for any type of restricted cache, both the off-line cache scheduling problem and the related interval scheduling problem is hard.

3.1.1 Interval Scheduling Problems

Restricted Interval Scheduling (RIS(m)) is defined in Definition 1.3.2. We will refer to the problem in which each interval can be scheduled on m machines as RIS(m). Previous work on the problem is described in Section 2.2.2. The problem has been shown to be **NP**-complete if intervals can be scheduled on three or more machines, that is when $m \geq 3$. [1] We will improve that result and show that any interesting variation of RIS(m) is hard. First we define Companion Machine Interval Scheduling, which is the simplest version of RIS(2).

Definition 3.1.1. The m-Companion Machine Interval Scheduling (CMIS(m)):
INSTANCE: A set M of machines $M_1, \ldots, M_k$, a set of m "companion" machines $B_1, \ldots, B_m$, an integer K, and a set I of n intervals (s_i, f_i, σ_i) where $s_i \in Z^+$ is the starting time of interval i, $f_i \in Z^+$ is the end time of interval i, and $\sigma_i \in M$ is the machine on which interval i can be scheduled. Additionally, any interval can be scheduled on any of the companion machines.
QUESTION: Can at least K of the intervals be legally scheduled?

Theorem 3.1.1. *CMIS(1) is **NP**-complete.*

Proof. We reduce 3-OCC-MAX-2-SAT to CMIS(1). 3-OCC-MAX-2-SAT is a restricted form of MAX-2-SAT where each variable occurs at most three times. This problem has been shown to be **NP**-complete and **APX**-complete [4].

Let $U = \{u_1, u_2, \ldots, u_a\}$, $C = \{c_1, c_2, \ldots, c_b\}$ and K be an instance of 3-OCC-MAX-2-SAT. For each variable u_i create two machines, M_{u_i} and $M_{\overline{u_i}}$. We also add the single companion machine B_1.

For each variable u_i create the following four intervals $(i, i+1, M_{u_i})$, $(i, i+1, M_{\overline{u_i}})$, $(0, |U| + |C| + 2, M_{u_i})$, and $(0, |U| + |C| + 2, M_{\overline{u_i}})$. This results in $4|U|$ intervals. For each clause c_k we create two intervals. If variable u_i appears in clause k in its positive form, we create the interval $(|U| + k, |U| + k + 1, M_{u_i})$. If variable u_i appears in clause k in its negative form, we create the interval $(|U| + k, |U| + k + 1, M_{\overline{u_i}})$. This results in $2|C|$ intervals. Thus a total of $4|U| + 2|C|$ intervals are created, clearly polynomial in the original input size.

An example instance is shown in Figure 3.1. The intervals form an instance of CMIS(1). The machine name above each interval indicates the machine, in addition to B_1, on which the interval can be scheduled. The intervals can be broken into three groups: the long variable intervals, the short variable intervals, and the clause intervals. The key observations about this reduction are that the variable intervals, both long and short, are used to enforce a variable assignment, and the schedulable clause intervals correspond to the satisfiable clauses.

Suppose we can satisfy K of the clauses in 3-OCC-MAX-2-SAT. We now show that we can schedule $3|U| + |C| + K$ of the intervals. Consider each variable u_i. If u_i is true, we use machine $M_{\overline{u_i}}$ to schedule one of the long intervals corresponding to u_i

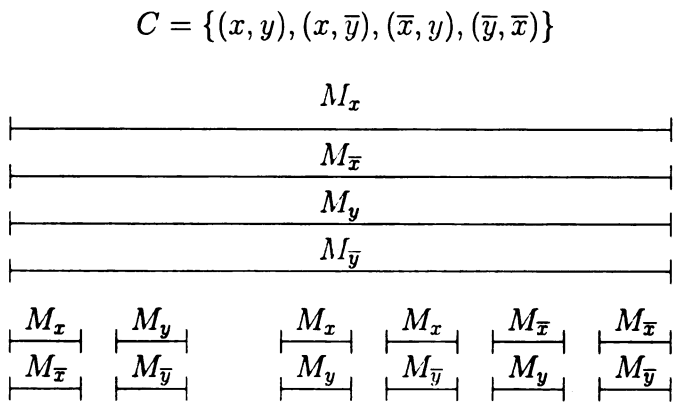

Figure 3.1: Example reduction from 3-OCC-MAX-2-SAT

and we use machines M_{u_i} and B_1 to schedule the two short intervals corresponding to u_i. If u_i is false, the roles of machines $M_{\overline{u_i}}$ and M_{u_i} are reversed. Exactly $3|U|$ of the variable intervals will be scheduled. Note that it is not possible to schedule more than $3|U|$ of the variable intervals because for each variable, there are 4 overlapping clauses and only 3 available machines.

Now consider each pair of clause intervals. If the corresponding clause c_i is satisfied, then one of the literals in c_i must be true. Without loss of generality, we can assume that this literal is u_j. Because u_j is true, the long interval that could be placed on M_{u_j} was not scheduled and machine M_{u_j} is free. We can therefore schedule the pair of intervals that correspond to clause c_i using machines M_{u_j} and B_1. If the clause is not satisfied, one interval can be scheduled on B_1. It is important to note that for each u_j, the clause intervals either use machine M_{u_j} or machine $M_{\overline{u_j}}$ but not both. A total of $|C| + K$ of the clause intervals will be scheduled. Thus we have scheduled exactly $3|U| + |C| + K$ intervals.

Now suppose there exists a schedule S that contains $3|U| + |C| + K$ intervals. We are going to create a modified schedule S' that contains at least $3|U|+|C|+K$ intervals, such that exactly one of the long variable intervals for each variable is scheduled, no long variable interval is scheduled on a companion machine, all $2|U|$ short variable intervals are scheduled, and $|C| + K$ of the clause intervals are scheduled. Note it is always possible to schedule $3|U|$ of the variable intervals as described above, and it is always possible to schedule $|C|$ of the clause intervals using the companion machine.

We construct S' as follows. If there is a pair of clause variables such that neither is scheduled on B_1, we schedule one of the intervals on the companion machine. This cannot hurt and may improve the schedule.

Suppose for a variable u_j, we use both M_{u_j} and $M_{\overline{u_j}}$ to schedule a long variable interval. This means that one of the short intervals corresponding to u_j is not scheduled. We modify the schedule by dropping one of the long variable intervals and replacing it with the unscheduled short variable interval.

Suppose that for a variable u_j, we do not schedule either of its corresponding long intervals. We look at the clause intervals. Because the variable u_j appears at most three times, it either appears in its positive form at most once, or it appears in its negative form at most once. It follows that there is at most one clause interval that can be scheduled on machine M_{u_j}, or there is at most one clause interval that can be schedule on machine $M_{\overline{u_j}}$. We drop that single clause interval and use the freed machine to schedule a long variable interval. This may require us to modify how the short intervals corresponding to u_j are scheduled. However, regardless of which long variable interval is scheduled, it is always possible to schedule the two short variable

intervals. The similar argument applies if B_1 were used to schedule a long interval, with the added note that if B_1 is used to schedule a long interval, only $2|U| + 1$ of the variable intervals can be scheduled.

The schedule S' will have at least $3|U| + |C| + K$ intervals scheduled. For each variable, exactly one long variable interval will be scheduled and it will not be on B_1. This will define a valid truth assignment for 3-OCC-MAX-2-SAT. At least K clause intervals will be scheduled on a non-companion machine, and these will correspond to the satisfied clauses in the instance of 3-OCC-MAX-2-SAT. □

It follows from this result that any interesting restricted interval scheduling problem is hard. We prove some specific cases below.

Corollary 3.1.1. *CMIS(m) is* ***NP****-complete for* $m \geq 1$.

Proof. We reduce CMIS(m) to CMIS($m + 1$). Given an instance I of CMIS(m), construct an instance I' of CMIS($m+1$) which contains all of the same intervals plus the additional machine B_{m+1} and a new machine G_{k+1}. Let $e_1, e_2, \ldots e_l$ be all the distinct interval endpoints that appear in I. For each $1 \leq i \leq l - 1$ we add 2 copies of the interval (e_i, e_{i+1}, G_{k+1}) to I'. This construction is illustrated in Figure 3.2.

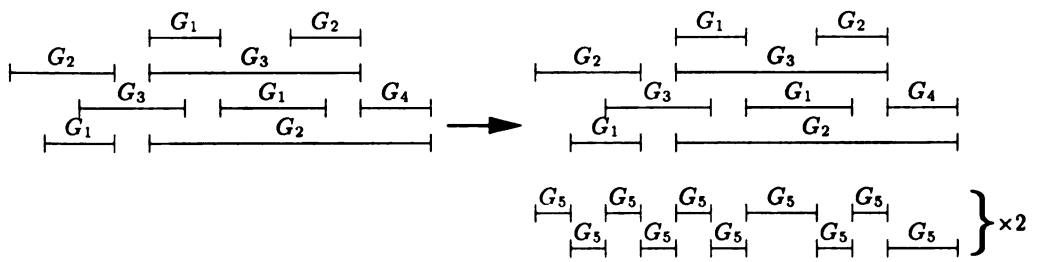

Figure 3.2: Example reduction from CMIS(m) to CMIS($m + 1$)

If we can schedule K intervals in I, we can clearly schedule $K + 2l - 2$ intervals in I' by using the same schedule for the original intervals and using the machines B_{m+1} and G_{k+1} to schedule the additional $2l - 2$ intervals. Suppose we can schedule $K + 2l - 2$ intervals in I'. Suppose the schedule does not contain one of the additional $2l - 2$ intervals. We will refer to the interval as j. Because j is as short as any of the original intervals, we can drop the interval scheduled on B_{m+1} and replace it with j without changing the cost of the schedule. Given that all $2l - 2$ of the additional intervals are scheduled, the remaining K intervals form a legal schedule for I.

CMIS(1) is **NP**-complete. It follows then that CMIS(m) is **NP**-complete for $m \geq 1$. □

Corollary 3.1.2. *RIS(m) is **NP**-complete for $m \geq 2$.*

Proof. CMIS(1) is an instance of RIS(2). It follows that RIS(2) is **NP**-complete. □

This improves the results of Arkin, *et. al.* [1] who only showed that RIS(m) is **NP**-complete when $m \geq 3$. Their proof is based on 3-SAT and breaks down if MAX-2-SAT is used. It is possible to modify their proof to work with MAX-2-SAT, but it is much more complicated than the proof presented here.

Corollary 3.1.3. *The m-way Skewed Interval Scheduling (SIS(m)) problem is **NP**-complete. The m-way Skewed Interval Scheduling problem is a variation of RIS(m) in which the machines are partitioned into m sets and each interval can be scheduled on exactly one machine from each group.*

Proof. CMIS(m) is an instance of SIS($m + 1$). Each companion machine forms a set and the remaining machines form the last set. □

3.1.2 Cache Scheduling

We can now show that off-line restricted cache scheduling is **NP**-complete. The basic observation is that when bypassing is allowed, an instance of cache scheduling in which each memory item is referenced exactly twice is identical to a restricted interval scheduling problem. We first consider the Companion Cache Structure Scheduling problem, CCSS(m, n). This is the off-line scheduling problem for the Companion Cache Structure CCS(m, n) defined in Section 1.5.1.

Definition 3.1.2. Companion Cache Structure Scheduling: (CCSS(m, n))

INSTANCE: We are given a CCS(m, n), a set of memory items R, and a function $g : R \rightarrow \{M_1, \ldots M_m\}$, that determines where in the main cache each item can be placed. We are also given a sequence of memory requests, $s \in R^*$, and an integer K.

QUESTION: Can the memory items be legally placed in the cache such that at least K requests in the sequence s are to items currently in the cache? In other words, does there exist a schedule with at least K hits?

Theorem 3.1.2. *CCSS(m, n) with bypassing is **NP**-complete.*

Proof. We reduce CMIS(n) to CCSS(m, n). For each machine in CMIS(m) create a cache location. The companion buffer consists of the n cache locations corresponding to the n companion machines. For each interval (s_i, f_i, σ_i) create a memory item x_i and let $g(x_i) = \sigma_i$. In other words, memory item x_i can be placed in the cache positions that correspond to the machines on which the interval (s_i, f_i) can be scheduled. Sort the end points $s_1, f_1, s_2, f_2, \ldots s_n, f_n$ of the intervals. If two end points are equal, then the end points are sorted as follows. Finishing points come before starting

points. Otherwise the endpoint from the lower indexed interval is first. A sequence of memory references is generated as follows. For each end point in the sorted list, reference the memory element that corresponds to the endpoint's interval.

The set of intervals defined by the resulting sequence of memory references and the original set of intervals are equivalent in the sense that the number of intervals that can be scheduled in both sets is the same. If an interval is scheduled, the reference corresponding to its endpoint will be a hit. Therefore, the number of intervals scheduled in the interval scheduling problem is equal to the number of hits in the cache scheduling problem. □

Theorem 3.1.3. *CCSS(m, n) without bypassing is* ***NP****-complete.*

Proof. If bypassing is not allowed, interval scheduling is not an exact model of the caching problem. However we can modify the proof of Theorem 3.1.1 using the ideas in the proof of Theorem 3.1.2 to reduce 3-OCC-MAX-2-SAT to CCSS($m, 1$) without bypassing. Let I be an instance of 3-OCC-MAX-2-SAT. We construct an instance I' of CCSS($m, 1$) without bypassing as follows. For each variable x we create the memory items $L_x, L_{\overline{x}}, U_x$ and $U_{\overline{x}}$. For each clause c_i we create the memory items C_{iu} and C_{iv}, where u and v are the literals that appear in clause c_i. The request sequence s is defined as follows. First each L_x and $L_{\overline{x}}$ is referenced. Then for each variable add $U_x U_{\overline{x}} U_x U_{\overline{x}}$ to the sequence. For each clause we add $C_{iu} C_{iv} C_{iu} C_{iv}$ to the sequence. Finally we add each L_x and $L_{\overline{x}}$ to the sequence.

We will show that I' has a schedule in which $3|U| + |C| + k$ references will be hits if and only if k clauses can be satisfied in I. Suppose we can satisfy k clauses

in the instance of 3-OCC-MAX-2-SAT. Consider a clause i that was not satisfied. One of the literals in clause i must appear only once. If not, we can modify the truth assignment to create an assignment that satisfies clause i and satisfies at least k clauses. Each variable only appears at most 3 times. If a variable only appears in the positive or negative form, then obviously clause i can be satisfied. Therefore clause i contains a variable that appears twice in its positive and once in its negative form (or vice versa). Let us assume that clause i contains the literals x and y. If x appears in another clause, then $\overline{x}$ can only appear in a single clause j. If we assign x to be true, we satisfy clause i, and the only clause that might no longer be satisfied is clause j. Therefore, there are still at least k clauses satisfied.

We now consider the modified truth assignment. If x is true, we use B_1 and $M_{\overline{x}}$ to schedule the U_x and $U_{\overline{x}}$ references, and to schedule the C_{ix} and C_{iu} references. We place L_x in M_x and $L_{\overline{x}}$ in B_1. If x is false, we swap the roles of $M_{\overline{x}}$ and M_x. This is the same idea used in Theorem 3.1.1. Because bypassing is not allowed, we must place the references corresponding to the unsatisfied clauses in the cache. Essentially we are forced to schedule the "short" intervals. For each unsatisfied clause, if x is the literal that appears a single time, we use M_x and B_1 to schedule the C_{ix} and C_{iu} references.

There are $2|U| + 2|C|$ hits for the "short" intervals. For each variable, one of the final L_x or $L_{\overline{x}}$ references will be a hit, unless that variable appeared in an unsatisfied clause. There are $|C| - k$ unsatisified clauses, so we have a total of $3|U| + |C| + k$ hits.

□

Theorem 3.1.4. *CCSS(m, n) with reorganization is* ***NP****-complete.*

Proof. Cache reorganization is equivalent to preemptive interval scheduling. In preemptive interval scheduling, an interval can be moved from one machine to another without being lost. We again consider the reduction used in the proof of Theorem 3.1.1. We show that in the optimal schedule for this type of instance, each interval is scheduled on one machine even if preemption is allowed. It follows that CCSS(m, n) with reorganization is **NP**-complete.

For each variable there are four intervals that must be scheduled on three machines. Preemption cannot help. At most three of the intervals can be scheduled; specifically one of the long intervals and two of the short intervals. Likewise preemption does not allow us to schedule any additional clause intervals. □

Theorem 3.1.5. *Off-line restricted cache scheduling (RCS) is* ***NP****-complete.*

Proof. Any interesting restricted cache is going to "contain" a CCS($m, 1$). Therefore CCSS($m, 1$) is an instance of every other restricted cache scheduling problem, and every restricted cache scheduling problem is **NP**-complete. □

3.1.3 Inapproximability Results

We now show that CMIS(1) belongs to the class **APX**-complete [31]. **APX**-complete is a class of problems that are hard to approximate. Unless $P = NP$, there does not exist a polynomial time approximation scheme for **APX**-complete problems.

Theorem 3.1.6. *CMIS(1) is* ***APX****-complete.*

Proof. In Section 3.3 we will show that CMIS(1) has a 2-approximation. Therefore the problem is in **APX**. We will now show that the reduction used in Theorem 3.1.1 is an L-reduction. L-reductions are defined in Section 1.6.1.

Let I be an instance of 3-OCC-MAX-2-SAT. $R(I)$ will be an instance of CMIS(1) created using the reduction in Theorem 3.1.1. By definition of the reduction,

$$OPT(R(I)) = OPT(I) + 3|U| + |C|.$$

Because each clause can contain at most two variables and each variable must appear at least once, $|U| \leq 2|C|$. Also, for any instance of MAX-2-SAT, at least $\frac{3}{4}$ of the clauses are satisfiable. It follows that

$$\begin{aligned} OPT(R(I)) &= OPT(I) + 3|U| + |C| \\ &\leq OPT(I) + 7|C| \\ &\leq OPT(I) + \left(\frac{28}{3}\right) OPT(I) \\ &\leq \left(\frac{31}{3}\right) OPT(I) \end{aligned}$$

Given a schedule s for $R(I)$ we can modify it as described in the proof of Theorem 3.1.1 to create a schedule s' that corresponds to a valid truth assignment t. The short intervals not scheduled in s' will correspond exactly to the unsatisfied clauses. Therefore

$$OPT(I) - c(t) = OPT(R(I)) - c(s') < OPT(R(I)) - c(s).$$

Therefore, the reduction is an L-reduction, with $\alpha = \frac{31}{3}$ and $\beta = 1$, and CMIS(1) is **APX**-complete. □

Corollary 3.1.4. *CMIS(1) does not have a PTAS for $m \geq 2$ unless P=NP.*

For the reasons stated in Thereom 3.1.5, the above results apply to any restricted cache scheduling problem.

3.2 Algorithms

In this section we present algorithms to optimally solve some cache scheduling and interval scheduling problems.

3.2.1 Optimal Off-line Solution

RIS(m) can be solved optimally using dynamic programming. Arkin and Silverberg present an optimal solution for this problem with running time $O(n^{k+1})$, where k is the total number of machines [1]. We now give a more straightforward and efficient dynamic programming solution.

Theorem 3.2.1. *RIS(m) with k machines and n intervals can be solved in $O(kn^k)$ time.*

Proof. Order the n intervals by starting time. The dynamic programming solution works from back to front. Let $S(j_1, j_2, \ldots, j_m)$ be the number of intervals that can be scheduled given that no interval before j_i is scheduled on machine M_i for $1 \leq i \leq m$. In other words, this is the subproblem which consists of intervals such that the earliest

starting interval that can be be scheduled on machine M_i is interval j_i of the original problem. Note that the subproblem is not just a subset of intervals. In some cases, intervals in the subproblem will only be schedulable on a subset of the machines on which they were schedulable in the original problem.

We define the functions $f_i, \ldots f_m$ and $g_i, \ldots g_m$. Let $f_i(j)$ be the earliest starting interval after j that can be scheduled on machine i. Let $g_i(j)$ be the earliest interval after j that can be scheduled on machine M_i and that does not overlap with j. Note $g_i(j)$ is the adjacency function. The optimal solution is $S(h_1, h_2, \ldots h_k)$, where h_i is the earliest interval that can be scheduled on machine M_i. The idea behind the dynamic programming solution is simple. Consider the earliest available interval i, and either schedule j on any of the machines, or reject it, and then solve for the remaining intervals. If j is scheduled on machine M_i, we must remember that machine M_i is unavailable for the length of interval j. As an example, consider $S(j_1, j_2, j_3, j_4, j_5)$, and suppose that the earliest interval can be scheduled on on machines M_1, M_2 and M_4. In other words, $j_1 = j_2 = j_4 \leq j_3, j_5$. In this case

$$\begin{aligned} S(j_1, j_2, j_3, j_4, j_5) = \max(&S(f_1(j_1), f_2(j_2), j_3, f_4(j_4), j_5), && \textit{(reject the interval)} \\ &1 + S(g_1(j_1), f_2(j_2), j_3, f_4(j_4), j_5), && \textit{(schedule on } M_1\textit{)} \\ &1 + S(f_1(j_1), g_2(j_2), j_3, f_4(j_4), j_5), && \textit{(schedule on } M_2\textit{)} \\ &1 + S(f_1(j_1), f_2(j_2), j_3, g_4(j_4), j_5)) && \textit{(schedule on } M_4\textit{)} \end{aligned}$$

In order to write out the dynamic program in a closed form we introduce some additional notation. Let $E = \{i \mid \forall l \; j_i \leq j_l\}$. E is the set of machines on which the earliest available interval can be scheduled. Let

$$F_i(j_i) = \begin{cases} f_i(j_i) & \text{if } i \in E \\ j_i & \text{otherwise} \end{cases}, \qquad G_i(j_i, l) = \begin{cases} g_i(j_i) & \text{if } i = l \\ F_i(j_i) & \text{otherwise} \end{cases}$$

Note that strictly speaking, F_i and G_i are also functions of E which is a function of $j_1, \ldots, j_k$, but we will abuse notation for brevity's sake. The earliest available interval is either scheduled on a machine, or it is rejected. The function $F_i(j_i)$ represents an interval being rejected and the function $G_i(j_i, l)$ represents an interval being scheduled. In the case that an interval is not the earliest ending interval, both functions are the identity function.

The complete dynamic programming solution is

$$S(j_1, j_2, \ldots j_k) = \max(S(F_1(j_1), \ldots F_k(j_k)), \quad \max_{l \in E}(1 + S(G_1(j_1, l), \ldots G_k(j_k, l))))$$

There are at most n^k values to calculate. Calculating any one value is an $O(k)$ operation, because in the worst case an interval can be scheduled on all k machines. Therefore the dynamic programming solution has a complexity of $O(kn^k)$. We can improve this slightly when we consider that an interval can be scheduled on at most m machines. The complexity is $O(m\left(\frac{mn}{k}\right)^k)$. Clearly, for large n and k this is impractical. □

As mentioned before, interval scheduling is not an exact model of restricted cache scheduling (RCS). In order to exactly model cache scheduling with bypassing, we need to consider the adjacency constraints. The adjacency constraint prevents adjacent intervals from being scheduled on different machines. The dynamic programming

solution can be extended to handle adjacency constraints, but the solution becomes considerably more complex.

Theorem 3.2.2. *RCS with k cache locations and $2n$ items referenced can be solved in $O(kn^{2k+1})$ time.*

Proof. Again we consider scheduling the earliest interval. However, instead of only scheduling a single interval at each step of the dynamic program, we need to consider scheduling groups of adjacent intervals. For example, suppose j_1 is the earliest interval that can be scheduled on machine M_1. We can schedule j_1 on machine M_1, or we can schedule j_1 and $g_1(j_1)$ (the earliest starting interval that does not overlap j_1 that can also be scheduled on M_1) on machine M_1, or we can schedule j_1, $g_1(j_1)$ and ${g_1}^2(j_1)$ on machine M_1, etc. Also, if we schedule j_1 on machine M_1, then we cannot schedule $g_1(j_1)$ on a different machine, so we need to remember which sets of consecutive intervals have been scheduled on which machines.

Our dynamic programming solution will have the form $S(j_1, u_1, j_2, u_2, \ldots, j_m, u_m)$. The meanings of j_i and u_i are the following. If $j_i = u_i$, then j_i is the earliest interval that can be scheduled on machine M_i. If $j_i \neq u_i$, then the set of intervals $\{g_i^n(u_i) | n \geq 0 \wedge g_i^n(u_i) < j_i\}$ have been scheduled on machine M_i. We must avoid scheduling any interval in this set a second time, and we cannot schedule any interval adjacent to this set on a different machine.

Let $E = \{i \mid \forall l\ j_i \leq j_l \wedge j_i = u_i\}$ and let $E' = \{i \mid \forall l\ j_i \leq j_l\}$. F_i and G_i are defined the same manner. As before, E is the set of machines on which the earliest interval

can be scheduled and thus is the set of machines on which we consider scheduling groups of intervals. If E is not empty, then the dynamic program is defined as follows.

$$S(j_1, u_1, j_2, u_2, \ldots j_k, u_k) = \max(S(F_1(j_1), F_1(u_1), \ldots F_k(j_k), F_k(u_k)),$$
$$\max_{l \in E} \left(\max_{\substack{1 \le v \le n, \\ \exists i \exists w \, g_i{}^w(u_i) = g_l{}^v(f_l) < f_i}} (v + S(G_1(j_1, l, v), u_1, \ldots G_k^v(j_k, l), u_k)) \right)$$

If E is empty, then E' is the set of machines on which it is safe to forget the previously scheduled groups of intervals. In this case the dynamic program is defined as follows

$$S(j_1, u_1, j_2, u_2, \ldots j_k, u_k) = S(j_1, H_1(j_1, u_1), \ldots j_k, H_k(j_k, u_k)),$$

where $H_i(j_i, u_i) = \begin{cases} j_i & \text{if } i \in E' \\ u_i & \text{otherwise} \end{cases}$.

There are n^{2k} values to calculate. Calculating any one value is an $O(kn)$ operation. Therefore the dynamic programming solution has a complexity of $O(kn^{2k+1})$. □

3.2.2 Optimal Solution for RIS(2)

If intervals can only be scheduled on at most two machines, and we are only interested in whether or not all the intervals can be scheduled, then things are easier. RIS(2) can be solved in polynomial time. The algorithm is similar to the algorithm for solving 2-SAT.

Theorem 3.2.3. *Determining if all of the intervals in an instance of RIS(2) can be scheduled is solvable in polynomial time.*

Proof. We construct a directed graph as follows. If interval j can be scheduled on machine M_x we create a vertex v_{xj}. Because each interval can be scheduled on at most two machines, there are at most $2n$ vertices. We create a directed edge (v_{xj}, v_{yk}) if intervals j and k overlap, $x \neq y$ and interval k can also be scheduled on machine M_x.

We apply an all pairs shortest path algorithm to this graph. If for any interval j schedulable on machines M_x and M_y, there is a path from v_{xj} to v_{yj} and a path from v_{yj} to v_{xj}, then it is not possible to schedule all of the intervals. Otherwise a schedule exists.

The basic idea is as follows. Consider an interval j that can be scheduled on machines M_x and M_y. If we schedule j on M_x, none of the intervals that overlap j can be scheduled on M_x and we are forced to schedule those intervals on another machine. This is the meaning of the directed edges in the graph. In turn, this will force other intervals to be scheduled on specific machines and may eventually force us to schedule j on M_y, in which case we cannot schedule j on M_x. If it is also the case that scheduling j on M_y eventually forces us to schedule j on M_x, it is clear that j cannot be scheduled. □

It is interesting to compare this result to the results of Kolen and Kroon [34]. There the authors showed that RIS(m) is **NP**-complete if there exist more than two classes of machines, where two machines are in a different class if there exists an

interval that can be scheduled on one machine but not the other. In RIS(2) there are more than two classes of machines. In general, each machine is in its own class. However, because each interval in RIS(2) can only be scheduled on 2 machines, the problem is solvable in polynomial time.

If intervals can be schedule on more than 2 machines, then even determining if all of the intervals can be scheduled or not is **NP**-complete.

3.3 Approximation Algorithms

Given that RIS(m) is **NP**-complete we have to be content with approximation algorithms. We have already shown that a PTAS does not exist for RIS(m) unless $\mathbf{NP} \neq \mathbf{P}$. In this section we present approximation algorithms for the optimization version RIS(m).

3.3.1 Graph Theoretical Model

Interval scheduling and cache scheduling with bypassing can both be formulated as independent set problems. This model is very useful for describing approximation algorithms. For the interval scheduling problem, we construct a graph G as follows. For each interval i create a vertex v_{ix} if interval i can be scheduled on machine x. There will be at most nk vertices where n is the number of intervals and k is the size of the cache. We add the edge (v_{ix}, v_{jx}) if interval i and j overlap and they both can be scheduled on machine x. This enforces the constraint that overlapping intervals cannot be scheduled on the same machine. Note that intervals that share endpoints

do not overlap. We add the edge (v_{ix}, v_{iy}) if interval i can be scheduled on machines x and y. This enforces the constraint that an interval can only be scheduled on one machine. A set of independent vertices in G corresponds to a valid schedule. A maximum independent set in G corresponds to an optimal solution to the scheduling problem. A similar model is used in [24].

The graph G has an interesting structure. For a fixed x, let G_x be the subgraph induced by the vertices v_{ix}. This graph represents all of the intervals that can be scheduled on machine M_x and is an *interval graph.* Interval graphs are well studied and have many interesting properties [27]. The independent set problem for an interval graph is solvable in polynomial time. This is because an interval graph is guaranteed to have a *simplicial* vertex. A vertex is simplicial if its neighbors form a clique. Clearly a simplicial vertex can be part of a maximum independent set, because only one of its neighbors could take its place. Removing a simplicial vertex and its neighborhood results in another interval graph. The vertex corresponding to the earliest ending interval is simplicial in an interval graph.

For a fixed i, let H_i be the subgraph induced by the vertices v_{ix}. This graph is a clique which represents the machines that interval i can be scheduled on and the fact that the interval can only be scheduled on one machine. If we look at a vertex v_{ix} that is simplicial in G_x, then the neighbors of v_{ix} form at most two cliques. One clique is the neighbors in G_x, and the other clique is the neighbors in H_i. All of the vertices corresponding to the earliest ending interval have the property that their neighbors form at most two cliques.

For restricted cache scheduling with bypassing (RCS), we also have to enforce the constraint that two adjacent intervals cannot be scheduled on separate machines. We do this by adding edges (v_{ix}, v_{jy}) if interval i is adjacent to interval j and $x \neq y$ to G. This additional edge is neither part of a clique nor part of an interval graph. For this reason, the vertices corresponding to the earliest ending interval have the property that their neighbors form at most three cliques.

It follows from the results of Section 3.1 that the independent set problem is **NP**-complete and **APX**-complete for these two families of graphs.

3.3.2 Earliest End Time Algorithms

The earliest end time algorithm (EET) is a greedy algorithm that sorts the intervals by end time, and then schedules intervals as they fit. EET is optimal for interval scheduling on identical machines. This is because interval scheduling on identical machines can be represented as an interval graph and the earliest ending intervals correspond to simplicial vertices in the interval graph.

Theorem 3.3.1. *There exists an optimal EET schedule for RIS(m).*

Proof. Let G be a graph representing an instance of RIS(m). Consider a maximum independent set that does not contain any of the vertices corresponding to the interval with the earliest end time. Because each of these vertices are simplicial in the interval subgraphs, each of these vertices can have at most one neighbor in the maximum independent set. We can therefore add one of the vertices that corresponds to the earliest ending interval and remove its one neighbor, without changing the size of the

maximum independent set. We then remove the scheduled vertex and its neighbors and repeat. □

Unfortunately, EET as described above is not a deterministic algorithm because it may be possible to schedule the earliest ending interval on more than one machine. In order to make the algorithm deterministic, EET will always schedule an interval on the lowest numbered machine available. This algorithm is a 2-approximation for RIS(m).

Theorem 3.3.2. *EET is a 2-approximation algorithm for RIS(m).*

Proof. We will prove this theorem two ways. First, let S be the schedule produced by EET, and let OPT be the optimal schedule. Consider an interval i scheduled on some machine M_x in OPT that is not in S. There must be exactly one interval j in S on machine M_x that overlaps i and has an earlier endtime than i. Otherwise i would have been in S. It is not possible that there is a second interval k scheduled on machine M_x in OPT that is not in S because of j. In order for k to not be in S because of j, k must be scheduled on M_x in OPT, it must have a later end time than j and it must overlap j. However, this means that i and k must overlap, and therefore OPT cannot schedule both i and k on machine M_x. It follows from this that OPT can schedule at most twice as many intervals as S.

Secondly, we can consider the graph formulation of the problem. There always exists at least one vertex whose neighbors form at most two cliques. These are the vertices that are simplicial within the interval subgraph for a specific machine, and include the vertices representing the earliest ending interval. Because this vertex's

neighbors form two cliques, at most 2 of the vertex's neighbors can be in the optimal solution. Therefore choosing such vertices will result in a 2-approximation. □

This bound is tight as shown in the instance of CMIS(1) depicted in Figure 3.3. Each interval is labeled with the machine on which it can be scheduled. Each interval can also be scheduled on the companion machine C_1. EET will schedule all of the short intervals on the appropriate M_i and one of the long intervals on B_1, for a total of $n+1$ intervals. The optimal solution schedules all of the short intervals on B_1 and the long intervals on the appropriate M_i for a total of $2n$ intervals. This example can be generalized for RIS(m) for any value of m.

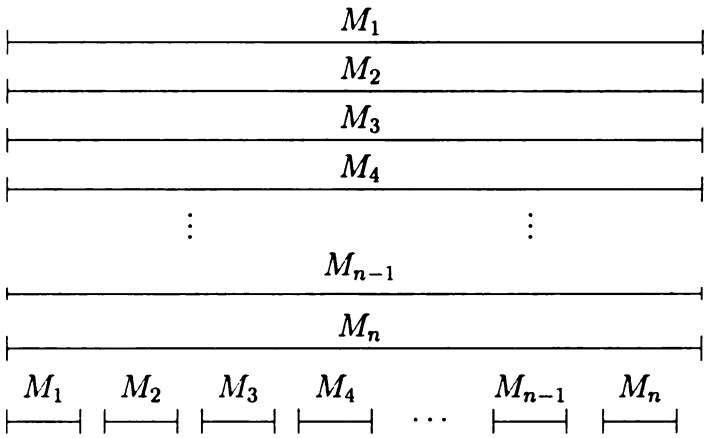

Figure 3.3: EET lower bound.

EET would be optimal if each interval was scheduled on the correct machine. For a deterministic algorithm, we can construct an instance where it always chooses incorrectly. If we randomly choose the machine on which to schedule an interval, we can improve the performance of EET. We consider the randomized EET algorithm

(RAN-EET), which randomly places the earliest ending interval on one of the available machines.

Theorem 3.3.3. *Randomized EET (RAN-EET) is a* $\left(\frac{2m-1}{m}\right)$*-approximation algorithm for RIS*(m).

Proof. The proof is inductive in nature. Let G be a graph corresponding to an instance I of RIS(m). Let $R(G)$ be the value of RAN-EET for the graph G. Let $OPT(G)$ be the optimal value.

If $OPT(G) \leq 2$, then clearly $R(G) = OPT(G)$. Otherwise, consider a graph G. RAN-EET randomly places the earliest ending interval on one of the available machines. The algorithm then removes the vertex corresponding to the placement of the interval, and all of the vertex's neighbors to form a new graph, and then repeats. There are two possibilities. With probability $\frac{1}{m}$, RAN-EET chooses the correct machine, in which case we name the new graph G'. Clearly $R(G) = 1 + R(G')$ and because this was the optimal choice, $OPT(G) = 1 + OPT(G')$. Otherwise with probability $\frac{m-1}{m}$, RAN-EET chooses an incorrect machine, in which case we name the new graph G''. $R(G) = 1 + R(G'')$ and $OPT(G) \leq 2 + OPT(G'')$, because the subgraph $G - G''$ consists of at most two cliques and can have at most two independent vertices in it. G' and G'' are illustrated in Figure 3.4.

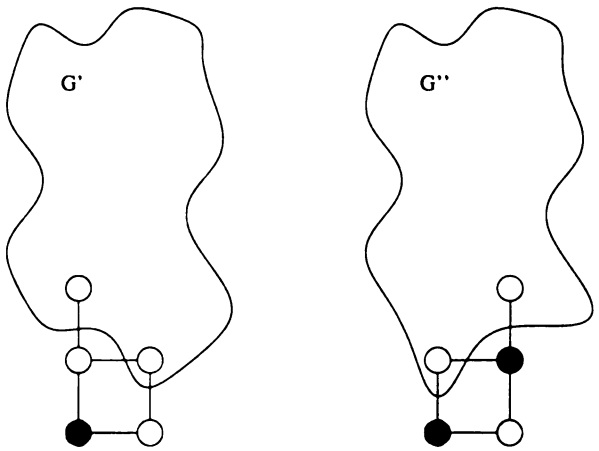

Figure 3.4: The choices of RAN-EET and the resulting subgraphs.

We can now calculate the expected value of $R(G)$.

$$\begin{aligned}
E|R(G)| &= \left(\frac{1}{m}\right)(1+E|R(G')|)+\left(\frac{m-1}{m}\right)(1+E|R(G'')|) \\
&\geq \left(\frac{1}{m}\right)\left(1+\left(\frac{m}{2m-1}\right)OPT(G')\right) \\
&\quad +\left(\frac{m-1}{m}\right)\left(1+\left(\frac{m}{2m-1}\right)OPT(G'')\right) \\
&\geq \left(\frac{1}{m}\right)\left(1+\left(\frac{m}{2m-1}\right)(OPT(G)-1)\right) \\
&\quad +\left(\frac{m-1}{m}\right)\left(1+\left(\frac{m}{2m-1}\right)(OPT(G)-2)\right) \\
&\geq \left(\frac{m}{2m-1}\right)OPT(G)+1-\left(\frac{1}{2m-1}+\frac{2m-2}{2m-1}\right) \\
&\geq \left(\frac{m}{2m-1}\right)OPT(G)
\end{aligned}$$

□

We can prove similar results for restricted cache scheduling. Let RCS(m) be a restricted cache scheduling problem in which each item can be placed in m cache

locations. In the graph representation of RCS(m), there exist vertices whose neighbors form 3 cliques.

Corollary 3.3.1. *EET is a 3-approximation algorithm for RCS(m) with bypassing.*

Corollary 3.3.2. *RAN-EET is a $\left(\frac{3m-2}{m}\right)$-approximation algorithm for RCS(m) with bypassing.*

3.3.3 Linear Programming Relaxations

RIS(m) can be formulated as an integer programming (IP) problem. For each interval i that can be scheduled on machine M_j, let the variable x_{ij} indicate if interval i has been scheduled on machine M_j. For each machine M_j, the intervals schedulable on M_j define an interval graph G_j. Let C_{jk} be the k^{th} maximal clique in G_j. The linear program version of RIS(m) is:

(IP) Maximize $\sum x_{ij}$

subject to

$$\sum_{j} x_{ij} \leq 1 \quad \forall i \text{ (schedule interval } i \text{ on at most one machine).}$$

$$\sum_{i \in C_{jk}} x_{ij} \leq 1 \quad \forall j \text{ (do not schedule overlapping intervals on machine } M_j).$$

$$x_{ij} \in \{0, 1\}$$

Interestingly enough, the linear programming (LP) relaxation of these IPs tend to have integral solutions. An LP can be expressed in the form $Ax = b$, where A is a

matrix, and x and b are vectors. A matrix A is *totally unimodular* if the determinant of every square, nonsingular submatrix of A, including A itself, is ± 1 [48]. If the matrix A is totally unimodular, then the LP has an optimal integer solution. In the LP formulation of restricted interval scheduling, the submatrix of A that corresponds to the second set of constraints for a machine M_j is the *clique matrix* of the interval graph G_j. The clique matrix of an interval graph is totally unimodular [46]. Therefore, in the restricted interval scheduling case, large sub-matrices of the matrix A are totally unimodular and it is for this reason we believe that the LP's often have integral solutions.

We cannot prove an upper bound on the approximation ratio of an LP relaxation, but we can prove a lower bound.

Theorem 3.3.4. *The integrality gap between the IP formulation of RIS(m) and its LP relaxation is* $\geq \frac{7}{6}$.

Proof. The proof is by example. Consider Figure 3.5. This is the graph representation of a restricted interval scheduling problem. We can divide this example into groups of four intervals, skipping the first interval. In each group, the optimal solution schedules three of the intervals. Each vertex corresponds directly to a variable in the LP, and the numbers besides each vertex represent the optimal assignment of that variable. The fractional LP solution for each group is 3.5. Therefore, the gap is $\frac{7}{6}$.

□

It follows that an LP based approximation algorithm cannot have an approximation ratio better than a $\frac{7}{6}$.

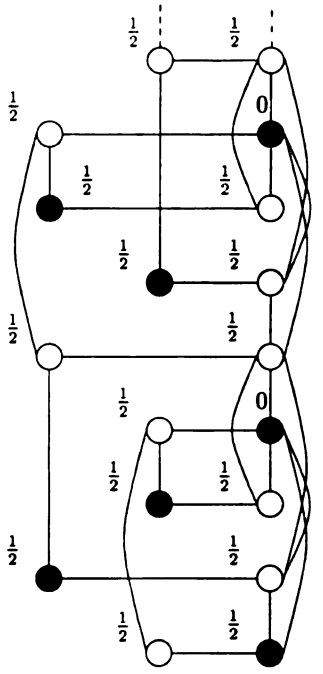

Figure 3.5: Integrality Gap Example.

3.3.4 Empirical Results

Despite the inapproximability results, random instances of this problem tend to be easy to approximate. We consider instances where each interval can be scheduled on 2 machines. In the 100 interval inputs, there are 8 machines, and all endpoints are in the range 0 through 64. In the 250 interval inputs, there are 10 machines, and all endpoints are in the range 0 through 120. In both cases the length of each interval is between 1 and 20, and the optimal solution schedules approximately 40% of the intervals.

We consider four algorithms. EET is the basic earliest end time algorithm. MSV and AIL are variations of EET that employ different heuristics to determine on which machine to schedule an interval. MSV (Maximize Simplicial Vertices) chooses the machine that results in the most simplicial vertices. AIL (Avoid Interval Loss), will

choose the machine that results in the fewest unscheduled intervals being lost. LP is an LP relaxation of the integer programming. Table 3.1 compares the performance of these different algorithms.

100 Intervals, 8 machines			
Algorithm	Avg Performance	Worst Performance	%Optimal
EET	93.16	85.10	1
MSV	93.95	86.36	5
AIL	95.59	87.76	9
LP	99.65	95.35	86
250 Intervals, 10 machines			
EET	93.24	87.38	9
MSV	94.14	89.19	13
AIL	95.91	91.89	6
LP	99.60	96.36	73

Table 3.1: Performance of Approximation Algorithms for RIS

3.4 Summary

We have shown that any interesting variation of Restricted Interval scheduling is **NP**-complete and **APX**-complete. As a consequence, off-line Restricted Cache Scheduling is also **NP**-complete and **APX**-complete. We have designed optimal, exponential algorithms for RIS and RCCS. We have also shown that the simple EET algorithm is a 2-approximation for interval scheduling, and have done empirical studies to show how EET and other algorithms perform.

Chapter 4

On-line Restricted Cache Scheduling

This chapter describes the results for on-line restricted cache scheduling problems. We evaluate on-line algorithms using the competitive analysis technique. Competitive analysis is defined in Section 1.6.2. For the most part we focus on the Companion Cache Structure (CCS(m, n)), defined in Section 1.5.1, as it is the fundamental restricted cache. We consider many variations of the basic Companion Cache Scheduling problem, including features such as bypassing and reorganization. We also include results for other types of restricted caches.

4.1 Companion Cache Scheduling

The Companion Cache Structure is defined in Section 1.5.1. A CCS(m, n) consists of a direct-mapped cache of size m, which we will call the main cache, and a small

fully-associative cache of size n, which we call the companion buffer. We will use M_1, M_2, ... M_m to refer to the positions in the main cache, and B_1, B_2, ... B_n to refer to the positions in the companion buffer. Two items are of the same *type* if they can occupy identical positions in the cache. We will use X, X′, Y, Y′, Z, Z′ to refer to memory items. X and X′ will refer to different items of the same type.

We first look at the Least Recently Used (LRU) replacement policy. When a miss occurs, LRU will place the item in the least recently used cache position, which can either be in the main cache or in the companion buffer. The first result is that LRU is not competitive, assuming reorganization is not allowed.

Theorem 4.1.1. *LRU is not competitive for CCS(2, 1).*

Proof. The proof is by example. Let X and X′ be items that can be placed in cache position M_1 and let Y and Y′ be items that can be placed in cache position M_2. Consider the input sequence in Figure 4.1.

Sequence:	Y′	X	Y	X′	Y	X	Y	X′	Y	X	Y	X′	...
LRU:	$\underline{M_2}$	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_1}$	B_1	$\underline{M_1}$	B_1	$\underline{M_1}$	B_1	$\underline{M_1}$	B_1	$\underline{M_1}$	
OPT:	$\underline{M_2}$	$\underline{M_1}$	$\underline{M_2}$	$\underline{B_1}$	M_2	M_1	M_2	B_1	M_2	M_1	M_2	B_1	

Figure 4.1: LRU is not competitive for CCS(2, 1)

Misses are underlined. OPT only misses on the first four references. LRU misses on every reference to X or X′. We are assuming that LRU initially places items in the main cache. If LRU initially places items in the companion buffer, then the counter example is the same sequence starting with the first reference to Y. It is interesting to note that LRU does not miss on every reference. In traditional caches,

the worst case for any on-line algorithm, which includes LRU, is when it misses on every reference. □

In any restricted cache, we can find three items such that they all share a location in one of the component caches, and two share a location in another component cache. It follows that LRU is not competitive for any restricted cache. Note that this is true whether or not bypassing is allowed.

4.1.1 Companion Cache Scheduling with Bypassing

For this section, we will assume that bypassing is allowed but reorganization is not allowed. We will first focus on CCS(m, 1). We divide the input sequence into phases. A phase consists of a minimal set of consecutive references to items that cannot all simultaneously fit in the cache. This is a generalization of the phase definition used in the analysis of traditional on-line caching as described in Section 2.1.2. The first phase begins with the first item in the request sequence. Within a phase, an item is a *leader* if its first occurrence in the phase is followed by the first occurrence in the phase of a different item of the same type. An item is a *follower* if its first occurrence is preceded by the first occurrence of a different item of the same type. An item can be both a leader and a follower. The reference to the second follower ends a phase. This is because we can place all of the non-followers in the main cache and one of the followers in the companion buffer, but there is no room in the cache for the other follower. The last phase ends when the input sequence ends. Figure 4.2 illustrates the phases for a given request sequence. Note that different phases can contain a

different number of distinct items, and that in phase 2, Z′ is both a leader and a follower. Because all the items in a phase cannot all simultaneously fit into the cache, any algorithm must miss at least once per phase.

X	Y	Y′	X	Z	X′	Y	X	Y′	Z′	Z	Z	Z′	Y	Z″	X	Y	Z
		phase 0					phase 1					phase 2			phase 3		

Figure 4.2: Example phases for CCS(3, 1)

We now define the algorithm Main Cache First (MCF) as follows. Whenever a miss occurs, the newly accessed item is placed in the direct mapped cache if it is the first item of its type in the phase. Otherwise the item is placed in the companion buffer. In Figure 4.3, we show how MCF behaves on the LRU counter example. The input consists of only two phases.

Sequence:	Y′	X	Y	X′	Y	X	Y	X′	Y	X	Y	X′	...
MCF:	$\underline{M_2}$	$\underline{M_1}$	$\underline{B_1}$	$\underline{B_1}$	$\underline{M_2}$	M_1	M_2	B_1	M_2	M_1	M_2	B_1	
OPT:	$\underline{M_2}$	$\underline{M_1}$	$\underline{M_2}$	$\underline{B_1}$	M_2	M_1	M_2	B_1	M_2	M_1	M_2	B_1	
		phase 0						phase 1					

Figure 4.3: MCF's behavior on the LRU counter example.

Theorem 4.1.2. *MCF is* 5*-competitive on CCS(m,* 1*) if bypassing is allowed.*

Proof. The outline of the proof is as follows. Each miss that occurs in the MCF schedule will be charged to one of the phases. Misses will not necessarily be charged to the phase in which they occur. Let c_i represent the cost charged to phase i. We will also determine a lower bound on OPT_i, the number of misses that occur in phase i of the optimal schedule. The total cost of MCF will be $\sum c_i$ and the total cost

of the optimal algorithm will be greater than $\sum OPT_i$. We will show that for each phase, $\frac{c_i}{OPT_i} \leq 5$. Therefore, $\frac{\sum c_i}{\sum OPT_i} \leq 5$ and MCF is 5-competitive.

MCF will only miss on an item once per phase. If an item is put in the main cache, it will remain in the main cache throughout the phase. If an item is put in the companion buffer, it must be a follower. The first item placed in the companion buffer will remain in the companion buffer until the second follower is referenced, which ends the phase.

Consider a reference to item X in phase i in the MCF schedule. This reference is a miss if X is not currently in the cache. We divide the misses into 4 different categories.

- Shared misses. Both OPT and MCF miss on X. This includes the very first reference to X, which is known as a *compulsory* miss.
- Companion Buffer misses. X is placed in the companion buffer. X must be a follower.
- Companion to Main misses: X was last in the companion buffer, and X is now placed in the main cache. This means that when X was last referenced it was a follower.
- Main to Main misses: X was last in the main cache and X is now placed in the main cache.

Companion Buffer, Companion to Main, and Main to Main misses are mutually exclusive. Shared misses take precedence over the other categories. If MCF places

an item in the companion buffer, but OPT also misses on that item, the miss will be classified as a Shared miss, not as a Companion Buffer miss.

Shared misses are charged to MCF in the phase in which they occur. Let m_i^S be the cost charged in phase i by Shared misses. It is also the case that the cost of OPT in phase i must be at least m_i^S. Companion Buffer misses will be charged to the phase in which they occur. Let m_i^{CB} be the cost associated with Companion Buffer misses. Companion to Main misses will be charged to the phase in which the item was placed in the companion buffer. Let m_i^{CM} be the cost associated with Companion to Main misses. Because there are at most two followers in a phase, $m_i^{CB} \leq 2$ and $m_i^{CM} \leq 2$, for all i.

Last we consider the Main to Main misses. OPT does not miss on the reference to X because otherwise it would have been a Shared miss. In order to have a Main to Main miss on a reference to X in phase i, there must have been an *intervening reference* to some item X'. We will only consider the earliest such intervening reference. Because MCF will only place an item in the main cache if it is the first item of its type in the phase, the phase in which X was last in the main cache, the phase in which X' forced X out of the cache, and the phase i in which the Main to Main miss occurred, must all be different phases. This is illustrated in Figure 4.4. We charge each Main to Main miss to the phase in which the intervening reference occurs. We will refer to X as a *saved* item. Main to Main misses will later be referred to as Intervening misses.

We look at phase j. Let m_j^I be the number of intervening references in phase j. In other words, OPT must save m_j^I items during phase j. We are going to show that

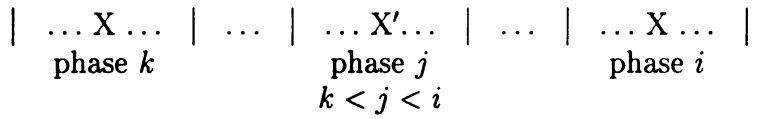

Figure 4.4: Illustration of a Main to Main (Intervening) miss.

$OPT_j \geq 1 + m_j^I$. Let d be the number of distinct types that appear in phase j. By the definition of a phase, there are exactly $d+2$ distinct items in phase j. There are a total of $d+1$ locations in which to place these items, the d locations in the main cache corresponding to the d types, plus the companion buffer. However, m_j^I of the cache locations must be used to hold saved items, so we are left with $d+1-m_j^I$ locations in which to place $d+2$ items. The result is that OPT must miss at least $1+m_j^I$ times in phase j.

The total cost charged to MCF in phase i is

$$c_i = m_i^S + m_i^{CB} + m_i^{CM} + m_i^I.$$

The cost of OPT in phase i is

$$OPT_i \geq \max(1 + m_i^I, m_i^S).$$

If $m_i^S \geq 1 + m_i^I$, then

$$\begin{aligned}
\frac{c_i}{OPT_i} &\leq \frac{m_i^S + m_i^{CB} + m_i^{CM} + m_i^I}{m_i^S} \\
&\leq \frac{m_i^S + 4 + m_i^I}{m_i^S} \\
&\leq \frac{2m_i^S + 3}{m_i^S} \\
&\leq 5.
\end{aligned}$$

The other case is if $1 + m_i^I > m_i^S$. Then

$$\begin{aligned}
\frac{c_i}{OPT_i} &\leq \frac{m_i^S + m_i^{CB} + m_i^{CM} + m_i^I}{1 + m_i^I} \\
&\leq \frac{m_i^S + 4 + m_i^I}{1 + m_i^I} \\
&\leq \frac{2m_i^I + 5}{1 + m_i^I} \\
&\leq 5.
\end{aligned}$$

□

This bound is tight, as the example in Figure 4.5 shows. The two phases shown each have length 5 and are repeated indefinitely. MCF misses on every reference in a phase. Other than the six compulsory misses, OPT only misses on the first reference of each phase, which it chooses to bypass. This lower bound example is interesting because the most recently evicted item is not always the next item requested. In traditional cache scheduling, the worst case for an on-line algorithm is when the most recently evicted item is always the next item in the sequence.

Sequence:	...	X	X'	Z'	Y	Y'	Z	Z'	X'	Y'	Y	...
MCF:		$\underline{M_1}$	$\underline{B_1}$	$\underline{M_3}$	$\underline{M_2}$	$\underline{B_1}$	$\underline{M_3}$	$\underline{B_1}$	$\underline{M_1}$	$\underline{M_2}$	$\underline{B_1}$	
OPT:		$\underline{\cdot}$	M_1	M_3	M_2	B_1	$\underline{\cdot}$	M_3	M_1	B_1	M_2	

Figure 4.5: A Worst Case Input for MCF on CCS($m, 1$)

We can generalize this result for CCS(m, n). The definition of a phase remains the same. Each phase will now contain $n + 1$ leaders and $n + 1$ followers. We generalize the MCF algorithm as follows. Each item is placed in the main cache if it is the first item of its type encountered in the phase. Otherwise, the item is placed in the least recently used location in the companion buffer.

Theorem 4.1.3. *MCF is* $(2n+3)$*-competitive on CCS(m, n) if bypassing is allowed.*

Proof. The proof is a straightforward generalization of the proof of Theorem 4.1.2. As before, MCF will miss at most once on an item per phase. Items placed in the main cache will remain in the main cache throughout the phase. Because items are placed in the companion buffer using a least recently used policy, each follower will be placed in a different location in the companion buffer and will remain in the companion buffer throughout the phase, except for the first follower. The first follower will be evicted by the reference to the $n + 1^{st}$ follower, which ends the phase.

Misses are divided into the same four categories and on-line misses are charged to phases as in Theorem 4.1.2. For each phase $c_i = m_i^S + m_i^{CB} + m_i^{CM} + m_i^I$. Clearly for each phase $OPT_i \geq m_i^S$. Each phase contains exactly one more item than can be simultaneously put in the cache, and each saved item uses one of the needed cache

locations, so $OPT_i \geq 1 + m_i^I$. The only real difference is that $m_i^{CB} \leq n + 1$ and $m_i^{CM} \leq n + 1$. Therefore,

$$\begin{aligned} \frac{c_i}{OPT_i} &\leq \frac{m_i^S + m_i^{CB} + m_i^{CM} + m_i^I}{\max(m_i^S, 1 + m_i^I)} \\ &\leq \frac{m_i^S + (2n+2) + m_i^I}{\max(m_i^S, 1 + m_i^I)} \\ &\leq 2n + 3 \end{aligned}$$

□

This bound is tight. The example in Figure 4.5 can be generalized by replacing the single pair Y, Y′, with n pairs of different types.

We now look at the First In First Out (FIFO) replacement policy. When a miss occurs on an item X, FIFO will consider the set of items occupying cache locations in which X can be placed, and evict the item in that set that has been in the cache the longest. Without loss of generality, we assume that items will be placed in the main cache initially. FIFO is competitive on a CCS(m, n), even though LRU is not. In traditional on-line cache scheduling, both FIFO and LRU are strongly competitive.

In order to show that FIFO is competitive we will use the phases described before, even though FIFO does not use phases. Unlike MCF, it is possible for FIFO to miss an item twice in a phase, as the example for CCS$(m, 1)$ in Figure 4.6 shows. We will call multiple misses on the same item *Repeat misses*. We first prove a bound on the number of Repeat misses in a phase.

Sequence:	X	Y	Y′	Z	X′	Z′	X	X′	Z′	Y′	Y
FIFO:	$\underline{M_1}$	$\underline{M_2}$	$\underline{B_1}$	$\underline{M_3}$	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_3}$	$\underline{M_2}$	$\underline{B_1}$

Figure 4.6: FIFO can miss an item more than once per phase.

Lemma 4.1.1. *FIFO will have at most $2n$ Repeat misses in a phase.*

Proof. We will say that an item X currently in the cache is *fresh* if FIFO placed X in the cache in the current phase. An item X is *stale* if FIFO placed X in the cache in a previous phase. In order for a Repeat miss to occur, FIFO must evict a fresh item. We will show an item Y can be responsible for evicting at most one fresh item in a phase. We will also show that Y must be a follower, or a repeated leader. The phase ends with the reference to the $n + 1^{st}$ follower, so at most $2n + 1$ fresh items will be evicted, and at most $2n$ of the evicted items can be requested a second time for a total of $2n$ repeat misses.

Suppose FIFO places Y in the main cache and evicts the fresh item Y′. Between the time Y′ was put in the main cache and the time Y was put in the main cache, all of the companion buffer locations must have been used, or else Y would have been placed in the companion buffer. Y is a follower or a repeated leader and so at least one follower has appeared. In order for Y to evict a second item, it must first be evicted itself. This requires all of the companion buffer locations to be used a second time. In order for FIFO to reuse a companion buffer location for an item Z, FIFO must have placed an item Z′ of the same type in the main cache. Otherwise Z would be placed in the main cache. Therefore, n more followers must appear before the

companion buffer locations will be reused, and $n + 1$ followers will appear and the phase will end before Y can be evicted.

Suppose FIFO places Y in the companion buffer and evicts the fresh item X. FIFO must have placed an item Y′ in the main cache between the time X was put in the cache and the time Y was put in the cache, or else Y would be put in the main cache. Therefore, Y is a follower or a repeated leader. It is also the case that all of the companion buffer locations must be used between the time X was referenced and Y was referenced, or else X would not be evicted. In order for Y to evict a second item, it must be evicted itself, which requires all of the companion buffer locations to be reused a second time, which will not happen until $n + 1$ followers have appeared in the phase. □

At this point we can prove that FIFO is $m + 3n + 1$ competitive. Each phase contains at most $m + n + 1$ distinct items, and FIFO can miss more than once on at most $2n$ items, for a total of $m + 3n + 1$ misses in a phase. OPT must miss at least once per phase. We can do better and prove a competitive ratio independent of m.

Theorem 4.1.4. *FIFO is* $(5n + 4)$*-competitive on CCS*(m, n) *when bypassing is allowed.*

Proof. The proof is similar to the proof of Theorem 4.1.2. Each miss in the FIFO schedule will be charged to one of the phases, and at the same time we will find a lower bound on the cost of the optimal off-line schedule. In each phase, FIFO does not necessarily use the main cache first. For this reason, categorizing the misses with

respect to whether the item is or was in the main cache or the companion buffer is not helpful. Instead we divide the misses into the following categories.

- Shared misses. Both FIFO and OPT miss on the reference to X.
- Repeat misses. This is not the first appearance of X in the phase.
- Follower misses. X is a follower in the phase.
- Intervening misses. A miss on X is an Intervening miss if there was an item X′ of the same type as X referenced since X was last referenced, and if the previous reference to X, the reference to X′, and the current reference to X all occurred in separate phases. This is the case illustrated in Figure 4.4.
- Pair misses. X was a leader or a follower the last time it was placed in the cache.
- Other misses. All other misses.

The categories are listed in order of precedence and each miss is placed in exactly one category. For example, in order for a miss to be an Intervening miss, OPT cannot miss on that reference; otherwise it would be a Shared miss. Likewise the miss cannot be the second miss on that item in the phase, and the item cannot be a follower in the phase. Otherwise the miss would be a Repeat miss or a Follower miss.

Shared misses are charged to the phase in which they occur. Let m_i^S be the number of Shared misses in phase i. Clearly $OPT_i \geq m_i^S$. Follower misses are also charged to the phase in which they occur. Let m_i^F be the number of follower misses in phase i. For each phase, $m_i^F \leq n + 1$.

Repeat misses are charged to the phase in which they occur. Let m_i^R be the number of repeat misses in phase i. By Lemma 4.1.1, there can be at most $2n$ Repeat misses in a phase, so $m_i^R \leq 2n$.

Intervening misses are handled the same as Main to Main misses in the proof of Theorem 4.1.2. The Intervening misses are charged to the phase in which the intervening reference occurs. Let m_i^I be the number of intervening misses charged to phase i. OPT did not miss on X or else this would have been a Shared miss, and the situation described in Figure 4.4 is true; therefore $OPT_i \geq 1 + m_i^I$.

Pair misses are charged to the phase in which the item was a leader or a follower. Let m_i^P be the number of Pair misses charged to phase i. It is not possible for both a leader and a follower of the same type and in the same phase to be responsible for a Pair miss. Consider an X and X' that appear in the same phase. If X and X' later appear in different phases, one of them will be a Shared miss or an Intervening miss. If they appear in the same phase, one of them is a Follower miss. Therefore, for each phase, $m_i^P \leq n + 1$.

We now consider the Other misses. X cannot be a follower, there cannot be an intervening reference, and X was not a leader or follower the last time it was referenced, or else the miss on X would have been classified as a Follower, Intervening, or Pair miss. If FIFO last put X in the main cache, X would still be in the main cache, because no item of the same type was referenced since X was last referenced. Therefore, FIFO must have put X in the companion buffer the last time it was put in the cache. The situation is illustrated in Figure 4.7. The reference to X' in phase $j-1$ must exist, or else FIFO would have placed X in the main cache in phase j.

FIFO will put X in the main cache in phase i because FIFO has not used X's main cache location since phase $j-1$.

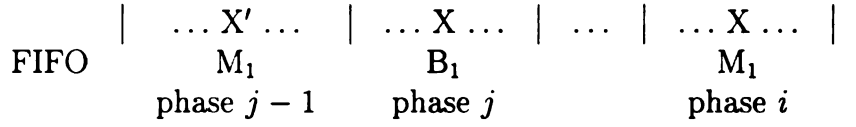

Figure 4.7: Illustration of a FIFO Other miss.

Suppose that OPT does miss on X in phase j. Because X is neither a leader nor a follower in phase j, this is an extra miss for OPT. OPT would still have $1 + m_j^I$ misses in phase j even if X did not appear in the phase. We charge the miss on X in phase i to m_j^E, and $OPT_j \geq 1 + m_j^I + m_j^E$.

Otherwise, we charge the miss on X in phase i to the same phase we charged the miss on X in phase j. Note that if FIFO did not miss on X in phase j, we charge the miss on X in phase i to the phase we would have charged the miss in phase j. The miss on X in phase j must be an Intervening miss or a Pair miss. Therefore, we possibly charge each Intervening miss or Pair miss twice. Note, that it is not possible for an Other miss on X in a phase later than i to also be added to the charge on the miss on X in phase j because FIFO puts X into the main cache in phase i.

The total cost charged to phase i is the following.

$$\begin{aligned} c_i &= m_i^S + m_i^F + 2m_i^I + 2m_i^P + m_i^R + m_i^E \\ &\leq (5n+3) + m_i^S + 2m_i^I + m_i^E \end{aligned}$$

The terms $2m_i^I$ and $2m_i^P$ appear because in some cases we charge the Intervening and Pair misses twice. Because OPT_i is bounded by $1 + m_i^I$, double charging the Intervening misses is not detrimental. The cost for OPT in phase i is the following.

$$OPT_i \geq \max(m_i^S, 1 + m_i^I + m_i^E)$$

Therefore,

$$\begin{aligned} \frac{c_i}{OPT_i} &\leq \frac{(5n+3) + m_i^S + 2m_i^I + m_i^E}{\max(m_i^S, 1 + m_i^I + m_i^E)} \\ &\leq 5n + 4 \end{aligned}$$

□

We do not believe this bound is tight. We have not been able to find an input sequence on which FIFO is worse than $(2n+3)$-competitive. FIFO behaves the same as MCF on the example in Figure 4.5.

We now present a lower bound argument for CCS$(n+1, n)$.

Theorem 4.1.5. *No on-line algorithm is better than* $(2n+2)$*-competitive for CCS*$(n+1, n)$ *if bypassing is allowed.*

Proof. We consider the adversary that always requests the most recently evicted item. In CCS$(n+1, n)$ there are $n+1$ types of items and $2n+1$ cache locations. The adversary will use 2 items of each type for a total of $2n+2$ items, and will reference each item once to begin the input sequence. Let l be the length of the input sequence. Because the adversary always requests the item most recently evicted, the cost for

any on-line algorithm is l. Because there are $2n+2$ items, some item is referenced no more than $\frac{l}{2n+2}$ times. OPT simply bypasses this item and keeps the other $2n+1$ items in the cache for a cost of at most $\frac{l}{2n+2}+2n+1$, where $2n+1$ is the cost of the compulsory misses for the other items. Therefore, no on-line algorithm is better than $(2n+2)$-competitive for CCS$(n+1,n)$. □

Corollary 4.1.1. *No on-line algorithm is better than 4-competitive for CCS(2, 1) if bypassing is allowed.*

As noted earlier, in the worst case example described in Figure 4.5, the most recently evicted item is not the next item in the request sequence. Against an adversary that always requests the most recently evicted item, MCF is $(2n+2)$-competitive. This is because there will be at most $2n+2$ distinct items in a phase, and MCF will miss at most once on each item in each phase. For CCS$(2,1)$, there are only two types, and there can be at most 4 distinct items in a phase. Therefore, MCF is 4-competitive and optimal for CCS$(2,1)$.

4.1.2 Companion Cache Scheduling without Bypassing

We now consider the case when bypassing is not allowed. We define a B-phase to be a maximal set of consecutive items that simultaneously fit in the cache. For CCS(m,n), each B-phase now contains exactly n leaders and n followers. Using the B-phases, we have a new version of MCF, which we will call MCF_2.

Theorem 4.1.6. *MCF_2 is $(2n+2)$-competitive for CCS(m,n) when bypassing is not allowed.*

Proof. It is not true that an algorithm must have a cache miss during a B-phase. For this reason we will define *periods*. A period begins with the second reference in a B-phase, and it includes and ends with the first reference of the next B-phase. Every period is guaranteed to have at least one miss. By definition, the items in a period and the item preceding the period cannot all fit into the cache, and because bypassing is not allowed, the item X preceding the period must be in the cache after it is referenced. It follows that some item referenced in the period cannot be in the cache after X is referenced, and there must be at least one miss during the period. Figure 4.1.2 illustrates the definition of B-phases and periods for CCS$(m, 1)$. Cache misses are underlined. Note that there are no cache misses in B-phase 2.

Y	Y′	X′	X	Y′	X′	Y	X′	X	Y′
$\underline{M_2}$	$\underline{B_1}$	$\underline{M_1}$	$\underline{M_1}$	B_1	$\underline{B_1}$	M_2	B_1	M_1	$\underline{M_2}$
B-phase 0			B-phase 1			B-phase 2			
period 0			period 1			period 2			

Figure 4.8: B-phases and Periods for CCS$(m, 1)$

The proof is similar to the bypassing case, except that we will use periods in the accounting scheme. Let c_i be the cost of MCF_2 in period i, and OPT_i be the cost of OPT in period i. Misses will still be categorized the same way. In each period MCF_2 places at most n items in the companion cache. Therefore, $m_i^{CB} \leq n$ and $m_i^{CM} \leq n$. Shared misses are handled in the same way. Clearly, $OPT_i \geq m_i^S$.

We now consider Main to Main misses. The situation described in Figure 4.4 does not apply to periods. The two references to X and the reference X′ need not occur in three separate periods. It is possible that the intervening reference to X′ occurs

in the same period as one of the references to X. In order for this to happen, either the intervening reference or the Main to Main miss must be the last reference in the period, which is equivalent to being the first reference of a B-phase. For this reason, we can only argue that $OPT_i \geq m_i^I$.

There must be at least one miss per period, so $OPT_i \geq \max(1, m_i^S, m_i^I)$. Therefore, for each period

$$\frac{c_i}{OPT_i} \leq \frac{2n + m_i^S + m_i^I}{\max(1, m_i^S, m_i^I)} \leq 2n + 2$$

□

Corollary 4.1.2. *MCF_2 is 4-competitive for CCS(m, 1) when bypassing is not allowed.*

We do not believe that this upper bound is tight. The best lower bound examples we can find have a competitive ratio of $2n + 1$.

It is worth noting that MCF_2 is a different algorithm than MCF. MCF_2 is 3-competitive on CCS(2, 1) when bypassing is not allowed because each B-phase can contain at most 3 distinct items, and MCF_2 misses at most once on an item in a B-phase. If bypassing is not allowed, then MCF is at best 4-competitive for CCS(2, 1). If bypassing is allowed, then MCF_2 is at best 6-competitive. Figure 4.9 demonstrates the different behavior of the two algorithms on different input sequences.

We now consider FIFO when bypassing is not allowed.

Theorem 4.1.7. *FIFO is $(4n + 2)$-competitive when bypassing is not allowed.*

Proof. The proof is the same as the proof of Theorem 4.1.4. The only difference is that a B-phase contains only n followers. □

	X	X′	Y	Y′	X′	X	Y′	Y	X	X′	Y	Y′	X′	X	Y′	Y	...
MCF:	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_2}$	$\underline{B_1}$	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_2}$	$\underline{B_1}$	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_2}$	$\underline{B_1}$	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_2}$	$\underline{B_1}$	
MCF_2:	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_2}$	$\underline{M_2}$	B_1	M_1	M_2	$\underline{M_2}$	M_1	B_1	M_2	$\underline{M_2}$	B_1	M_1	M_2	$\underline{M_2}$	
OPT:	$\underline{M_1}$	$\underline{M_1}$	$\underline{M_2}$	$\underline{B_1}$	M_1	$\underline{M_1}$	B_1	M_2	M_1	$\underline{M_1}$	M_2	B_1	M_1	$\underline{M_1}$	B_1	M_2	

(a) MCF vs MCF_2 without bypassing

	X	X′	Y	Y′	Y	X′	X	X′	Y	Y′	Y	X′	X	X′	Y	Y′	...
MCF:	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_2}$	$\underline{B_1}$	M_2	$\underline{M_1}$	$\underline{B_1}$	M_1	M_2	$\underline{B_1}$	M_2	M_1	$\underline{B_1}$	M_1	M_2	$\underline{B_1}$	
MCF_2:	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_2}$	$\underline{M_2}$	$\underline{B_1}$	$\underline{M_1}$	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_2}$	$\underline{M_2}$	$\underline{B_1}$	$\underline{M_1}$	$\underline{M_1}$	$\underline{B_1}$	$\underline{M_2}$	$\underline{M_2}$	
OPT:	$\underline{\cdot}$	$\underline{M_1}$	$\underline{M_2}$	$\underline{B_1}$	M_2	M_1	$\underline{\cdot}$	M_1	M_2	B_1	M_2	M_1	$\underline{\cdot}$	M_1	M_2	B_1	

(b) MCF vs. MCF_2 with bypassing

Figure 4.9: Different behaviors of MCF and MCF_2

When bypassing is not allowed, we can find cases where MCF_2 performs better than FIFO. In the input described in Figure 4.9(a), FIFO behaves the same as MCF, so FIFO is at best 4-competitive on CCS(2, 1), whereas MCF_2 is 3-competitive on CCS(2, 1).

We now prove a lower bound using a language theory argument.

Theorem 4.1.8. *No algorithm is better than 3-competitive for CCS(2, 1) when bypassing is not allowed.*

Proof. We consider the adversary that always requests the most recently evicted item. There are four items, X, X′, Y and Y′. The request sequence is a string over the alphabet $\Sigma = \{X, X', Y, Y'\}$. We define an NFA F_1 that accepts all the input strings the adversary may generate for any on-line algorithm. Each state of the machine represents a possible cache configuration and is an accepting state. The state [X, Y, X′] represents X and Y in the main cache, and X′ in the companion buffer.

If this is the current cache configuration, then the next item requested will be Y′. If the on-line algorithm places Y′ in the main cache, then the next configuration will be [X, Y′, X′]. If the on-line algorithm places Y′ in the companion buffer, then the next configuration will be [X, Y, Y′]. The entire NFA is depicted in Figure 4.10. Without loss of generality, we assume that X, Y and X′ are in the cache at the beginning. That means that the initial states are [X, Y, X′] and [X′, Y, X].

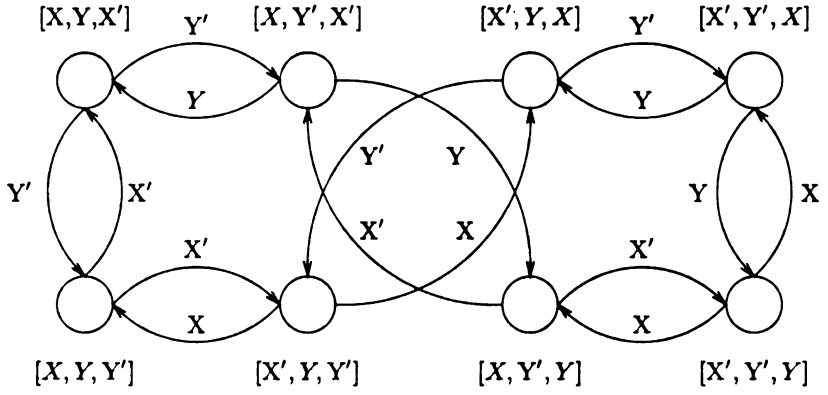

Figure 4.10: The NFA F_1.

We define a second NFA F_2. The strings accepted by F_2 represent input sequences for which at most a third of the references are misses in the optimal solution, not counting the initial compulsory misses. Note, F_2 does not represent all possible input sequences for which at most a third of the references are misses in the optimal solution. For each cache configuration there will be 4 states, all of which are accepting states. Transitions between states corresponding to the same cache configuration represent hits. Transitions between states corresponding to different cache configurations represent misses. The NFA is constructed such that between a pair of misses there are at least two hits, unless there were three or more hits between the previous two misses, in which case there is at least one hit between the pair of misses. A subset of

the NFA is depicted in Figure 4.11. Only three of the eight configurations are shown, and only the first configuration has all of its states and transitions labeled. For any input sequence accepted by F_2, at most a third of the references will be misses in the optimal solution.

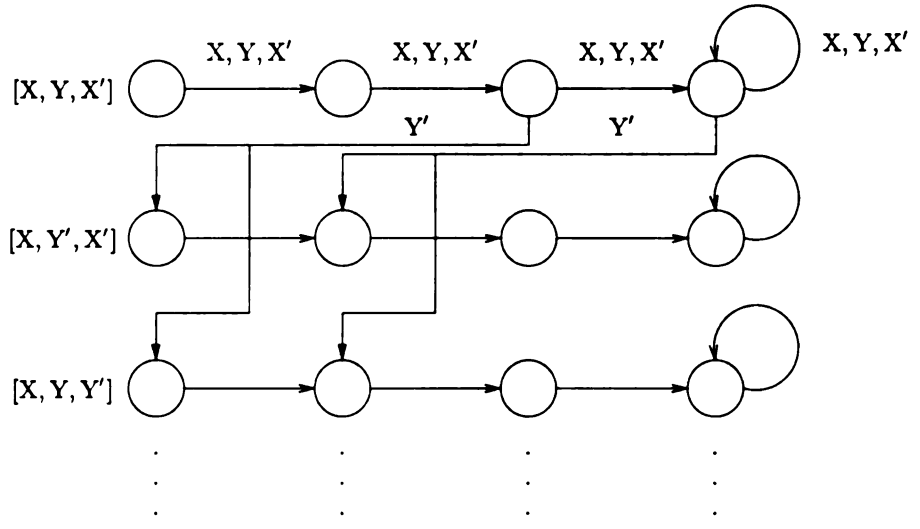

Figure 4.11: Part of NFA F_2.

Using standard NFA constructions it can be shown that the language accepted by F_1, $L(F_1)$, is a subset of $L(F_2)$. We first construct deterministic machines equivalent to F_1 and F_2. We can then use the difference construction to construct a machine that accepts $L(F_1) - L(F_2)$. The resulting machine has no reachable accepting states, so it follows that $L(F_1)$ is a subset of $L(F_2)$. Clearly any on-line algorithm misses on every reference against the adversary that always requests the most recently evicted item. For every possible input sequence generated by any on-line algorithm against this adversary, the optimal algorithm will fault at most a third of the time. Therefore, no on-line algorithm is better than 3-competitive for CCS(2, 1). □

In theory the above argument could be used to prove a lower bound for CCS($n+1, n$) for any given value of n. Unfortunately, the deterministic equivalent of F_2 becomes exceedingly large. For CCS(2, 1), F_2 produces a deterministic machine with nearly 13000 states. For $n = 2$, F_2 becomes so large that constructing the corresponding deterministic machine is computationally intractable. The same argument could also be used for the case where bypassing is allowed, but even for CCS(2, 1) finding the deterministic equivalent of F_2 is intractable, and it is not clear which adversary to use.

Using other techniques we can prove the following general lower bound.

Theorem 4.1.9. *No algorithm is better than $(n+1)$-competitive for CCS($n+1, n$) when bypassing is not allowed.*

Proof. The proof is similar to the proof of Theorem 4.1.5 We consider the adversary that always requests the most recently evicted item. In CCS($n+1, n$), there are $n+1$ types of items and $2n+1$ cache locations. The adversary will use 2 items of each type for a total of $2n+2$ items, and will reference each item once to begin the input sequence. Let l be the length of the input sequence. Because the adversary always requests the item most recently evicted, the cost for any on-line algorithm is l. Because there are $2n+2$ items, some item is referenced no more than $\frac{l}{2n+2}$ times. Let this item be X, and let X' be the other item of the same type. In one off-line solution, X and X' share the same location in the main cache. The other $2n$ items are each given their own location either in the main cache or in the companion buffer. In the worst case, each reference to X' will be a miss, and each following reference to

X will be a miss, and all other references will be hits, not counting the compulsory misses. The total cost for off-line is at most $\frac{2l}{2n+2} + 2n + 1$. Therefore, no on-line algorithm is better than $(n+1)$-competitive for CCS$(n+1, n)$. □

4.1.3 Companion Cache Scheduling with Reorganization

We now consider caches with the ability to reorganize. When a hit occurs, the cache is free to move items within the cache. Because the exact location in which an item is located is important, reorganization may be beneficial. Some proposed restricted cache designs such as the assist cache and the victim cache allow reorganization [32, 13].

If the cache is allowed to reorganize itself, we can define a competitive variant of LRU for CCS(m, n) as follows. Whenever a miss occurs, the item is placed in the main cache. If the item currently occupying that location in the main cache was referenced more recently than the least recently used item in the companion buffer, then it replaces the least recently used item in the companion buffer. Otherwise it is evicted from the cache entirely. If a hit occurs on an item in the companion buffer, the referenced item is swapped with the item of the same type in the main cache. We will call this algorithm Reorganize LRU (RLRU). At all times, the main cache will contain the most recently used item of each type, and the companion buffer will contain the most recently used items not in the main cache. This algorithm is essentially the proposed replacement strategy for victim caches.

Theorem 4.1.10. *RLRU is* $(2n+2)$*-competitive for CCS*(m, n) *when bypassing and reorganization are allowed.*

Proof. The proof is similar to Theorem 4.1.3. The input sequence is again divided into phases, even though the algorithm does not use phases.

First we prove that for any item, RLRU will miss on a reference to that item at most once per phase. Consider an item X. The first time X appears in a phase, it will be placed in the main cache. In order for RLRU to miss on the next reference to X, an X′ must be referenced to force X out of the main cache. One of the following two things must then happen: either X is evicted from the cache entirely when X′ is referenced, because all the items in the companion buffer were referenced more recently than X; or else X was moved to the companion buffer and later evicted.

Suppose X is moved to the companion buffer when X′ is referenced. In order for an item Y to possibly evict X from the companion buffer, it must be referenced after X. Furthermore, for Y to evict X, a follower of Y must be referenced after X. Because a LRU policy is used during reorganization, X will not be evicted before n such followers have been referenced. X′ is also a follower, so X will not be evicted before $n+1$ followers have appeared in the phase. In other words, X will not evicted before the last reference in the phase.

Suppose X is not moved to the companion buffer and is instead evicted from the cache. In order for all the items in the companion buffer to have been referenced more recently than X when X′ is referenced, n items referenced after X must be moved into the companion buffer before X′ is referenced. This requires each of these items to

have a follower referenced before X'. Therefore, X' must be the $n+1^{st}$ follower, and the end of the phase.

The remainder of the proof closely follows the proofs of Theorem 4.1.3 and Theorem 4.1.4. We divide the on-line misses into the following categories.

- Shared misses. Both RLRU and OPT miss on the reference to X.
- Follower misses. X is a follower in the phase.
- Intervening misses. See Figure 4.4.
- Pair misses. X is a Pair miss if X was a leader or a follower the last time it was put in the cache.

The categories are listed in order of precedence. A miss will be classified as an Intervening miss only if it is not a Follower miss or a Shared miss.

Shared and Follower misses are charged to the phase in which they occur. Let m_i^S and m_i^F be the number of Shared and Follower misses in phase i. Pair misses are charged to the phase in which the item was last put in the cache, and m_i^P is the number of Pair misses charged to phase i. As in Theorem 4.1.4, it is not possible for both a leader and a follower of the same type and in the same phase to be responsible for a Pair miss, so $m_i^P \leq n+1$. Intervening misses are charged to the phase in which the intervening reference occurs. There are no other types of misses. If X is the only item of its type in the phase, it will be put in the main cache and will remain in the main cache until another item X' is referenced. The next miss on X must be either a Follower miss or an Intervening miss.

Clearly reorganization does not help OPT with Shared misses. For Intervening misses there are still $1+m_i^I$ more items than can be fit in the cache and reorganization does not help, and $OPT_i \geq 1 + m_i^I$. Therefore,

$$\begin{aligned} \frac{c_i}{OPT_i} &\leq \frac{m_i^S + m_i^P + m_i^F + m_i^I}{\max(m_i^S, 1 + m_i^I)} \\ &\leq \frac{m_i^S + (2n+2) + m_i^I}{\max(m_i^S, 1 + m_i^I)} \end{aligned}$$

We consider two cases. First, if $OPT_i \geq 2$, then

$$\begin{aligned} \frac{c_i}{OPT_i} &\leq \frac{m_i^S + (2n+2) + m_i^I}{\max(m_i^S, 1 + m_i^I)} \\ &\leq \frac{2 + (2n+2) + 1}{2} \\ &\leq n + \frac{5}{2}. \end{aligned}$$

For $n \geq 1$, $n + \frac{5}{2} \leq 2n + 2$.

The other case is if $OPT_i = 1$. If $OPT_i = 1$, then $m_i^S \leq 1$, $m_i^I = 0$ and OPT must miss on either a leader or a follower. If OPT misses on a follower, then $m_i^F \leq n$ because one of the followers will be a Shared missed instead of a Follower miss and

$$\begin{aligned} \frac{c_i}{OPT_i} &\leq \frac{m_i^S + m_i^P + m_i^F}{1} \\ &\leq 2n + 2. \end{aligned}$$

If OPT misses on a leader, it must bypass the leader. If it does not bypass the leader, OPT must miss more than once in the phase. Let X be the leader that OPT bypasses. X cannot appear more than once in the phase, because then OPT would miss more than once in the phase. Therefore, at the end of the phase in the RLRU schedule, some follower X′ of X will be in the main cache location occupied by items of type X. This is because the RLRU algorithm moves each item referenced to the main cache. Because X was bypassed, it cannot be responsible for a Pair miss. Because X′ is in the main cache, the next reference to it is either a hit, a Shared miss, an Intervening miss or a Follower miss, so it too cannot be responsible for a Pair miss. Therefore, $m_i^P \leq n$ and

$$\begin{aligned} \frac{c_i}{OPT_i} &\leq \frac{m_i^S + m_i^P + m_i^F}{1} \\ &\leq 2n + 2 \end{aligned}$$

□

This bound is tight, as the example in Figure 4.12 shows for $n = 1$. It is worth noting that if RLRU did not reorganize on a hit in the companion buffer, the algorithm would be subject to the same counter example used in Theorem 4.1.1.

Sequence:	X	X′	Y	Y′	X′	X	Y′	Y	...
RLRU:	M_1	M_1	M_2	M_2	M_1	M_1	M_2	M_2	
	↓		↓		↓		↓		
	B_1		B_1		B_1		B_1		
OPT:	M_1	B_1	M_2	·	B_1	M_1	·	M_2	

Figure 4.12: Lower Bound Example for RLRU

4.1.4 Companion Cache Scheduling with Extra Resources

We now consider extra resource analysis of Companion Cache scheduling. The idea behind extra resource analysis is that for some problems, an on-line algorithm's lack of future knowledge can be offset if the on-line algorithm has extra resources at its disposal. We will consider the case where the on-line algorithm is given a larger companion buffer. This is a very natural idea, as the companion buffer is essentially an extra resource.

Let n be the size of the off-line algorithm's companion buffer and let $n+e$ be the size of the on-line algorithm's companion buffer. In other words, the on-line algorithm has e extra locations in the companion buffer. We will analyze the MCF algorithm defined in Section 4.1.1. Phases will be defined with respect to the on-line algorithm's cache.

Theorem 4.1.11. *MCF is $\frac{2n+3e+3}{e+1}$-competitive on a CCS$(m, n+e)$ when MCF has e extra locations in the companion buffer.*

Proof. The proof is very similar to Theorem 4.1.2. The key difference is that for each phase, $OPT_i \geq 1+e+m_i^I$. Phases are defined with respect to the on-line algorithm's cache. Each phase will contain $n+e+1$ followers, and will contain $e+1$ more items than can fit in the off-line algorithm's cache. If the off-line has m_i^I items "saved" in phase i, the off-line algorithm must have at least $1+e+m_i^I$ misses in phase i.

The Companion Buffer and Main to Companion Buffer misses are both bounded by $n + e + 1$. Therefore,

$$\frac{c_i}{OPT_i} \leq \frac{m_i^S + m_i^{CB} + m_i^{CM} + m_i^I}{\max(1 + e + m_i^I, m_i^S)} \leq \frac{2n + 3e + 3}{1 + e}$$

□

This bound is tight. We can construct a lower bound for any n and e using $n+e+2$ pairs of different types. We divide the types into three groups: A, B, C. There are $e + 1$ types in group A, $e + 1$ types in group B, and n types in group C. We assume that each pair is ordered. In the odd phases we first reference the $e+1$ pairs in group A, referencing the first element of each pair before the second. We then reference the second element of each pair in group B. We then reference the pairs in C. In the even phases we first reference the $e + 1$ pairs in group B, then the second element of each pair in group A, and then the inverted pairs in group C.

Each phase will contain $2n + 3e + 3$ references. MCF will miss on every reference. The off-line algorithm can use the n positions in the companion buffer and the main cache to hold all of the items in group C. It will choose to bypass the first item in each pair from group A and B, and will use the main cache to store the other item. OPT will only miss on the $e + 1$ items it bypasses each phase.

Figure 4.13 illustrates the lower bound for $n = 2, e = 2$, where group A contains X, X', Y, Y', Z, Z' group B contains R, R', S, S', T, T' and group C contains U, U', V, V'.

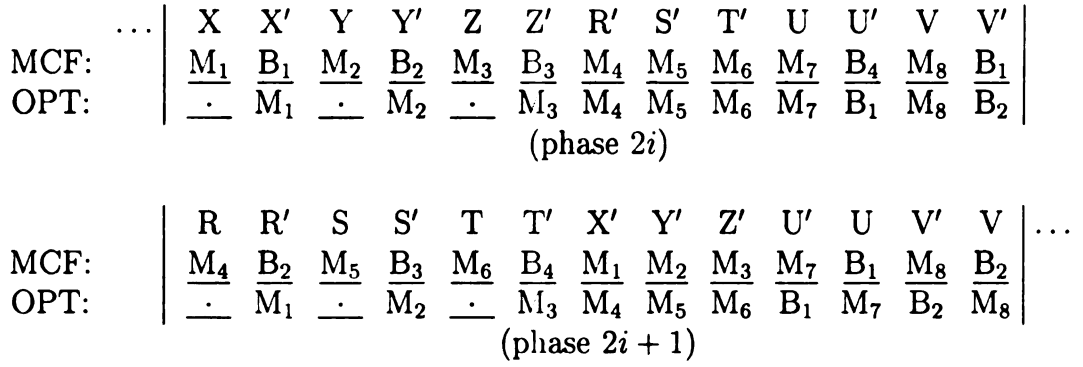

Figure 4.13: Lower Bound Example with Extra Resources

If the size of the companion buffer is doubled, that is $e = n$, the competitive ratio is 5. As the size of companion buffer is increased to infinity, that is $e \to \infty$, the competitive ratio approaches 3.

A curious anamoly occurs if we look at the case where off-line has a simple direct mapped cache of size m, and on-line has a CCS(m, n). If bypassing is allowed, then FIFO (or MCF) is 2-competitive on a direct mapped cache. Surprisingly, if the on-line algorithm has a CCS(m, n), the competitive ratio actually becomes worse, even though the on-line algorithm has a larger cache. If we consider the generalized example of Figure 4.5, FIFO (or MCF) will miss $2n+3$ times each phase. OPT, which only has a direct-mapped cache, will only miss $n+1$ times per phase, for a competitive ratio of $\frac{2n+3}{n+1}$. When $n = 1$, the competitive ratio is 2.5. Similarly, when bypassing is not allowed, then FIFO is 1-competitive. However, if the on-line algorithm has a CCS(m, n), the competitive ratio is $\frac{2n+1}{2n}$.

4.2 Other Restricted Caches

All of our on-line scheduling results have focused on the Companion Cache Structure. We now consider some more complex restricted caches and their corresponding scheduling problems.

4.2.1 The Set-Associative Companion Cache

We can generalize the results of Section 4.1.1 to restricted caches that consist of a set associative cache paired with a fully associative cache. We define the set associative companion cache SACC(b, m, n) as a b-way set associative cache of size bm paired with a fully associative cache of size n.

As before, we will use $\mathrm{B}_1, \ldots \mathrm{B}_n$ to refer to the positions in the companion buffer, and we will use $\mathrm{M}_1^1, \mathrm{M}_1^2, \ldots, \mathrm{M}_1^b, \mathrm{M}_2^1, \ldots, \mathrm{M}_m^b$ to refer to the locations in the main cache. The locations $\mathrm{M}_1^1 \ldots \mathrm{M}_1^b$ form one of the m sets.

The definition of types and phases remains the same. Each item can occupy b locations in the main cache and n locations in the companion buffer. We generalize the MCF algorithm as follows. Within a phase, the algorithm will not put an item in the companion buffer unless it has already seen b items of that type in the phase. A marking algorithm will be applied to both the companion buffer and the sets within the main cache. That is, main cache locations within a set will not be reused until all the locations within the set have been used.

Theorem 4.2.1. *MCF is* $((n+1)(b+1)+b)$*-competitive on SACC*(m, b, n) *if bypassing is allowed.*

Proof. The proof is a generalization of the proof of Theorem 4.1.2. Items put into the companion buffer are charged the same way as before. Otherwise, we only consider the first miss of each type in a phase. These misses we divide up as before into Shared, Companion to Main, and Main to Main misses. Because we only consider the first miss of each type in a phase, each miss we consider can represent b actual misses in the on-line schedule. Otherwise, Shared and Companion to Main misses are charged the same way as before. We only need to consider the Main to Main misses.

In order for a Main to Main miss to occur on an item X, there must have been b different items of the same type as X placed into the main cache since X was last in the main cache or else X would not have been evicted from the main cache. Some of these references must be in a phase other than the phase X was last placed in the main cache and the current phase. There are $b+1$ items including X itself. Either OPT misses on one of these references, in which case we charge the miss to that phase, or else OPT uses one of the companion buffer locations to save an item, in which case we charge the miss to the phase in which the companion buffer is used. The cost of these misses is m_i^I.

The total cost charged to phase i is

$$c_i = bm_i^S + m_i^{CB} + bm_i^{CM} + bm_i^I.$$

The cost of OPT in phase i is

$$OPT_i \geq \max(1 + m_i^I, m_i^S).$$

Therefore,

$$\begin{aligned}
\frac{c_i}{OPT_i} &\leq \frac{bm_i^S + m_i^{CB} + bm_i^{CM} + bm_i^I}{\max(1 + m_i^I, m_i^S)} \\
&\leq \frac{b((n+1) + m_i^S + m_i^I) + (n+1)}{\max(1 + m_i^I, m_i^S)} \\
&\leq b((n+1)+1) + (n+1) \\
&\leq (b+1)(n+1) + b
\end{aligned}$$

□

When $b = 1$ this is the same result as Theorem 4.1.2. This bound is tight, if an arbitrary marking algorithm is used on main cache locations. Figure 4.14 illustrates the lower bound for SACC$(2, m, 1)$. MCF misses 8 times each phase and OPT only misses once each phase.

...	X	X′	X″	Z″	Z′	Y	Y′	Y″	Z	Z′	Z″	X″	X′	Y″	Y′	Y	...
MCF:	$\underline{M_1^1}$	$\underline{M_1^2}$	$\underline{B_1}$	$\underline{M_3^1}$	$\underline{M_3^2}$	$\underline{M_2^1}$	$\underline{M_2^2}$	$\underline{B_1}$	$\underline{M_3^1}$	$\underline{M_3^2}$	$\underline{B_1}$	$\underline{M_1^2}$	$\underline{M_1^1}$	$\underline{M_2^2}$	$\underline{M_2^1}$	$\underline{B_1}$	
OPT:	$\underline{\cdot}$	M_1^1	M_1^2	M_3^2	M_3^1	M_2^1	M_2^2	B_1	$\underline{\cdot}$	M_3^1	M_3^2	M_1^2	M_1^1	B_1	M_2^2	M_2^1	

Figure 4.14: A Worst Case Input for MCF on SACC$(m, 2, 1)$

This lower bound relies on the fact that when MCF reuses the M_1^x locations in the second phase, it reuses the most recently used location. This does not violate the premise of a marking algorithm, as it is free to use any of the M_1^x locations. If we modify MCF so that it uses FIFO or LRU to determine which of the M_1^x locations to use, then the best lower bound example we can find has a performance ratio of $(b+1)(n+1)+1$.

4.2.2 The Skew-Associative Cache

We now look at the skew associative cache. As described in Section 1.5.2, a b-way skew associative cache of size bn (SKEW(b,n)) has b banks containing n cache locations each, and each item can be placed in one location in each bank. As in Section 4.2.1, we will use $M_1^1, M_1^2, \ldots, M_1^b, M_2^1, \ldots, M_m^b$ to refer to the locations in the cache. The locations $M_1^1, \ldots M_n^1$ forms one of the b banks in the cache.

The definitions of type and phase remain the same. There are n^b different types in a SKEW(b,n). We will say an item X is of type $[i_1, \ldots i_b]$ if X can be placed in location $M_{i_j}^j$ for $1 \leq j \leq b$. We will still use X and X′ to refer to items of the same type.

For the CCS(m, n), we introduced the MCF algorithm. MCF is a very intuitive scheduling algorithm for CCS(m, n) and it has the property that no item could be missed more than once in a phase. There is no counter part to the MCF algorithm for a skew cache. A CCS(m, n) is an asymmetrical cache design, so it makes sense to favor one of the cache components over another. This is not the case in a skew cache. All of the banks are the same. Furthermore, there is no on-line algorithm that misses each item at most once per phase.

Theorem 4.2.2. *No on-line algorithm misses each item at most once per phase.*

Proof. Consider an item X of type $[i_1, \ldots, i_b]$ that is requested at the beginning of a phase. Suppose the on-line algorithm places the item in location $M_{i_m}^m$. Let the next b requests be to b different items of type $[j_1, \ldots, j_m = i_m, \ldots, j_b]$ where $i_k \neq j_k$ for $k \neq m$. All b items of the second type can fit in the cache if X is placed elsewhere, so

the phase has not ended. However, one of these $b+1$ items cannot be in the on-line algorithms cache, and this item can be requested a second time in the phase, resulting in a second miss on that item in the phase. □

Theorem 4.2.3. *FIFO is at best* $(4n-2)$*-competitive on a SKEW(2,n).*

Proof. Let X and X′ be items of type $[1,1]$ and Y and Y′ be items of type $[n,n]$. Let Z_1 be an item of type $[2,1]$, Z_2 be an item of type $[2,2]$, Z_3 be an item of type $[3,2]$, etc. and Z_{2n-3} be an item of type $[n,n-1]$.

It is possible to fit one of the pairs, all of the Z_i items, and one item from the remaining pair into the cache at the same time. Figure 4.15 shows how we can construct a phase such that FIFO misses $4n-2$ times and OPT misses only once. FIFO will miss on X,X′,Y and Y′ once per phase, and FIFO will miss on each Z_i twice per phase, for a total of $4n-2$ misses per phase. OPT will only miss once. Note, that in the next phase, X′ will appear before X and Y′ will appear before Y. For this reason, this lower bound also applies to the non-bypassing case. OPT can use M_2^n for both Y and Y′ and still miss only once per phase.

	Z_1	Z_2	Z_3	...	Z_{2n-4}	Z_{2n-3}	Y	Y′	Z_{2n-3}	Z_{2n-4}	...	Z_3	Z_2	Z_1	X	X′	...
FIFO	$\underline{M_2^1}$	$\underline{M_2^2}$	$\underline{M_3^1}$	...	$\underline{M_2^{n-1}}$	$\underline{M_1^n}$	$\underline{M_2^n}$	$\underline{M_1^n}$	$\underline{M_2^{n-1}}$	$\underline{M_1^{n-1}}$	...	$\underline{M_2^2}$	$\underline{M_2^1}$	$\underline{M_1^2}$	$\underline{M_1^1}$	$\underline{M_1^2}$	
OPT	M_2^1	M_2^2	M_3^1	...	M_2^{n-1}	M_1^n	M_2^n	.	M_1^n	M_2^{n-1}	...	M_3^1	M_2^2	M_2^1	M_1^2	M_1^1	

Figure 4.15: A lower bound example for FIFO

□

The above lower bound example is perhaps best visualized as a bipartite graph. Let the vertices represent the cache locations. In particular, let the vertex l_i represent

M_i^1 and let the vertex r_i represent M_i^2. Each item is represented by an edge. An item of type $[i, j]$ is represented by the edge (l_i, r_j). Because some types are represented by more than one item, we are actually constructing a multi-graph. The graph that corresponds to the lower bound example of Theorem 4.2.3 is shown in Figure 4.16.

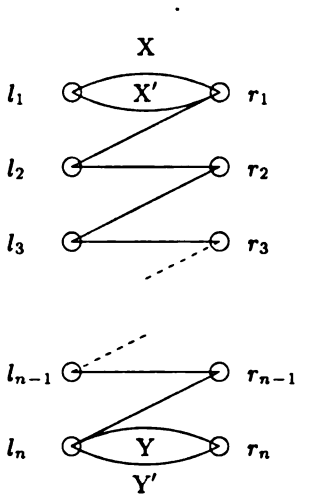

Figure 4.16: Graph Representation of FIFO Lower Bound

The graph contains two cycles, one representing the pair X and X′, and the other representing the pair Y and Y′. The graph also contains a path of length $2n - 3$ connecting the two cycles. All the items in the path can be put in the cache, however this will require using one of the cache locations needed by the pair X and X′, or the pair Y and Y′. When the pair is requested, FIFO will evict an item in the path, and will be forced to "walk" the path, moving each item in the path to a new cache location. When the end of the path is reached, the other pair is requested, and FIFO is forced to "walk" the path a second time.

In general, if the graph corresponding to a set of items contains two cycles connected by a path of length l, then FIFO will have a competitive ratio of

$$2l + |\text{cycle}_1| + |\text{cycle}_2|.$$

This highlights another difference between restricted cache scheduling and traditional fully associative cache scheduling. In a fully associative cache of size n, FIFO will be n-competitive on any collection of $n + 1$ distinct items. In a SKEW(2,n) of size $2n$, FIFO can be anywhere between 2-competitive and $(4n - 2)$-competitive on a collection of $n + 1$ distinct items.

4.3 Summary

We have shown that LRU, which is strongly-competitive for traditional caches, is not competitive for any restricted caches. This is another example of the fundamental difference between restricted cache scheduling and traditional cache scheduling. For the CCS(m, n), we have defined the MCF algorithm, which is competitive, and nearly optimal. We considered the scheduling problem with and without bypassing, reorganization, and extra resources, and defined competitive algorithms for these different cases. We have shown that FIFO is competitive, but not optimal, for CCS(m, n). We have proved that no on-line algorithm can be better than $(2n+2)$-competitive for CCS(m, n), and using language theory we have proved that no on-line algorithm can be better than 3-competitive for CCS$(m, 1)$. We also extended these

results to more complicated restricted caches, such as the set associative companion cache, and the skew cache.

Chapter 5

Cache Design Analysis

Our previous work has studied the behavior of different algorithms on specific cache designs. The goal of this chapter is to develop a theoretical model with which to directly compare different cache designs.

5.1 Metrics

In Section 1.4 we stated that the fully associative cache is the ideal cache design because any item can be placed anywhere in the cache. Unfortunately large fully associative caches cannot be built. The goal of restricted caches is to increase the number of choices regarding item placement with minimal increases in hardware. In general, the more choices there are about item placement, the lower the achievable miss rate will be. In this section we develop a model to quantify this concept of cache flexibility.

5.1.1 Size and Associativity

There are two parameters that describe in some sense a cache's flexibility. The first is the overall size of the cache, and the second is the number of locations each item can be placed in. Clearly the larger either of these values is, the more flexible the cache is. These two parameters completely describe a traditional set associative cache.

These two parameters are not sufficient to describe restricted caches. Both a CCS($2n-1,1$) and a 2-way set associative cache of size $2n$ (SA(2,n)) have $2n$ total locations, and each item can be placed in exactly 2 locations. However the two cache designs are not equivalent. We would suspect the SA(2,n) to be more flexible and have a lower miss rate because in a CCS($2n-1,1$) one of the locations can be occupied by all of the items and the other locations can be used by $\frac{1}{2n-1}$ of the items, whereas in a SA(2,n) each location can be occupied by the same number of items. An even better example is a SA(2,n) and a SKEW(2,n). Again both caches have the same number of locations, and each item can be placed in two locations. Furthermore, the mapping of items to locations in both cases is even. Each location can be occupied by the same fraction of the items. However, the intuition behind the design of the skew cache is that it is more flexible than the same sized set associative cache.

5.1.2 A New Flexibility Metric

We introduce the following method to quantify the flexibility of a cache. We consider the probability that r items chosen at random can simultaneously fit in the cache. The more flexible a cache is, the more likely a random set of items will fit in it. The

inspiration for this metric is the concept of the *working set.* The working set is the set of items currently needed by the program. Due to temporal and spatial locality, the working set tends to be relatively small and remains unchanged for relatively long periods of time. If the entire working set can be placed in the cache, the program will not generate any cache misses until the working set changes.

For a cache C, we define the flexibility function $F(C, r)$ as follows.

$$F(C, r) = \text{probability that any } r \text{ items fit in cache}$$

We will assume that any item is equally likely to be chosen. To simplify the math, we will assume that the main memory is essentially infinite. This allows us to focus on types of items (determined by the cache structure), and we can assume that each type of item is equally likely to be chosen regardless of earlier choices.

5.2 Flexibility of Different Cache Designs

In this section, we calculate $F(C, r)$ for different cache designs.

5.2.1 Set-Associative Caches

We first consider the simplest case, the direct-mapped cache. A direct-mapped cache is a 1-way set associative cache (SA(1,n)). A set of r items will all fit into a direct-mapped cache if all of the items are of a different type. A direct-mapped cache of size n has n different types. The number of ways of choosing r items of n types

such that no type is chosen more than once is $\frac{n!}{(n-r)!}$. The total number of ways of choosing r items of n types is n^r. Therefore, the probability that r items will fit into a direct-mapped cache of size n is

$$F(SA(1,n),r) = \frac{n!}{(n-r)!n^r}$$

For an b-way set associative cache of size bn (SA(b,n)), there are n different types. A set of r items will all fit into the cache if no type appears more than b times. The number of ways of choosing r items of n types such that no type is chosen more than b times is defined by the following multinomial expression.

$$g_b(n,r) = \sum \binom{r}{k_1, k_2, \ldots, k_n}$$

such that $\sum k_i = r$ and $0 \leq k_i \leq b$. Therefore,

$$F(SA(b,n),r) = \frac{g_b(n,r)}{n^r}$$

such that $\sum k_i = r$ and $0 \leq k_i \leq b$.

We can define $g_b(n,r)$ recursively as follows.

$$g_b(n,r) = \sum_{i=0}^{b} \binom{r}{i} g_b(n-1, r-i)$$

The base cases are $g_b(n,0) = 1$, $g_b(1,r) = 1$ if $r \leq b$, and $g_b(1,r) = 0$ if $r > b$. The recurrence is based on the fact that $\binom{r}{k_1,k_2,\ldots,k_n} = \binom{r}{k_1}\binom{r-k_1}{k_2,\ldots,k_n}$. We can then define

$F(SA(b,n),r)$ as follows.

$$F(SA(b,n),r) = \sum_{i=0}^{b} \binom{r}{i} F(SA(b,n-1),r-i) \left(\frac{n-1}{n}\right)^{r-i} n^{-i}$$

This definition has the advantage that it is easy to calculate and is not sensitive to integer overflow.

5.2.2 Companion Caches

We can define $F(CCS(\mathrm{m,n}),r)$ in a similar fashion. There are m types of items. A set of r items will fit into the cache if the total number of excess items chosen is less than or equal to n. An item is an excess item if an item of the same type has already been chosen. Again we can define this using a multinomial expression.

$$h_1(m,n,r) = \sum \binom{r}{k_1, k_2, \ldots, k_m}$$

such that $\sum k_i = r$ and $\sum u(k_i - 1) \leq n$, where $u(k) = k$ if $k \geq 0$, and $u(k) = 0$ if $k < 0$.

This immediately generalizes for SACC(b,m,n), that is a b-way set associative cache of size bm paired with a companion buffer of size n. The number of ways of choosing r items that fit into such a cache is the following.

$$h_b(m,n,r) = \sum \binom{r}{k_1, k_2, \ldots, k_m}$$

such that $\sum k_i = r$ and $\sum u(k_i - b) \leq n$, where $u(k) = k$ if $k \geq 0$, and $u(k) = 0$ if $k < 0$.

Therefore,

$$F(SACC(b,m,n),r) = \frac{h_b(m,n,r)}{m^r}.$$

For calculation purposes, we can define this recursively as follows.

$$F(SACC(b,m,n),r) = \left(\sum_{i=0}^{b} \binom{r}{i} F(SACC(b,m-1,n),r-i) \left(\frac{m-1}{m}\right)^{r-i} m^{-i}\right)$$
$$+ \left(\sum_{i=1}^{n} \binom{r}{i+b} F(SACC(b,m-1,n-i),r-(i+b)) \left(\frac{m-1}{m}\right)^{r-i-b} m^{-i-b}\right)$$

For CCS$(m,1)$ this can be simplified considerably.

$$\frac{r!\left(\binom{m}{r} + \frac{m}{2}\binom{m-1}{r-2}\right)}{m^r}$$

5.2.3 Skew Caches

We now consider SKEW(b,n), the b-way skew associative cache. Determining the flexibility of a skew cache is harder because simply counting the number of times each type appears in a set of r items does not tell us if that set of r items will fit into the cache. For example, in the set represented in Figure 4.16, no item appears more than twice, yet this set cannot fit into the cache. On the other hand, some sets of items where items of the same type appear twice do fit into the cache. Therefore, we will use a graph model to determine if r items will fit into a skew cache.

We can model a set I of r items in a SKEW(b,n) as a b-partite hypergraph with r edges. We construct the graph $G_{b,n}(I)$ as follows. Each vertex represents a cache location, and each item X in I is represented by a hyperedge incident to the vertices that correspond to the cache locations in which X can be placed. Note that because there may be multiple items of the same type, the resulting graph is actually a multi-hypergraph. The r items can all fit into the cache if we can match each item to a cache location, which is the same as matching each edge to one of its incident vertices in $G_{b,n}(I)$. We will call such a matching an edge-vertex matching. Therefore, to determine if a set I of r items can fit into a SKEW(b,n), we need to determine if the $G_{b,n}(I)$ has an edge-vertex matching. To determine how many sets of r items can fit into a SKEW(b,n), we need to determine how many b-partite hypergraphs with r edges have an edge-vertex matching.

We first consider SKEW(2,n). In this case, $G_{2,n}(I)$ is a bipartite graph, such as the one pictured in Figure 4.16. To determine the probability that r items will fit into SKEW(2,n), we need to determine how many bipartite graphs with at most n vertices on the left, and at most n vertices on right, and with r edges have an edge-vertex matching. These graphs are subgraphs of the multigraph formed by doubling all of the edges in $K_{n,n}$.

Theorem 5.2.1. *A graph G has an edge-vertex matching if and only if each connected component of the graph is a tree or unicyclic.*

Proof. We only need to consider connected components. If a connected component is unicyclic, then it consists of a set of trees rooted at a cycle. If the component is a

tree, we can arbitrarily root it at any vertex. Consider the leaves of the trees. Only one edge is incident to each leaf, so for each leaf we can safely match the incident edge with the leaf vertex. We remove the leaves and repeat this procedure until only the single cycle is left. The cycle will contain n vertices and n edges, and the edges can be matched to the vertices in two ways. Therefore if a component is a tree or unicyclic, an edge-vertex matching exists.

If an edge-vertex matching exists in a connected component, then $|E| \leq |V|$. This immediately implies that the connected component is a tree or unicyclic. □

We now need to count the number of subgraphs G of $K_{n,n}$ with r edges such that all the components of G are trees or unicyclic. We consider components that span a subset of l vertices from the left side and m vertices from the right side. Each component is either a tree, or unicyclic. Let $T(l,m)$ be the number of spanning trees on $K_{l,m}$. This is determined by the following equation defined in [44].

$$T(l,m) = l^{m-1}m^{l-1}$$

We now count the unicyclic graphs. There are two types of unicyclic graphs. The first type contains a cycle of length two. This is simply a spanning tree with one of the $l+m-1$ edges duplicated. Clearly there are $(l+m-1)T(l,m)$ such graphs. The other type of unicyclic graph contains a cycle of length greater than two. To count these, we first choose a cycle, and then find a forest that spans the remaining vertices such that the roots of each tree are vertices in the cycle. For this we need to know the number of spanning forests on $K_{l,m}$ such that j trees are rooted at vertices on the

left and k trees are rooted at vertices on the right. We denote this as $H(l, m, j, k)$ and it is also derived in [44].

$$H(l, m, j, k) = (lk + mj - jk)l^{m-k-1}m^{l-j-1} \tag{5.1}$$

Putting the two types of unicyclic graphs together, we get the following equation for the total number of unicyclic graphs that span $K_{l,m}$, which we denote as $U(l, m)$.

$$U(l, m) = (l + m - 1)T(l, m) + \sum_{i=2}^{\min(l,m)} \binom{l}{i}\binom{m}{i}\frac{(i)!(i-1)!}{2} \times \left(\sum_{j=0}^{l-2i}\sum_{k=0}^{m-2i} \binom{l-2i}{j}\binom{m-2i}{k} H(l-2i, m-2i, j, k)\right) \tag{5.2}$$

The outer summation considers cycles containing $2i$ vertices. There are $\binom{l}{i}\binom{m}{i}$ ways to choose the vertices in the cycle and there are $\frac{i!(i-1)!}{2}$ ways of ordering the vertices in the cycle. The second line of the equation counts the number of ways to span the remaining vertices with trees rooted in the cycle. The number of trees rooted in vertices on the left is j, and the number of trees rooted in vertices on the right is k. The number of spanning forests for specific values of i, j and k is determined with Equation 5.1.

Putting the trees and unicyclic graphs together, we can define a recurrence relation for the number of subgraphs of $K_{l,m}$ with r edges that have an edge-vertex matching,

which we denote as $M(l, m, r)$.

$$M(l,m,r) = M(l-1,m,r) + \sum_{i=1}^{r}\sum_{j=1}^{r+1-i}\binom{l-1}{i-1}\binom{m}{j}T(i,j)M(l-i,m-j,r-(i+j-1)) + \sum_{i=1}^{r-1}\sum_{j=1}^{r-i}\binom{l-1}{i-1}\binom{m}{j}U(i,j)M(l-i,m-j,r-(i+j)) \quad (5.3)$$

The base cases are the following.

$$\begin{aligned} M(l,m,0) &= 1 \quad \textit{(There is one graph with 0 edges.)} \\ M(0,m,r) &= 0 \quad \textit{(There are no bipartite graphs with} \\ M(l,0,r) &= 0 \quad \textit{0 vertices on the left or right)} \end{aligned}$$

The idea behind the recurrence relation is the following. We assume that the vertices are ordered, and we use the vertex ordering to order the components. The first component is the component that contains the lowest ordered vertex on the left. The second component is the component that contains the lowest ordered vertex on the left that is not in the first component, and so on. This ordering defines a unique way of constructing each graph one component at a time, and prevents us from counting the same graph more than once.

We consider any subgraph G of $K_{l,m}$ with r edges and look at the component that contains the first vertex on the left. The first line of Equation 5.3 is the case when the

first vertex on the left is not contained in any component. In this case, G is subgraph of $K_{l-1,m}$, so we remove the first vertex on the left and consider the subgraphs of $K_{l-1,m}$ with r edges. The second line in Equation 5.3 is the case when the component that contains the first vertex on the left is a tree that spans i vertices on the left and j vertices on the right. There are $\binom{l-1}{i-1}\binom{m}{j}T(i,j)$ such trees. Each tree has $i+j-1$ edges. The rest of the graph G is therefore a subgraph of $K_{l-i,m-j}$ with $r-(i+j-1)$ edges. The third line in Equation 5.3 is the case when the component that contains the first vertex on the left is unicyclic and spans i vertices on the left and j vertices on the right. There are $\binom{l-1}{i-1}\binom{m}{j}U(i,j)$ such graphs and each graph has $i+j$ edges. The rest of the graph G is therefore a subgraph of $K_{l-i,m-j}$ with $r-(i+j)$ edges.

Equation 5.3 only counts the number of subgraphs of $K_{n,n}$ with r edges that have an edge-vertex matching. To calculate $F(SKEW(2,n),r)$ we need to consider the number of ways a graph can be constructed by choosing one edge at a time. In general this is $r!$. However, if an edge appears twice in a graph, then the number of ways of constructing that graph is $\frac{r!}{2}$, and if a graph contains d edges that appear twice, then the number of ways of constructing that graph is $\frac{r!}{2^d}$. Note that if an edge appears more than twice, the graph does not have an edge-vertex matching.

To account for this, we change the way we count unicyclic components. Let $U'(l,m)$ be defined as follows.

$$U'(l,m) = \frac{(l+m-1)T(l,m)}{2} + \sum_{i=2}^{\min(l,m)} \binom{l}{i}\binom{m}{i}\frac{(2i)!(2i-1)!}{2} \times \left(\sum_{j}^{l-2i}\sum_{k}^{m-2i} \binom{l-2i}{j}\binom{m-2i}{k} H(l-2i, m-2i, j, k)\right)$$

The first line counts the number of components containing a cycle of length two. We divide this result in half, to account for the fact that there are only half as many ways to construct a component with a duplicate edge. The remainder of the equation counts components containing cycles of length four or greater, and is the same as in Equation 5.2.

We can now define $M'(l,m,r)$ the way $M(l,m,r)$ is defined in Equation 5.3, using $U'(l,m)$ instead of $U(l,m)$. The end result is

$$F(SKEW(2,n),r) = \frac{r! \cdot M'(n,n,r)}{n^{2r}}$$

To extend this result to SKEW(b,n) for $b > 2$ we need to count the number of b-partite hypergraphs with r edges that have an edge-vertex matching. Unfortunately we have been unable to count hypergraphs with this property. A hypergraph may have any number of connected "cycles" and still have an edge-vertex matching. For a given set of items it is easy to determine if the items fit into the cache by framing

it as a matching problem, but counting the number of sets with a matching appears to be hard.

5.3 Comparisons

Using this metric we can compare the overall flexibility of different cache designs of different sizes.

5.3.1 Different Cache Designs

The flexibility of different caches of size 64 are plotted in Figure 5.1. In the case of the companion caches, the size of the main cache is 64. The total size of the companion caches is actually larger than 64, Figure 5.2 is a similar plot for graphs of size 128.

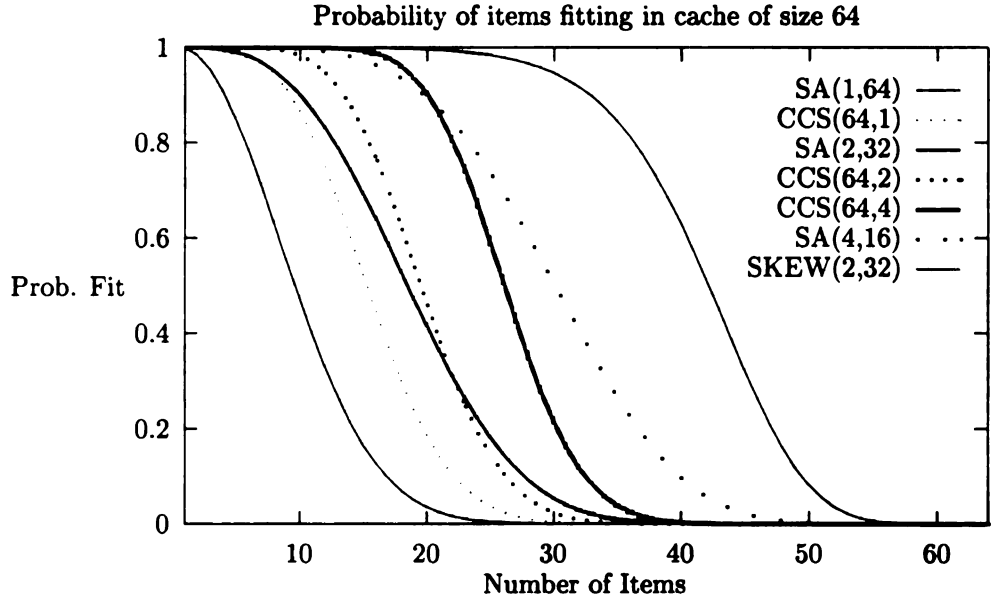

Figure 5.1: Flexibility of Caches of size 64

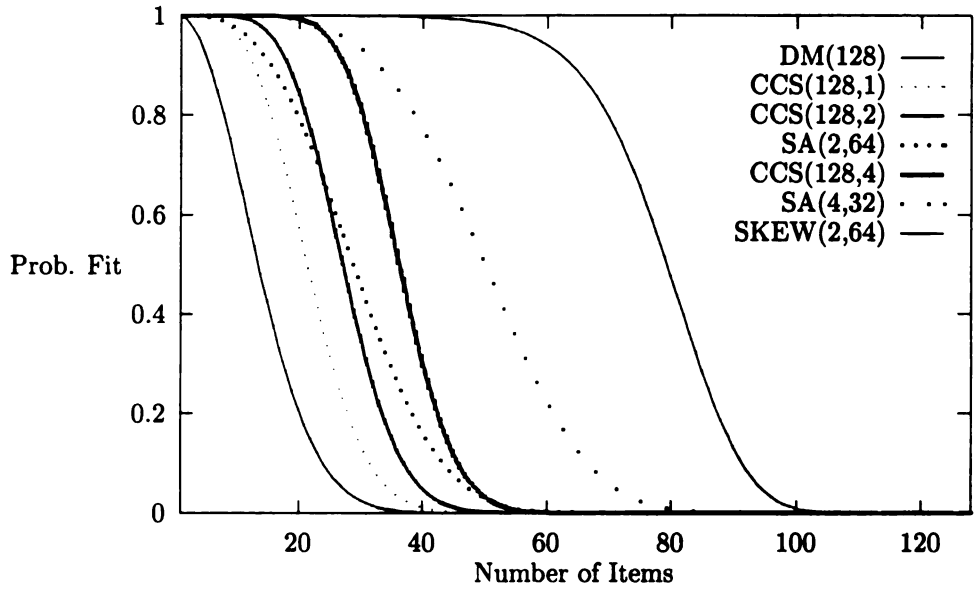

Figure 5.2: Flexibility of Caches of size 128

The following conclusions can be drawn from the graphs. CCS($m, 1$) has a much higher probability of fitting a set of items than a direct mapped cache. CCS($m, 2$) compares favorably with a 2-way set associative cache. However, the larger the cache becomes, the worse CCS($m, 2$) appears in comparison. Finally, a SKEW(2,n) compares very favorably with a 4-way set associative cache. This is consistent with simulation results found in [49].

5.3.2 Different Sized Caches

In this section we compare the flexibility of different size caches. The larger a cache is, the more likely a random set of items will fit into it, regardless of the cache design.

We compare a SKEW(2,32) with a direct mapped cache, a 2-way set associative cache, and a 4-way set associative cache. The sizes of the caches were chosen so that

the total area under each curve is the same. A fully associative cache of size 41 would also have a similar total area under its curve. The results are shown in Figure 5.3.

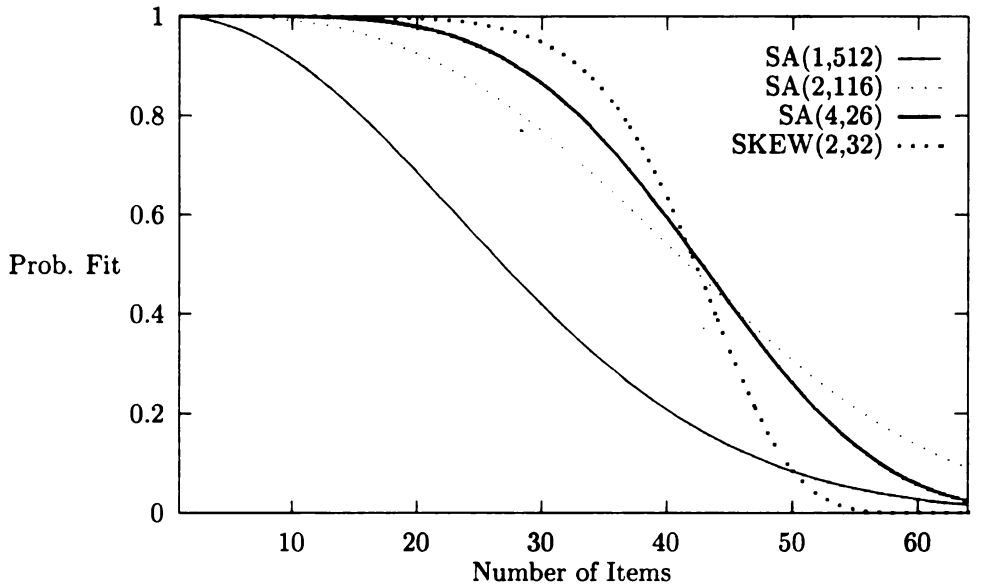

Figure 5.3: Flexibility of Different Size Caches

Unfortunately as size increases, our flexibility metric appears to becomes increasingly pessimistic. This is because in a real system, where spatial locality of reference is present, the probability that multiple items of the same type are requested is lower than if items are randomly chosen. As the cache size increases, the number of types increase, and the number of conflicts decreases. The lower the associativity, the benefits of few conflicts becomes more dramatic. We can determine the optimal miss rate for these caches using trace data. As seen in Table 5.1, the set associative caches do not need to be as large as our flexibility metric suggests. The advantages of a larger cache are documented in [29].

Cache	Trace File #1 #misses	Trace File #2 #misses
SA(41,1)	8827	9955
SA(4,26)	4957	6899
SA(2,116)	3694	5295
SA(4,14)	8220	9831
SA(2,50)	8063	8204

Table 5.1: Simulation results of Different Sized Caches

5.3.3 Comparing CCS(m, n) and Set Associative Caches

As seen in Figure 5.1 and Figure 5.2, as the overall cache size increases, the flexibility of CCS(m, n) decreases relative to a set associative cache for a fixed n. In this section we analyze this phenomena more closely.

We compare CCS(m, n) and $SA(b, \frac{m}{b})$. We will say that a cache C_1 dominates a cache C_2 if for all r, $F(C_1, r) \geq F(C_2, r)$. This means that for any r, a random set of r items is more likely to fit into cache C_1 than into C_2. Unfortunately, as r increases, $F(C_1, r)$ goes to 0, and as the cache size increases, $F(C_1, r)$ is relatively close to 0 for a significant percentage of the values of r. Because it does not make much sense to compare $F(C_1, r)$ and $F(C_2, r)$ when they are both essentially zero, we modify our definition of domination to only consider the "interesting" part of the range. Therefore, we will say that a cache C_1 *ϵ-dominates* a cache C_2 if for all r, $F(C_2, r) \geq \epsilon \rightarrow F(C_1, r) \geq F(C_2, r)$.

For different associativities b and $\epsilon = .01$, we can calculate how large n must be in order for CCS(m,n) to ϵ-dominate $SA(b, \frac{m}{b})$. The relation between m, n, and b for $\epsilon = .01$ is listed in Table 5.2.

Size of m	Size of n		
	2-way	4-way	8-way
64	3	7	12
96	3	9	17
128	4	11	20
160	4	12	24
192	4	14	27
224	5	15	31
256	5	16	34
288	5	18	37
320	5	19	40
352	6	20	42
384	6	21	45
416	6	22	48
448	6	23	50
480	6	24	53
512	6	25	56
1024	8	37	92
2048	11	55	
4096	14	83	

Table 5.2: Values of n needed for CCS(m, n) to ϵ-dominate SA(b,$\frac{m}{b}$) for $\epsilon = .01$

As stated earlier, our flexibility metric is somewhat pessimistic, so the companion buffer need not be so large to achieve similar performance. As the cache size increases, and if spatial locality is in effect, the number of conflicts decreases dramatically, so only a few extra cache locations would actually be necessary.

5.4 Summary

The goal of this section is to develop a model with which to compare cache designs that is independent of specific algorithms and input sequences. We define a metric that attempts to measure how flexible different cache designs are, and captures more information than just the size and associativity of the cache. We show how to calculate

this metric for different cache designs, and then compare different cache designs using this metric.

Chapter 6

Conclusion

Improved cache performance is essential to overall computer performance and becomes more important as the gap between processor speeds and memory speeds continues to increase. New cache designs such as restricted caches are a promising approach to improved cache performance. As demonstrated in this dissertation, restricted caches give rise to a number of interesting and challenging scheduling problems. In this chapter we briefly summarize the contributions made in this dissertation, and discuss further work that can be done in the area of restricted cache scheduling.

6.1 Contributions

The contributions of this thesis can be summarized as follows

- We have shown that the optimal off-line restricted cache scheduling problem is **NP**-complete and **APX**-complete. This is an important result for cache researchers interested in optimal cache performance. We have also shown that

a number of related restricted interval scheduling problems and independent set problems are hard, extending the work of [1].

- We have designed optimal algorithms and polynomial time approximation algorithms for the off-line cache scheduling problems.

- We have presented the first results on on-line restricted cache scheduling. The results show that on-line restricted cache scheduling differs greatly from traditional on-line cache scheduling, and in many ways is a significantly more difficult problem. For the most basic restricted cache, CCS(m, n), we have designed the near optimal MCF algorithm. We presented upper bounds for FIFO, and general lower bounds for the bypassing and the non-bypassing case. We also considered reorganization, and more sophisticated restricted caches such as the SACC(b, m, n) and the skew cache.

- We have defined a metric with which to measure the flexibility of a cache design. The goal is to develop a theoretical model that can be used to directly compare different cache designs.

6.2 Future Work

This section presents additional work that needs to be done in the area of restricted cache scheduling, and also describes other interesting open problems related to this work.

6.2.1 Off-line Cache Scheduling

Given that the off-line problems are for the most part **NP**-complete, future work must focus on approximation algorithms. It seems likely that algorithms with approximation factors much better than 2 exist. Proving an upper bound for the LP relaxation is a good starting point.

The graph model of the off-line problem brings up an interesting independent set problem. In an interval graph, there exist vertices whose neighbors form a single clique, and for an interval graph the independent set problem is easily solvable. In a general graph, each vertex's neighbors may form as many as n cliques, and the best approximation ratio possible is $\sqrt{n}$, assuming $\mathbf{P} \neq \mathbf{NP}$. We have looked at graphs containing vertices whose neighbors form 2 or 3 cliques and have shown that the independent set problem for this graphs is **APX**-complete. An interesting open question is the exact relationship between the number of cliques in a vertex's neighborhood and the approximability of the independent set problem.

6.2.2 On-line Cache Scheduling

For the Companion Cache, we would like to close the gaps between the proven upper and lower bounds for on-line restricted cache scheduling. Improving the lower bound arguments for the bypassing case would require determining exactly how the worst case adversary behaves. The language theory proof for the non-bypassing lower bound is interesting, but does not scale well and does not prove a general lower bound. In both cases, accounting for the fact that the on-line and off-line algorithm can choose

to place items in different locations, and the fact that location is important, will be challenging. We do not believe the upper bounds for MCF in the non-bypassing case, and FIFO in any case, are tight. Improving the upper bound for MCF would be particularly important, as we believe this algorithm is optimal.

We have only touched on the on-line scheduling problem for the skew cache. We would like to find general upper and lower bounds for this problem.

We would also like to study the performance of randomized on-line algorithms. Randomization is another method of overcoming the on-line algorithm's lack of future knowledge. If the on-line algorithm is allowed to make random choices, then the adversary cannot as easily construct a worst case input. There has been considerable work done with randomized identical cache scheduling. However, none of the techniques used generalize to the restricted cache case. The techniques break down because location is important in a restricted technique, and simply keeping track of which items are in the cache is not sufficient. The location of each item must also be accounted for.

Restricted caches have a natural ability to reorganize. As items are repeatedly requested, they will tend to move around in cache (by means of eviction) until it is possible for the current working set to fit in the cache. A theoretical model of this phenomena would be useful for the understanding of restricted cache performance. Our study of flexibility was a result of our initial efforts to study this reorganization property.

Finally, cache scheduling is an instance of the k-server problem. We can generalize restricted cache scheduling to define the restricted k-server problem, where only

certain servers are eligible to service each request. To the best of our knowledge, nothing is known about this problem.

6.2.3 Cache Design Analysis

To improve the usefulness of our flexibility metric we would need to better account for locality. Unfortunately this is likely to greatly complicate the math involved. Another possibility is to better determine which parts of the range of the flexibility function are most significant with respect to cache performance. Also finding closed forms, or good approximations for the flexibility functions of different cache designs, would make working with them more practical. On the purely theoretical side, characterizing and counting general hypergraphs with the edge-vertex matching property is an interesting problem, in addition to being necessary to applying this metric to skew caches with associativities greater than two.

Bibliography

[1] E.A. Arkin and E.B Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18:1–8, 1987.

[2] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and intractability of approximation problems. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 13–22, 1992.

[3] James Aspnes, Yossi Azar, Amos Fiat, Serge Plotkin, and Orli Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 623–631, 1993.

[4] G. Ausiello, P. Crescenzi, Gambosi G., Kann V., A. Marchettie Spaccamela, and M. Protasi. *Complexity and Approximation. Combinatorial Optimization Problems and their Approximability Properties.* Springer-Verlag, Berlin, 1999.

[5] Baruch Awerbuch, Yossi Azar, and Serge Plotkin. Throughput-competitive on-line routing. In *Proceedings of the 34th IEEE Annual Symposium on Foundations of Computer Science*, pages 32–40, 1993.

[6] Yossi Azar, Andrei Z. Broder, and Anna R. Karlin. On-line load balancing. In *Proceedings of the 33rd IEEE Annual Symposium on Foundations of Computer Science*, pages 218–225, 1992.

[7] Yossi Azar, Bala Kalyanasundaram, Serge Plotkin, Kirk R. Pruhs, and Orli Waarts. On-line load balancing of temporary tasks. *Journal of Algorithms*, 22(1):93–110, January 1997.

[8] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):282–288, 1966.

[9] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pages 249–259, 1991.

[10] Mark Brehob, Richard J. Enbody, and Nick Wade. Analysis and replacement for skew-associative caches. Technical Report TR97-32, Michigan State University, Department of Computer Science and Engineering, 1997.

[11] Martin C. Carlisle and Errol L. Lloyd. On the k-coloring of intervals. *Discrete Applied Mathematics*, 59:225–235, 1995.

[12] Michael W. Carter and Craig A. Tovey. When is the classroom assignment problem hard? *European Journal of Operational Research*, 40:S28–S39, 1992.

[13] K.K. Chan, C.C. Hay, J.R. Keller, G.P. Kurpanek, F.X. Schumacher, and J. Zheng. Design of the HP PA7200. *Hewlett-Packard Journal*, February 1996.

[14] Marek Chrobak and John Noga. LRU is better than FIFO. *Algorithmica*, 23(2):180–185, 1999.

[15] V. Reddy Dondeti and Hamilton Emmons. Algorithms for preemptive scheduling of different classes of processors to do jobs with fixed times. *European Journal of Operational Research*, 70:316–326, 1993.

[16] V.R. Dondeti and Hamilton Emmons. Fixed job scheduling with two types of processors. *Operations Research*, 40:S76–S85, 1992.

[17] Ulrich Faigle and Willem M. Nawijn. Note on scheduling intervals on-line. *Discrete Applied Mathematics*, 58:13–17, 1995.

[18] A. Fiat and A. Karlin. Randomized and multipointer paging with locality of reference. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, pages 626–634, 1995.

[19] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.

[20] Matteo Fischetti, Silvano Martello, and Paolo Toth. The fixed job schedule problem with spread-time constraints. *Operations Research*, 35:849–858, 1987.

[21] Matteo Fischetti, Silvano Martello, and Paolo Toth. The fixed job schedule problem with working-time constraints. *Operations Research*, 37:395–403, 1989.

[22] Matteo Fischetti, Silvano Martello, and Paolo Toth. Approximation algorithms for fixed job schedule problems. *Operation Research*, 40:S96–S108, 1992.

[23] Marshall L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management Science*, pages 1,18, 1981.

[24] Virginie Gabrel. Scheduling jobs within time windows on identical parallel machines: New model and algorithms. *European Journal of Operational Research*, 83:320–329, 1995.

[25] M.R. Garey and D.S. Johnson. *Computers and Instractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[26] Ilya Gertsbakh and Helman I. Stern. Minimal resources for fixed and variable job schedules. *Operations Research*, 26:68–85, 1978.

[27] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[28] 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.

[29] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Francisco, 1996.

[30] S. Irani, A. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. In *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 228–236, 1992.

[31] D.S. Johnson. A catalog of complexity classes. In *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, pages 67–161. Elsevier Science Publishing Company, Amsterdam, 1990.

[32] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, volume 18, pages 364–373, May 1990.

[33] Nakajima K., Hakimi S.L., and Lenstra J.K. Complexity results for scheduling tasks in fixed intervals on two types of machines. *SIAM Journal of Computing*, 11:512–520, August 1982.

[34] Antoon W.J. Kolen and Jeo G Kroon. On the computation complexity of (maximum) class scheduling. *European Journal of Operational Research*, 54:23–38, 1991.

[35] Antoon W.J. Kolen and Jeo G Kroon. On the computation complexity of (maximum) shift class scheduling. *European Journal of Operational Research*, 64:138–151, 1993.

[36] E. Koutsoupias and C. Papadimitriou. On the k-server conjecture. In *Proceedings of the 25th Symposium on Theory of Computing*, pages 507–511, 1994.

[37] Jeo G Kroon, Marc Salomon, and Luk N. Van Wassenhove. Exact and approximation algorithms for the operational fixed interval scheduling problem. *European Journal of Operational Research*, 82:190–205, 1995.

[38] Leo J. Kroon, Marc Salomon, and Luk N. Van Wassenhove. Exact and approximation algorithms for the tactical fixed interval scheduling problem. *Operations Research*, 45:624–638, 1997.

[39] E.L. Lawler and J. Labetoulle. On preemptive scheduling of unrelated parallel processors. *Journal of the Association for Computing Machinery*, 25:612–619, October 1978.

[40] Jan Karel Lenstra, David B. Shmoys, and Eva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[41] Richard J. Lipton and Andrew Tomkins. Online interval scheduling. *Proceedings of the Fifth Annual Symposium on Discrete Algorithms*, 54:302–311, 1994.

[42] M. Manasse, L.A. McGeoch, and D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.

[43] Lyle A. McGeoch and Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.

[44] J.W. Moon. *Counting Labelled Trees.* Canadian Mathematical Monographs, 1970.

[45] W.M. Nawijn. Minimum loss scheduling problems. *European Journal of Operational Research*, 56:364–369, 1992.

[46] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, New York, 1982.

[47] C.H. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[48] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, New Jersey, 1982.

[49] A. Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 169–178, 1993.

[50] A. Seznec. Skewed associativity enhances performance predictablility. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 256–275, 1995.

[51] A. Seznec and F. Bodin. Skewed-associative caches. In *Proceedings of PARLE' 93*, 1993.

[52] 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.

[53] Frits C.R. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2:215–227, 1999.

[54] Eric Torng. A unified analysis of paging and caching. *Algorithmica*, 20:175–200, 1998.

[55] Gerhard J. Woeginger. On-line scheduling of jobs with fixed start and end times. *Theoretical Computer Science*, 130:5–16, 1994.