Reducing Cache Pollution in Time-Shared Systems

by

Daisy T. Paul

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2001
[February 2001]

Author . . . .
Department of Electrical Engineering and Computer Science
January 19, 2001

Certified by . . . .
Larry Rudolph
Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Reducing Cache Pollution in Time-Shared Systems

by

Daisy T. Paul

Submitted to the Department of Electrical Engineering and Computer Science
on January 19, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis proposes the Lame Duck cache replacement policy, a LRU-based scheme that accounts for effects of context switching on cache behavior. Context switching results in multiple working sets residing in cache simultaneously, given sufficient cache sizes. Traditional LRU may evict cache lines belonging to the working set of a previously scheduled process in preference to stale cache lines belonging to the currently running process. Such behavior results in the unnecessary eviction of active cache lines in each process' working set. In order to reduce this pollution, Lame Duck constrains the size of the cache footprint of the currently executing process part-way through its time quantum. For the remainder of the time quantum, the process may bring new data into the cache; however, a cache line belonging to the process is evicted to make room for the new data.

Experiments show that Lame Duck and its variants can reduce cache pollution, thereby mitigating cold start misses incurred after a process regains control of the processor after a context switch. Lame Duck performance is heavily influenced by workload characteristics, time quantum length, and cache size/associativity. This thesis concludes with a general guideline describing scenarios for which Lame Duck does and does not perform well.

Thesis Supervisor: Larry Rudolph
Title: Research Scientist

Acknowledgments

The number of people I need to thank for enriching my experiences at MIT are countless. Each person I have met here has challenged me both emotionally and academically, helping me become the person I am today. I continually realize how lucky I am.

I would like to thank my advisor, Larry Rudolph, for his guidance and support in completing this thesis. He had faith in my abilities and gave me the opportunity to follow it through with my research.

Derek Chiou and David Chen listened to my ideas, discussed research with me, and continually told me to stop worrying. We met as officemates; I leave MIT with two more good friends.

Manish Jewtha, Rebecca Xiong, Sandeep Chatterjee, James Hoe, Gookwon Suh, Todd Mills, Ali Tariq, George Hadjiyiannis, Daniel Engels and Vinson Lee are new friends. They made the many days and late nights at LCS bearable, and even fun. James coined the term "Lame Duck" and Vinson proofread this thesis many, many times. I thank Manish for making me laugh, spastically.

I thank my MIT friends from undergrad for their support during this past year, especially Andres Tellez, Marissa Long, and Rochelle Pereira. For them I will always be greatful.

Ten years after we met, Lydia Kang and Trupti Rao still make time to listen to me and send care packages. Some friendships do last a lifetime.

Finally I thank my family. They have always believed in me and been supportive of my decisions. Without you, none of this would have been possible.

Love, Daisy :)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis proposes and evaluates the Lame Duck cache replacement policy, an algorithm aimed at making effective use of caches in multiprogrammed systems. Most caches today make use of the Least Recently Used (LRU) cache replacement policy. Although LRU often performs close to optimal, there is still room for improvement. Lame Duck, a variant of LRU, avoids some of the mistakes made by LRU while also taking advantage of the algorithm's strong points.

Lame Duck leverages knowledge of how context switches affect cache performance. Its goal is to curtail cache pollution by preventing processes from unnecessarily evicting cache lines belonging to the working set of another process. Ideally, assuming there is sufficient space, we would ensure that the working sets of all running processes reside concurrently in cache. We make strides towards attaining this goal by estimating how much cache space the process needs (as opposed to can use) and limiting the size of each process' memory footprint accordingly. The memory access pattern of the particular application and its allocated time quantum are primary influences on these estimations.

After motivating the Lame Duck policy and describing an implementation, this thesis discusses its use and evaluates its performance. Our investigation revealed that some workloads perform well under the Lame Duck policy, while others, with specific cache configurations perform better under LRU. Therefore, a variation of the Lame Duck policy was developed. The Lame Duck Variation is presented, discussed, and evaluated, especially with respect to the original algorithm.

The rest of this chapter begins with a discussion of current cache architecture and its defects, as well as a list of assumptions and terminology used throughout the thesis. We set the stage for our replacement policy by discussing suboptimal cache allocations granted by LRU, and describing the idea of time-adaptive caching. We conclude the chapter by providing intuition behind the Lame Duck cache replacement policy and a summary of the rest of the thesis.

1.1 Caches

Caches are typically small, fast, local memories designed to reduce memory latency and bandwidth requirements by maintaining copies of data that will likely be accessed by the processor in the near future. The cache examines every memory request to determine if it can satisfy that request from data it caches. The greater the number of memory requests satisfied out of the cache, the greater the reduction in average memory latency and memory bandwidth demands. Thus it is important to constantly keep the cache up to date with data that will be accessed in the near future.

The cache can be easily abstracted as a matrix in which the columns are termed "ways" and the rows are termed "sets". When a piece of data is brought into the cache, specific bits of its address are used to associate a cache line with a set. (A cache line is the standard unit of data transfer between main memory and the cache.) The cache line can be placed in any free column in the set. If the set is full, some other cache line from the set must be discarded to make room for the new cache line. The cache replacement policy decides what data to replace. If the replacement algorithm discards the cache line least recently used, the LRU (least recently used) replacement policy is being used. LRU assumes that the recent past predicts the near future. Thus, the cache line used furthest in the past is the chosen candidate for replacement.

A cache's replacement policy directly affects the degree to which the cache can decrease the amount of communication between the processor and memory. A perfect cache is an oracle; it decides what to cache based on future reference patterns. Since the cache does not have a priori knowledge of what information the processor will need, the replacement policy does its best to predict which data in the cache will be useful. Elements in the cache are weighted according to the replacement policy's estimation of how likely it will be used next. (The random replacement policy essentially assigns equal weights to all cache lines.) If the replacement policy is a bad predictor, the processor will not find the data it is looking for in cache. This results in a cache miss, and requires the processor to request data from another source (a lower level of cache, or main memory). The cache then incorporates the data fetched in response to the cache miss. (A "lower cache" is a cache further from the processor core. A "higher cache" is closer to the processor core. The distance of the cache from the processor core is referred to as the "cache level." A level one ($L1) cache is, generally speaking, the first cache the processor sends a memory request to.) If the processor finds the data it is looking for in cache, the cache access is termed a cache hit.

Modern day caches are designed to account for spatial and temporal locality. Spatial locality refers to the idea that memory accesses are likely to be clustered around a set of addresses. Temporal locality refers to the idea that memory access are likely to request the same piece of data multiple times within a short period of time. Many applications have fairly regular memory reference patterns that exhibit "local" behavior. As a result cache designers have selected the LRU replacement policy as the standard for cache replacement policies.

1.2 Assumptions and Terminology

We use the term "working set" to refer to the set of addresses which a process will use in the near future. To reduce thrashing, caches should be larger than the working set of the currently running process.

We use the term "memory footprint" to refer to the amount of cache space used by a given process. By this definition, the memory footprint of a process cannot be larger than the size of cache.

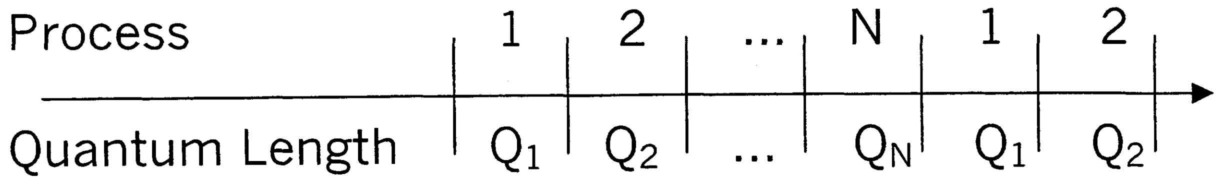


Figure 1-1: Round robin scheduling policy.

useful data between context switches. Caching of virtual addresses would necessitate a cache flush on every context switch in order to avoid address aliasing between process address spaces. Because our cache is modified to include process identification tags, virtual addresses can be cached. However, the cache lookup mechanism must be changed to include a process id check to determine whether or not a piece of data resides in cache.

Although our results are only representative of data caches, our cache simulator allows us to easily extend the same set of experiments to instruction caches. We leave this as future work.

We assume that processes exhibit self-similar memory access patterns. This assumption is discussed and substantiated in Section 4.1 of Chapter 4.

1.3 Design Criteria for Early Caches

Early caches were fast and transparent to software; caches were designed to improve performance without software involvement. Comparable processor and memory speeds required the cache to respond quickly to have a real benefit. In addition, compilers and software were not sufficiently sophisticated to explicitly manage fast storages that were distinct from memory. Because of the need for speed and transparency, cache replacement protocols were implemented in simple hardware and were uniform across the entire address space.

Constraints on chip space required cache sizes to be small. Small caches restricted the cache to contain the working set of a single application at a given time. In a time-shared system, this meant that every time quantum began as a cold start regardless of what cache replacement strategy was used. Exploiting locality within a program was the primary consideration in choosing a cache replacement algorithm.

We use the term "time quantum" to refer to a small unit of time for which a process is allowed to execute before the next process gets control of the CPU on a time-sharing system.

We discuss time quantum length in terms of the number of memory references that occur during that time-slice.

We use the term "cache miss penalty" to refer to the number of cycles needed to obtain a piece of data from main memory in response to a cache miss.

We use the term "self-similar" to refer to a process exhibiting memory access patterns over small amounts of time (t) which reference k distinct cache lines belonging to set j during the first half of the time slice, and k distinct cache lines belonging to set j during the second half of the time slice, for every set j. K is some number less than $(t/2)$. If the set access pattern of a process is symmetric about the halfway point through the time slice, then the process is self-similar.

We recognize that context switches can be categorized into two types, voluntary and involuntary. Voluntary context switches result from a page fault or a system call. Involuntary context switches occur when the scheduler switches the current process out so a new process can execute. In our discussions, we refer almost exclusively to involuntary context switches. However, our algorithm can also be used to mitigate the effects of voluntary context switches on cache performance.

We assume no particular operating system; our results should be applicable to any system that schedules processes in the same way Unix systems do and that does not make any attempt to take cache behavior into account when deciding which process to run next. A scheduler that tried to schedule processes to avoid context-switch-related cache misses might not yield the same results.

Unless specifically stated otherwise, we assume the concurrent execution (through timesharing) of multiple processes using a round-robin policy. Each process has a fixed time quantum, as show in Figure 1-1. We also assume a cache capacity too small to hold the working sets of all running processes, but large enough to hold at least the working set of one process and part of the working set of a second process.

We assume the caching of physical addresses, enabling the cache to maintain

Structured applications exhibit "local" behavior, thereby lending itself to perform well in caches. For these applications, past memory references are a very good indicator of future memory references and LRU approximations are near optimal. Locality of reference within a particular process coupled with the fact that caches and miss penalties were small made a uniform policy such as the LRU replacement policy good enough.

1.4 Technological Setting of Today's Caches

As a result of changing technologies and novel applications, assumptions made on system characteristics are no longer valid. Miss penalties are no longer small. Current processors can take on the order of 100 cycles to access main memory [8, 14] while an on-chip L1 cache can be accessed in 1 cycle. The gap between processor cycle times and memory access times is expected to continue widening. Current trends indicate that processor speeds double every 18 months [2]. Meanwhile, memory latency remains virtually unchanged by comparison, decreasing by only 7% per year [12, p. 429]. Processor clock rates are now six or seven times that of the bus clock rate. AMD Athlon processors operate at speeds up to 1.5GHz; the upper limit for front-side bus speeds are currently 266MHz [2]. Memory is the bottleneck in today's systems, making cache replacement policies more important than ever.

Compilers and users have become significantly more sophisticated than when caches were first designed. Though users and compilers cannot always give complete information about the memory usage of a program, a significant amount of information can now be made available at compile-time as well as run-time. This makes interaction between software and cache practical.

Caches and applications themselves have changed such that LRU is not always good enough. Cache sizes are larger, yet application working set size has not grown at the same rate. Modern caches no longer contain only a fraction of one working set; they are large enough to contain working sets of multiple applications simultaneously. The standard LRU replacement policy does not take advantage of this fact. Instead, it blindly assumes locality of reference, even if two temporally close memory accesses belong to different processes. Under LRU, the cache throws away data that has not been accessed recently (such as live data of a swapped out application) over data that has been accessed recently (such as the current application's recently accessed but dead data).

Applications can no longer be generalized as being coded in traditional paradigms. Certain computing paradigms, such as object-oriented codes, access memory less regularly than applications developed in years past. In addition, applications do not use memory for the same uses as they once did due to a number of factors including increased communication between different devices and new streaming applications such as decompression, video and graphics, and so on. With these less regular memory references, standard LRU algorithms perform suboptimally.

One approach to dealing with this is to increase cache sizes. By storing more data in the cache, a safety net is created, giving the replacement policy more leeway to make a mistake in determining which cache lines are needed most. While some programs could make use of a large amount of cache space, many applications could perform better if they simply made better use of the cache, which is often underutilized [6].

In light of the suboptimal behavior of LRU, and the new technological advances that allow for knowledgeable cache replacement policies, we explore software management of caches. Specifically, we focus on the interaction between cache replacement policy and the existence of multiple working sets residing in cache concurrently. Having multiple working sets in cache simultaneously complicates how data in the cache should be treated. LRU treats all cache lines the same, and is therefore unable to account for relative timing differences between addresses from different processes; cache lines accessed by the current process at the end of a time quantum are treated the same as cache lines accessed by the next scheduled process a cycle of time quanta ago, (assuming round robin scheduling). In the next section we will discuss the effects of context switching on cache performance.

1.5 Effect of Context Switching on Cache Performance

The job of a cache replacement policy becomes ever more difficult in a multi-programmed system. Multiple processes are logically run concurrently, and multiple partial working sets are stored in the cache simultaneously, assuming a large enough cache. Once a process' time quantum has elapsed, a context switch takes place. State of the old process is saved away, and state of the new process must be loaded, including a new address space. The new process has dedicated processor time for the length of its time quantum, during which time it brings cache lines into the cache. If the cache is not large enough to hold the working sets of all running processes, processes compete sequentially for cache space.

For the sake of simplicity, we discuss a two-process workload, each process running with time quantum length Q under the LRU replacement strategy, and scheduled to execute in a round-robin fashion. Let A and B denotes the two processes. We assume the capacity of the cache is large enough to contain the entire working set of B, in addition to some cache lines belonging to A. During process B's active time quantum, (process A's dormant phase), process B encroaches on process A's cache allocation. When process B incurs a cache miss, LRU throws out the oldest element available. Since process B does not evict cache lines used during it's time quantum (unless the time quantum is long enough to have written the entire cache), it usually discards cache lines belonging to process A. Conversely, each miss during the execution of process A evicts a cache line belonging to process B whenever possible. We see that under LRU, the memory footprint of a process grows on each cache miss. As one footprint grows, the footprint of another process diminishes in cache. As a result, process B pollutes the memory footprint of process A, and vice verse.

Each time process A decrements the number of cache lines allocated to process B, the hit rate of process B may be reduced. Process B needs to reload the cache lines evicted by process A when it resumes control. This number of reloads is dependent on cache size, workload, and time quantum. For very small caches (that barely holds the working set of a single process), this number is very high. Cache capacity issues would force cache lines to be reloaded frequently regardless of the replacement algorithm. After a context switch, each process brings as much of its working set into cache as possible.

The probability of reloading a cache line remains high for moderate sized caches if the workload consists of a large number of processes, all of which make use of the cache. Here we use "moderate-sized" cache to describe a cache that can hold the working sets of a few processes, but not of a large number of processes. Consider an example of a workload of twenty-six processes interleaved via a round-robin scheduling policy. Let letters A through Z denote processes one through twenty-six respectively. Although the working set of process A may reside in cache after process B finishes executing for its time quantum, the probability that a cache line belonging to process A resides in cache greatly decreases with the execution of every process. Once process Z is scheduled to execute, most (if not all) of process A's cache lines have been evicted from cache.

We describe a new scenario in which processes A and B have small working sets, but access large amounts of data (have large footprints). Even if the cache can hold the working sets of both processes, long time quantums increase the probability that process A evicts cache lines belonging to the working set of process B. The more dedicated processor time a process is given, the more chances it has to evict cache lines belonging to other processes.

We have seen how context switches can degrade the individual and aggregate performance of processes. As such, multiprogrammed systems call upon us to redefine our yardsticks for cache performance. We must now consider whether we are trying to optimize the performance of a particular "important" process, or whether we want to optimize the cache to enhance overall performance. Optimizing overall performance may not mean trying to obtain the highest possible hit rate for the currently running application. It may make sense for the currently running process to give up a few of its own cache lines if the space could be much better used for another process. LRU does not know how to handle these new complications, and as a result does not always choose the optimal cache line to replace.

If we are trying to optimize cache performance over the entire workload, LRU is clearly not the optimal replacement strategy for streaming applications. Streaming applications contain large chunks of data accessed (sequentially or in stride) only once. Keeping such data in cache leads to the eviction of valuable data that will be accessed in the near future. Even though the streaming application may suffer a few cache misses as a result of limiting the number of cache lines allocated to streaming data, the cost would be amortized over the number of cache hits gained by allocating those lines to another process who could store useful data there.

1.6 Time-Adaptive Caching

Another approach for optimizing overall performance entails minimizing the amount of cache allocated to each process during a time quantum such that each process gets only as many cache lines as it needs to obtain close to optimal hit rates during that time quantum. Unlike LRU, this caching scheme takes into account the affect of context switching on cache performance. It recognizes that multiple processes can be logically run concurrently and that multiple working sets can be stored in the cache simultaneously. The goal is to decrease the number of cold start misses processes suffer after a context switch. Under the LRU replacement policy, the number of these misses can be very large; the entire footprint of a process can be evicted from cache before the process regains control of the processor.

Time-adaptive caching does not assume locality among memory references between processes. Instead it observes the age of the current process (time since it was last swapped in), the time quantum allocated to the process, the process owner of cache lines, and the age of stored data to decide which cache line to replace. Part-way through the time quantum of the current process, it may no longer be beneficial to augment the footprint of the current process. New data is still brought into the cache; however, a cache line belonging to the current process is evicted to make room for the data. By preventing addresses of the next process form being overwritten, some cache misses incurred as a result of the context switch are circumvented. We refer to the maintainance of a fixed cache allocation, in conjunction with updating data in the cache allocation as "recycling."

The caching scheme builds off of the strong points of the LRU replacement policy by managing the cache under LRU for a large part of the time quantum. Software is used to dynamically decide at what point during the time quantum LRU begins to perform suboptimally and a context-aware replacement strategy should be put in place. For certain workloads and cache configurations, a time-adaptive cache replacement policy may not be useful. Again we refer to the two-process workload of processes A and B. If a cache is small compared to the working set of process B, the gain in allocating an additional cache line to process B outweighs the cost of the miss that will be incurred when process A context switches back into the processor. We want to fill the cache with as much of the current working set as possible. Since LRU continually adds to the cache allocation of process B, LRU replacement is the algorithm of choice. If the size of cache is large enough to hold the entire working set of process B, in addition to some cache lines belonging to process A, the idea of recycling becomes feasible and offers potential performance increases.

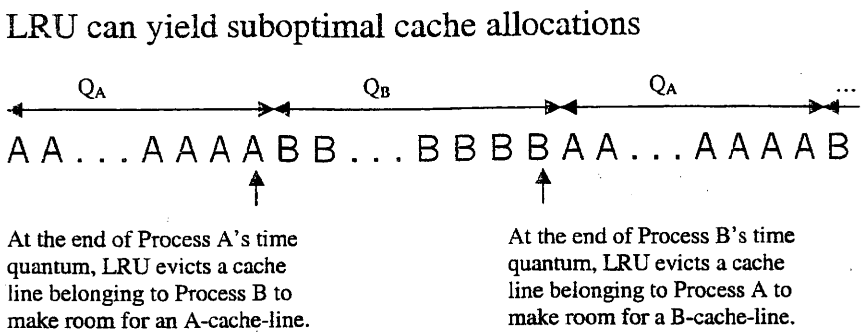
We argue that for large enough cache sizes, part-way through the time quantum of process B, the marginal reduction in misses obtained by increasing the cache allocation of process B is equal to the marginal increase in misses obtained by decreasing the cache allocation of process A. The intuition relies heavily on the effect of diminishing returns. There is a point at which allocating more cache to a process only marginally increases hit rate. The desire is to determine what amount of cache is allocated to a process when the miss rate of the process asymptotically begins to approach a constant (be it zero, or a plateau). At this point, we want to restrict the size of the footprint of the currently running process in order to keep items belonging to the other process in cache. The replacement policy should no longer be standard LRU; it should replace the least recently used line of process B from this point onwards, thereby disallowing process B to increase its cache allocation and retaining cache lines of process A in cache.

The above argument is the basis for the Lame Duck Policy. We are not the first to make such an observation. Stone, Turek, Wolf [19] argue that such a tradeoff point exists; however, they approximate the threshold time in a different manner and seem to have never fully explored all the issues.

We do not argue that the dormant process can always make better use of cache lines than the currently running processes. Limiting the cache allocation of the running process to only as many cache lines as it needs makes more efficient use of cache lines and leave cache lines belonging to other process in cache. The hope is that those saved lines contain data that will be useful to the process the next time it is switched in. This argument can be trivially extended to an n-process workload. Forcing the currently running process to use less space mitigates the cold start of the dormant processes resulting from cache pollution.

We return to the two process workload of A and B introduced in the previous section. Processes A and B are running with time quantum length Q under the LRU replacement strategy, and scheduled to execute in a round-robin fashion. Two desires seem intuitive: First, immediately before process A gets switched in, cache elements belonging to process A should not be ejected from cache. Secondly, immediately before process A gets switched out, cache elements belonging to process A should not replace cache elements belonging to process B.

Consider the behavior of LRU in this scenario. At the end of process A's time quantum, the furthest used cache line most probably belongs to process B. Even though process B is about to obtain control of the processor, data values it needs are evicted from the cache. A similar situation occurs at the end of process B's time quantum; cache lines owned by process A have not been accessed since the last time quantum, making it likely that one of A's cache lines is evicted from the cache. This goes against our intuition. If the two processes are perfectly interleaved, LRU does the right thing. However, because every Q memory accesses made by each process are grouped together, LRU makes the wrong decision. Assume that we do not evict one of B's cache lines on the last memory reference of A's time quantum. Process A's hit rate does not decrease as a result of using one less cache line during it's time quantum.


LRU can yield suboptimal cache allocations

At the end of Process A's time quantum, LRU evicts a cache line belonging to Process B to make room for an A-cache-line.

At the end of Process B's time quantum, LRU evicts a cache line belonging to Process A to make room for a B-cache-line.

Figure 1-2: Traditional LRU considers only the global age of a cache line when determining which line to evict. This can result in counter-intuitive cache allocations. At the end of a time quantum, LRU does not prepare the cache for the onset of a new process' execution. Instead it continues to augment the footprint of the ending process, thereby decreasing the footprint sizes of other processes. In the two-process case, it is clear that increasing the cache allocation to process A at the end of its time quantum diminishes the footprint of the next scheduled process. If LRU prevented Process A from obtaining another cache line on the last cache miss in the time quantum, Process B might have one less cache miss when it regains control of the processor.

It does provide the process B with one more resident cache line once it regains control of the processor. It is this behavior that Lame Duck strives to exploit. Figure 1-2 examines a simple case in which LRU can yield suboptimal cache allocations.

The question remains, how do we approximate the time at which to fix the cache allocation of the currently running process? When has the process acquired the minimum number of cache lines it needs to obtain the same hit rate during the rest of the time quantum as if we let the footprint of the process grow for the entire time quantum? One method proposed by Stone, Turek, Wolf approximates the incremental value of a cache line to each process given the current cache allocation and number of remaining references in the time quantum (This method is discussed in detail in Chapter 2.) The method we propose relies on self-similar memory access patterns exhibited by individual processes and the number of references remaining in the time quantum. In the next section, we introduce a particular time-adaptive caching scheme, our Lame Duck cache replacement policy that depends on this method.

1.6.1 Lame Duck Cache Replacement Policy

We explore the idea of time-adaptive caching by choosing a metric to estimate a threshold time past which the cache allocation of the executing process should not be augmented. The Lame Duck cache replacement policy restricts footprint growth once the time quantum midpoint is reached. Lame Duck is the topic of the rest of this thesis.

Restricting growth at halfway through the time quantum does not mean that every process gets the same memory allocation. By allowing the footprint of the process to grow under standard LRU for the first half of the time quantum, each footprint grows at the rate of its associated process. Assume process A has a large working set, and process B has a small working set. Given the same time quantum, and restricting growth at one half the time quantum, process A will own a larger percentage of the cache than process B. Under both the Lame Duck replacement policy and LRU, cache allocations are reflective of cache usage of individual processes. In addition, Lame Duck works to make cache allocations reflective of the cache *needs* of individual processes.

Our ability to estimate when to restrict the footprint growth of a process relies heavily on the assumption that the memory access patterns of a process are self-similar. Memory references made during the first half of the time quantum access sets in the same way during the second half of the time quantum. This assumption has two direct implications. First, the number of distinct virtual addresses accessed during the first half of the time quantum is roughly equal to the number of distinct virtual addresses accessed during the second half of the time quantum. Second, set access patterns made during both halves of the quantum are similar. From this we conclude that the number of cache lines needed during the second half of the time quantum is equal to the number of lines needed during the first half of the time quantum. If this is true, the amount of cache space allocated to the process at halfway though the time quantum should be enough to hold all virtual addresses accessed during the second half of the time quantum.

Although there is no formal proof that memory access patterns during a time quantum tend to look this way, in practice we have seen this to be true for a number of different workloads and time quantums. A complete discussion is presented in the Validation section of Chapter 4.

Not all applications exhibit self-similar memory access patterns. As a result, cache allocations obtained by a process at the time quantum half may not be sufficient for Lame Duck to maintain cache performance while constraining cache footprints. We address the problem by proposing a Variation to the Lame Duck policy (LDV), in which we allow the cache allocation to grow past the halfway point of the time quantum if it owns less than half of the cache lines in the relevant set. Allowing the process to continue growing after the time quantum midpoint avoids worst case scenarios in which sets heavily accessed during the second half of the time quantum were not accessed during the first half of the time quantum.

We explore the potential benefits (and pitfalls) of Lame Duck using a hypothetical cache replacement policy, Intelligent Lame Duck. Intelligent Lame Duck constrains cache footprint size in the same manner as original Lame Duck. In addition, it leverages information about the needs of the currently executing process to make intelligent decisions about what data to evict from cache. By combining the use of local knowledge with dynamic partitioning, Intelligent Lame Duck can determine if a choice of workload, cache configuration and time quantum is ever able to benefit from constraining footprint size at the time quantum midpoint. Intelligent Lame Duck approximates best-case performance of original Lame Duck.

1.7 Thesis Overview

Having presented the motivation for and the intuition behind the Lame Duck cache replacement strategy, we use the next chapter to present the work that lay the foundation for this thesis. Specifically, we discuss past work regarding the effect of context switching on cache performance and the partitioning of memory between competing processes. We discuss an algorithm put forth by Stone, Turek and Wolf that is similar to the one presented in this thesis, pointing out the differences, while also explaining how this thesis further explores the problem space they delineated.

In Chapter 3, we develop the Lame Duck algorithm, two variations of the algorithm (LDV, Intelligent Lame Duck), and talk about implementation issues. Although very little additional hardware is needed to implement the Lame Duck replacement policy, we opted to measure performance using a software simulation environment. The chapter concludes with a description of the simulation tools we used.

In Chapter 4, we evaluate the original Lame Duck replacement policy, LDV and Intelligent Lame Duck. We provide performance numbers in terms of hit rates and cost of memory operations.

In Chapter 5, we conclude the thesis with an assessment of the algorithms presented, as well as some parting thoughts and observations. We also give future work and directions that should be explored.

Chapter 2

Related Work

Cache performance under multiprogrammed workloads has been studied extensively over the last two decades. Thiebaut and Stone [20] created a model for cache-reload transients, the set of misses incurred as a result of having to reload program data after an interruption in execution. Based on estimates of instruction execution time, they interleaved traces of single processes to simulate a multiprogrammed environment. Agarwal et al. [4, 5] generated accurate multiprogramming traces and compared the average behavior of uniprocess caches and multiprocess caches with either cache flushing on context switch or the use of process identifiers on cache lines. Agarwal proposed a few small hardware additions to reduce the number of caches misses resulting from multitasking. In addition, he built upon the work of Thiebaut and Stone by creating a more detailed analytical model of the cache. Mogul and Borg [17] estimated the cache performance reduction caused by a context switch, differentiating between involuntary and voluntary context switches. They suggested that for time-sharing applications, the cache-performance cost of a context-switch dominates the cost of performing a context switch.

Context switching can cause a decrease in cache performance by violating locality of reference. When an operating system switches contexts, the instructions and data of the newly-scheduled process may no longer be in the cache(s). One possible solution to this problem is cache partitioning. Cache partitioning consists of various techniques used to increase the probability that a process will have cache lines in the cache when it regains control of the processor. The technique may try to maximize performance for a process specifically designated as important, or may try to maximize the aggregate performance of all processes.

One approach to cache partitioning is to dedicate physical cache space to a process/address range. Kirk [15] analyzed the partitioning of an instruction cache into a static partition and a LRU partition. Sun Microsystems Corporation holds a patent [18] on a technique that partitions cache between processes at a cache column granularity. Chiou [7] proposes to include as a part of a process state, a specified bit mask hat indicates which columns can be replaced by that process. While this technique isolates processes from each other within the cache, Chiou's mechanism, named column caching, allows for dynamic cache partitioning between different address ranges. Regions of memory that compete for the same cache lines are mapped to different regions of the cache (e.g. different cache columns), and non-conflicting memory regions are mapped to the same regions of cache.

Liedtke et al. [16] developed an OS-controlled application-transparent cache-partitioning technique that relies on main memory management. Main memory is divided into a set of classes, called colors, whose members are physical pages that are associated with a cache bank. In order to avoid conflicts by multiple tasks, virtual addresses are translated such that one or more colors are assigned exclusively to one cache bank. This scheme provides tasks/address regions with exclusive use of their partition. Although there is the potential for significant waste of main memory, they attempt to avoid this by using memory coloring techniques.

The work we have discussed up to this point dedicates physical cache space to a process/memory region. An alternative approach is to limit the amount of cache space a process is allocated. Stone, Turek, and Wolf [19] pursued this idea and introduced an algorithm that dynamically determines a threshold time past which the currently running process is not allocated additional cache space. This threshold time is dependent both upon the remaining time quantum and the marginal reduction in miss rate due to an increase in cache allocation.

Because of the large similarities between the algorithm presented in this thesis and the one put forth by Stone et al., we will now spend some time developing and discussing what shall hereafter be referred to as the STW cache replacement algorithm. One observation worth noting is that previous work on cache partitioning does not explicitly take into account the effects of context-switching on cache performance. Both the STW and Lame Duck replacement strategies do make such considerations.

The goal of Stone et al. is to distribute cache lines to each process so as to make equivalent the incremental value gained by giving each of the processes a single cache line in addition to its allocation. Their model defines the optimal fixed allocation of cache as the allocation for which the miss-rate derivative with respect to the cache size is equal for all processes. Formally stated, when processes A and B share a cache of size C, partitioning the cache by allocating C_A lines to process A, and C_B lines to process B ($C_A + C_B = C$) maximizes the hit ratio when the miss-rate derivative of process A, as a function of cache size C_A equals the miss-rate derivative of process B in a cache of size C_B.

In later work, Thiebaut, Stone, Wolf [21] define miss-rate derivative as the incremental performance gain to a process of an additional cache line in addition to its current allocation. For a two process workload, A and B, the frequency-weighted miss-rate derivative of process A can be approximated by the equation: $\frac{M_A(C_i)-M_A(C_{i+1})}{Ref_{A+B}} = \frac{(\# \ Amisses \ in \ C_i \ - \# \ Amisses \ in \ C_{i+1})}{Total \ \# \ of \ A \ and \ B \ refs}$ The numerator of this fraction is the number of misses in a cache of size C_i that becomes hits in a cache of size C_{i+1}. This method of partitioning has the convenience of automatically weighing the derivatives by frequency of reference.

The product of the miss-rate derivative of a process with respect to the curent cache allocation, $\frac{dM_p(x)}{dx}$, times the number of reference remaining in the time quantum, q, is the number of misses they expect to save by increasing the cache allocation to the currently running process. They account for p additional misses resulting from processes having to reload the cache line displaced by the currently running process. Thus the net reduction on misses is equal to $\frac{qdM_{cur\_process}(x)}{dx} - p$. When allocating a new cache line to the currently running process yields equivalent performance gain and performance degradation (between the current process and one of the other processes) it is time to stop allocating cache lines to the currently running process. The threshold time q(x) can be obtained by setting the previous equation to zero, and solving for q. If the remaining time quantum is less than q(x) for a cache allocation of x, then the replacement policy should not increase the cache allocation to the currently running process. The replacement policy should replace the least recently used eligible line of the currently running process, thereby retaining the lines of other processes in cache. If the remaining time exceeds q(x), then the replacement policy should replace the least recently used line in the cache (which is unlikely to belong to the currently running process), and thereby increase the cache allocation of the currently running process.

It is clear that there are many similarities between the STW and Lame Duck replacement policies. Both policies build upon LRU, and select a point during the time quantum at which to constrain the cache allocation of the currently running process. However, different metrics are used to determine when to begin recycling over cache lines. STW starts recycling when the marginal reduction in misses obtained by increasing cache allocation equals the number of cold start misses created by reducing the cache allocation of other processes. Lame Duck relies on self-similar memory access patterns of individual processes and begins recycling once the process has acquired enough cache space to perform well during the rest of the time quantum. The Lame Duck Variation also relies on the percentage of cache lines in the set that are owned by the current process.

Although Stone et al. discuss the algorithm behind their multiprogramming cache replacement strategy, they have not evaluated the performance of the STW policy. The effect of time quantum lengths, cache configurations and workload have not been looked at with respect to STW. In addition, issues of implementation details and hardware cost have never been addressed. In a number of ways, the research presented in this thesis builds upon the work of Stone, Turek, and Wolf. We present a slightly different cache replacement algorithm, and perform a much more detailed evaluation of its performance. Using LRU as our standard for comparison, we explore the source of performance gain and performance degradation resulting from invocation of the

Lame Duck policy. Finally, we not only search for the answers to our own questions, but we look to answer questions proposed, but left unanswered, by Stone, et al.

Thiebaut, Stone, and Wolf present a extension of the model developed by Stone, Turek, and Wolf. They describe an adaptive online algorithm for managing disk caches shared by several identifiable processes. Again, they define the optimal fixed allocation of cache as the allocation for which the miss-rate derivative with respect to the cache size is equal for all processes. Initially, disk cache is partitioned among the processes in such a manner as to obtain equivalent miss-rate derivatives with respect to cache size among all processes. Cache partitions are updated using the "Robin Hood" philosophy: take from the rich and give to the poor. On a cache miss, the LRUth cache line of the process having the lowest frequency-weighted miss-rate derivative is evicted. This differs from the STW algorithm, in which miss-rate derivatives are used to calculate threshold times, after which LRU information determines which cache line of the currently running process should be evicted. Simulation results exhibit a relative improvement in the overall and read hit-ratios in the range of 1% to 2% over results generated by a LRU replacement algorithm. Thiebaut et al. maintain that their algorithm can be applied to any buffering system managed by an LRU algorithm that has sufficient processing capability to perform the required operation in real time. Because current microprocessor cache architectures can not support the necessary processing capability, this thesis will not address similarities between Lame Duck and the cache replacement policy presented by Thiebaut et al.

Chapter 3

Lame Duck Cache Replacement Policy

In this thesis we propose a new cache replacement policy called the Lame Duck Policy. This policy takes into account the affect of context switching on cache performance, while normal LRU makes no such considerations. The goal is to decrease the number of cold start misses processes suffer from after a context switch. Under the LRU replacement policy, the number of these misses can be very large; the entire footprint of an inactive process can be evicted from cache before the process regains control of the processor.

3.1 The Algorithm

The Lame Duck Policy works as follows. During the first half of the time quantum, the normal LRU replacement policy is used to determine which cache line to evict on a cache miss. Under this policy, the footprint of the currently running process grows in cache on every cache miss. Each time a process brings a line into the cache, a tag associated with that line is set to the process id of the process making the memory reference. Once half of the time quantum has elapsed, LRU is performed over the cache lines tagged with the process id of the currently running process. (If no lines in the set belong to the process, the process is given the LRUth line. But it will not be granted any additional lines in the same set until its next time quantum.) In essence, the process recycles lines it accessed during the first half, throwing out old data to bring in the new. The contention is that number of the cache lines obtained during the first half is sufficient to hold the memory values needed for the second half.

By recycling lines belonging to the currently running process, we increase the probability that cache lines belonging to other processes will still be in the cache the next time the process is switched in. Experimental results show that this hypothesis does not always hold true, as will be discussed in detail in the Validation section ofChapter 4. However, we will present the problem here in order to introduce a variation on the Lame Duck Policy.

Persistent data between context switches is likely to exist if all processes have short time quantums or the workload consists of a small number of processes. Short time quantums allow the number of partial footprints existing simultaneously in cache to increase, since each process only has a short amount of time during which to evict cache lines belonging to other processes. A small number of processes allows for increased size of partial footprints existing simultaneously. Fewer processes means less competition for space in cache. If a large number of processes run large time quantums in a round robin fashion, by the time any individual process gets to run again, its entire footprint is evicted from cache, thereby forcing the process to cold start.

The Lame Duck Policy aims to increase the number of distinct time quantum lengths and workloads for which persistent data exists between time quantums. By limiting the time the footprint of a process can grow in cache to the first half, we keep processes from eviciting lines belonging to other processes during the second half. This effectively halves time quantums in terms of the effect it has on footprints sizes in cache.

Although the process may access approximately the same number of cache lines during the second half of the time quantum as during the first, the distribution of those lines throughout the cache may not be the same. If a process does not access a set during the first half of its time quantum that it needs in the second half, then it would only be allocated one line in the set. If the process heavily uses multiple cache lines in that set during the second half, the Lame Duck Policy does the wrong thing. The footprint of the process should grow to own more of the set. In this case, LRU does the right thing. Under the Lame Duck policy, the number of lines a process owns in any given set once it starts recycling is

$$MAX(X,\ MIN(\#lines\ process\ needs,\ 1))$$

$$X = (\#lines\ owned\ at\ half\ way\ though\ the\ time\ quantum)$$

This scenario of where Lame Duck fails can be generalized to sets in which less lines are accessed in the first half of the time quantum than are needed in the second half of the time quantum. This disparity is made worse if many lines of a previously unaccessed set are heavily accessed during the second half. We address this problem with the Variation of the Lame Duck Policy.

3.1.1 Variation of the Lame Duck Policy

Algorithmically, the Lame Duck Variation (LDV) differs from the original Lame Duck Policy only by placing an additional constraint on when to begin recycling lines belonging to the currently running process. The normal LRU replacement policy is used until half of the time quantum has elapsed. Once half the time quantum has elapsed, LDVonly allows recycling to be done if the current process owns greater than or equal to half of the sets of $Ln. ($Ln is the cache level in which the Lame Duck Policy is being used.) If the current process owns less than half the sets of $Ln, then a cache line is evicted under the LRU replacement policy, thereby allowing the current process to own an additonal line in the set. Once the process owns half the lines in the set, then recycling may begin.

This policy averts the extreme case in which the current process only owns one line in a set that is heavily accessed during the second half of a time quantum. It allows for a much larger degree of disparity between cache line distribution of the first and second halves of a time quantum. The number of lines a process owns in any given set when it starts recycling is

$$MAX(X,\ MIN(\#lines\ process\ needs,\ 0.5 * associativity\ of\ \$Ln))$$

$$X = (\#lines\ owned\ at\ halfway\ though\ the\ time\ quantum)$$

Although this policy ensures that the currently running process owns at least half the lines of its ideal allocation, this may not be good enough. Every line within a set may be accessed repeatedly by a process for short periods of time, e.g. as seen in the SPEC95 `Swim` benchmark [3]. If there is significant pressure on particular sets, neither version of the Lame Duck Policy does the right thing. The policy should allow the process to own the entire cache line, regardless of how many lines in the set were owned at halfway through the time quantum. (If the process owns all the lines of the set at the halfway point, then it will own all the lines of the set after halfway point). A more extreme case, as seen in `Swim`, is that the sets accessed during the first half of the time quantum are not accessed at all during the second half of the the quantum (`Swim`: Q=1000, Q=10000). So even though `Swim` owns entire cache lines (most of certain cache lines) at Q/2, they are the wrong cache lines. As a result, the footprint of `Swim` grows during the second half and only uses the newly acquired space. The cache lines it acquired during the first half is of no use to it.

We address this problem using set hashing. This greatly reduces set contention in workloads that maintain pressure on a small group of sets. This makes owning entire sets unnecessary, thereby increasing the probability that the process will get the space it needs during the second half of the time quantum under LDV.

In general, this policy would not be used with fully associative caches. Since a fully associative cache consists of one set, LDV would require that the currently running process own half of the cache before recycling would begin.

3.1.2 The Intelligent Lame Duck Policy

Practically speaking, we can not assume that a replacement policy has global knowledge of access patterns of each process. However emerging compiler technology in conjunction with profiling can provide the cache with information about individual processes, enabling the cache to decide what data is useful to the currently running process. Local knowledge can be leveraged to make intelligent decisions about which cache lines should reside in cache. Taking the idea one step further we propose Intelligent Lame Duck, a hypothetical cache replacement policy that combines the use of local knowledge with dynamic partitioning.

For the first half of the time quantum, Intelligent Lame Duck uses an LRU replacement policy. It constrains footprint size at the time quantum midpoint, and uses information about future accesses to decide which cache line belonging to the current process to evict on a cache miss. We do not assume the cache has knowledge of how the memory access patterns of all scheduled processes interact, so the cache is only allowed to consider ideal information of the currently executing process.

Using information about future memory accesses allows Intelligent Lame Duck to approximate the best-case performance for Lame Duck. If LRU information is a good predictor for a particular single process, Lame Duck and Intelligent Lame Duck will virtually mirror each other's performance for that process; (Lame Duck can not perform any better than Intelligent Lame Duck). However, if the metric used by LRU is not good enough to keep shared variables in cache between quantum halves, Intelligent Lame Duck will outperform Lame Duck. As a result, studies of Intelligent Lame Duck answers the question, "Does it ever make sense, for this workload, cache configuration and time quantum, to constrain the footprint of a process at the time quantum midpoint?"

Sometimes the answer may be "no." There are instances in which LDV and/or LRU outperform Intelligent Lame Duck. This is true for workloads whose performance is severely limited by cache capacity.

Often, the answer is "yes." Basic examples include the cases in which the LDV just barely outperforms LRU and cases in which LRU just barely outperforms LDV. Intelligent Lame Duck embodies our outlook for the future; as applications and system characteristics change and become more complex so must caching algorithms. In Section 4.3 of the next chapter we compare Intelligent Lame Duck performance to the original Lame Duck, LDV and to Ideal, thereby allowing us to see how well Intelligent Lame Duck bridges the gap between LRU performance and optimal performance.

3.2 Hardware Support

In order to incorporate the Lame Duck replacement policy into caches, a process identification tag must be associated with each cache line. Initially all process tags are set to an invalid process ID (pid), e.g. -1. When a cache line is brought into the cache, the process tag is set to reflect the identity of the process that requested the cache line. When checking for a cache hit, the pid and tag comparison of an address can be done simultaneously. Combinational logic used to compute the least recently used element is modified to allow computation of the least recently used element among cache lines having the pid of the currently running process.

We recognize that a 16-bit process id associated with each cache line is not cheap. An ownership bit approximates cache line/process associations. A single bit is added to each cache line to indicate that the currently running process owns the cache line. Ownership bits are cleared before the beginning of each time quantum. When a cache line is first accessed, the ownership bit is set, indicating that it belongs to the currently running process. Instead of performing process ID comparisons, additional combinational logic need only compute the least recently used element among cache lines having its ownership bit set.

The problem with ownership bits is that all cache lines belonging to the currently running process residing in cache cannot be detected. Ownership bits only highlight cache lines belonging to the process accessed during the current time quantum. Cache lines belonging to the current process accessed during a previous time quantum are "lost children." Since recycling relies on ownership bits to know which cache lines it can evict, once recycling begins, these lost children cannot be evicted during this quantum. As a result, the least recently used cache lines belonging to the current process may not be evicted via recycling. Instead cache lines accessed in the first half of the time quantum will be thrown out before persistent cache lines from a previous time quantum. This can keep persistent cache lines around for longer than necessary.

Process identification tags avoid this problem. PIDs enable the processor to detect cache lines belonging to the currently running process even after multiple time quantums have elapsed. The cache can then recycle over the process' entire footprint in cache as opposed to the part of the footprint accessed during the first half of the time quantum.

3.3 Simulation Environment

While minimal modifications to current cache architectures are required to support Lame Duck, simulation tools were used to evaluate the performance of Lame Duck. A trace-driven approach was selected to allow for easy modification of cache characteristics, thereby providing the ability to quickly rerun experiments over a whole range of parameters. Additional benefits include ideal cache simulation, portable traces that can run on machines other than the ones the traces were generated on, and to facilitate multithreaded/multitasking experimentation.

The Lame Duck policy is evaluated against applications from two benchmarks suites, DIS (Data-Intensive Systems Benchmark Suite) [1] and the SPEC CPU (1995, 2000) Benchmark Suites [3]. Instruction and data references for these applications are generated by Simplescalar, a high-performance instruction-level simulator. Simplescalar was augmented to produce traces in PDATS format [13] , a trace-knowledgeable compressed format. The traces are further compressed with the Gnu compression utility gzip [9] and stored in a gzip'ed format.

The traces are processed by our cache simulator `hiercache`. `Hiercache` has support to model a number cache replacement algorithms: LRU, psedo-ideal, Lame Duck, Lame Duck Variation, and Intelligent Lame Duck. LRU is approximated by ordering data within the cache structure. Every time a cache line is accessed, it is moved into the first column of the appropriate set. The Nth column contains the least recently used cache lines for all sets in the cache. Iideal (pseudo) is done by looking forward withing the traces and choosing the cache line that will be accessed furthest within the future for replacement [7]. The number of references to look forward can be set as a parameter. Despite its name, it is not truly ideal since it does not consider the cost of pushing out modified data, which can be significantly more costly than pushing out clean data.

The Ideal replacement algorithm gathers information from the instruction/data stream before the cache is simulated, and adds lookahead information to the LRU simulation. Ideal chooses the cache line that will be accessed furthest in the future for replacement. The "next-access-time" is maintained for each cache line to make this process easy. The process is started from column 0. When a cache line containing data that will be accessed further in the future than the data looking for a cache line, they are swapped. The process is iteratively performed until the cache line that the original address was located on is reached, where the current comparison data will be inserted. The algorithm produces optimal replacement statistics for a range of cache configurations, just like the LRU simulator.

Chapter 4

Experimental Results

Workload, time quantum, cache size/associativity, and replacement policy are the key factors in determining cache performance. In this chapter, we evaluate how these factors affect the performance of Lame Duck (and its Variant) with respect to two metrics, hit rate and memory operation cost (approximated in cycles). We compare Lame Duck performance to LRU and Ideal. While improvement over the standard is important, it is also important to measure how well Lame Duck mirrors optimal cache performance. Cache performance is optimal when the overall miss-rate of the cache is minimized.

Before proceeding to the performance analysis, we present experimental support for the premises upon which Lame Duck is framed.

4.1 Validation of Assumptions

Lame Duck relies largely on the premise that processes exhibit self-similar footprints; each process accesses cache lines in a fairly regular way. Self-similarity over a time quantum implies that the number of distinct cache lines accessed during the first half of a time quantum is nearly equivalent to the number of cache lines accessed during the second half of the time quantum. Moreover, the number of cache lines accessed within a set during the first half of a time quantum is nearly always equal to the number of cache lines accessed in that set during the second half of a time quantum. These

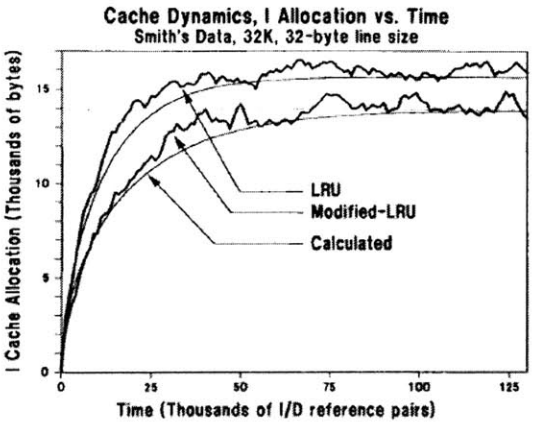


Figure 4-1: Cache allocation to a process approaches a steady-state value about which the allocation varies slightly. Stone, Turek, Wolf simulate this behavior under a LRU replacement and their own Modified-LRU. If the process has not obtained its steady-state allocation by the time quantum midpoint, our assumption of self-similar cache footprints does not hold. However, short transients prevent most time quanta and workloads from falling in this range.

characteristics do not hold for all time quantum lengths and workloads. However, we assume these characteristics without the loss of generality because it is applicable to a large number of time quantum lengths and workloads.

For typical applications, the graph of footprint size verses time indicates that cache allocation changes rapidly if the new allocation is quite different from the present allocation. The rate of change slows considerably as the current allocation comes close to the steady-state allocation. The region of time over which footprint growth significantly slows to a steady-state allocation is often referred to as the "knee" of the graph. Past this time, cache allocation varies over a range centered on a long-term asymptote. Stone et al. simulate and model this behavior as shown by the LRU and Calculated pair of curves in Figure 4-1[19, p. 1062].

If the footprint has not grown to its steady-state allocation by half of the time quantum, our assumption of equivalent footprint sizes during time quantum halves does not hold. In this case, the footprint continues to grow steadily during the second half of the time quantum. We argue that because the transient in cache allocation is very small, most time quanta and workloads do not fall in this range.

Our experiments indicate that this assumption is valid. We analyze various multiprocess workloads using a round-robin scheduling policy. We specifically look at four different time quanta, Q=100, 1000, 10000, 100000. Each workload execution is simulated for 1000 distinct time quanta (1000*time quantum length) data cache references. In each case we measure the number of distinct virtual addresses accessed in each time quantum half. Figure 4-2 reveals that the number of virtual addresses accessed during the first half of the time quantum is approximately equal to the number of virtual addresses accessed during the second half of the time quantum. This indicates that cache allocation at the midpoint of the quantum is sufficient in a fully associative cache in which each cache line only contains data for one virtual address (cache-line size of 1 word).

For a fully associative cache having a cache line size of one word, there is a strong correlation between virtual address access patterns and cache allocation. However, this correlation may not exist for larger cache line sizes. For example, assume that 100 distinct virtual addresses are accessed during each half of the time quantum. A cache line size of one indicates that 100 distinct cache lines are accessed in each quantum half. A cache line size of eight indicates that between 13 and 100 distinct cache lines are accessed in each quantum half; it does not indicate that the number of cache lines accessed in each time quantum half is equal. If the referenced virtual addresses are distributed over the same number of cache lines in each quantum half, then virtual address access patterns are a good indication of whether or not the cache allocation granted to the process at $\frac{Q}{2}$ is sufficient to maintain cache performance in a fully associative cache.

To obtain more precise cache allocation statistics, we measure the number of distinct cache lines accessed in each time quantum half, assuming a cache line of eight 32-bit words and an infinite number of cache columns, each containing 128 sets (4KB column). Figure 4-3 indicates that the cache allocation obtained at halfway through the time quantum is large enough to hold the cache lines that will be accessed during the second half of the time quantum in an infinite-sized fully associative cache.

In order to determine whether or not allocation is sufficient on a per set basis, we look at the distribution of these cache lines within a 4KB page. By measuring the

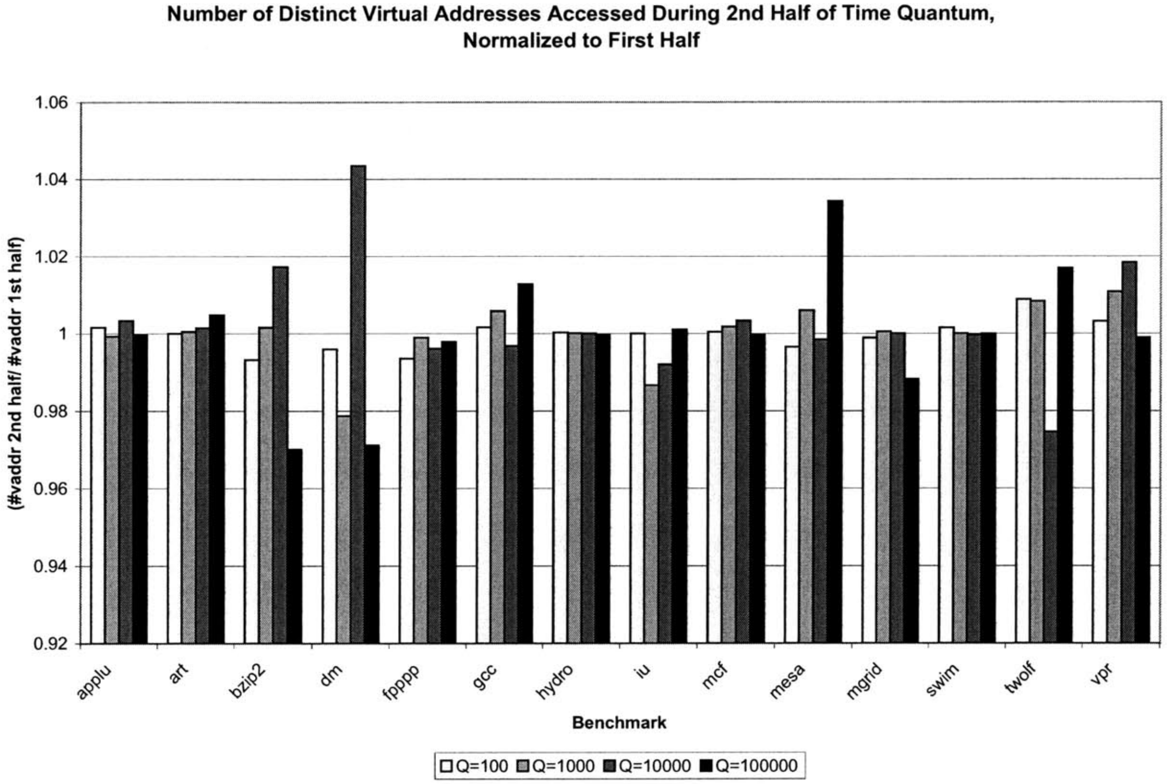


Figure 4-2: Bar graph detailing the average number of distinct virtual addresses accessed in the second half of the time quantum, normalized to the the number of distinct virtual addresses accessed during the first half of the time quantum. Results are shown for multiple DIS and SPEC benchmarks, and time quantum lengths of 100, 1000, 10000, and 100000 memory references.

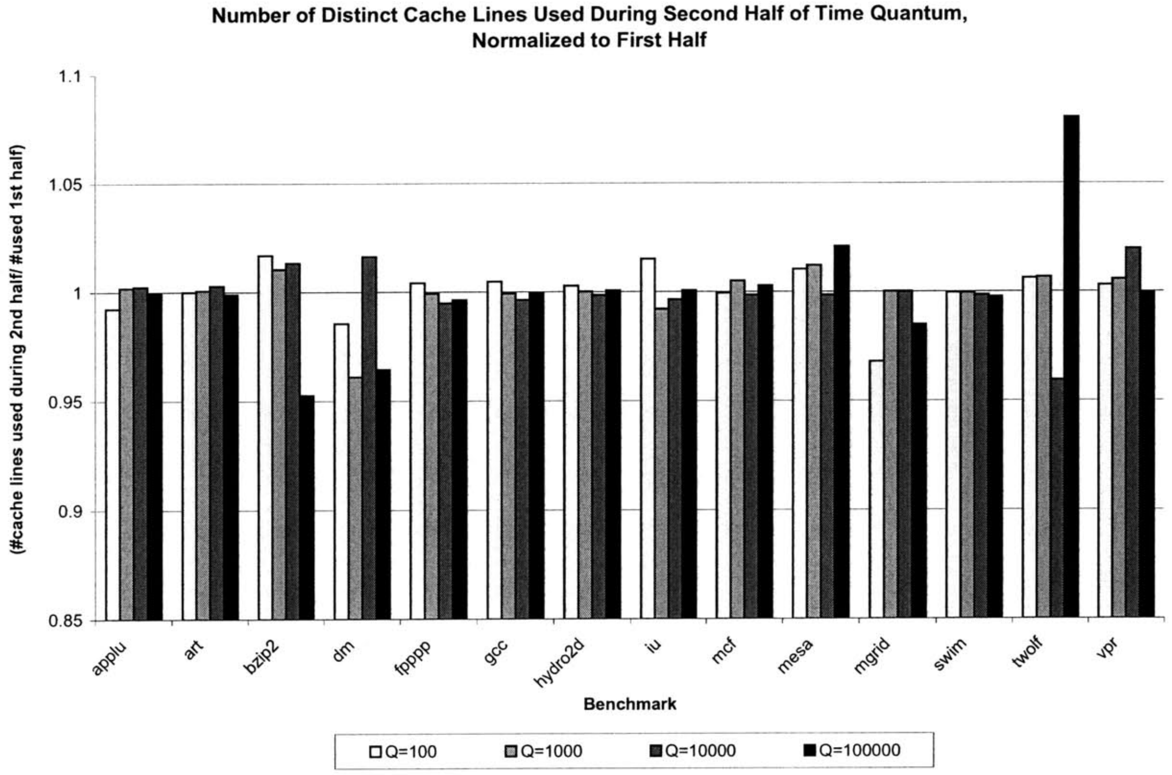


Figure 4-3: Bar graph detailing the average number of distinct cache lines accessed in the second half of the time quantum, normalized to the the number of distinct cache lines accessed during the first half of the time quantum. A 32-byte cache line and a 4KB column is assumed. Results are shown for multiple DIS and SPEC benchmarks, and time quantum lengths of 100, 1000, 10000, and 100000 memory references.

number of cache-line-sized chunks of memory that map to the same position within a page, we can determine the number of distinct cache lines mapping to the same set that are accessed during a time quantum half. If the number of distinct cache lines accessed within the set during the first half of the time quantum is greater than or equal to the number of cache lines that are accessed within the set during the second half of the time quantum, for all sets, cache allocation at the time quantum midpoint is technically sufficient for a set-associative cache. Figure 4-4 indicates that, on average, the application uses the same number of cache lines in a particular set during each quantum half. The difference between cache needs during time quantum halves is characterized by the normal distribution function. For most applications the standard deviation is quite small, although certain applications can vary its memory access patterns widely between quantum halves. We notice a correlation between these applications (requiring high associativity) and applications that perform poorly under the Lame Ducks scheme. We also notice that increasing time quantum lengths lead to a larger variance in cache access patterns between quantum halves. For smaller time quanta, as in 100 memory references, cache allocations granted within a set during the first half of the time quantum may differ from the cache allocation needed after the time quantum midpoint is almost always sufficient. If not, an additional one or two cache lines is generally sufficient.

Assuming an infinite-sized cache allows us to be conservative in determining which cache allocations fall in this category. If we take cache capacity into account, cache allocations obtained at $\frac{Q}{2}$ may consist of entire sets (if we are referring to a set associative cache), or the entire cache (if we are referring to a fully associative cache). In this case, the cache allocation to the process is constrained by the cache itself. The process can not obtain a larger cache allocation during the second half of the time quantum. So cache allocations obtained at $\frac{Q}{2}$ are large enough to maintain the performance of the process, despite the fact that footprint characteristics may differ between quantum halves.

By showing that the cache allocation obtained at $\frac{Q}{2}$ is enough space to hold the cache lines needed after $\frac{Q}{2}$, we argue that with a smart replacement policy the pro-

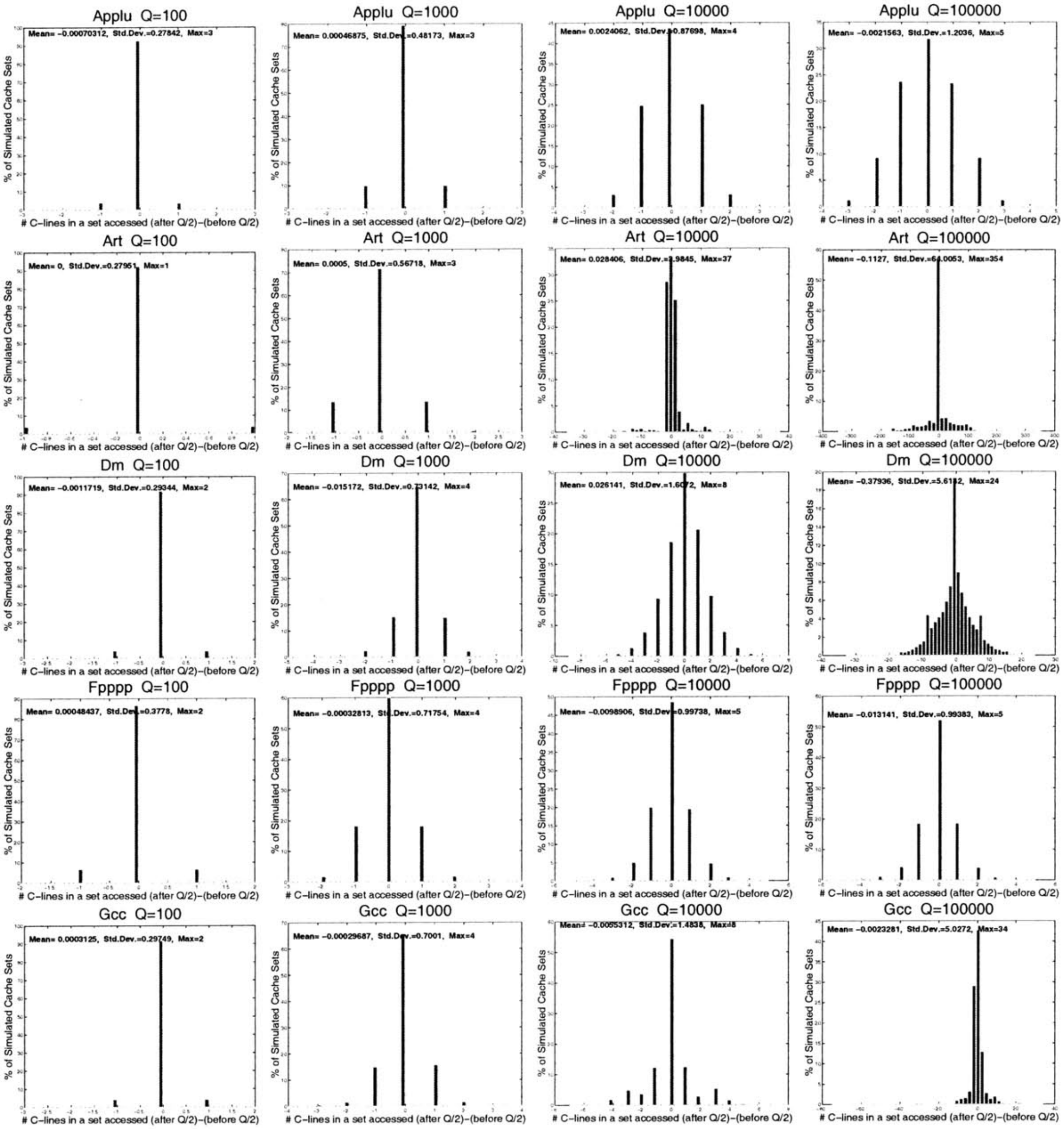
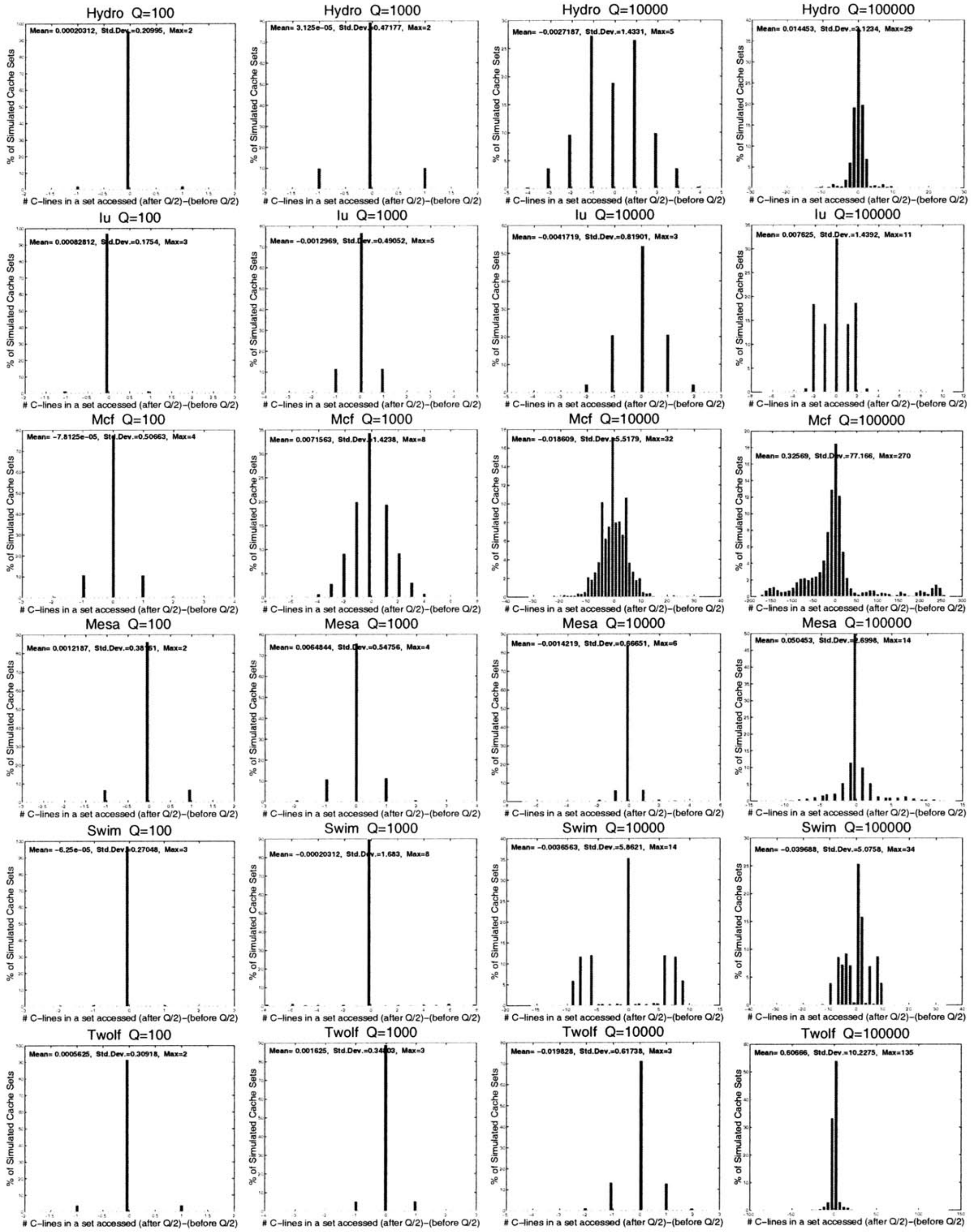


Figure 4-4: Histograms detailing the difference between the number of cache lines accessed during the second half of the time quantum and the number of cache lines accessed during the first half of the time quantum, on a set by set basis. A x-coordinate value of negative two indicates that two fewer cache lines were accessed during the second half of the time quantum for the y-coordinate percentage of sets. Histograms are shown for two DIS benchmarks (Dm and Iu) and multiple SpecCPU benchmarks. Data for time quantum lengths of 100, 1000, 10000, 100000 are shown. Notice that most applications exhibit a normal distribution about 0, and the standard deviation increases with time quantum length.


Hydro Q=100
Mean= 0.00020312, Std.Dev.=0.20995, Max=2
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Hydro Q=1000
Mean= 3.125e-05, Std.Dev.=0.47177, Max=2
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Hydro Q=10000
Mean= -0.0027187, Std.Dev.=1.4331, Max=5
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Hydro Q=100000
Mean= 0.014453, Std.Dev.=3.1234, Max=29
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
lu Q=100
Mean= 0.00082812, Std.Dev.=0.1754, Max=3
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
lu Q=1000
Mean= -0.0012969, Std.Dev.=0.49052, Max=5
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
lu Q=10000
Mean= -0.0041719, Std.Dev.=0.81901, Max=3
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
lu Q=100000
Mean= 0.007625, Std.Dev.=1.4392, Max=11
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Mcf Q=100
Mean= -7.8125e-05, Std.Dev.=0.50663, Max=4
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Mcf Q=1000
Mean= 0.0071563, Std.Dev.=1.4238, Max=8
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Mcf Q=10000
Mean= -0.018609, Std.Dev.=5.5179, Max=32
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Mcf Q=100000
Mean= 0.32569, Std.Dev.=77.166, Max=270
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Mesa Q=100
Mean= 0.0012187, Std.Dev.=0.38161, Max=2
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Mesa Q=1000
Mean= 0.0064844, Std.Dev.=0.54756, Max=4
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Mesa Q=10000
Mean= -0.0014219, Std.Dev.=0.66651, Max=6
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Mesa Q=100000
Mean= 0.050453, Std.Dev.=2.6998, Max=14
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Swim Q=100
Mean= -6.25e-05, Std.Dev.=0.27048, Max=3
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Swim Q=1000
Mean= -0.00020312, Std.Dev.=1.683, Max=8
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Swim Q=10000
Mean= -0.0036563, Std.Dev.=5.8621, Max=14
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Swim Q=100000
Mean= -0.039688, Std.Dev.=5.0758, Max=34
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Twolf Q=100
Mean= 0.0005625, Std.Dev.=0.30918, Max=2
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Twolf Q=1000
Mean= 0.001625, Std.Dev.=0.34403, Max=3
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Twolf Q=10000
Mean= -0.019828, Std.Dev.=0.61738, Max=3
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)
Twolf Q=100000
Mean= 0.60666, Std.Dev.=10.2275, Max=135
% of Simulated Cache Sets
# C-lines in a set accessed (after Q/2)-(before Q/2)

cess(es) can maintain the same level of performance by constraining its footprint at $\frac{Q}{2}$ by allowing its footprint to grow until the end of its time quantum.

4.2 Cache Allocation During a Time Quantum

If we graph the cache needs of a process during a single time quantum against percentage of time quantum elapsed, we obtain a convex curve (resembling a hill) that equals zero at the beginning and end of a time quantum. At the beginning of its time quantum, the current process only accesses a few cache lines and therefore needs only a few cache lines to ensure a cache hit. Initially, the curve looks very much like the transient portion of the cache allocation curve. As the quantum elapses, the process uses more of its working set, thereby increasing the number of cache lines it must keep resident in cache. As the time quantum ends, memory references of the process are made from a smaller set of cache lines. Finally on the last memory reference of the time quantum, the process requires that only one cache line be resident in cache to obtain a cache hit.

If we look at the cache needs of a process over multiple round-robin cycles of time quanta, it becomes clear that it is advantageous for the process to maintain ownership of at least a few cache lines during its dormant phase. These cache lines help mitigate the effect of cold-start misses incurred when the process becomes active again. The second graph in Figure 4-5 confirms that this is true by graphing Ideal cache allocations to a process during a time quantum. Again we have a convex curve, but this hill lies on top of a plateau. This plateau is space allocated for shared variables that will be accessed during the next active time quantum. The hill is space allocated for local variables. In this manner, the Ideal replacement policy limits the amount of cache pollution a process can cause to the footprints of other processes.

LRU cache allocations to a process during a time quantum are often in excess of how much space the process actually needs. We compare hit rates of L1 cache size N KB using the LRU replacement policy with an L1 cache of size $\frac{N}{2}$ KB using the Ideal replacement policy. For every workload and time quantum simulated, Ideal

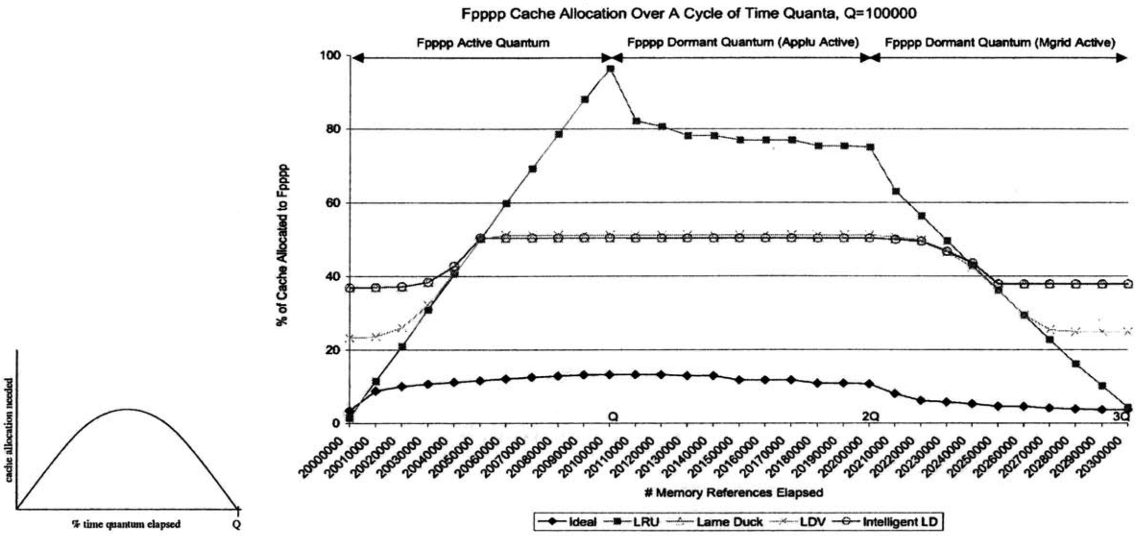


Figure 4-5: (a) Cache allocation required by a single process during a single time quantum, (b) Fpppp Mgrid and Applu are scheduled in a round-robin fashion, with equivalent time quanta of 100,000 memory references. The cache allocation granted to Fpppp over a cycle of time quanta is graphed as a percentage of a 16-way set associative 64 KB $L1. Cache allocations are shown for Ideal, LRU, Lame Duck, LDV and Intelligent Lame Duck replacement policies.

outperformed a cache twice the size running LRU. Ideal makes more efficient use of the cache than LRU, and so it outperforms LRU even if the number of cache lines accessed during quantum halves is not equal.

Lame Duck dynamically constrains the footprint size of each process, thereby curtailing the excess cache allocation to any particular process and maintaining dedicated space for shared variables. We define shared and local variables as the two types of addresses encountered during a time quantum. Shared variables are addresses accessed during both quantum halves. Local variables are addresses accessed exclusively during a time quantum half. (The status of an address may change over different time quanta; currently we are only concerned with access patterns within a single time quantum.) By constraining footprint growth, Lame Duck attempts to mitigate the cold start of processes by leaving useful data in the cache for each process to use the next time it is executed. By definition, Lame Duck tries to keep shared variables in the cache.

For some workloads, Lame Duck cache allocations over a time quantum closely mirror the characteristic "plateau" found in Ideal cache allocations, while LRU's allocations are significantly different. Lame Duck curtails excess cache allocations granted under the LRU scheme. By virtue of constraining footprint size, Lame Duck dedicates some amount of cache space to the current process. The policy updates the partition every time quantum. Although the partition is not updated at very small time intervals, Lame Duck is an improvement upon the statically defined partitions used in the aforementioned cache partitioning scheme.

4.3 Performance Analysis

In this section we present the results of trace-driven simulations of set associative and fully associative two-level caches in which $L1 cache sizes range from 16 KB to 256 KB, and $L2 cache sized range from 32 KB to 640KB. Each column contains 128 sets, and each cache line contains eight 32-bit words yielding a 4KB column. The simulations compare caches managed by a conventional LRU algorithm to those managed by our Lame Duck algorithm. The results are grouped into three sections which evaluate the original Lame Duck Policy, LDV and the Intelligent Lame Duck Policy respectively.

For simplicity, all simulations contain two or more processes running in a round-robin fashion having identical time quanta. All processes execute for the entire duration of the simulation; at no point during the simulation does the number of scheduled jobs or the round-robin ordering change. In addition, each process executes for an entire time quantum length. Each workload is simulated for 50-150 million references, depending on the number of processes interleaved.

Simulations are constructed in the following way:

1. Specify a workload that exhibits certain characteristics:
   {small/large working sets, data sets, number of processes, amount of streaming data, etc.}

2. Choose replacement policy:
   {Lame Duck, LDV, Intelligent Lame Duck, LRU, Ideal}

3. Choose cache associativity:
   {set associative, fully associative}

4. Choose a cache size:
   \$L1::$\{16, 32, 64, 96, 128, 192, 256\}KB$
   \$L2::$\{32, 64, 80, 128, 256, 320, 640\}KB$

5. Choose a time quantum length:
   $\{100, 1000, 10000, 100000\}$ memory references

6. Simulate workload execution using `hiercache`

4.3.1 Performance Metrics

Each simulation is evaluated on the basis of two metrics, hit rate and run-time cost. Hit rates are calculated as the percentage of cache hits among all memory references, and are obtained with respect to individual processes as well as aggregate workload. Run-time cost is approximated by calculating the average number of cycles the processor must wait on each memory operation based on statically designated costs. Table 4.3.1 lists the cycle costs associated with each memory operation outcome for a two-level cache.

Hit rate allows us to compare efficiency of cache usage between various replacement policies. Identical simulations (workload, cache characteristics, time quantum) are carried out for each replacement policy. We compare Lame Duck with LRU and Ideal, to show not only how the policy does with regards to the standard but also how closely the policy models the behavior of an "oracle", the optimal attainable performance.

Cycle cost is a useful metric because it allows combined \$L1 and \$L2 hit rates to be compared easily between replacement policies. For some workloads, Lame Duck and LRU hit rates are nearly equal for \$L1 caches. At times, Lame Duck yields hit rate degradation in \$L1 caches coupled with a significant increase in \$L2 hit rates. We incorporate \$L1 and \$L2 hit rates into a single value by using the memory access cost model in Table 4.3.1.

As the gap between processor and memory speeds widen, the cost associated with cache misses will increase.

	cost
hit in \$L1	1
miss in \$L1, hit in \$L2	10
miss in \$L1, miss in \$L2	100

Table 4.1: Cost of Memory Operations Approximated in Cycles

4.3.2 Lame Duck

First we look at Lame Duck performance in fully associative caches. Although most caches are set-associative, fully associative caches are easier to think about; the effects of set contention can be disregarded.

Fully Associative Caches

For fully associative caches, Lame Duck hit rates are very close to those of LRU. \$L1 hit rates are within 0.1% of each other. For workloads consisting of a large number of applications (in our studies, larger than 5 applications), Lame Duck hit rates are typically the lower of the two. This finding underlines the fact that Lame Duck must be used with discretion; it does not perform well for all workloads and time quanta. For workloads consisting of a large number of applications (each desiring cache space), Lame Duck is unlikely to yield performance improvements. In this case, Lame Duck constrains footprint size despite the large probability that a process' entire footprint will be evicted over a cycle of time quanta (assuming a round-robin scheduling policy). This scheme proves detrimental to cache performance when cache misses incurred as a result of preventing footprint growth that cannot be amortized over cache hits from saved state. Figure 4-6 shows Lame Duck and LRU hit rates for a 36 process workload on a fully associative 64KB \$L1 cache. Notice that for increasing time quanta hit rates for both policies increase significantly, yet Lame Duck does not outperform LRU.

Lame Duck performance cannot be so easily generalized for workloads containing fewer applications. Memory access patterns of each application and time quantum length both play crucial roles in determining where Lame Duck performs well.

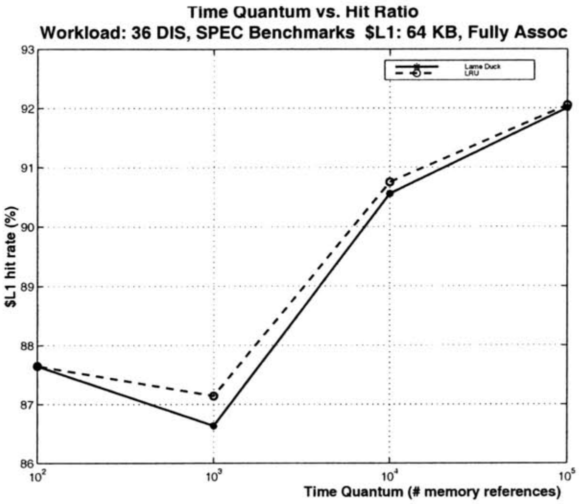


Figure 4-6: Hit rates obtained for the interleaved execution of 36 DIS and SPEC benchmarks under Lame Duck and LRU replacement policies. Hit rates are given for a fully associative two-level cache, 64 KB \$L1 and 80 KB \$L2

A textbook use for Lame Duck is in conjunction with streaming applications. Lame Duck prevents the streaming application from polluting the entire cache with stagnant data and evicting the working sets of other processes. The application, Mcf[3], streams over 12 million addresses. Memory references in this range are sparse and exhibit some locality. Interleaved with this application is Data Management (Dm) [1]. Data Management also has a large footprint; its data set spans nearly seven million addresses. However, they only account for a minority of the memory accesses; 80% of all references are stack references (approximately 4K addresses).

While Mcf exhibits little locality, Dm exhibits strong locality. Lame Duck improves Dm hit rate as we suspect. However, Lame Duck's effect is limited since both applications have large footprints and tend to pollute each other's cache allocation. Figure 4-7 shows Lame Duck and LRU hit rates for a time quantum length of 10,000 references and various cache sizes.

Workloads consisting of applications having small working sets also benefit from Lame Duck. Fpppp, Mgrid, Applu are SpecCPU Benchmarks [3], each of which can contain their working set in 8KB of cache. Although each process contains many small streams, they exhibit strong locality. Each cache line is accessed multiple times consecutively, resulting in few capacity misses; most cache misses are cold-

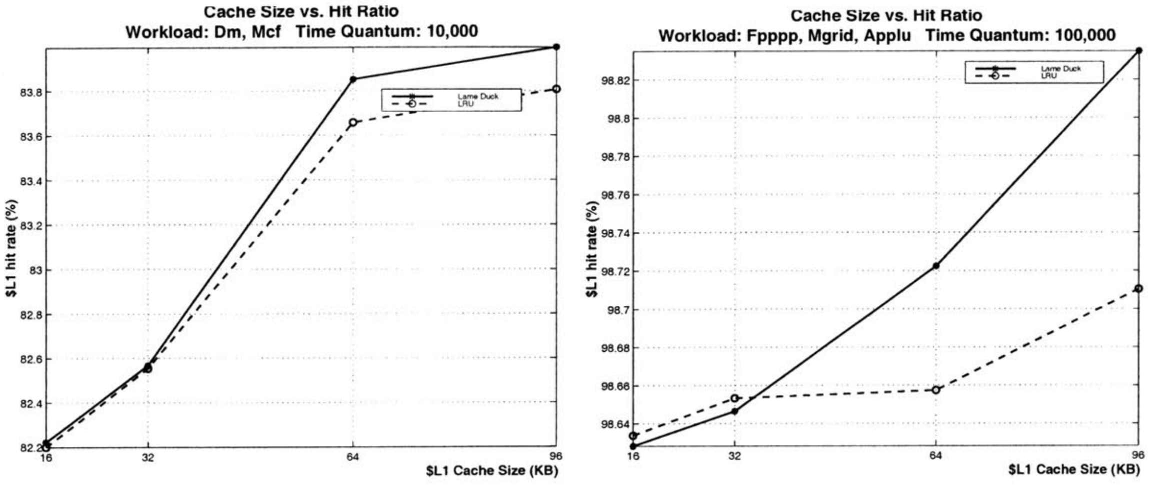


Figure 4-7: (a) Lame Duck and LRU hit rates obtained by simulating Dm and Mcf in fully associative $L1 caches of sizes 16KB, 32KB, 64KB, and 96KB for a time quantum length of 10,000 memory references. (b) Lame Duck and LRU hit rates obtained by simulating Fpppp, Mgrid, and Applu in fully associative caches of sizes 16KB, 32KB, 64KB, and 96KB for a time quantum length of 100,000 memory references.

start misses. Figure 4-7 shows Lame Duck and LRU hit rates for a time quantum length of 100,000 references and various cache sizes. We see an increase in cache hit ratios from 98.71% to 98.84%, a 10% reduction in miss rate. For smaller time quanta and the same cache size, Lame Duck $L1 hit rates are slightly less than LRU (within 0.05%)

Although increases in $L1 hit rates are small, significant increases in $L2 hit rates boost Lame Duck performance with regards to the cost metric. Figure 4-8 graphs cycle costs of Lame Duck and LRU. The graph indicates that for a time quantum of 100,000 memory references and a fully associative two-level cache (64KB $L1 and 80KB $L2) using the Lame Duck replacement policy, memory operations cost almost 6% less than it would on the same cache using an LRU replacement policy. Admittedly, gains are not universal, even for this workload. For smaller time quanta, memory operations made under the Lame Duck policy cost more than under LRU.

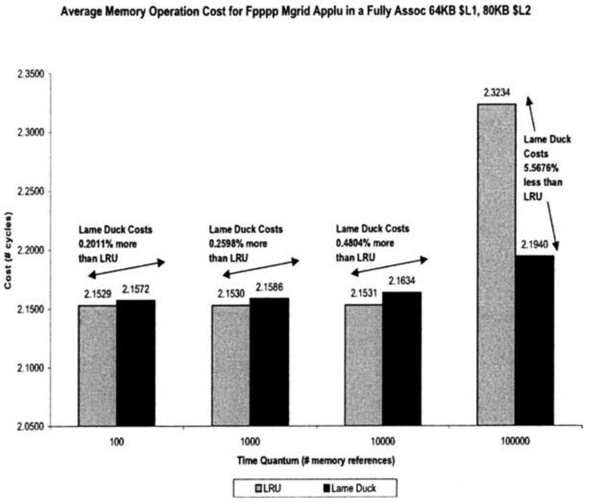


Figure 4-8: Average cost (in cycles) of LameDuck and LRU memory operations based on $L1 and $L2 hit rates and statically defined costs. Fpppp Mgrid and Applu memory costs are calculated for various time quantum lengths assuming a fully associative two-level cache, $L1 64 KB and $L2 80 KB.

Set Associative Caches

We notice that that Lame Duck exhibites significantly different behavior when one considers set associative caches. For example, workloads consisting of application having large footprints, such as `Mcf` and `Dm`, yield Lame Duck performance gains for a fully associative cache; however, performance decreases for set associative caches.

Lame Duck performance also decreases for workloads in which a large number of processes aggressively compete for cache space. Figure 4-9 shows $L1 hit rates and memory operation costs for an application consisting of nine applications in a two-level cache (96KB $L1, 128KB $L2). Lame Duck hit rates perform up to 35% worse than LRU hit rates, and memory operations cost up to 40% more than LRU. It is not surprising that Lame Duck does not perform well for this type of workload in a set associative cache since it did not yield significant benefits for the same in a fully associative cache.

In previous sections we argue that self-similar footprints result from fairly regular cache line access patterns, and as such the number of cache lines accessed within a set during the first half of a time quantum is always nearly equal to the number of cache lines accessed in that set during the second half of a time quantum. Thus cache

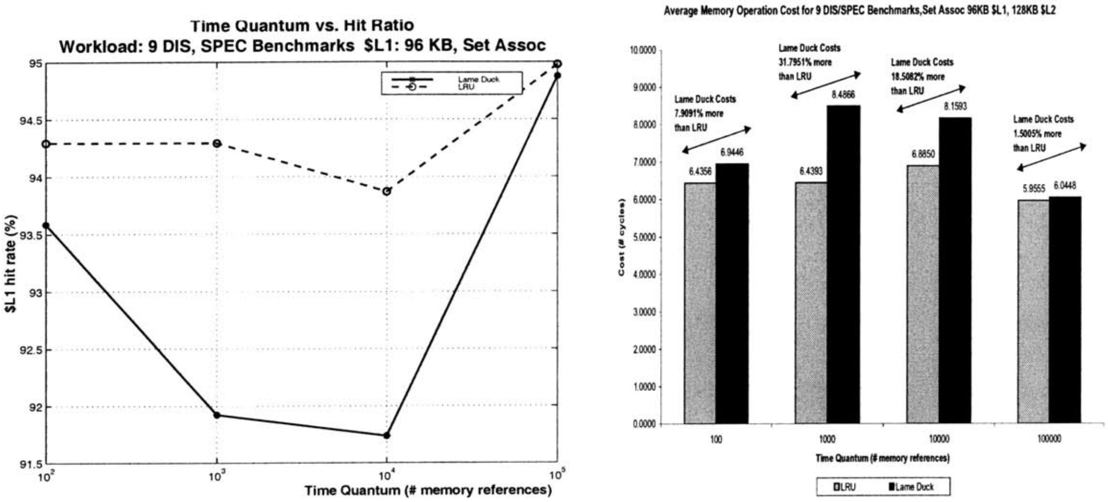


Figure 4-9: Memory accesses made by nine Spec and DIS Benchmarks were simulated on a 24-Way set associative 96 KB $L1 cache. (a) Lame Duck and LRU hit rates. (b) Lame Duck and LRU memory operation costs.

allocations obtained at time quantum midpoints are technically enough to maintain cache performance while constraining footprint growth. However, Lame Duck simulations for set associative caches indicate that constraining process footprints at $\frac{Q}{2}$ generally yields sub-LRU hit rates and memory operation costs. We explain this by noting that constraining footprint size at halfway through the time quantum can effectively halve cache associativity available to each process. There is an important distinction between halving cache capacity and halving cache associativity. Cache designers often choose to decrease column size over decreasing cache associativity. One reason is that higher associativity provides the cache with greater flexibility in determining cache line placement, which leads to less set contention.

Not all processes have self-similar footprints. As a result, cache needs during the first half of a time quantum provide no insight into cache needs during the second half of the time quantum. A process may access many lines in a particular set after the time quantum midpoint. If the process owns only a few cache lines in that set, or worse yet does not own any lines in that set, the application's hit rate will suffer very noticeably. This is particularly problematic for applications having "hot sets" in which a process uses a limited number of cache sets, but many cache lines mapping to that set are accessed. A good example is an application that strides through a large array such that all accessed indices map to a few sets.

Swim, a Spec95 benchmark, exhibits a very particular memory access pattern. The application consists of ten interleaved streams. These streams move very slowly and in unison. Each stream accesses cache lines from the same set, and each cache line is accessed multiple times, as depicted in Figure 4-10. However, once the application moves past the current set of addresses, it does not access them again. This peculiar behavior makes Swim a nightmare for Lame Duck. The fact that multiple streams are accessing the same set simultaneously means that the application requires high associativity to store its working set in cache. For typical streaming applications this would not matter since each address is only accessed once; Swim differs radically since it exhibits local temporal locality. Each stream accesses the same cache line on the order of twenty times consecutively. Afterwards the cache line is not accessed again. Swim requires high associativity to do well; in a cache having low associativity, constraining the size of Swim's footprint at any point during the time quantum is a bad idea. Figures 4-10 graphs $L1 and $L2 hit rates for Swim, Mgrid and Image Understanding, the latter is a Data Intensive Benchmark while the two former applications are Spec95 benchmarks. Hit rate decreases in $L1 and $L2 lead to high memory operation costs, up to 80% more than LRU costs. Figure 4-10 depicts costs for a 16KB $L1 cache.

This is not to say that Lame Duck performs poorly for set associative caches on the whole. Workloads consisting of multiple applications having small working sets exhibit performance gains under Lame Duck for set associative caches. Two such workloads consist of interleaved traces of three Fpppp applications, and the second of Fpppp-Mgrid-Applu. The former shows strong performance gains for both hit rate and memory operation cost, with $L1 hit rates outperforming LRU by as much as 22.6% and costs up to 11.8% lower than LRU. The performance gains of the latter are more modest and are specific to long time quantum lengths, e.g. 100,000. Figure 4-11 shows hit rates and memory costs for the Fpppp application in a 16KB $L1 cache and 32KB $L2 cache.

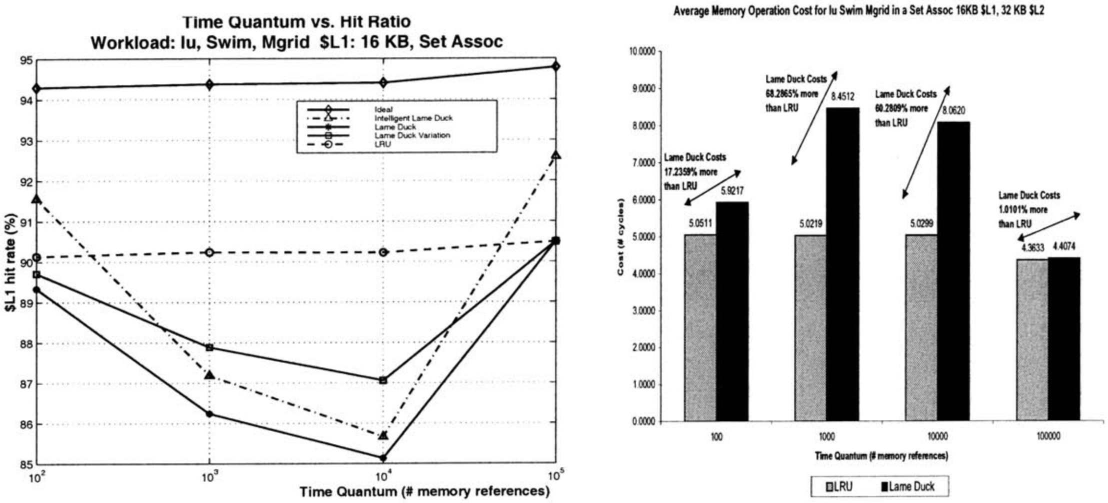


Figure 4-10: (a) Iu Swim Mgrid $L1 hit rates. (b) Comparison of Lame Duck and LRU average cost of a memory operations. Costs were calculated by averaging data from 1000 time quantum samples.

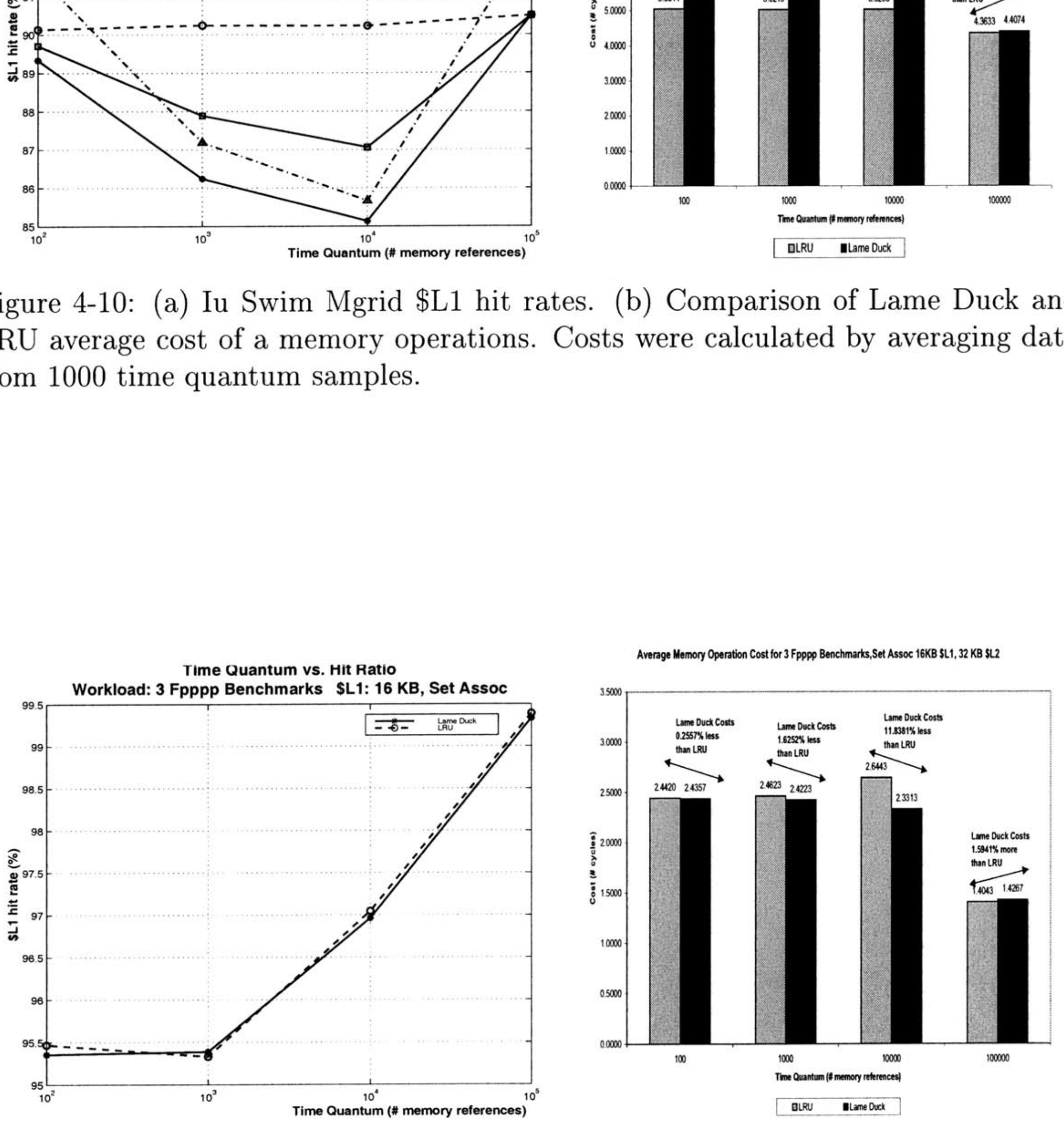


Figure 4-11: (a)Fpppp hit rates for a 16KB $L1. (b) Fpppp memory operation costs obatined simulating a 16KB $L1 and 32 KB $L2.

4.3.3 Lame Duck Variation

While Lame Duck performance gains over LRU do exist for set associative caches, the gains are specific to a certain time quantum lengths and cache configurations. Which simulations yield positive Lame Duck performance is highly dependent on workload characteristics. In order to address poor performance in set associative caches, we study a variation of the Lame Duck cache replacement policy. Instead of constraining cache footprint size at the time quantum midpoint, we allow a process to continue growing until it owns half of the cache lines in the targeted set. This guarantees each process at least of the cache available for its use. While this does not remove the effect of halving cache associativity to a process, it ensures that the process obtains at least half of its optimal allocation within a set, even if new sets are accessed during the second half of the time quantum. Note that this algorithm is not meant for fully associative caches since that would require the process to own half of the cache before recycling cache lines.

This modification avoids many of the bad cases for original Lame Duck. For example, for workloads consisting of six, nine, twelve and thirty-six interleaved processes, LDV performs just as well if not better than LRU for multiple caches size and time quanta.

For applications having large footprints, simulations using LDV on a set associative cache behaves similarly to Lame Duck in a fully associative cache; Lame Duck (fully associative) and LDV (set associative) outperform LRU for the same time quanta and cache sizes. Figure 4-12 shows hit rates for `Mcf` and `Dm` in various cache sizes and associativities, given a time quantum of 10,000 memory references.

For applications having small working sets, LDV once again outperforms the original Lame Duck policy. Unlike LRU, LDV achieves better or equivalent hit rates as compared to LRU for short time quanta (e.g. 100) as well as long time quanta.

Performance improvements made possible by changes to the original Lame Duck policy do not hold for all workloads. Applications requiring highly associative caches to perform well, such as `Swim`, are still a source of severe hit rate decreases and mem-

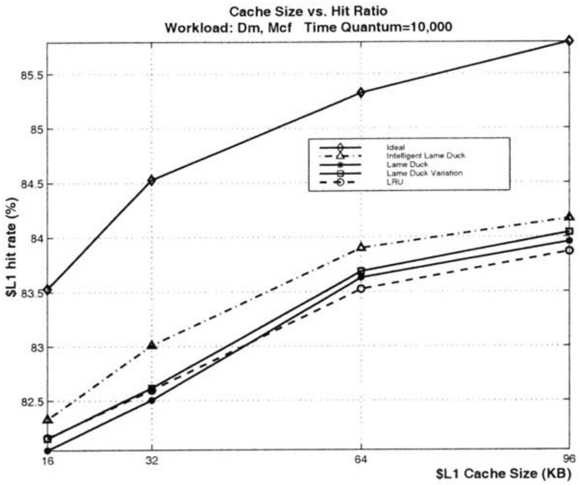


Figure 4-12: Dm Mcf $L1 hit rates as simulated in $L1 caches of size 16KB, 32KB, 64KB, and 96KB for time quantum length of 10,000.

ory cost increases (with respect to LRU; LDV obtains better hit rates and memory costs than Lame Duck). This is not surprising since we have already established that the manner in which LDV constrains footprint size can effectively halve cache associativity.

We solve this problem by introducing random hashing [10, 11]. Random hashing decreases set contention for any individual set by re-mapping cache lines to sets other than the one indicated by the specified bits of the relevant physical address. This is key to improving Swim cache performance; hashing removes Swim's need for high associativity by decoupling the streams within the application. As a result, hit rates for Lame Duck, its Variation, and LRU improve drastically. However, since Swim with hashing behaves similar to typical streaming applications, Lame Duck is able to outperform LRU, both with regards to hit rate and memory operation costs. Figure 4-13 graphs memory operation costs obtained under LDV using a 96KB $L1 cache and 128KB $L2 cache. Values for simulations with and without hashing are shown.

Even though LDV improves upon Lame Duck performance, LDV as a whole performs the same as LRU. We can at least partially explain why it does not do better by noting that although Lame Duck does not assume locality between processes, it uses the same metric as LRU to decide what cache line should be evicted. Constraining footprint size prevents cache pollution to a degree, but the right cache lines still

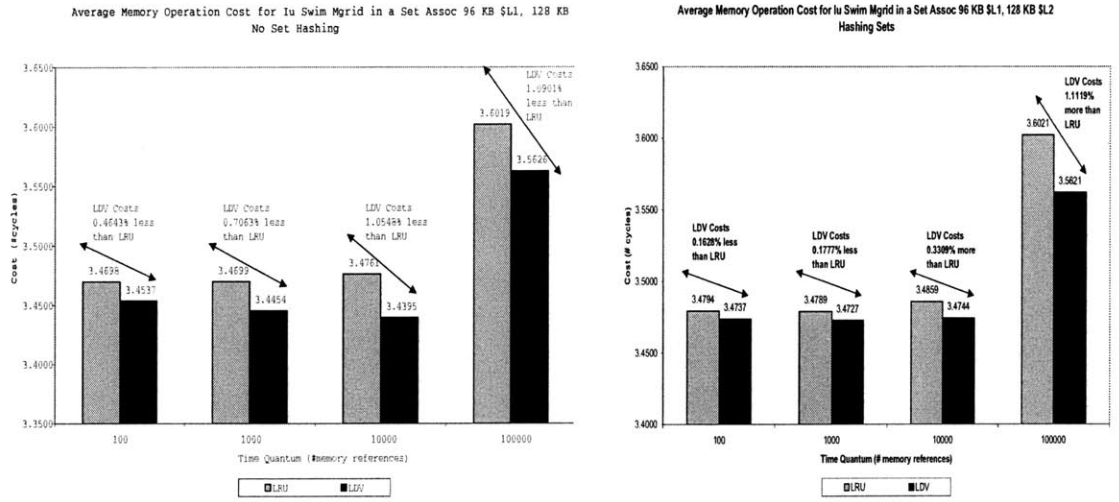


Figure 4-13: (a)Memory operations costs without hashing the address, as calculated via simulation of a 96 KB $L1 and 128 KB $L2 cache. (b) Memory operation costs with hashing for the same cache configuration.

need to be kept in cache. The goal of Lame Duck is to keep shared variable in cache between time quantum halves. If Lame Duck does not do this well then constraining space will decrease cache performance; this is specifically true if locality arguments do not apply to a particular process. By allowing a process to increase the size of its footprint, the replacement policy is allowed a margin of error. An intelligent replacement policy would use the cache more efficiently, thereby making recycling of cache lines beneficial.

4.3.4 Intelligent Lame Duck

For workloads consisting of applications with large footprints, Intelligent Lame Duck $L1 hit rates outperform both Lame Duck and LRU. In order to measure how close the policy performs to optimal, we compare Intelligent Lame Duck hit rates with Ideal hit rates. Although Intelligent Lame Duck does not make significant progress in bridging the gap between LRU and Ideal hit rates, we notice that Intelligent Lame Duck performs almost exactly the same as using an Ideal policy on half the cache size.

For workloads whose performance is severely limited by cache capacity, it is unlikely that that a given process will have cache lines resident in cache when it regains

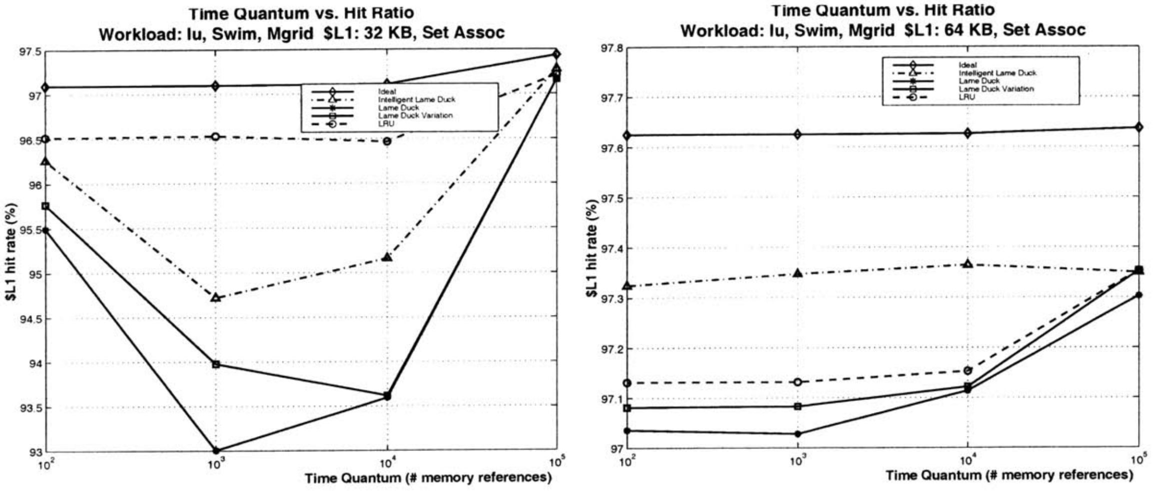


Figure 4-14: Intelligent Lame Duck hit rates obtained for Iu Swim Mgrid in a (a) 32 KB $L1 cache. (b) 64 KB $L1 cache. Intelligent constraining of cache footprint size can improve Swim's cache performance.

control of the processor. Even if the cache has access to future information, it may be of little use if no single process can fit its footprint in the cache. Since constraining cache footprints size is detrimental to such workloads, we expect a noticeable difference between Intelligent Lame Duck and Ideal hit rates for the same cache size. If cache capacity does not place severe constraints on cache performance, Ideal of cache size N KB and Ideal in a cache of size $\frac{N}{2}$ KB is virtually equal.

For set associative caches, LRU and LDV obtain better hit rates and memory operation costs than Intelligent Lame Duck on applications requiring high associativity. `Swim` (without hashing), our running example of such an application, obtains the best hit rates under LRU for most cache sizes and time quanta. For a 32KB $L1 cache, LRU obtains better hit rates than Intelligent Lame Duck, but Intelligent Lame Duck obtains better hit rates than LDV. In this case, efficient use of the cache outweighs the value of a slight increase of cache allocation to the process. A significant increase in cache allocation, as given by an LRU replacement policy, outweighs efficient use of effectively half the cache. Figure 4-14 depicts `Iu`[1] `Swim` `Mgrid` hit rates obtained using a 32KB $L1 cache.

We notice that for a $L1 set associative cache of size 64 KB, Intelligent Lame Duck outperforms both LRU and LDV. In this case, halving cache associativity by

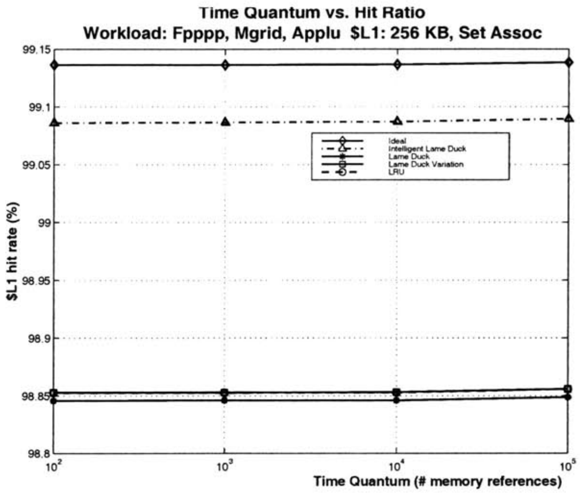


Figure 4-15: Intelligent Lame Duck hit rates are very close to optimal for workloads consisting of applications having small working sets.

constraining footprint size does not pose a significant limitation on the cache performance for this workload, as long as the space allocated to the process is used efficiently. Figure 4-14 depicts `Iu Swim Mgrid` hit rates obtained using a 64KB $L1 cache. These results show that although in the general case it almost never makes sense to constrain the footprint of `Swim` at the time quantum midpoint, in particular cases it can lead to performance gains over LRU.

Intelligent Lame Duck also obtains hit rates very close to Ideal for workloads consisting of applications having small working sets. For a 256 KB $L1 cache, Intelligent Lame Duck hit rates are within 0.05% of Ideal. Figure 4-15 shows `Fpppp Mgrid Applu` hit rates obtained by both replacement policies for various time quantum lengths.

Workloads consisting of applications having large footprints perform well in caches using the Intelligent Lame Duck replacement policy. Simulations interleaving traces of `Dm` and `Mcf` yield Intelligent Lame Duck hit rates that surpass LRU and Lame Duck Variation hit rates. As we expect, Intelligent Lame Duck hit rates are not very close to optimal; in fact, they are closer to LRU than to Ideal. We attribute this behavior to the streaming nature of `Mcf` and LRU's inadequacy in dealing with streams.

For the first half of the time quantum, the cache allocation of each process grows as under a LRU replacement policy. This allows `Mcf` to pollute the cache with data that will not be accessed again, resulting in the eviction of many cache lines belonging to

`Dm`. Although Lame Duck can help curtain the damage streaming applications inflict on the cache footprints of other processes, (as is shown by Intelligent Lame Duck's improvement over LRU), it can not outperform a policy that never allows streaming data to be cached. As a result, Intelligent Lame Duck hit rates for this workload are not comparable to Ideal hit rates obtained using a cache half the size. Figure 4-12 shows `Dm Mcf` hit rates for a 64 KB $L1 cache. Notice the gap between Intelligent Lame Duck hit rates and Ideal hit rates. Ideal for a $L1 cache of size 32 KB is also plotted on this graph.

In addition, Ideal hit rates for a cache of size N can be noticeably higher than Ideal hit rates for a cache of size $\frac{N}{2}$, (increased cache size decreases capacity misses, which is often a source of hit rate degradation for processes having large footprints). This further widens the gap between bit rates obtained by Intelligent Lame Duck and Ideal.

We only simulate Intelligent Lame Duck in $L1 caches, so no memory operation cost approximations were calculated for comparison with LRU and other Lame Duck policies.

Chapter 5

Conclusions and Future Work

5.1 Summary of Contributions

The main contribution presented in this thesis is the exploration of a cache replacement policy that considers the effect of context switching between multiple processes upon cache performance. While LRU often provides near-optimal hit rates, in truth it is a static policy that assumes locality of reference between memory accesses of different processes. This false assumption can result in sub-optimal cache allocations to processes and the degradation of cache hit ratios.

We present Time-Adaptive algorithms, a class of cache replacement policies that does not assume locality of reference between memory acceses made by different processes. In order to prepare the cache for the end of one process' execution and the onset of another, Time-Adaptive algorithms constrain the footprint of the currently executing process at some point through the time quantum. The policy fixes the cache allocations to each process, allowing the current process only to evict its own data from cache. By constraining footprint growth, Time Adaptive algorithms mitigate pollution of cache footprints and keep more cache lines resident in cache between active phases of a process, resulting in the reduction of cold start misses incurred after a context switch.

We build upon the work of Stone, Turek, and Wolf whose "Multiprogramming Replacement Policy"[19] falls into the category of Time-Adaptive Caching. Their work set the foundation for this research and posed the following questions: "For multiprogrammed systems, what is a practical means for implementing a limit on cache allocation? Does such a scheme produce sufficient benefit to justify its implementation? How large do caches have to be for non-LRU replacement to be worthwhile?" We make progress on answering these questions through our studies of the Lame Duck cache replacement strategy, a specific Time-Adaptive algorithm that strives to improve overall cache hit rates by constraining footprint size once the midpoint of the time quantum has been reached.

Lame Duck implementation requires minimal modifications to current cache architectures. It allows interaction between the operating system and cache hardware, enabling the cache to dynamically change cache replacement policy based on time quantum lengths and the amount of time quantum elapsed. Lame Duck's metric for deciding when to limit cache allocation to a process leverages information about self-similar cache footprints. The simplicity of the metric makes it practical to implement, but not necessarily worth the resources.

Lame Duck performance is equally dependent on a number of factors, making the generalization of "good cases" and "bad cases" extremely difficult. Memory access patterns of each interleaved process, time quantum, and cache size all play a role in deciding when, if ever, constraining footprint growth and recycling of cache lines improves cache hit rates.

Our simulations show that while Lame Duck does outperform LRU for certain workloads and time quanta in fully associative caches, it yields little and often negative benefit for set associative caches. The reasons for this are two-fold. Lame Duck effectively halves the cache associativity available to a process by constraining the size of process footprints at the time quantum midpoint. In addition, the LRU metric for deciding which cache line should be evicted on a cache miss is not good enough.

Consider identical workloads simulated on two caches, the first of size N KB using the LRU replacement policy and the second of size $\frac{N}{2}$ and half the associativity using an Ideal replacement policy. For many workloads, the latter simulation will obtain higher hit rates than the first, indicating that LRU does not use cache space as efficiently as it could. By constraining the cache footprint of a process we make this problem worse, resulting in lower hit rates (even though the cache allocation may technically be enough space to hold all of the needed data).

In order to alleviate these problems we proposed a variation of the Lame Duck policy that allows the process to increase its cache allocation until it owns half of the cache lines in the set. While this does not remove the effect of effectively halving the cache associativity available to each process, it ensures that the process is granted at least half of its optimal allocation within a set. Our simulations show that LDV performs much better than the original Lame Duck, and that LDV often meets or surpasses LRU hit rates for set associative cache configurations.

We provide a general guideline of when LDV can yield gains in cache hit rates and when it will not; the lists are not exact and are not complete. The seemingly infinite number of possible combinations of workloads characteristics, cache sizes, and time quanta makes exact delineation nearly impossible. The usefulness of cache partitioning must be determined on a case by case basis. In addition, many of the measurements are subjective ("long enough", "small enough") and workloads often meet criteria on each list. An example is `Swim`, a streaming application requiring high associativity.

Our simulations show that LDV should not be used in conjunction with

- cache sizes that are not large enough to hold the working set of one process in addition to part of the working set of another process in the workload. If a process is constrained by the size of cache, it should be allowed to use the entire cache.

- time quanta lengths so long that by the time quantum midpoint the running process has evicted all cache lines belonging to other processes.

- time quantum lengths so short that the time quantum midpoint occurs during the cache allocation transient of the process. The goal is to curtail cache allocations once they have reached steady-state, not to stifle processes by denying them sufficient cache space.

- workloads consisting of a large number of processes such that regardless of time quantum length and cache recycling efforts, no process will have cache lines resident in cache upon return from a context switch. This is applicable to muliple scheduling policies, but is easily understood in the context of a round-robin scheduling policy.

- workloads containing one of more processes that do not exhibit self-similar cache footprints.
- workloads containing one or more processes requiring high associativity to obtain high hit rates. Processes that have "hot sets" can suffer from a large increase in cache misses since Lame Duck effectively halves the available cache asscociativity.

LDV yields hit rates superior to LRU for

+ cache sizes large enough to hold multiple working sets of the workload in cache concurrently.
+ cache sizes small enough that it can not concurrently hold the cache footprints of all processes in the workload.
+ a middle range of time quanta, neither too large nor too small, that can only be specified with knowledge of cache size and process characteristics.
+ workloads containing one or more streaming processes. By constraining the cache footprint of the streaming process, Lame Duck mitigates the cache pollution it gives rise to.
+ workloads consisting of processes having small footprints. Constraining the cache footprint of any individual process has little effect on cache performance. The cost of a few additional cache misses at the end of a time quantum is amortized over the reduction in cold-start cache misses incurred after a context switch.

Although LRU incorrectly assumes locality of reference among memory references (made by different processes) surrounding context switches, Lame Duck is not the solution. The hit rate gains afforded by Lame Duck (and LDV) are not significant enough or broad enough to merit diverting both hardware and software resources to it. Lame Duck studies do indicate that constraining cache footprints during a time quantum can be beneficial; however, better metrics are needed to decide at what point during a time quantum to begin constraining the cache footprint of the current process, and to decide which cache line should be evicted on a cache miss once cache recycling has begun. Although our work does not propose a viable solution to the reduction of cache pollution in time-shared systems, it does provide insight into when and where a similar policy can be useful. Lame Duck sets the stage for new Time-Adaptive algorithms that may follow.

5.2 Future Work

There are several topics requiring further investigation that lead on from this work.

5.2.1 Instructions per Cycle

Lame Duck performance was evaluated on the basis of cache hit rates and approximations of memory operation costs, e.g. the number of cycles needed to complete a memory operations. Our cost approximations are very loose and do not account for the cost of pushouts, stale data that is pushed out of the cache and written to memory (generally as a result of evicting a cache line). Cache-line writebacks use precious CPU cycles, and can interfere with process execution. Instruction level simulators such as Simplescalar can be modified to provide accurate informations detailing the number of Instructions per Cycle (IPC) in multiprogrammed environments. Although Lame Duck does not perform well with regards to hit rates, it may increase IPC.

5.2.2 Scheduling Policy

All of the experiments used to evaluate Lame Duck assume a round-robin scheduling policy and equal time quanta for each process. These assumptions simplify the study of how context switching effects cache performance. However, most modern operating systems use a priority-based scheduling policy with varying time quanta. While many of the conclusions we have drawn are valid for different scheduling policies and time quanta, it is important to fully understand the potential benefits and pittfalls of Lame Duck in more realistic settings.

5.2.3 Workload

In addition to applications, today's microprocessors spend a significant amount of time executing small code segments for small processes such as interrupt handlers and daemons. Various problems with our system-level simulator prevented us from obtaining accurate instruction traces of such processes. As a result, the workloads we have simulated consist solely of DIS and SPEC benchmarks. Interrupt handlers, and the like, might prove to be the class of workloads for which Lame Duck universally obtains higher hit rates than LRU. If an interrupt process gains control of the processor for a short period of time, LRU will allocate cache space to the process on every cache miss. Lame Duck preserves the data belonging to the interrupted process by allowing the interrupt handler a small cache allocation, but then quickly curtailing its acquisition because of its short time quantum.

Bibliography

[1] www.aaec.com/projectweb/dis.

[2] www.geek.com/procspec/procmain.htm.

[3] www.spec.org.

[4] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4), November 1988.

[5] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), May 1989.

[6] Doug Burger, Alain Kägi, and James R. Goodman. The Declining Effectiveness of Dynamic Caching for General Purpose Microprocessors. Technical Report 1261, Computer Sciences Department, University of Wisconsin, Madision, WI, January 1995.

[7] Derek Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching.* PhD thesis, Massachusetts Institute of Technology, August 1999.

[8] Compaq. *The Alpha Architecture Handbook*, October 1998.

[9] Free Software Foundation, http://www.gnu.org/manual/gzip/index.html. *Gzip User's Manual.*

[10] Antonio Gonzalez and Nigel Topham. Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers*, 48(5), February 1999.

[11] Antonio Gonzalez, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. Eliminating cache conflict misses through xor-based placement functions. *Proceedings of the 1997 International Conference on Supercomputing*, 1997.

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

[13] Eric E. Johnson and Jiheng Ha. PDATS: Lossless Address Trace Compresssion for Reducing File Size and Access Time. In *Proceedings of 1994 IEEE International Phoenix Conference on Computers and Communications*, 1994.

[14] R.E Kessler. The Alpha 21264 Microprocessor: Out-Of-Order Execution at 600 Mhz. In *Hot Chips 10*, August 1998.

[15] David B. Kirk. Process dependent static cache partitioning for real-time systems. *Real-Time Systems Symposium, 1988., Proceedings*, December 1988.

[16] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems. *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, June 1997.

[17] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, 1991.

[18] Basem Nayfeh and Yousef A Khalidi. Us patent 5584014: Apparatus and method to preserve data in a set associative memory device, December 1996.

[19] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), September 1992.

[20] Dominique Thiébaut and Harold S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4), November 1987.

[21] Dominique Thiébaut, Harold S. Stone, and Joel L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6), June 1992.