Extending the Reach of Microprocessors: Column and Curious Caching

by

Derek T. Chiou

S.B., Massachusetts Institute of Technology (1989)
S.M., Massachusetts Institute of Technology (1992)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1999

Author ....
Department of Electrical Engineering and Computer Science
August 11, 1999

Certified by .... [illegible]
Arvind
Professor of Computer Science
Thesis Supervisor

Certified by .... [illegible]
Larry Rudolph
Principal Research Scientist
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Extending the Reach of Microprocessors: Column and Curious Caching

by

Derek T. Chiou

Submitted to the Department of Electrical Engineering and Computer Science
on August 11, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis proposes column caching and curious caching, two mechanisms that enable caches to be dynamically customized, improving performance and resource control and enabling novel functionality. Column caching provides the ability to partition a cache between address regions while curious caching provides the ability for devices other than the master to insert data into the cache. Column and curious caching provide simple, controlled ways to dynamically change the traditionally static replacement policy that treats all memory and cache locations the same.

These mechanisms were conceived during the design of StarT-Voyager, a high-performance parallel system. That effort demonstrated how current memory hierarchies and bus protocols interfere with fast communication. Though the mechanisms were originally developed for communciation, they are surprisingly useful elsewhere. For example, column/curious caching can minimize pollution, reduce miss rates, improve multitasking/multithreading performance, reduce or eliminate read latencies and implement new functionalities such as bus-accessable SRAM within the cache.

This thesis motivates column and curious caching by high-performance communciation, evaluates these adaptive mechanisms for communication and other uses and proposes various implementations designed for different constraints. It demonstrates how these simple mechanisms can enable substantial performance improvements and support a wide range of additional functionality.

Thesis Supervisor: Arvind
Title: Professor of Computer Science

Thesis Supervisor: Larry Rudolph
Title: Principal Research Scientist

Acknowledgments

I would like to thank my two advisors, Arvind and Larry Rudolph, for their support and advice throughout my graduate career. I have been in Arvind's group since I was a junior, many, many years ago. Arvind provided me with the opportunity to design whole systems, something I will always be grateful for. His ability to organize ideas and clearly see the real problems was a tremendous help with this thesis.

Larry worked with me on a daily basis, challenging me to improve every aspect of the research work but also helping to rein me in when I was heading off course. His encouragement and advice were essential to finishing this thesis. Larry teaches by example and makes the quest for knowledge fun.

My reader, Srinivas Devadas, provided very helpful comments to drafts of the thesis and during our weekly meetings. His energy is truly inspiring.

Boon S. Ang has been my partner in crime since he joined our group. We have worked on every major project together, from software for Monsoon and the original *T, to designing StarT-NG and StarT-Voyager. We traveled together, worked late nights together and complained about grad school together. He read through this thesis, providing very detailed comments and suggestions. I would like him for his friendship and all of his support, both professionally and personally.

I would also like to thank Dan Rosenband and Mike Ehrlich, my colleagues who implemented StarT-Voyager. They took our high-level, imprecise and often changing specs and turned them into a real machine. Throughout the process, they demonstrated professionalism and dedication second to none. I learned a lot from both of them.

I greatly enjoyed my interactions with all my fellow graduate students, past and present, especially Alex Caro, James Hoe, Xiaowei Shen and Matteo Frigo. Alex was not only a great lab mate, but a great roommate for six years as well. James could always entertain us with his many stunts, stories, jokes and is always generous with his deep technical knowledge. I enjoyed being his officemate all these years. Xiaowei Shen not only taught me a lot about cache coherence protocols, he also taught me all I know about classical Chinese literature and patiently listened to my poor Chinese. My other officemate Matteo Frigo and I had many enjoyable discussions on a variety of topics, from algorithms to his dislike of Microsoft to the Italian political system.

In the last few months, Krste Asanovic has been very helpful in critiquing column and curious caching as well as educating me in the detailed micro-architecture of processors and caches. My summer UROPs, David Chen, Boris Zbarsky, Richard Schalck and Carl Steinbach have helped to built the infrastructure necessary to test these ideas. Peter S. Magnusson and Virtutech provided SimICS, the simulator we used to generate the memory references fed to the cache simulators.

Last, but not least, I would like to thank my family and friends who always supported me during this time. My grandparents, my mother and Brian and Jeff have always patiently supported me. I am lucky to have the best friends a person could ever ask for, especially Angelina, Ning, Henry and Lisa, Lisa, Erica and Ronald. I know that I can always count on you.

In memory of my father

Contents

Chapter 1

Introduction

This thesis proposes *column and curious caching*, two simple mechanisms that enable the effective use of caches in a changing application landscape. Modern caches (i) treat all cached data the same, applying the same replacement algorithm to all memory accesses, (ii) treat the cache monolithically, always selecting the cache-line to replace from the entire set, and (iii) only insert cache-lines in reaction to requests from its *master(s)* as opposed to its snoopers. The first two properties assume that reference patterns of all memory locations are very similar, an assumption that is quickly becoming less valid. The third property limits cache performance by making reads inherently more expensive than writes and disallowing pushes into the cache by others.

Column caching provides the ability to dynamically partition the cache between different memory regions, while curious caching provides the ability for devices other than the master to initiate the insertion of data into a cache. Mapping regions of memory that would otherwise interfere with each other to different regions of the cache and memory regions that coexist to the same regions of the cache can improve performance and reduce cache usage. Non-master insertion allows a producer to insert data into a consumer's cache, dramatically reducing fetch latency for that data.

Column and curious caching were devised to address problems encountered when designing StarT-Voyager, a parallel system built from commercial, off-the-shelf (COTS) systems. The memory hierarchies hindered fast communication, requiring complex work-arounds to get reasonable, but not optimal, performance. Column and curious caching give greater control over the memory hierarchy, enabling much more efficient data movement.

This thesis examines ways to make better use of caches and bandwidth. It motivates the mechanisms, develops them, discusses their use and evaluates them. The rest of this chapter contains a discussion of current cache architecture and its defects, a list assumptions and terminology used throughout the thesis and a summary of the rest of the thesis.

1.1 Optimizing the Common Case

The computer architect's battle mantra, "optimize the common case", has fueled the incredible increases in processor performance. The mantra has been successfully applied in virtually all areas of computer architecture, from instruction sets to branch prediction to caches and even to circuits. To this end, architects have devoted the lion's share of design and processor resources to the common case. Caches are perhaps the most glaring example currently occupying up to 75% of area[40] and 75% to 95% of the transistors on a processor die[21, 15].

As general-purpose processors grow faster, their application space grows as well. For example, general-purpose processors can now run applications such as real-time MPEG compression or high-performance routing that required specialized hardware just a year or two ago. New applications, however, sometimes exercise processor functionality in ways previously thought to be "uncommon", invalidating old assumptions and exposing suboptimal areas in the processor architectures. Thus, yesterday's assumptions create today's processors that can now run tomorrow's applications that invalidate yesterday's assumptions.

A common approach to dealing with a new common case is to add mechanisms that deals with those specific cases. For example, stream buffers were added to provide prefetching ability and buffering for stream data and spatial/temporal cache splits were provided to separate spatially accessed data from temporally accessed data in the cache. Though adding new mechanisms can be advantageous in some cases, doing so statically splits processor resources and can be inefficient if the mechanisms are not frequently used.

Better is to modify existing structures to actively support the new common cases while providing backward-compatibility. Doing so enables dynamic resource sharing yielding both the performance of dedicated mechanisms and higher resource utilization while preserving the peak performance of any single mechanism.

We take this approach, proposing modifications to caches that can improve performance over a range of metrics. Though caches are one of the linchpins of processor performance, they have reached a point of diminishing returns for many applications. Often times, caches are larger than is necessary to run a single process, opening an opportunity for optimization. Our proposed modifications provide software control of hardware, letting software provide hints that hardware executes. Thus, complex functionality is supported with simple and easy-to-implement hardware.

1.2 Caches

Caches are small, local, fast memories designed to reduce memory latency and bandwidth requirements by maintaining copies of data that will be accessed in the near future. Conceptually, a cache sits between the data requester and the main data store. To keep cycle times low while maintaining a large amount of state, caches themselves may be divided into many levels forming a hierarchy. The cache examines each memory request to determine whether it can satisfy that request from data it caches. If the cache can satisfy the request, i.e., a cache *hit*, the memory operation is not propagated to the next hierarchy level. If the cache cannot satisfy the request, i.e., a cache *miss*, a potentially modified request is passed to the next level of the memory hierarchy. The greater the number of memory requests satisfied out of the cache, the greater the reduction in average memory latency and bandwidth demands. Thus, it is important to keep the data that will be accessed in the near future in the cache.

Caches incorporate data fetched in response to a cache miss. Depending on the cache architecture, there may be a choice of cached data to replace. The *cache replacement algorithm* decides what data to replace and is usually some variant of the least-recently-used (LRU) algorithm. LRU assumes that the recent past predicts the near future making least-recently-used data least likely to be used in the near future and is thus likely to be the best candidate for replacement. The only other common replacement algorithm is random, where the data to replace is selected at random.

Traditionally, caches are transparent to software and thus are managed completely by hardware. A cached memory address can reside within any *cache-line* that is a member of its assigned *set*. The set assigned to an address is determined by extracting specific bits from the address.

1.3 Uniform, Hardware Replacement Algorithms

Caches were developed in an earlier era of computing when the ratio of processor speed to memory speed was much smaller than it is today. Because memory at that time was fairly similar in speed to processors and, therefore, the cache had to respond quickly to have a real benefit. In addition, cache sizes were quite small. Because of their small size, early caches could not contain the entire working set of many programs, making frequent replacements inevitable.

Caches were also designed to be transparent to software, that is, they were designed to improve performance without software involvement. The reason for transparency is simple; when caches first appeared in computer systems, neither compilers nor software were sufficiently sophisticated to explicitly manage fast storages that were distinct from normal memory. The B registers and T memories, fast explicitly-managed memories found in the Cray I[61] and subsequent Cray architectures, are examples of fast, explicit memories that were generally too difficult for compilers to use.

Because of the need for cache speed and transparency, cache replacement protocols were implemented in simple hardware and were uniform across the entire address space. With the exception of uncached space, mostly used for I/O, and the frequent separation of instruction and data caches, all cached memory was treated the same way. Because caches had to be small, and hit rates were relatively low, a uniform policy could perform reasonably well compared to an optimal scheme. Cache capacity issues would force to be frequently reloaded regardless of the replacement algorithm. Another side effect of a small cache is that accurate cache studies could be made by looking at individual, rather than groups of applications that might be scheduled together, since each time applications were swapped, the cache had to be refilled with the application's data.

A perfect cache is an oracle; it decides what to cache based on future reference patterns. Because the replacement algorithm is implemented in hardware, it only has past references to assist in predicting the future[1]. Often times, however, the past produces a good enough prediction of the future. Many applications have fairly regular memory reference patterns that exhibit "local" behavior, that is, if an address is referenced in the near past, it and/or its neighbors would probably be accessed in the near future. Cache designers took advantage of this observation, selecting least-recently-used algorithms as the *de facto* standard of cache replacement protocols. Coupled with the fact that caches and miss penalties were small, the LRU replacement policy was good enough.

1.4 The Times They Are A Chang'in

Times, however, have changed. Processor cycle times are getting faster much more quickly than memory, making the cache miss penalty high and this penalty is getting higher. Current processors take around 100 cycles to access memory[20, 46], an order of magnitude more than a generation ago. This trend is expected to continue. Processor clock rates are now five or six times that of the bus clock rate.

In order to compensate for this ever increasing miss penalty, architects try to

[1]Speculation and multiple outstanding operations could permit a processor to "lookahead" somewhat in the memory reference stream. Even with perfect control prediction, however, memory addresses would need to be computed and therefore could not be predicted. If there was good data prediction, caching would not be necessary.

decrease the number of misses by increasing cache size and adding extra levels of caches. The HP PA-8500 processor contains 1.0MB of L1 data cache (4-way set associative) and 0.5MB of L1 instruction cache (also 4-way set associative), both on-chip. The AMD K6-3[2] contains a 256 KB L2 (4-way set associative) cache and 64KB L1 cache, both on-chip. As caches grow in size, their percentage of die resources grows as well. Current processor are dominated by caches that take 80% or more of the total die space. These numbers are still increasing. In addition to on-chip caches, processors such as the Sun UltraSPARC[69], Compaq Alpha[20] and SGI R10000[54], support off-chip caches of 4 MB or more. As time goes on, caches are expected to get larger and deeper.

Though caches are larger, however, application memory footprint size has not necessary grown at the same rate. Rather than containing a fraction of a single programs working set, current caches are large enough to contain multiple application working sets *simultaneously*. Caches, however, generally do not take advantage of this fact. The standard LRU replacement policy throws away data that has not been accessed recently (such as live data of a swapped out application) over data that has (such the current application's recently accessed but dead data). As computers become faster and more capable and the number of jobs per processor increases, the chance that live data gets thrown away, even though there is sufficient space, increases as well.

Applications and systems have also changed substantially. Certain computing paradigms, such as object-oriented codes, access memory less regularly than applications coded in more traditional paradigms. In addition, applications do not use memory as regularly or for the same uses as they once did due to a number of factors including much more communication between different devices, new streaming applications such as decompression, video and graphics, larger register sets and better compilers that eliminate redundant memory requests and so on. With these less-regular memory references, standard LRU algorithms perform sub-optimally. To further aggravate the problem, the bigger and deeper caches incur more latency with each additional layer, making misses more costly.

In addition, compilers and users have become significantly more sophisticated than when caches were first designed. Users have been able to do the same by hand. Advanced parallelizing compilers are able to automatically partition programs so that they run well on distributed memory machines. 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 information could potentially make a replacement algorithm much more accurate since it can give a much more accurate estimate of future accesses.

1.5 What Do We Want?

Processor architects want to minimize perceived memory access times, especially to data whose latency cannot be masked, to improve performance. The standard approach is to throw resources at the problem resulting in very large caches. This brute force approach is necessary for some applications but is overkill for many applications. Because the caches are managed in a very simple and straightforward fashion, they are often vastly under-utilized[15].

Despite their problems, however, standard caches still work very well for many applications. Rather than adding additional mechanisms, our approach is to modify caches to allow software (and/or additional smart hardware) to tune how caches work when such tuning improves performance. Such modified caches must be backward-compatible, enabling others to assist in controlling the cache, but also able to revert back to normal cache behavior.

What sort of cache management should be provided and how should it be implemented? Providing fast, explicitly addressed SRAMs instead of caches[2] is a solution, but not general due to its reliance on as yet undeveloped software. The automatic management of caches as well as the ability to use the same name (address) to address fast cache memory and the slower DRAM main store make caches almost essential for general compiler-generated codes or codes that have unpredictable memory access

[2]Obviously, you can provide both at a cost.

patterns. Explicit SRAMs which reside in an orthogonal address space as in DSPs, require copying thus requiring changes to legacy code. Very few users are willing to spend the time to hand tune code to run better with explicitly addressed SRAMs; in fact, it is impossible to do so in many cases because of the dynamic nature of the programs.

So, we need caches. What kind of software cache management is appropriate? Most current caches do not allow software management of caches[38, 28]. Some current microprocessors have cache management instructions which can flush or clean a given cache-line, prefetch a line or zero out a given line[51, 69]. Others permit locking down cache-lines within the cache, essentially removing those cache-lines as candidates to be replaced[22, 23]. All of these operations are on a cache-line basis, requiring a separate operation per cache-line. In addition, the exact address must be specified in the instruction which requires either predicting or tracking cache-line addresses. The first is a difficult task, akin in complexity to mapping memory to a separate address space, the latter incurs overheads. Cache-line locking does not give control over which cache-line is replaced, but rather which cache-lines *cannot* be replaced assuming they are in the cache when the operation is executed (a subtle but significant difference). Though cache management operations and cache-line locking are extremely useful, they require detailed knowledge and do not cover all cases.

This thesis describes an efficient, easy-to-implement, safe and fully backward-compatible set of cache management mechanisms which give software dynamic control over the cache. The mechanisms will still allow the cache to operate as a standard cache and are guaranteed to be safe; software cannot violate memory semantics. We developed these mechanisms during the design of StarT-Voyager, a high-performance parallel machine built from stock systems. Some of the deficiencies of those stock systems lead us to these mechanisms.

1.6 Contributions

The contributions of this thesis are as follows.

- Design and evaluation of the Column Caching cache partitioning mechanism that opens opportunities for performance enhancement and resource control by allowing software to dynamic split the cache between different memory regions. Various designs compatible with a wide range of cache architectures are presented.

- Design and evaluation of the Curious Caching mechanism that opens opportunities for performance enhancement and resource conservation by allowing data to be safely inserted by devices other than the cache's master. Curious caching makes memory hierarchies communication-friendly, eliminating the asymmetry between transmitting and receiving data and can emulate bus memory, even for other bus devices, within a cache.

- Design and implementation of StarT-Voyager, a high-performance message passing/shared memory machine. StarT-Voyager provides hardware and firmware flexibility that allows the efficient implementation of almost any conceivable communication mechanism and provides a test-bed for communication mechanism research.

1.7 Assumptions and Terminology

Many assumptions throughout this thesis provide a base for discussion, but do not limit the described mechanisms and techniques.

Though we discuss caches in the context of general-purpose microprocessors, the mechanisms are applicable to caches that do not have processors as masters (disk drives, digital-signal processors, etc).

We use "cache-line" to mean the physical entry in the cache where a cache-line of data is stored. A "higher cache" is a cache closer to the processor core while a "lower cache" is a cache further from the processor core. For example, the L1 is a higher cache than the L2 cache.

We use "bus transaction" to mean the entire bus transaction including the bus operation, address, etc. "Bus operation" is the specific operation (read, write, flush, sync) of the bus transaction.

We assume an least-recently-used (LRU) replacement algorithm unless otherwise noted. Variants of the LRU algorithm are used in almost all commercial general-purpose microprocessors.

Unless otherwise stated, we assume an invalidation-based MESI cache coherence protocol. MESI stands for the four possible states a cache-line can be in: MODIFIED (cached data is most up-to-date copy), EXCLUSIVE (data is unmodified but is not cached by other caches that not direct ancestors or descendants), SHARED (data is unmodified but other caches may have copies) and INVALID (cache-line contains invalid data).

We use `read-exclusive` to mean a read between memory hierarchy levels indicating that the reader wants an exclusive copy of the data, presumably to allow it to write. `read` means a read between memory hierarchy levels indicating the reader would be satisfied with a shared copy, implying others can have shared copies of that data as well.

Because of our experience with the PowerPC architecture, we often use its terminology and assume its functionality. Specifically, the `dclaim` bus operation is issued to get write permission for data cached in a shared state. The `flush` instruction (that can also appear on the bus as a `flush` bus operation) flushes an address out of the cache, performing a `pushout` bus operation if the cached data is modified. The `clean` instruction forces a writeback of the specified address if the cached data is in modified state, but keeps an exclusive copy in that case.

1.8 Rest of Thesis

In the next chapter, we describe the StarT project from the 88110MP, to StarT-NG and StarT-Voyager. These machines were all designed to maximize performance within the given constraints. StarT-Voyager built around a stock microprocessor is used to illustrate the difficulties that standard memory hierarchies impose on high-performance communication.

In Chapter 3, we describe *column caching*, a mechanism to partition a cache between different address ranges. Column caching enables an application and/or an operating system to explicit manage of cache allocation, potentially improving performance, reducing memory footprint and providing additional functionality.

In Chapter 4, we describe *curious caching*, a mechanism that enables a cache to insert data it snoops on the bus. This mechanism used properly can dramatically reduce memory latencies for shared data as well as dramatically reduce bandwidth requirements. This mechanism works well with column caching to provide these benefits while containing pollution and providing new functionalities within cache.

In Chapter 5, we evaluate StarT-Voyager, column and curious caching, providing preliminary performance numbers. We also describe how StarT-Voyager could efficiently emulate column and curious caching and how column and curious caching could improve StarT-Voyager performance and simply its design.

In Chapter 6, we conclude the thesis with some parting thoughts and observations. We also give some future work and directions that should be explored.

Chapter 2

Communication and Caches

The column and curious caching mechanisms emerged from our exasperation with the design defects of modern processor memory hierarchies that we discovered during the design of StarT-Voyager, a high-performance parallel system based on commercial-off-the-shelf (COTS) processors. This chapter describes those defects, how they can impact high-performance communication that operate over the memory hierarchies and how they can be attacked.

Though this discussion assumes message passing communication, the memory hierarchy issues apply to a wide range of memory patterns that do not perform well in conventional caches. For example, emulating message passing on top of shared memory hardware has precisely the same problems that we encountered for StarT-Voyager's message passing. Likewise, rapid context switching between several threads, even if each thread's memory usage works well within a cache, can also perform sub-optimally.

In this chapter we briefly describe the StarT project and its design challenges, and give an overview of the final design that was implemented. Specifically we discuss the difficulties of implementing high-performance communication on top of off-the-shelf processors.

2.1 The StarT Project

StarT was originally a collaborative effort between MIT and Motorola to build a high-performance parallel system from off-the-shelf or slightly modified components. It was our intention to build a research prototype that would be one or two generations ahead of a commercial product. Thus, it was deemed reasonable to make small changes to a microprocessor to achieve very high communication performance. These changes, if successful, could be propagated into the mainstream processors. The significant changes would be concentrated within the processor itself; the rest of the hardware and the operating system would only have modifications necessary to use the modified processor.

The first machine in the StarT family, *T[59], was based on a modified Motorola MC88110 (88110MP) and the Arctic[13] network routing chip. The 88110MP augments the MC88110 core with an internal network interface implemented as a functional unit. A diagram of an 88110MP processor is shown in Figure 2-1. The network interface is accessed via register operations, and is thus very fast. It utilizes the MC88110's dual instruction issue capability and wide internal datapath allowing up to 256 bits to be moved into the network buffer each cycle. A simple message could be formatted and launched in 6 cycles. Reading an incoming message is also done with simple register operations, albeit at a lower bandwidth of 128 bits/cycle.

Due to many issues ranging from the resources required to add the network interface functional unit to the 88110 processor core, uncertainty surrounding the operating system and the new alliance between Motorola and IBM to design and build the PowerPC processors, we were strongly encouraged to change processors. We decided to design a new machine, StarT-NG, around stock SMPs based on the PowerPC 620 (Figure 2-2).

Though we could not modify the processor, the 620 supported a special *co-processor* interface that allowed the back-side L2 cache interface to simultaneously support a memory-mapped device. A region of the physical address space was statically mapped to a special region of the L2 cache interface. It was possible to have

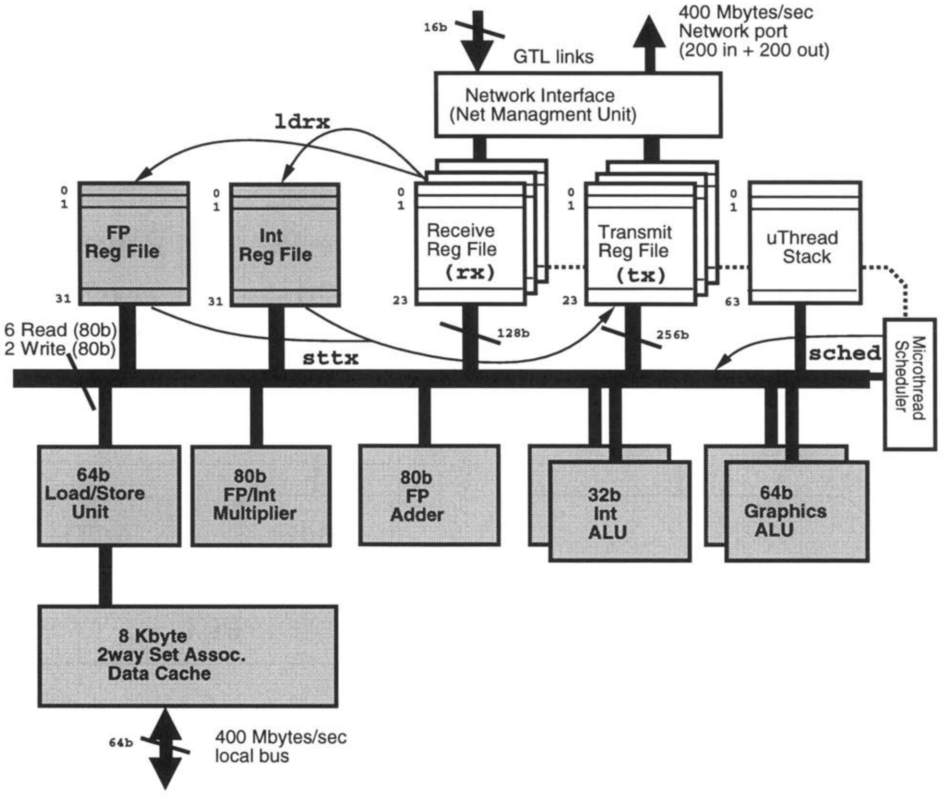


Figure 2-1: An 88110MP node (taken from [59])

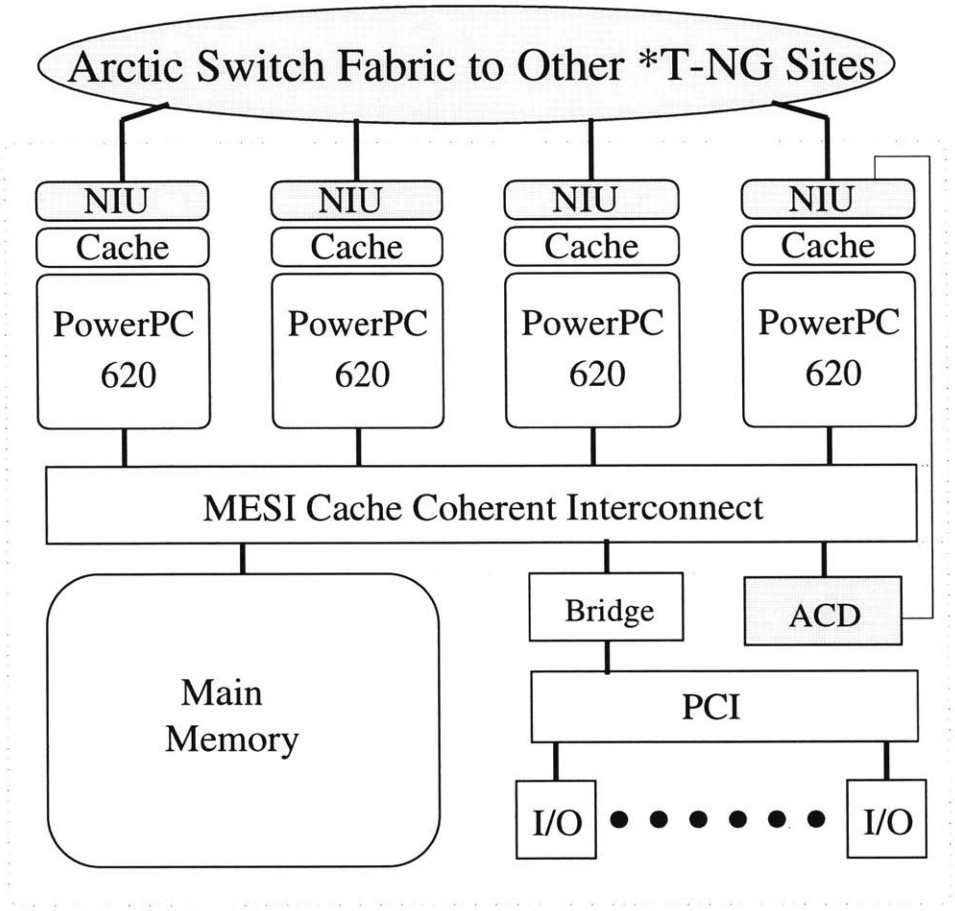


Figure 2-2: A StarT-NG site: the white areas comprise the base SMP. The grey areas are our additions.

a memory-mapped device, such as a network interface, share the back-side L2 interface with the L2 address tag and data RAMs. The memory-mapped device could be cached in the L1 cache but not in the L2 cache.

The 620 L2 interface was quite fast with a 128-bit data path and potentially running at 1/2 the processor clock rate. Though the network interface was moved off of the processor and thus incurred additional latency, the closeness of the interface and the available bandwidth could still produce extremely aggressive performance. A network interface supporting message passing was designed. Shared memory support that relied on one processor on each SMP node to coordinate the moving of data and act as a coherence protocol engine was also provided. A complete overview is provided elsewhere[19].

In October 1995, we were informed that the PowerPC 620 was indefinitely delayed, forcing us to redesign StarT once more. Rather than relying on unavailable technology, we decided to design around an existing processor, its systems and a stock operating system. We selected the PowerPC 604e processor since it was Motorola's highest performance processor at that time. Soon after, our relationship with Motorola dissolved. As we had disagreed with Motorola on the aggressiveness of the machine, we scrapped the joint design in favor of a new, higher performance design based on the 604e. This machine was named StarT-Voyager[6, 8, 7]. Because we could not modify the 604e or its bus, we were forced to implement all communication over the standard memory bus and thus had to design around around the given memory model, cache semantics and memory bus. Communication had to be encoded within standard memory operations that would be subjected to the associated cache and bus protocols. Some communication mechanisms, such as shared memory, do fit better into the cache and bus protocols provided than other communication mechanisms, such as message passing, that do not necessary fit well within standard cache and bus protocols.

During this design process[1], we found that standard processors and their memory

[1]The author was responsible for *T's run-time system and Boon S. Ang and the author were principal architects of StarT-NG and the architects of StarT-Voyager.

- Destruction of one-to-one mapping between memory operations and bus operations.
- Uncached operation performance degradation relative to cached operation performance.
- Multiple word cache-lines one result of which are bus operations for cache-line transfers.
- Coherency protocols to maintain a coherent view of memory given multiple devices accessing memory one result of which is specialized coherency support in bus protocols.
- Weak memory models requiring memory barriers to enforce ordering.

Figure 2-3: Cache Characteristics that Impact High-Performance Communication

hierarchies are not well suited for high-speed communication. Instead, they are optimized for memory operations with significant locality. Thus, a significant amount of the StarT-Voyager design effort went into developing an interface between the processor and it caches and the network interface made the best out of the memory hierarchy.

2.2 Providing High-Performance Communication to COTS Microprocessors

In this section, we detail how the memory hierarchy, memory models and bus protocols get in the way of high-performance communication. In Figure 2-3 we summarize the specific cache characteristics that affect communication discussed in the next section[34]. The following section describes how a high-performance message passing mechanism might be implemented on top of such caches. We finish with a description of how StarT-Voyager attacks the problems.

2.2.1 The Evolution of Caches

The very first computers did not have a memory hierarchy. The memory system itself was quite simple. There was no cache. All memory requests were serviced by the memory itself over the bus in the order the memory operations appear within the instruction stream. Because each instruction would generally take several cycles to finish, demands on the memory were fairly light, eliminating the need to heavily optimize memory.

Processors, however, have become very fast. Though memories have improved in speed, they have not kept up with processors. Standard processors depend on fast access to memory, or at least a perceived fast access to memory. Caches were introduced to reduce memory latency and potentially reduce required bandwidth. The replacement algorithm and miss penalty together play a key role in determining the effectiveness of the cache. Originally, cache replacement algorithms were not so critical since cache miss latencies were low, masking replacement errors. The miss penalties seen today, however, are so large that even small miss ratios can significantly impact performance.

The introduction of caches, however, has had a profound effect on the memory system. For example, with caches bus transactions do not correspond one-to-one with memory operations. Caches intercept and satisfy memory operations, often completely avoiding the bus. Caches can also cause more than one bus transaction to occur in response to a single memory operation such as when a modified cache-line is written back to make room for new data.

Another impact of caching is that uncached operations have become relatively slower. Most processor pipelines and bus interface units, including those found in the 604, are not optimized for uncached transfers since they are generally used only for I/O operations or control register operations that are not speed critical. Most buses issue an address for each word of data being read or written from/to uncached space, cutting into the critical address bus bandwidth. Generally, burst transfers are not allowed from or to uncached space. Though store-gathering buffers (to aggregate uncached stores) are becoming common, no such structure exists for reads since each read may return different data or signal different events if they occur at different times. Care must be taken even with store gathering since bus devices that depend on uncached stores must be able to handle the burst data.

In order to both exploit spatial locality and reduce tag overhead, cache-lines became larger than a single word. Doing so, however, requires either additional bookkeeping to specify the valid words within a cache-line or that data is moved from/to lower cache levels in full cache-line blocks rather than single words. Virtually all modern caches take the latter approach. Of course, moving cache-lines rather than single words generally improves bus efficiency.

In order to further improve performance, bus protocols were tuned for caches. Special cache-line bus operations were introduced and highly optimized. Bus operations of sizes other than cache-line size were not as optimized because they are used less frequently.

In order to improve overall system performance and better utilize memory, additional processors were introduced on the same memory bus. Since multiple processors could be writing the same memory location at the same time, the original assumption that a value within storage would not change between two memory accesses was no longer true[2]. Thus, a cached copy of a value could become "stale". Protocols designed to keep caches coherent were developed to address this problem. Though many protocols have been proposed, the most common is the MESI protocol, for MODIFIED, EXCLUSIVE, SHARED, and INVALID, the four possible states of a cache-line[3]. The MESI protocol is an invalidation-based protocol, that is, it optimized for single-writer and multiple-reader situations. The protocol ensures that when a processor is writing cached data, no other processors can be caching that data. Multiple readers, however, can be caching the same data simultaneously.

Bus-based coherency protocols maintain a coherent view of memory by "snooping"

[2]Other bus devices, such as I/O devices, can also read and write storage, causing the same coherency problem.

[3]A real coherence protocol would require either several intermediate states or support for those cache-lines in transient states as well.

the bus operations of other bus devices. Snooping protocols require each cache to watch all bus transactions from other caches and react to the snooped transactions in a way that maintains coherence. For example, if a cache caches data in the MODIFIED state and snoops a read operation from another cache, it is required to return the modified data to the bus. The exact response, of course, are determined by the coherence protocol. The bus protocol, as opposed to the coherence protocol, may retry the snooped bus operation and writeback the data to memory or may return the data directly to the requesting cache while writing the data back to memory.

Obviously, bus protocols were modified to better support snoopy cache coherency protocols. For example, most bus protocols support multiple distinct read operations. The PowerPC 60X bus protocol includes a simple `read`, a `read-with-intent-to-modify` (which we will call `read-exclusive` throughout this thesis), a `read-atomic`, etc. The distinct bus operations notify the coherence protocols of the intended use of the requested data, giving the coherence protocol the information necessary to react appropriately. Thus, as cache coherency protocols became more complex, bus protocols followed.

As performance became more and more important, weak memory models were developed. In a weak memory model, memory operations to different addresses can proceed out-of-order. For example, two reads to different locations could appear on the bus in the opposite order that they appear in the instruction stream. Weak memory models are becoming more more prevalent within superscalar architectures, since they allow operations to proceed at their own pace, rather than being limited by their instruction order. The arguments necessary to generate the address of a particular read could take a long time to obtain because they depend on previous reads that have not yet been returned by the cache while a read that appears later in the instruction stream may have all of its arguments available immediately.

While weak memory models can dramatically improve performance, they eliminate the possibility of using a single standard memory operation to indicate the completion of previous memory events. Some sort of "memory barrier" that ensures previous memory operations have completed before the barrier and subsequent memory operations are allowed to complete becomes necessary. Memory barriers, however, are quite expensive. Most architectures implement them by forcing all outstanding operations to complete before the memory barrier is allowed to complete and instructions further in the instruction stream are allowed to issue. For example, a `sync` instruction takes about 20 processor cycles in a 166MHz PowerPC 604e. It is difficult to make memory barriers fast because each memory instruction may have a complex set of dependencies.

This memory system evolution dramatically improve memory performance as long as memory access patterns follow the optimizations' assumptions. The memory hierarchy evolution has specialized the memory system for cache-line-sized coherent, weak memory model cache accesses, neglecting other forms of memory accesses. If memory usage does not follow the assumptions, however, performance can suffer, sometimes greatly. As time goes on, the uses of memory and the fast memory bus have become more and more diverse and tend to fit less and less into the simple cached memory model assumed and optimized by modern memory systems.

2.2.2 Message Passing with Caching

This section describes a naïve message passing mechanism and how it might better deal with COTS caches and buses as well as how caches and buses get in the way of message passing paradigms. Consider a connection-based stream message passing interface where the first step to communicating is to establish a link between a single producer and a single consumer. Then, the producer generates data which is somehow transfered to the consumer for consumption. The consumer should see the data in the order that it was produced.

Given a standard memory system and assuming that control information such as the address and bus operation are kept separate from the message data, the producer would ideally write message data to a single address that identifies the stream input and the receiver reads from another address that identifies the stream output. Using a single address allows the producer/consumer to concisely specify the exact stream it is accessing. Assuming uncached writes to produce and uncached reads to consume, an uncached bus write to a stream-out address can indicate that a message has been launched and an uncached bus read to a stream-in address can indicate that a message has been received. Some handshaking is necessary for flow-control to ensure that buffers do not overflow. If uncached reads and writes to the same location can get out of order, memory barriers are necessary to keep data in order.

Unfortunately, it is impossible to implement such a scheme in an stock processor if maximum bandwidth is desired. Uncached reads and writes are slow and use the bus inefficiently. Caching the interface only makes things worse. Caches are designed to move cache-line sized blocks of data, not single words. Bandwidth would be wasted moving cache-lines of data when only single words need to be moved. In addition, transmits and receives would have to flush out data after each composition/consumption to avoid overwriting/reading the same data.

Rather than using a single address to specify a transmit or receive queue, a range of cached addresses can be used to create a circular buffer. Additional computation needs to be performed to compute each address and deal with wrap-around in the circular queue, though superscalar execution eliminates any significant performance impact. The buffers should be large to reduce signaling overheads and provide elasticity between the producer and the consumer rates.

Caches remove the one-to-one correlation between memory operations and bus transactions and, thus, there must be a way to signal that a message has been composed or consumed as well as a way to provide flow-control to avoid buffer overflow. One way to provide both signaling and flow-control is to specify bits within a block of data that indicates a message has been produced or consumed[19]. Polling on those bits can determine when it is safe to transmit or receive additional messages. This technique can eliminate the need for the memory barrier if there is such a signal in each word of data but does not eliminate the memory barrier if there is one signal for a group of words. It does, however, reduce the amount of state that can be moved at one time and may require undesirable overheads.

Another possible way to signal completion is to have the producer/consumer update a separate location indicating that data has been produced/consumed. That separate location could be a counter or a pointer. Either scheme allows the aggregation of signals. For weak memory models, a memory barrier is required between the production of data and the updating of a signaling location in order to ensure that the signaling write occurs after the data is produced. We assume uncached producer and consumer pointers to indicate message composition and consumption signaling.

Though cached circular buffers avoid the problems of the original single address specification of transmit and receive queues, they have their own set of problems. The sequential access of message buffers does not match well with standard cache LRU replacement algorithms. The message data that is streaming into or out of the cache tends to replace data that will actually be used again, *polluting* the cache with dead message data.

One way to minimize cache pollution is to make the message buffers small. Doing so, however, reduces elasticity between the producer and consumer, increasing the possibility of unintended throttling due to limited resources. It also increases overhead by reducing the ability for pointers to be amortized. Additional bus transactions for the pointer reads and writes as well as the synchronization operations become required; all are expensive operations.

Another way to minimize pollution is to insert cache-line `flush` instructions right after data is produced or consumed. Inserting `flush`s however, increases execution overhead, requires bookkeeping and tends to be expensive; they are almost as costly as a memory barrier in the PowerPC 604e. In addition, `flush` instructions must either be proceeded by a memory barrier or semantically include a memory barrier to ensure that data is not pushed out prematurely, adding additional overheads. Barriers can be aggregated then data is not flushed immediately, bringing back the pollution problem. Rather than having the processor explicitly flush data from the cache, another bus device can issue bus operations that force the cache to invalidate or write-back the desired data. The bus device starts issuing the flushes once it sees a pointer update indicating completion. Having another bus device perform the flushes does eliminate the overhead of processor-initiated `flush` instructions but significantly increases latency since a signal to the bus device must travel down the memory hierarchy and the externally launched `flush` back up again before the data is written back.

There is tension between coherence protocols and controlling cache pollution. If data is completely flushed from the cache to launch a message, after wrap-around a future message will require a cache reload. If data is just *cleaned*, that is the data is written back but an exclusive copy is kept, the bus transaction is not necessary but the cache becomes polluted.

Another problem with cached buffers is the inherent latency imposed by cache hierarchies. Moving data through multiple levels on the way to and from the bus is costly. In order to get produced data to the network interface quickly, a producer must explicitly flush produced data increasing processor overhead. The alternative of having an external bus device issue bus operations that force flushes increases latency. Thus, a producer must balance between latency and overhead; it cannot minimize both.

Such a tradeoff is not as available to a consumer who must issue a read to pull the message into its cache, requiring a round-trip to the memory bus. Generally, the consumer must wait for the data to return since instructions further in the instruction stream often depend on the read data. The consumer could prefetch the message, but that would require (i) it knows a message has arrived and (ii) the consumer does something else while waiting for the prefetch to return to mask the latency. Such a switch might not always be cost effective since it has its own overheads. On the other hand, if prefetching is done too early, the cache could become polluted.

Yet another problem is the sharing of message queues between different processes or different threads. In general, process/thread switches can occur at any time. Since messages take more than one instruction to transmit (check flow-control, compose and launch,) or receive (check for message arrival, receive, indicate consumption), locks must be taken to ensure sequential access to a single queue. Instead, it would be preferable to have an independent queue per process/thread. A large number of queues, however, requires a large amount of buffer space. Memory can be used to provide a large region of buffer space.

DRAM, however, is much slower than SRAM. Generally, buses support much higher bandwidth than the memory can provide. Also, buffering in DRAM will generally create more bus transactions than buffering in dedicated message buffers because message data is moved to and from the DRAM. For example, when a message is received from the network, it is written to the DRAM buffer. The receiver then reads the message from DRAM for a total of two bus transactions and two accesses to DRAM as opposed to one bus transaction if a dedicated message buffer was provided. Special message buffers, however, would probably be much smaller than memory due to its higher cost and special-purpose nature.

Figure 2-4 summarizes the problems facing message passing on stock hardware and the other issues with which it conflicts. There are no perfect solutions for implementing stream-based message passing within a standard memory hierarchy. We found the same to be true for all forms of message passing we looked at; all had unavoidable overheads.

2.3 StarT-Voyager Solutions

StarT-Voyager attacks these issues, producing excellent performance in the context of COTS processors and caches. StarT-Voyager is built from unmodified IBM PowerPC 604e-based SMPs, each with a processor card in one processor slot and the StarT-Voyager network interface unit (NIU) in a second processor slot as shown in Figure 2.3. The processor card consists of a 166MHz 604e, which we refer to as the application processor or aP, and a 512KB in-line L2 cache. The NIU consists of custom hardware, some SRAM, and a 604 microprocessor used as an embedded service processor (sP) to execute firmware.

Buffering space is provided by two banks of dual-ported SRAMs, each attached to one of the 604 data buses on one side and to a central bus, called the IBus on the other side. Multiple message passing queues are available to reduce or eliminate the atomicity problem. SRAM buffers are provided on the network interface to address the DRAM performance problem, though DRAM can be used as buffer if more queues

Pollution Message buffers can pollute caches since their access patterns do not match standard replacement algorithms.

Coherency Standard coherency protocols require a writer to get permission and (sometimes) have a current copy of the cache-line of data to write. Either a producer keeps a copy causing pollution or must issue reads to get permission/cache-lines. Conflicts with pollution.

Explicit Cache Management Generally, software-initiated cache management instructions are costly. Preferably, the data is written-back/flushed automatically after composition/consumption. Conflicts with pollution, latency.

Cache Latency Cache hierarchies are deep, creating latency. Conflicts with explicit cache management.

Synchronization The memory barriers implemented in modern microprocessors are expensive. Solutions require either modifications to the processor core, limits on the scope of the synchronization operation or both. Conflicts with pollution.

Receive/Transmit Asymmetry Transmitting a message is more efficient than receiving a message. Transmits are pushes, not requiring any information other than aggregatable flow-control to execute while receives are pulls and cannot continue until the received data is available. There are no general solutions possible with COTS hardware.

Atomicity Sharing of a single queue between different threads of control is expensive. Separate queues are desired but are potentially expensive to implement, especially if buffering space is tight. Conflicts with DRAM performance.

DRAM Performance DRAM is slow in terms of both latency and bandwidth (though bandwidth limitations are being fixed with SDRAM). But, DRAM is a large resource. Conflicts with atomicity and performance.

Figure 2-4: The problems faced by high-performance communication implemented through standard caches.

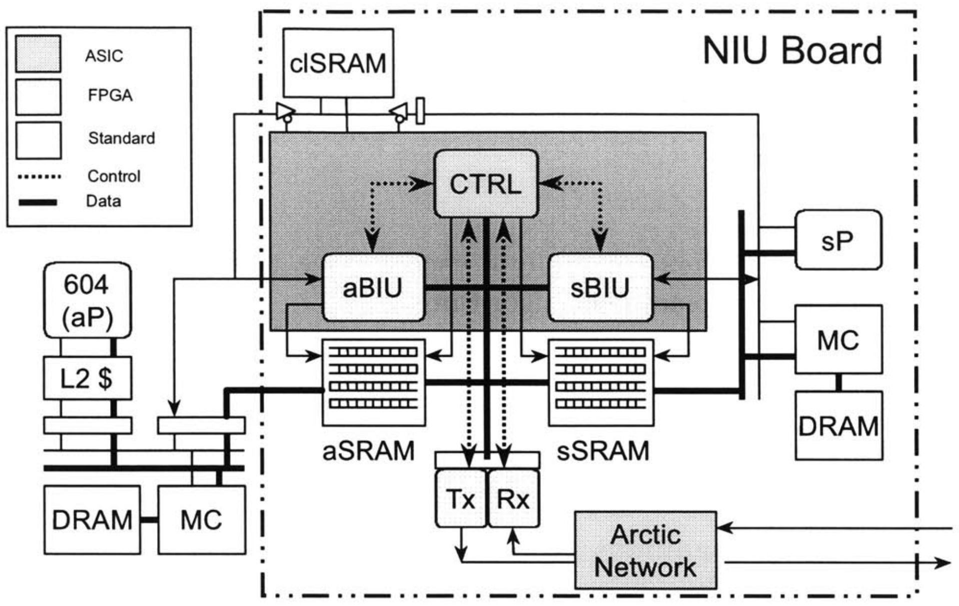


Figure 2-5: A StarT-Voyager node. The NIU contains 3 FPGAs, 1 LPGA, two dual-ported banks of SRAM (labeled aSRAM and sSRAM), an SRAM attached to the aP address bus, and an embedded processor. The NIU connects to the system bus of an SMP as well as a high performance interconnection network.

than are supported in the SRAM are needed. Unfortunately, there is nothing that we could do about cache latency and our solutions to cache pollution are simply balancing acts, allowing the user to trade one negative for another.

StarT-Voyager's message passing mechanisms that address the problems mentioned earlier in this chapter are now described. A more complete description of StarT-Voyager's message passing can be found elsewhere[7, 5]. Information about shared memory and its performance can be found in the Appendix of this thesis.

2.3.1 Basic

Basic messages are implemented in a cached (though they could be uncached if desired), circular buffer. Buffer sizes are configurable, allowing pollution to be traded for coherency, explicit cache management and synchronization. Up to 88 bytes (in 8 byte increments) of data can be sent in each message. Messages are launched by uncached pointer updates and are received by first reading an uncached pointer that points to the first free location in the receive buffer then reading the received data. By comparing to the previous pointer, a consumer can detect if messages have arrived. Pointers are aggregate-able, allowing amortization of memory barriers and uncached operations. Because several operations are necessary to produce/consume a message, there is a atomicity problem.

Hardware support to "reclaim" cached message buffer data is provided by the network interface. When production/consumption pointers are updated, the network interface issues the necessary `clean` or `flush` operations to clean/flush the appropriate address regions from the cache. This support eliminates the need for the processor to issue cache management instructions to pushout transmit messages or to flush receive messages but increases latency. This support can be turned off on a per queue basis.

2.3.2 Express

Express messages are uncached thereby eliminating the pollution problem. Writes to a single address indicate a specific queue and destination. Reads to a single address indicate a specific queue. Transmit compose/launch are combined into a single 32b uncached write. The lower address bits are used as a logical destination as well as extra 5 bits of data if necessary. The data from the write is the bulk of the transmitted data.

Receives from a specific queue are performed by reading a 64b value from a specific address. The 64b read will return either a valid message (containing the 37b of data plus some return destination information) or the contents of a set location, the "miss location". The 64b read can be performed either as a single 64b read or 2 32b reads.

The express message mechanism emulates hardware FIFOs. They eliminate pollution, pushout costs and cache latency problems because they are completely uncached. The uncached write/reads and FIFO emulation also eliminate the need to signal and synchronize when a message is composed or consumed though transmits still need to read counters for flow-control. Express messages, however, are limited in bandwidth. There is still a receive/transmit asymmetry but that is unavoidable in standard memory hierarchies. In addition, Express messages are safer for multithreaded codes, since each transmit/receive can be atomic, though the transmit flow-control still needs to be finessed.

Express messages have excellent latency, very poor bandwidth and good processor overhead. There is a potential impact on the TLB, since a wide range of addresses is used when stealing address bits for destination/data, but that impact can be limited by limiting the number of address bits used.

2.3.3 Tagon

The Tagon mechanism allows the Express and Basic mechanisms to specify a pointer to additional data stored within the SRAM buffers. The additional data area can be cached, providing burst transfer of blocks of data to Express messages. The Tagon mechanism is especially useful for multicasting to multiple locations (by writing the data once and sending multiple messages with that data) and for multithreaded sharing of queues (by giving each thread a separate region to compose Tagon data.)

On the receive side, Express messages with Tagon are split into two queues, the standard Express message and the Tagon part. The user must read the Tagon part separately and explicitly deallocate it by updating an uncached consumer pointer. For Basic messages, the received messages look as if they were composed as normal Basic messages.

Tagon buffer space size is configurable, allowing pollution to be controlled. Since Tagon data is launched when its corresponding Basic or Express message is launched, there is no additional issues with pointer aggregation. Hardware support can be provided to support a reclaim ability for Tagon data. When used aggressively, Tagon has the same pollution verses coherency, explicit cache management and synchronization issues as Basic messages. Atomicity can be finessed by dividing buffer space, but then increases the buffer space required.

2.4 Summary

To reiterate, the problems we faced are not inherently message passing problems, but can be found in a variety of memory reference patterns such as streaming data and rapid context switching between threads. Even aggressive shared memory has some of the same problems such as inherent read latencies (see Appendix for detail on StarT-Voyager's shared memory support).

Though StarT-Voyager attacked all the addressable issues given the PowerPC 604e and its caches, there is still considerable room for improvement. For example, containing cache pollution only comes by increasing other overheads. In addition, there is still the issue of the inherent latency to receive messages.

There are several possible solutions to these unresolved issues. For example, a fast memory barrier and fast `flush/clean` implementations would solve many of the remaining issues. The mechanisms described in the next two chapters, column and curious caching, however, are much more general, solving most of the problems discussed in this chapter while also having applicability across a wide range of other applications. In addition, they provide additional functionality not currently available in standard microprocessors and caches.

Chapter 3

Column Caching

A column cache is a cache modified so that the replacement algorithm can be controlled, by software, hardware, or both. The column cache replacement algorithm is dependent on parameters such as the address and memory operation that caused the replacement. Performance and cache resource efficiency can be improved by controlling the regions of memory that are allowed to replace particular cache locations.

Standard caches use a uniform replacement policy for all cached memory locations, creating a monolithic cache appropriate for caching memory references that exhibit uniform locality. Standard reference streams, however, rarely exhibit uniform locality. Logically, a standard reference stream is an interleaving of multiple streams, each with a unique *region of locality* (Figure 3-1). Within a region of locality, memory reference behavior is uniform but different regions of locality have different behaviors, that is their range and access frequency change over time. A particular region may appear to a uniform replacement algorithm to require more cache than another region when, in reality, the opposite is true, producing sub-optimal caching behavior.

Ideally, the replacement policy is an oracle that does optimal replacement based on knowledge of the future. Unfortunately, oracles are currently beyond the state-of-the-art but there are situations in which software may know future access patterns. Newer processors, such as the Intel IA-64[37], provide instructions that can specify in which level of the cache accessed data should be cached. Such control, however, is tied to instructions and thus provides control that may be too fine-grained or too

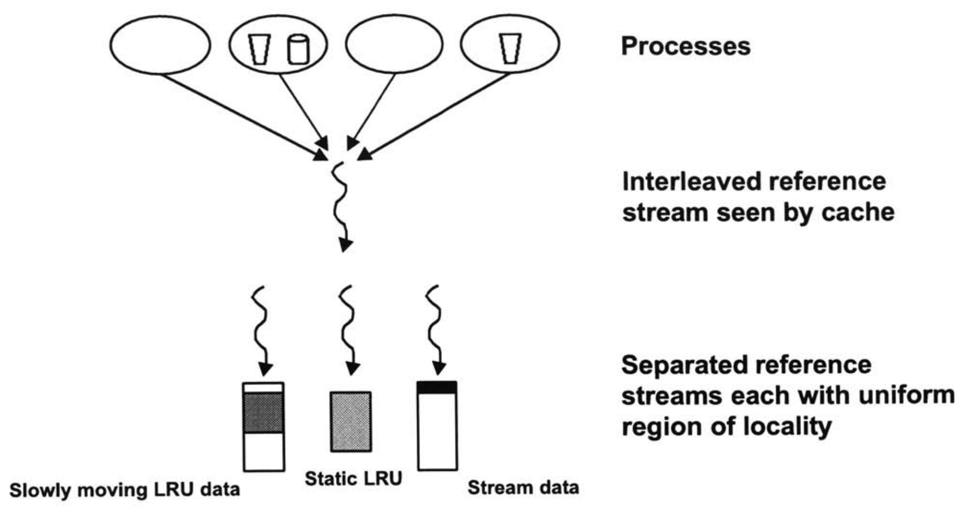

Processes
Interleaved reference stream seen by cache
Separated reference streams each with uniform region of locality
Slowly moving LRU data
Static LRU
Stream data

Figure 3-1: Multiple Regions of Locality. Multitasking operating systems multitask sophisticated applications that each use memory in a variety of different ways creating the memory reference stream seen by the cache. That stream seen by the cache can be logically decomposed into unique streams, each with a unique region of locality.

static. More dynamic and aggregatable control provides additional benefits.

One possible aggregation is for software to inform hardware of the regions of locality and their probable reference pattern. Logically, the predictability of future references based on past references is much higher in a single region of locality and thus hardware could potentially do a good job with replacement. Giving each region of locality its own, appropriately-sized cache using a specialized replacement policy can approximate an ideal replacement policy and improve performance and resource usage. It is possible for software to specify regions of locality and the cache size and replacement policy for each region.

Column caching is a mechanism that gives software the ability to make such specifications and cause the cache to act accordingly. Specifically, column caching enables software (or special hardware) to dynamically map regions of address space to cache partitions that are made up of *columns*. Cache partitions can overlap completely or partially, or be completely independent. Partitioning the cache isolates regions and eliminates conflicts resulting from standard replacement algorithms. Our implementation of partitioning, referred to as column caching, can improve hit rates and tune cache usage.

An alternative to column caching relies on the software control available in direct-mapped caches. Direct-mapped caches do not have a replacement policy since every physical address is always mapped to the same cache location. Such a cache provides software with substantial control since the software maps virtual addresses to physical addresses and thus can map memory regions to specific cache regions. Mapping pages to avoid conflicts and thus improve performance is called *page coloring*[12, 66] and has been shown to make a direct-mapped cache perform as well as a low-associative cache. Page coloring is orthogonal to column caching; the two techniques can be used together for mutual benefit.

This chapter first presents a motivating example for column caching. It then describes page coloring in more detail. Column caching is then introduced. Several of the possible uses for column caching are enumerated followed by a detailed implementation. Alternative designs are then explored. Related work and a summary finish up the chapter.

3.1 A Motivating Example

Consider a data compression application. Compression algorithms read a data stream into an input window and repeatedly search for common patterns within that window. It identifies the largest patterns, encodes them into smaller representations, and outputs these encodings. A large fraction of the memory accesses involve the repetitive searching within the window. While searching the window, other data structures are used to record patterns. Finally, there is a memory-mapped output buffer.

The three basic regions of memory, the input window, the patterns, and the output buffer are each accessed differently. The input window is read linearly, with some backtracking, as the application searches for patterns. It is never written. The patterns are stored in hash tables, a data structure that intentionally destroys spatial locality to evenly spread records. If the file is compressible, the patterns hash table will have temporal locality because a compressible file has repeated patterns. Thus some hash table references will be to a small number of repeatedly accessed locations,

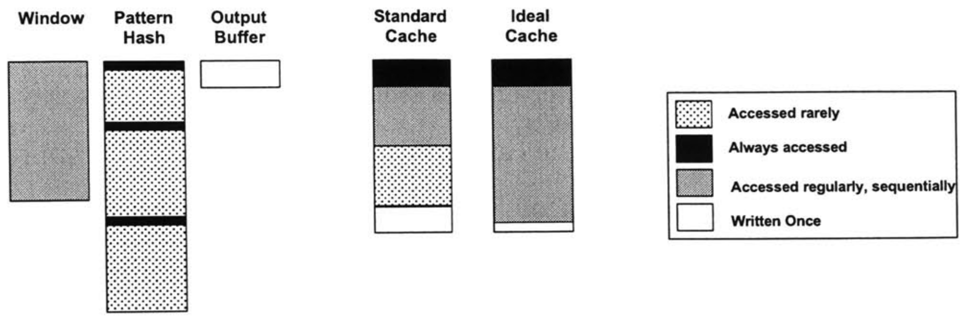


Figure 3-2: Compression. The 3 figures on the left, labeled *Window*, *Pattern* and *Output Buffer*, show the memory consumption, specific access locations and frequency and type of access. The standard cache shows how a standard LRU cache would cache these references. The ideal cache shows how these references should be cached to achieve best performance.

while the others will appear random. The output buffer, on the other hand, is written sequentially, but never read. It is accessed less frequently than the window or the hash table since it stores compressed data on the way to the disk.

The three memory regions compete for cache space (Figure 3-2). The input window and the pattern hash table are accessed at approximately the same rate (look in the window, then look for the pattern), while the output buffer is accessed far less frequently. A standard LRU replacement algorithm will cache the frequently accessed hash table entries and (approximately) split the rest of the cache between the window, the randomly accessed hash table entries and the output buffer. With limited cache space, the randomly accessed hash table entries should not be cached since they are rarely reused. Moreover, the output buffer does not need to be cached at all. Due to cache-lines being bigger than a single word, however, it may improve performance to cache a single cache-line of the output buffer.

To maximize performance, it is important to protect the window data from being replaced prematurely. Window data has a fairly long lifetime, but each cache-line within the window is not accessed frequently enough to keep it from being replaced by the pattern and output memory accesses. Thus, the window should be cached in its entirety since it will be reused several times. An ideal cache (also shown in Figure 3-2) would dedicate most of the cache to the window data, only caching the frequently accessed hash table entries and very little (if any) of the output buffer.

The LRU algorithm, however, replaces it with far less useful randomly accessed hash table entries and useless output buffer entries since they were more recently accessed.

To give an example, an ideal cache can improve gzip L2 hit rates by approximately a factor of two for L2 cache sizes of 32K and 64K and getting more the a 25% improvement for a 128K L2 cache, potentially improving performance by the same amount or more. In these examples, the L1 is assumed to be 8K, 2-way set-associative. L1 hit rates are very high to begin with due to a significant number of stack references and thus cannot be improved much by the ideal cache.

3.2 One Partitioning Solution: Page Coloring

Our approach is to provide the ability to *partition* the cache between different address regions, separating different regions of locality. Thus, in the example, a large partition of the cache is allocated to window data, a small region of the cache to the hash table and an even smaller region (or none at all) to the output buffer.

Page coloring[12] (see Figure 3-3) is a method to partitioning the cache that generally assumes a direct-mapped cache. Two virtual pages can be isolated from each other by mapping them to physical pages that do not overlap in the cache. Mapping virtual pages to the same cache page requires that the corresponding physical pages must be a multiple of the cache size apart.

Page coloring can have a significant performance benefit. Early research demonstrated that page coloring can generally make a direct-mapped cache perform as well as a two-way set-associative cache[12], providing better hit rates without the added hardware complexity. More recent research demonstrates that page coloring can improve overall performance by 20% to 30%[66].

Page coloring, however, has limitations. Once a page is allocated, the only way to relocate the page within the cache is by copying that page to another page that maps to another cache page, a costly operation. In addition to copying the page, the TLB and the page table entry also need to be updated in order to maintain transparency to the running program. Of course, the program can do the copying and deal with

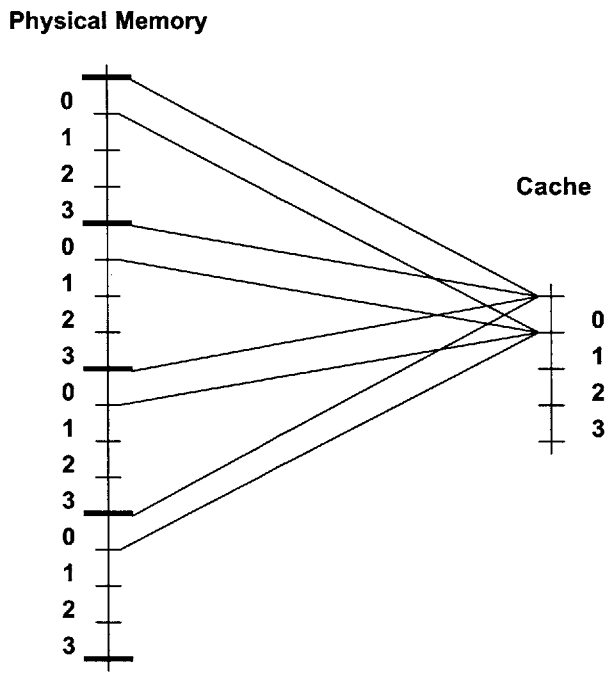


Figure 3-3: Page Coloring. When the operating system maps a virtual page to a physical page frame, it choses a physical page frame that resides in the desired region of the cache. For example, a 0 memory region must be selected to map to the 0 cache region. By judicious mapping, the operating system can either separate regions of memory from each other in the cache, or combine regions of memory within the cache.

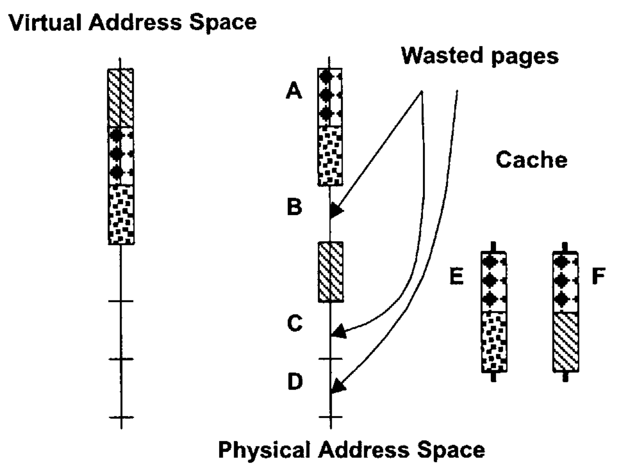


Figure 3-4: How page coloring can waste both memory and cache when used with a set-associative cache. We want to map the memory page marked A to the cache page marked E. Only one more page that maps to cache pages E or F can be allocated to avoid conflicting with A in the cache. If none of those pages are allocated, however, part of the cache pages E and F are wasted.

the address change itself. This will eliminate the need for page table manipulation but not eliminate copying.

Many remappings require many more than one page copy. They may also waste memory. If a page Q is being mapped to an exclusive cache page, it is possible that $m_p/c_p - 1$ pages (where m_p is the number of pages in memory and c_p is the number of pages in the cache divided by the associativity of the cache) need to be copied to give page Q exclusive access to that cache page, a tremendously expensive operation. Those $m_p/c_p - 1$ pages are wasted as well, since they would conflict with page Q in the cache. Thus, there are both performance and resource problems with page coloring in a direct-mapped cache.

Things are even worse in a set-associative cache. The same copying problem exists. The same memory wastage problem exists, but worse, because c_p will be smaller for a n-way set-associative cache compared to a direct-mapped cache of the same size. Of course, it is possible to use $n - 1$ of the wasted pages since the cache is capable of caching them without interfering with the isolated page. If no pages within the wasted $1/p$ range are used, however, $(n-1)C/nc_p$ space in the cache (where C is the total size of the cache) will be wasted as well. Figure 3-4 illustrates these problems.

Page coloring is also unable to map contiguous regions of memory addresses to a single cache page so it cannot be used with I/O streams. Virtually all I/O devices have memory-mapped buffers that occupy contiguous physical addresses.

Hardware support can improve page coloring. For example, a high speed memory copy implemented by the memory controller, coupled with a block invalidation for the cache(s) would dramatically improve the ability to move memory pages from one region of cache to another. These mechanisms would be generally useful as well as they would dramatically improve memory copying performance. Though improving remapping, however, they would be difficult to implement lazily. In addition, this support will does not deal with non-contiguous regions of memory.

3.3 Column Caching Overview

In column caching, each region of locality is conceptually mapped to its own separate cache. The mappings can be changed dynamically. Cache partitioning can guarantee that regions of locality interfere or do not interfere. Often times, different regions of locality can coexist — after all, standard caching generally performs very well. If a cache is correctly partitioned, where the reference patterns within each partition have uniform characteristics, it is much easier to accurately predict future references from past references and, therefore, get better hit rates.

A cache partitioning mechanism should have the following properties:

- Allow fast repartitioning of the cache and remapping of regions to those partitions.
- Allow flexible mapping between regions of memory and regions of cache.
- Use memory and cache resources efficiently.
- Have no negative effect on the clock speed nor slow down the critical lookup phase. Since caches are on the critical path in all processors, increasing latency can significantly impact performance.
- Be backward performance-compatible – that is, there should be no penalty if partitioning is not used.
- Be inexpensive to implement.

Page coloring violates the first three requirements. Page coloring does not allow fast dynamic repartitioning of the cache. It does not allow flexible mappings since it cannot map a contiguous region of memory to a single cache page. And, it does not use memory and cache resources efficiently since mapping a single memory page to a single cache page wastes memory and potentially cache space.

The simplest hardware approach, providing several separate caches, where each cache contains only specific data and only that cache is searched when looking for that specific data, violates all of the requirements. The static nature of the cache does not allow for fast repartitioning, flexible mapping and efficient use. In addition, there is negative impact on cycle time, since it must be determined where to look for the requested data. Finally, such a cache cannot be used as a single monolithic cache.

As we shall see shortly, our proposed mechanism, column caching, addresses *all* of these issues: it allows flexible and dynamic partitioning, uses resources efficiently, has no performance impact during lookups[1], runs as a normal cache if necessary and it is easy to implement with minimal extensions to standard set-associative caches. The rest of this we describes this mechanism and how it is able to achieve these design goals.

3.3.1 What is a Column?

We now describe reference column caching through the end of this section. Each column is one "way", or bank, of the n-way set-associative cache (Figure 3-5). Every cache-line in the set is searched during every access, making each column effectively a separate direct-mapped cache and allowing any memory location to be cached in any column.

In a standard cache, lower-order bits are used to select a *set* of cache-lines which are then associatively searched for the desired data. There are two control units normally associated with the cache. The *hit unit* determines whether or not there is the hit on each access using the address tags and cache state tags stored in the cache itself

[1]No performance impact during lookups is assuming the reference column caching implementation and not low-associativity designs.

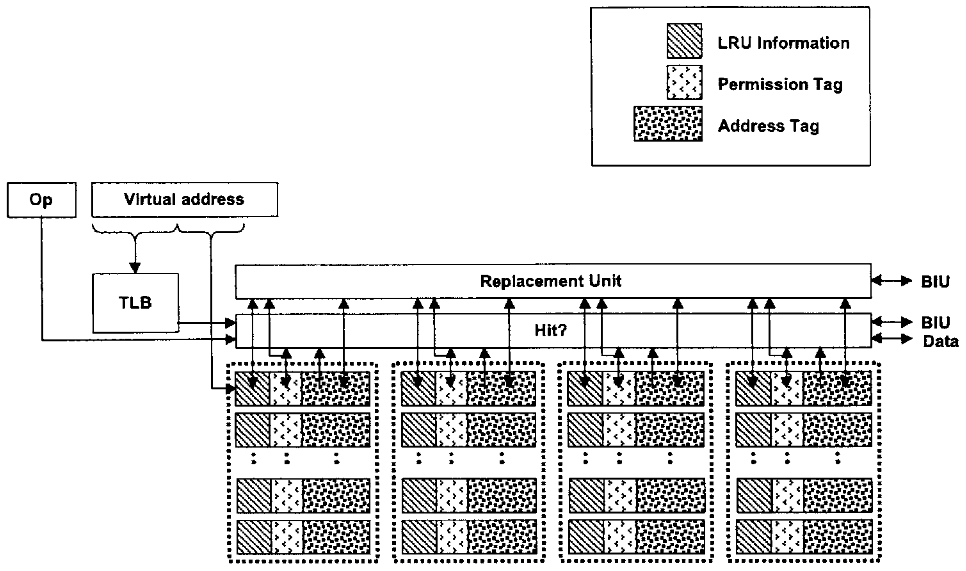


Figure 3-5: A four-way set-associative cache. Dotted boxes surround each possible column.

along with the requested physical address and the opcode of the request. Though physical caches are assumed throughout, column caching works just as easily with virtual caches. The *replacement unit* determines which cache-line should be replaced if replacement is necessary. We assume a least-recently-used (LRU) replacement algorithm throughout our base implementation and thus store LRU state with each cache-line. In addition, a permission tag and an address tag are stored for each cache-line.

Column caching incurs no performance penalty during lookup, since lookup is precisely the same as for a standard set-associative cache. By mapping all regions of memory across all columns, the cache becomes a normal set-associative cache. Repartitioning is graceful; if data is moved from one column to another (but always in the same set), the associative search will still find the data in the new location. A memory location can be cached in one column during one cycle, then re-mapped to another column on the next cycle. The cached data will not move to the new column instantaneously, but will remain in the old column until it is replaced. Once removed from the cache, it will be cached in a column to which it is mapped the next time it is accessed.

By aggregating columns into partitions, we provide set associativity to partitions as well as increase the size of partitions. Obviously, larger partitions require more columns, implying more set-associativity for the partition. Again, this functionality fits in well with standard set-associative caches.

An additional level of software management may be necessary if the column cache services multiple independent processes. Traditional caching allows each running process to use the entire cache. Such a policy is undesirable for latency-sensitive periodic processes and when context switch times must be minimized. The operating system can manage the entire cache, not surrendering mapping control to individual processes, optimizing cache usage across all processes. This scheme can improve overall throughput without any modifications to applications, but may not improve the performance of individual applications.

The operating system could manage the cache with hints from the running processes. Or, it could give each application a part of the cache to manage on its own, reserving the right to take back columns if necessary. These last two options, similar to user-level page management[45], allow applications to optimize their own cache usage without the operating system giving up all control.

The remaining high-level problem, then, is how to partition the cache so that data from specific memory regions are cached only in specified columns.

3.3.2 How To Partition?

Column caching partitions the cache by modifying the replacement policy implemented by the replacement unit. A standard set-associative cache replacement policy considers all cache-lines in a set as candidates for replacement. A column cache replacement policy, on the other hand, provides the ability to limit replacement candidates to a subset of cache-lines in a set. Selectively restricting replacement is essentially partitioning the cache.

Implementation is greatly simplified if the minimum mapping granularity is a page, since existing virtual memory translation mechanisms including the ubiquitous translation-look-aside-buffers (TLB) can be used to store mapping information that will be passed to the replacement unit. TLBs are designed to be accessed every

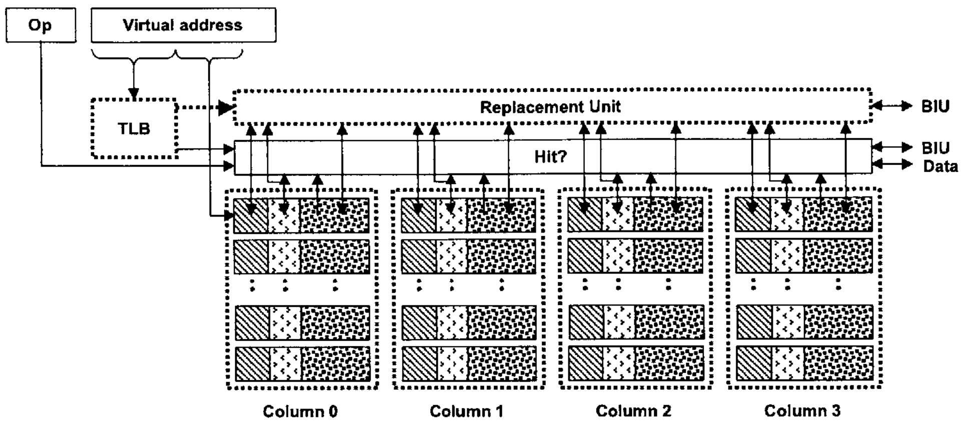


Figure 3-6: Basic Column Caching. Three modifications to a set-associative cache are necessary: (i) augmented TLB to hold mapping information, (ii) modified replacement unit that uses mapping information and (iii) a path between the TLB and the replacement unit that carries that information.

memory reference and are designed to be fast in order to minimize physical cache access time. Partitioning is supported by simply adding column caching mapping entries to the TLB data structures and providing a data path from those entries to the modified replacement unit.

Thus, column caching is implemented by three small modifications to a set-associative cache (Figure 3-6). The TLB must be modified to store the mapping information. The replacement unit must be modified to respect TLB-generated restrictions of replacement cache-line selection. A path to carry the mapping information from the TLB to the replacement unit must be provided. Similar control over the cache already exists in standard caches for uncached data, since the cached/uncached bit resides in the TLB.

Basic column caching specifies replacement candidacy using a *bit vector* in which a bit indicates if the corresponding column is a candidate for replacement. The bit vector is used by the replacement unit, along with standard replacement data such as LRU information normally associated with each cache-line, to decide which cache-line to replace. There may be a single bit vector per partitionable unit, or there may be several bit vectors depending on factors such as the memory operation. For example, there may be different bit vectors for data cached in response to a store than a load. Such differentiation is useful to allocate different amounts of space to a memory region depending on its use, .e.g., production verses consumption of a buffer.

The cache is made software-partitionable by allowing software to set the bit vector. It can be reset by the software at any time to allow repartitioning. Simply changing the bit vector does not move or invalidate data in the cache. Rather, a changed bit vector only affects how new memory addresses are brought into the cache. There is no need to move data out of a cache unless there is space pressure or the data is needed elsewhere exclusively. Lazily repartitioning in this fashion reduces the overhead of partitioning and the resources required to repartition. By simple changes to the replacement algorithm, however, data cached in a region where it is not mapped will quickly be replaced by data that is mapped to that region (more details will follow in Section 3.5.3). If all the bits in the bit vector are asserted for all memory addresses, the column cache behaves exactly like the original set-associative cache.

The bit vector have semantics other than indicating which columns are potential replacement candidates. Instead, the nth bit can indicate whether to replace into the nth most-recently-used position. A standard LRU cache, for example, would be emulated by having only the 0th bit set since replacement is always done into the most-recently-used position. In order to achieve true partitioning, however, this approach requires careful updating of LRU state.

To improve clarity, we define the terms that we will use throughout the rest of this chapter. We define a *cache page* as the minimum cache mapping region. Because the reference implementation of column caching assumes that the minimum unit of cache allocation is a column and a column is the size of a page, we often use column as a synonym of a cache page. Towards the end of the chapter, when discussing low-associativity column caching, we use the term "cache page" instead of column, since the latter means an entire way or bank of of the cache. A *set of cache pages* is equivalent to a set of cache-lines, just at page granularity.

A cache *partition* is a set of cache pages that are assigned to a specific region of memory. A partition may overlap with other partitions, i.e., they may contain cache pages that are assigned to other partitions as well.

3.3.3 Software Control: Virtual Partitions

To leverage existing virtual address translation mechanisms, especially the TLBs, our reference column caching mechanism uses pages as the basic granularity of mapping to columns. Thus, mapping information is mostly contained within page table entries and is accessed on every memory access just as translation information is accessed. Mapping, therefore, requires manipulation of virtual memory translation structures.

Mapping a page to a cache partition represented by a bit vector is a two phase process. Pages are mapped to a *tint* rather than to a bit vector directly. A tint is a virtual grouping of address spaces. For example, an entire streaming data structure could be mapped to a single tint, or all streaming data structures could be mapped to a single tint, or just the first page of several data structures could be mapped to a single tint. Tints are independently mapped to a set of columns, represented by a bit vector; such mappings can be changed quickly. Thus, tints, rather than bit vectors, are stored in page table entries.

The tint level-of-indirection is introduced to isolate the user from machine-specific information such as the number of columns, or the number of levels of the memory hierarchy. Another motivation is to make re-mapping easier. Imagine a scenario where initially each page is mapped to all columns, the default case that behaves exactly the same as a normal cache, and one page is remapped to its own dedicated column. This operation requires changing the mapping of all pages to columns. If bit vectors were stored within the page table entries, then all page table entries would have to be changed (top of Figure 3-7). In our tinting scheme, on the other hand, a new tint is allocated and assigned to the to-be-isolated page and the mappings between two tints and their bit vectors must be changed. The level of indirection allows the bit vectors of a group of pages to be changed simultaneously, rather than requiring each page to be changed individually.

Rather than eagerly tinting all pages, the appropriate tint can be automatically applied when a page table entry is allocated. If a region's tint is changed (re-tinted), each page table entry of that region needs to be updated and any corresponding TLB's

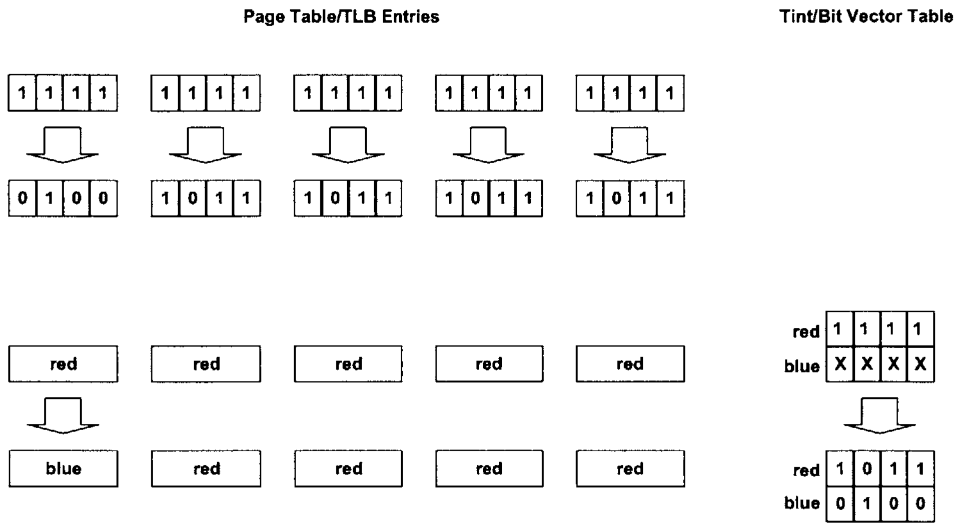


Figure 3-7: An example that demonstrates the advantage of storing tints instead of bit vectors in the page table entries. In the first example, where we store raw bit vectors in page table entries, in order to remap page 0 to use its own column and the rest of the pages to use the remaining columns, we need to change all page table entries. In the second example, using tints, only one tint and two tint/bit-vector table entries need to be changed. All pages start with the default tint, red. In order to give one page its own column, that page's tint is changed to blue. Tint blue corresponds to a bit vector that specifies that page to be cached in the second column. Tint red's bit vector is changed to remove the second column as a possible replacement column. The TLB entries for all former tint red pages must be flushed or modified in place to reflect the new bit vector.

`map_init_tints(num_tints)` OS call that creates a specified number of tints. Returned status indicates how many tints, if any, were actually created and a pointer that provides control access to the tints.

`map_init_columns(num_columns)` OS call that allocates a specified number of columns. Returned status indicates how many columns, if any, were actually allocated and a pointer that provides control access to the columns.

`map_malloc(size, tint)` User-level call that allocates memory and maps that memory to a certain tint.

`map_address_range_to_tint(*addr, size, tint)` User-level call that maps a preallocated region of memory to a certain tint.

`map_tint_to_bit_vector(tint, bitvector)` User-level call that maps a tint to a bit vector. Here, we assume a virtual bit vector that is translated to a physical bit vector by software and protected by hardware mechanisms (Section 3.5.2).

Figure 3-8: Possible Cache Mapping Services

either updated or flushed depending on the requirements of the processor architecture. Re-tinting should occur very infrequently since changing the mapping between tints and bit vectors is all that is necessary for most remappings.

Both mappings are achieved via user-level memory operations. Operating system calls are first made to initialize mapping data structures and to allocate control regions of memory in user-level space. User-level code access the control regions of memory directly to set mappings between address regions and tints, and mappings between tints and columns.

In Figure 3-8 we list a possible set of mapping services functions.

The `map_init` and `map_change` calls are operating system calls and, therefore, incur more overhead. They should be called very infrequently. The rest of the functions are user-level functions. Because they are composed of memory accesses to user-space, they should incur very little overhead. Helper functions to assist with tasks such as bit vector generation will also be provided.

Protection mechanisms to ensure that one process cannot map to columns assigned to another process should be provided. To achieve best performance, some of those protection mechanisms may need to be part of the hardware. Ideally, user-level code can update tints and bit vectors without operating system calls. The operating system must perform mappings for processes that do not use column caching explicitly, in order to avoid polluting mapped columns that are expected to persist across context switches.

3.4 Uses for Column Caching

Column caching enables tuning the tradeoff between cache-resources and performance. In a traditional system, the entire cache is used by the currently running process. Thus, the running process is allowed to use all the resources to maximize its performance. If the running process does not need all of the cache or if the running process is not performance critical, however, there may be better uses for regions of the cache. Column caching allows the operating system to target a "global maximum", rather than throwing all available resources to the task at hand. With proper management, processor throughput can improve with either no degradation or even improvement in individual process latency.

In the rest of this section, we elaborate on ways that cache partitioning can help achieve higher performance or maintain performance with fewer resources.

3.4.1 Controlling Pollution

Memory references that have short temporal locality can *pollute* a standard cache, replacing data that should remain cached. Pollution is caused by multiple regions of locality sharing the same monolithic cache, e.g., a stream and a stack or a FIFO queue slightly larger than the capacity of the cache. Pollution can also be caused by sparse, random accesses such as accesses to a hash table or by regions of memory that have different access patterns depending on the phase of the program. To avoid damaging the locality of other regions, the multiple regions of locality should be dynamically separated into different partitions, as the situation dictates.

A standard cache has no provision for controlling pollution; in fact, the standard

LRU replacement algorithm often aggravates pollution because it keeps recently accessed data (that is often not used again) and replaces less recently accessed data. Column caching can solve this problem by partitioning the different regions of locality into different partitions of the cache.

Explicit cache management mechanisms have been introduced into certain processor instruction sets, giving those processors the ability to limit pollution. Perhaps the most interesting are the new load/store instructions found in the Compaq Alpha 21264[46] that minimize pollution by invalidating the cache-line after it is used. Instructions to load into a specific memory hierarchy are provided by next generation Compaq Alphas and the EPIC/Merced/IA-64[37] instruction sets. Such instructions, if used properly, can improve performance and reduce cache pollution but statically bind specific cache behavior to specific instructions.

It has been demonstrated[73] that for several applications only a few memory instructions cause most of the cache misses. Simple hardware (or compilers) can track these “missing” memory instructions to determine whether to cache loaded data on an instruction-by-instruction basis, yielding fairly good results. It addresses the worse pollution, but cannot deal with data that should be cached, but should take less space that a standard replacement algorithm allocates it.

Most current and future processors provide support to allow uncached writes to be store-gathered, that is, consecutive writes to an uncached page will be gathered into a more-efficient burst transfer on the bus. This support allows regions of memory that would normally pollute the cache be mapped uncached but still use the fast burst path to the bus. Regions of memory mapped uncached for transmit must either be mapped cached for receive, requiring an additional virtual mapping to the same physical pages, or must have corresponding modifications to the receiving path to achieve reasonable receive bandwidth. In addition, not all stream data sequentially addresses the cache and there may be reason to cache the written data as well.

These sorts of cache management instructions and policies can be used with column caching to provide additional flexibility. Column caching, however, can provide many of the same benefits and is, in many ways, easier to conceptualize and use since it deals with regions of memory rather than individual instructions.

3.4.2 Constructive Interference

Purposely mapping a region of memory to a region of cache so that the current contents of the cache will be forced out is called *constructive interference.* For example, a memory region can be mapped to a small cache region, allowing a region to constructively interfere with itself. Such self-interference is very useful for memory regions that are accessed in a stream-like fashion, such as a buffer. For example, a message passing transmit buffer of size n can be mapped to a cache region of size m, where n is much larger than m. If messages are written out nearly continuously, not only will this mapping limit pollution, it will automatically push out message data as new messages are being composed in the cache. In contrast, a standard cache would simply become polluted by the transmitted data and delay moving the transmitted data out to the network.

A standard cache has no provision for doing constructive interference. Some processors support cache management operations such as `flush` or `clean`[51], but the instruction stream must execute one such instruction per cache-line consuming instruction dispatch and load-store unit slots, the exact cache-line address must be tracked creating run-time overheads, and synchronization must be inserted to ensure that the cache management operations are executed after the last access to the cache-line, to avoid ping-ponging the cache-line.

3.4.3 Enabling Compiler Optimizations

Significant progress has been made in the area of cache-aware compiler optimizations. Through careful allocation of memory, careful layout of data structures on top of that memory and careful scheduling and insertion of loads/stores to ensure that the correct data remains in the cache, compilers (or the extraordinary user) can improve cache performance within a standard cache. Such optimizations fall loosely into one of two categories: data layouts/instruction reordering that improve hit ratios and instruction rearrangement or instruction addition (prefetching) to ensure that data is cached before it is used extensively. Most of these optimizations depend on knowing specific details about the cache such as total size, block size, associativity, replacement strategy, etc. The compiler combines that knowledge with its program analysis with the assumption of a dedicated cache. Such hacks, however, are imprecise because (i) the cache replacement algorithm may not be fully understood or may vary depending on the particular implementation, (ii) instruction reordering can occur within virtually all modern superscalar processors and (iii) process switches can swap in other code that subsequently destroys the careful layout. Different cache instances also have different sizes requiring fancier layout code.

Column caching facilitates compiler-based cache mapping techniques by guaranteeing mappings, ensuring that mapped memory stays mapped and that they do not interfere with other regions of memory. For example, standard blocking techniques can benefit from cache partitioning that can prevent other data from being displaced by the blocked data. In addition to optimizing compilers for sequential machines, column caches are natural targets for compilers that optimize parallel codes by mapping memory to different nodes in a distributed system.

Cache-line pinning is available in processors like the Cyrix MII[23], while column-pinning is provided in other processors like the Motorola 8240[55]. The 8240 column-pinning allows software to specify that a specific column in the instruction cache not be replaced. Pinning eliminates a particular cache-line/column as a candidate for replacement, keeping the memory cached. Unfortunately, these mechanisms do not provide a way to determine whether the right data is in the cache when the pinning occurs, and potentially not pinning the data even though the instruction to do so was issued. In addition, at least for the cache-line mechanism, pinning requires an operation per cache-line to pin and probably another operation per cache-line to unpin.

3.4.4 Embedded SRAM

Column caching has the ability to create dedicated SRAM within the cache by mapping a single memory page to a single cache page. This emulated SRAM is better than SRAM, however; it is automatically swapped in and out if the cache page is remapped rather than requiring an explicit swapping as a standard embedded SRAM would. Of course, in order to guarantee performance, the load/store can be performed during remapping, just like with a dedicated SRAM. Such a structure is obviously of benefit to embedded applications as well as compilers that can detect and use embedded SRAMs.

3.4.5 Multiprogramming/Multithreading

The operating system can use column caches to improve multitasking. Current machines are fast enough to run a huge number of processes. Process switches, however, are limited by the cost of amortizing cache misses incurred by conflicts due to the multiple jobs, which are essentially cache pollution caused by multiple processes sharing the same cache (see Section 5.3.3). Such pressures can reduce the number of jobs a processor can realistically support.

Cache partitioning can improve the situation. For example, consider four running jobs (Figure 3-9). Each job uses memory in an LRU fashion and their working sets are smaller than the total cache. A mapped cache would allow efficient time sharing of the cache, while a traditional cache requires refilling the cache on each context switch. Notice that one job whose working set is twice as large as the rest can either share space with another job or be relegated to a smaller region of the cache.

Cache trashing, another form of pollution, has also been observed in multithreaded systems, especially those that perform context switches on cache misses such as the Alewife machine[47]. Column caching could eliminate thrashing by ensuring that memory operations from one thread cannot interfere with the memory operations from another thread by simply mapping the memory used by the two threads to different regions of the cache. Column caching can also ensure that a critical job's

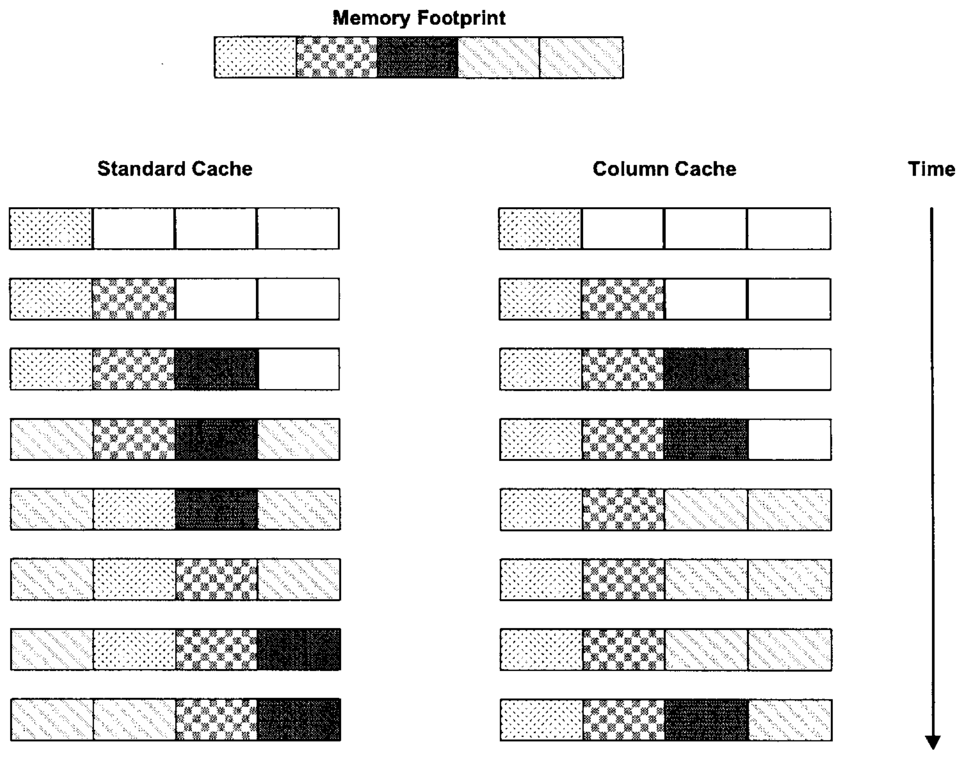


Figure 3-9: Four jobs scheduled in a round-robin fashion. Three jobs have a one page memory footprint, while the fourth job has a two page memory footprint. A standard cache always be cold when a job is swapped in because the other jobs have thrashed out its cache state. A column cache, however, allows state to be protected from other processes' references, allowing two out of the four jobs to run out of cache and another job to have half of its data in the cache at all times.

state is not polluted by other, less critical jobs.

3.4.6 Combining Instruction/Data Cache

Harvard architectures that imply separate instruction-data caches are a classic example of a statically-partitioned cache. Column caching can eliminate cache conflicts between instructions and data sharing the same cache while better utilizing the combined cache. Often times, programs with small instruction footprints have large data footprints (scientific codes) and programs with large instruction footprints have small data footprints. Certain commercial processors, such as the Cyrix 6x86MX[22] have fast, unified caches.

3.4.7 Speculative Execution Buffers

By partitioning the cache and specifying special replacement policies, part of a mapped cache can be used for speculative thread execution. In speculative execution, either (i) a branch is predicted and a path is speculated, (ii) a non-blocking request for a lock is issued and the thread is started on the assumption that the lock will eventually be returned or (iii) execution continues based on a speculated data value. Until the speculation is resolved, no writebacks can occur. Once that happens, writebacks are reenabled. If the speculation fails, the mapped regions of the cache are invalidated without being written back. Deadlock is possible in the locking case, but can be circumvented by timeouts. By partitioning the cache in a column cache, the speculation writes can be contained in a small part of the cache, potentially eliminating the need for special speculation store buffers, and speculation-initiated cached memory operations can be isolated to prevent pollution.

3.4.8 Multiple Memory Operations per Cycle

Column caching is compatible with the simpler multiple memory operation implementations that split odd addresses and even addresses to two banks and allow simultaneous access to those banks. There is no change necessary to column caching if the odd and even words accessed simultaneously must be from the same cache-line. If the odd and even words can be from different cache-lines, it is possible (but perhaps not desirable) to have different bit vectors for odd and even words.

3.4.9 Copying in Cache: Tag Access

Assume that user-level code has protected access to cache address tags, cache permission tags and LRU information. Having such a mechanism enables a lot of useful functionality. For example, examining address tags allows software to determine what has been accessed recently and may assist in adaptive mapping policies. By changing the address tags and permission tags from one region of memory to another region of memory, we have effectively done a memory copy within the cache.

Of course, coherence issues immediately arise when you allow the ability to change address and permission tags. Protection must be provided to (i) limit such access to regions of memory that the current process has permission to access, and (ii) restrict update abilities to eliminate the possibility of an unhandled condition such as having two modified cache copies of the same address.

Protection can be provided by the standard translation mechanisms. The paging mechanism can also ensure that a process is only able to read its own tags from the cache, ensuring privacy to other processes. Protecting other process's tags from being accessed, however, may not be necessary since the non-shared data still cannot be read.

To support our copying example, pages could be marked single-process and pinned to a processor, indicating that only a single process (and no other bus devices) will be using the data, ensuring that no other devices will be caching the data. Thus, it would be possible for a process to change address and permission tags without worrying about coherence with other processors' caches. If the hit and replacement algorithms account for the possibility of having two cached copies of the same data, potential machine errors are eliminated. Then, by careful mapping of addresses (for example, the copied-from page and the copy-to page are both mapped to a single cache page), it can be guaranteed that such an operation is possible.

Issuing instructions for each cache-line may become too expensive, motivating block operations. The memory copy in cache, for example, could be easily implemented as a block operation, allowing whole pages to be copied with a single instruction.

Such an ability can deal with data accesses whose stride modulo a column size is zero, thereby using the cache very inefficiently. The data could be first copied to a new space, completely within the cache. After the computation completes, the data can be copied back to the original space by simply changing the address tags then immediately flushing the data. Address tags would need to be as large as the entire address minus the cache-line index bits and write-backs would need to regenerate the address from the address tag alone, potentially making such a scheme impractical.

Though tag access mechanisms are useful in non-column caches, these mechanisms work better in column caches. In a normal cache, hardware support must be provided to ensure that changing an address tag is safe and that the intended data has not been replaced. Such support requires an atomic operation that checks the original tag then replaces it. Such complexity implemented within the cache will likely slow the entire cache down. Column caching can ensure that the specific addresses are not replaced from the cache, potentially making such mechanisms easier to implement.

3.5 Implementation Details

In this section we cover column caching implementation details. We start by discussing where tints and bit vectors are stored within page table and TLB entries. We then describe how protection of the partitioning mechanism might be provided. Modifications to the replacement unit and the replacement algorithm follow. The section finishes with the clock cycle impact of column caching.

3.5.1 Implementing Control Mechanisms

One simple implementation of the tint-to-bit-vector translation stores a page's tint in the page table entry and translates the tint to a bit vector when loading into the

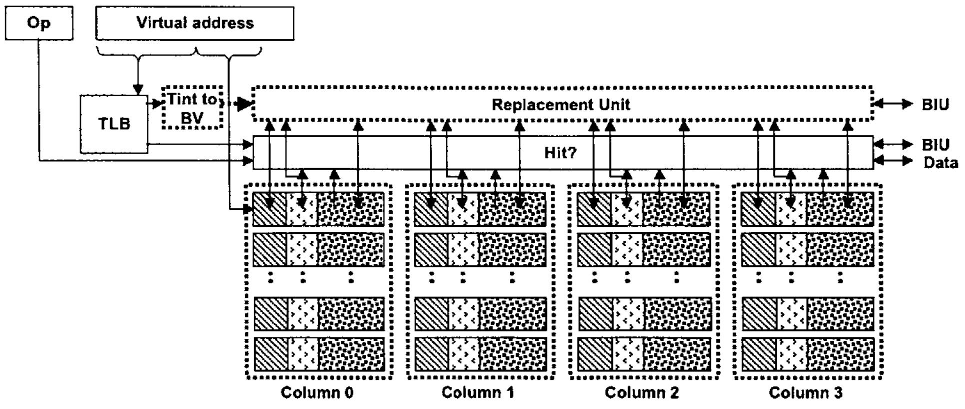


Figure 3-10: Adding a Tint-to-Bit-Vector translation unit between the TLB generating the bit vector and the replacement unit. This unit allows tints to be quickly remapped to different columns.

TLB. When tint-to-bit-vector mappings change, only the corresponding TLB entries need to be updated or flushed while the page table entries remain the same. The bit vector mappings are part of process state and thus must be saved and subsequently restored on process switches.

To eliminate the need to update the TLB on remapping, an additional lookup structure can be inserted (see Figure 3-10). When a TLB entry is loaded, the tint is stored in the TLB entry. When the TLB entry is accessed during a memory reference, the tint is read and passed to another translation unit that dynamically converts the tint to a bit vector that is passed to the replacement unit. The bit vector is only needed when replacement is necessary, probably removing this translation from the critical path. The conversion is a simple lookup, given a reasonable number of tints, since the tints are contiguous rather than sparse like the virtual address space. The resulting bit vector is passed to the replacement unit. If a tint-to-bit-vector mapping changes, only this tint-to-bit-vector needs to be updated. On the other hand, if a page is re-tinted, its page and corresponding TLB entry (if there is one) needs to be updated.

3.5.2 Protection

Protection mechanisms are provided to allow user-level access to mapping structures, avoiding expensive operating system calls every time a mapping needs to change. Protection can be imposed in one of two places: when the tint-to-bit-vector table is written or when the tint-to-bit-vector table is accessed for cache-line replacement. The former method requires no additional hardware but does require some protection during the table updates and updating or flushing the tint-to-bit-vector table when protections change. The latter method requires additional hardware but requires no changes when protections change. It is likely, however, that protections will not change often and thus the former method will probably work well with little effort.

The simplest implementation of the first method requires operating system calls to change tint-to-bit-vector mappings. Such a scheme requires no additional hardware but has fairly high overhead due to the operating system call.

Another implementation of the first method requires additional hardware in the form of a bit mask that encodes to which columns the running process can map. When the user writes a new bit vector, it is ANDed with the bit mask to produce the bit vector stored in the tint-to-bit-vector table. The bit mask requires maintenance. If the columns assigned to a running process changes, the operating system must update the bit mask. On a context switch, the operating system must save and restore the bit mask. If the columns assigned to a process change, the corresponding tint-to-bit-vector table entries must be updated accordingly.

An implementation of the second method uses the bit mask to mask bit vectors before they are passed to the replacement unit. Any such approach, however, adds time to the path between the TLB and the replacement unit. Performing the bit vector mask at this point eliminates the need to make changes in the tint-bit-vector mapping table when column permissions change since the bit mask ensures only valid mapping columns are passed to the replacement unit. It may be, however, that

An implementation of the second method uses "virtual bit vector" in the tinting-to-bit-vector map structure rather than an actual bit vector. Such an approach does add time to the path between the TLB and the replacement unit. The virtual bit vector assumes contiguous virtual columns rather than having to deal with the actual physical columns that are allocated to the process. During replacement, the virtual bit vector is translated into a physical bit vector in a way very similar to standard page translation. Using a virtual bit vector allows the operating system to change column allocation in a way transparent to the process. The translation process could be done in a way similar to page translation, requiring a bit vector page table with bit vector TLBs. Such a solution is very similar to simply translating tints, however, since the virtual bit vector can be considered a tint but may require fewer entries since there may be fewer cache partitions than tints.

There are other ways to implement partitioning without using bit vectors. One possibility is to specify a *replacement mode* per memory page. Replacement modes might include (i) use all unmapped columns, (ii) map to stream column, or (iii) map to unique column 0. There would be a corresponding *column mode* per column such as (i) unmapped, (ii) stream, or (iii) unique column 0) that define how the column should be used. The replacement mode might also specify the replacement policy used for its page.

The semantics of column caching might be that partitioning is a hint rather than an imperative. Doing so might make the mechanism less useful, but it may also make the implementation easier under certain circumstances.

3.5.3 Simple Replacement Algorithms for Column Caching

The replacement unit and the policy it implements is the center of modifications to implement column caching. Exactly how the replacement selection is restricted using a bit vector is very dependent on the details of the replacement algorithm itself.

The full range of replacement algorithms can be used with a bit vector to specify which columns are candidates for replacement. Pseudo-random replacement is the simplest replacement policy to implement for both standard caches and column caches. In a standard cache using a random replacement algorithm the cache-line to be replaced is selected at random. One implementation of a column cache with random replacement would generate a random number from 0 to the number of columns the requested address is mapped to minus one. That number will then be used to select the corresponding column.

Similar modifications can be made to least-recently-used replacement algorithms to implement column caching. One way to implement an LRU replacement policy within a column cache is to delay the computation of the replacement cache-slot until the bit vector has been read from the TLB. Then, the same LRU algorithm is run on the bit-vector-specified subset of the cache. LRU replacement information, as in a standard cache, must be maintained.

Another possible replacement unit implementation provides a replacement unit for each partition specified by a unique bit vector. After translation is done and the correct subset has been determined, the output of the appropriate replacement unit is used for replacement. The standard LRU bookkeeping information is kept as in a normal LRU algorithm. Providing a separate replacement unit per unique active bit vector allows the replacement algorithm to start before the bit vector is actually read. Of course, it consumes a significant amount of resources, since there must be logically one replacement unit per unique bit vector pattern present.

To really tune replacement, it is possible to make the replacement policy per partition specifiable by software. The replacement policy selection can be part of a TLB entry's state which is passed to the replacement unit along with the bit vector. Thus, the replacement policy can be selectable per page, though that replacement policy applies across all the columns that that page might reside.

3.5.4 Column-Caching-Specific Modifications to LRU: Reinforced Repartitioning

Standard LRU algorithms consider all cache-lines in a set equal, a policy not well suited for column caching. If a column cache is repartitioned, a standard LRU algorithm may prevent the repartitioning from occurring because frequently accessed memory locations are still cached in old columns. For example, tint red is initially

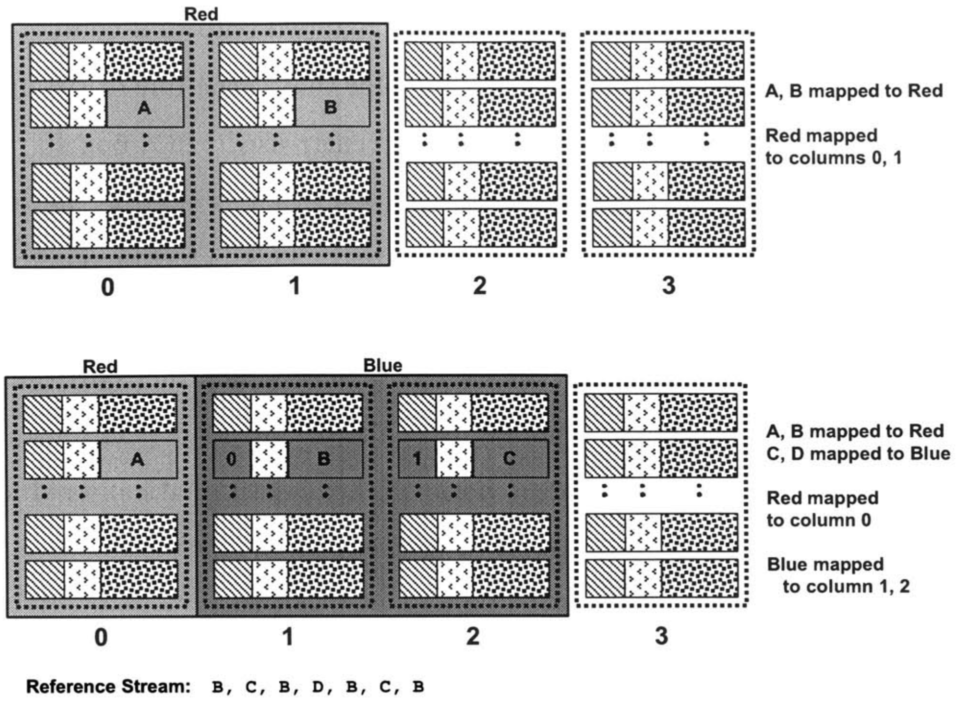


Figure 3-11: A standard LRU algorithm may prevent repartitioning from taking effect. The top figure shows the cache right before the remapping. The bottom figure shows the cache after the columns are remapped and the specified reference stream completes. When tint red is shrunk to a single column, it might still have data in columns it does not currently map. Data in the unmapped columns may still be consistently accessed, while the data in the mapped columns not accessed at all. A standard LRU algorithm, however, will continue to update the LRU data for that column and thus, given certain reference patterns, never replace it.

mapped to columns zero and one (Figure 3-11). Then, tint red is repartitioned to column zero while tint blue is mapped to columns one and two. Column one will continue to contain its old data while column two is forced to cache all of the tint blue data. Column zero, however, is full of data that will never be used again. This problem is due to the standard LRU replacement algorithm not accounting for column caching.

The following modification to the LRU algorithm fixes this problem. LRU algorithms maintain bits for each cache-line in order to keep track of which cache-lines are least recently used. After a cache-line is accessed, the LRU bits are updated to indicate that it was just accessed. Depending on the LRU algorithm, the update could be to all LRU bits in the set or some subset of the LRU bits.

If the cache-line caches a location that is not mapped onto that column our modification either (i) does not update the LRU bits for that cache-line or (ii) updates the LRU bits for the accessed cache-line to least-recently-used. In doing so, the cache-line will be replaced much sooner, since it will seem that it is either never accessed or immediately becomes the least-recently-accessed. This behavior reinforces repartitioning; cache-lines that do not belong in a specific partition are quickly replaced from that partition. It may be that an address belongs in another partition in which case it will be replaced from the unmapped column and read back into a mapped column. Additional support, such as a victim cache, can dramatically reduce or even eliminate the need to reload data replaced in this way.

3.5.5 Combining Random and LRU

Another possible modification to the LRU algorithm has potential benefits to standard caches as well as column caches. Standard LRU algorithms conceptually keep track of the absolute order in which cache-lines have been accessed. Instead, imagine if the accessed cache-line's LRU bits are set to zero while all other cache-lines' LRU bits are incremented by one. LRU bits cannot be incremented past a maximum value.

Using these values, the new replacement algorithm is as follows. When selecting a replacement cache-line, first find all cache-lines whose LRU bits are set to the maximum value. Choose one of those cache lines randomly for replacement. If none are at the maximum value, choose the cache-line whose LRU bits are at the largest value. This replacement algorithm combines random replacement with LRU. Intuitively, random replacement will perform better on memory reference streams that perform poorly under a LRU replacement algorithm, while LRU replacement performs better on streams that do have good LRU behavior. This intuition is supported by recent research[48].

3.5.6 Impact of Column Caching on Clock Cycle

Many caches precompute replacement cache-lines in parallel with the cache access. Upon a cache miss, the replacement line, address and bus operation information are sent to the bus interface unit that initiates a request for the line and coordinates the return of the requested data. If the replacement line currently contains modified data, that data also needs to be written back to the bus.

The bit vector is incorporated into the TLB and thus is not available until after the TLB is read. For a physically-addressed cache, a hit cannot be determined until after the TLB is read and the physical-address is constructed and thus, the extra cycle to read the bit vector is likely not on the critical path. If the cache set associated with the virtual address can be determined before translation completes, however, it is possible to determine the replacement cache-line before the TLB read completes. Determining the cache set from a virtual address requires either that (i) the cache banks (columns) are smaller than or equal to a virtual page in size or that (ii) the virtual-to-physical mappings are restricted so that the lower order bits needed to access the cache set are the same in the virtual address as the physical address.

Computing the replacement cache-line in a single cycle is difficult and is a timing bottleneck in many caches. There is, however, little reason for the replacement cache-line to be determined so early other than to allow the initiation of a pushout if necessary. In the fastest realistic system, data requested from the next memory hierarchy level takes at least three cycles to return. One cycle is needed to pipeline the request to the next memory hierarchy level, one cycle to read the data, and one cycle to pipeline the reply. Most systems take at least another two cycles from initiation to completion of a memory request since there are generally more pipeline stages. The exact replacement cache-line does not need to be decided until the data returns, giving the replacement algorithm at least 3 to 5 cycles to make a decision.

Since the replacement algorithm has several cycles to make a decision, it is likely that designers pipeline it. The bus operation to fetch the new line can be issued before replacement determination has completed. If the cache-line is not selected by the time the read completes, read buffers will buffer the data pending the decision. Since write-backs cannot be initiated until after the replacement line is selected, a few more pushout buffers may be necessary to balance the system.

Such a design removes replacement from the critical path and may even improve cache cycle times. Pipeline provides additional time to execute more complex replacement algorithms, such as column caching, without performance penalties. Thus, column caching should have no cycle time impact or may even improve cycle times as an artifact of pipelining the replacement policy.

3.6 Effect of Multi-Level Caches

Most modern memory hierarchies have more than one level of cache. The presence of column caching on each level of the memory hierarchy is not dependent on any other level. For example, column caching may only be provided in the L1 cache or only in the L2 cache or just in the L1 and L3 caches.

Generally, each level of the cache is different in size and in associativity, making it both undesirable and impractical to reuse the same partitioning across all levels. Our basic column caching implementation maintains a single tint for each page of memory. Providing the tint to each level of the memory hierarchy requires either (i) passing tint information from the TLB down through the hierarchy levels or (ii) duplicating the information in each memory hierarchy level. The latter is probably the preferred solution when successive hierarchy levels cross chip boundaries, since silicon is a far cheaper resource than bandwidth while the former is probably the preferred solution when the successive hierarchy levels are on the same chip.

The tint is converted to a bit vector at each memory hierarchy level independently so that different partitionings can be achieved at each level. Thus, each level in the memory hierarchy contains a table of tints to bit vector mappings. Software has the ability to change each mapping individually.

Controlling each level of the memory hierarchy may be too much trouble for the code generator (the user or the compiler) since details about the memory hierarchy might not be known at compile time. Instead, dynamically-linked libraries that contain specific information about the current hardware and that map common "patterns" to the respective cache levels can be provided to ease the mapping task for portable code. These libraries would allow the specification of tints, the mapping of regions of memory onto tints, and the specification of abstract mapping policies of tints onto a cache.

There are many possible mapping policies. The simplest would have every level in the hierarchy devoting approximately the same amount of space, modulo hardware constraints. An "expanding" policy, where the mapping at each lower level of the memory hierarchy is proportional to the highest level of the hierarchy, up to some limit, would probably be the default. Something like one of these first two policies for realistic behavior with inclusive caches, since every level of the hierarchy must contain everything cached in higher levels of the hierarchy. For non-inclusive caches, on the other hand, there is no dependence between levels of the cache; therefore, there is no restriction on how much space is allocated at each level. Polices that bypass the first level of cache, or first few levels of cache would also be provided for large data sets that have no spatial locality and not enough temporal locality to make caching in a small cache worthwhile.

Inclusive caches can implement a form of column caching by just implementing column caching in lower-levels of the memory hierarchy. By restricting the amount of space that a region of memory can take in a lower cache, ancestor caches are automatically limited as well. Though conflicts are not necessarily avoided, space in a higher cache is automatically limited to be smaller than or equal to the space allocated in a lower cache.

Some caches (MIPS R8000[35, 63]) load certain data such as floating point data into the L2 cache (a "streaming cache" in the MIPSR8000) rather than into the L1 cache since regions of memory containing floating point numbers tend to be accessed in ways that would thrash the L1 cache. Research has been done in techniques to dynamically determine whether a cache should be bypassed[42, 73]. Such approaches, however, are a fixed replacement policy which will perform well in some cases but will likely perform poorly in others. In addition, additional structures must be inserted (read buffers for example) in order to exploit the available spatial locality.

3.7 Cache Associativity

Our reference design for column caching imposes the following constraints: columns are each one page large and memory regions are mapped on a page basis. Even with current cache sizes, however, page-sized columns result in a large number of columns, requiring high-associativity caches. High-associativity caches are desirable to get the full benefit from column caching but are expensive to implement. We discuss how high-associativity caches might be efficiently implemented. We also discuss column caching implementations that use low-associativity caches.

3.7.1 High-Associativity

Since the minimum cache partition granularity in column caching is a column, more columns means a finer granularity of partitioning. High-associativity, however, can create high implementation overheads and, in extreme cases, impact cycle times, latency, or both; more cache-lines need to be searched on each access and the replacement algorithm has more possibilities from which to choose. Thus, there are greater costs to a cache of higher associativity. High associative designs, however, have not been aggressively investigated since research has shown that there is very little benefit of associativity higher than four for common benchmarks[33]. By restructuring cache organization, it is possible to implement a high associative cache with little to no performance penalty.

One way to implement fast, high-associativity is to use Context-Addressable-Memories (CAM) to do lookup instead of using the standard single comparator per column architecture. CAMs do require significantly more space, perhaps two to four times more space for the address tags than the standard approach, since there is conceptually a comparator associated with each address tag. High-speed implementations of high associativity, such as with TLBs that are often 64-way or 128-way fully associative caches, are possible with CAMs. The SA-1100 StrongARM processor[39] contains a 32-way set-associative cache implemented with CAMs.

Another option to implement high-associativity caches is to use on-chip tags and pipelined hit units and replacement units. Current processors dedicate a considerable amount of silicon area to on-chip L2 caches or include dedicated L2 cache interfaces on the processor die. Assuming we have access to the space that would be consumed by an on-chip L2 cache and we had a dedicated path to off-chip SRAM, a very high-performance, high associativity (perhaps 128 to 256 way) L2 cache could probably be implemented.

By putting only tags and logic on the processor die and placing the cache data on external SRAMs, like the next generation UltraSPARC[49], much more space is available for tags. Since the tags, the hit unit and the replacement unit for the L2 cache are all on the processor die, L2 hit determination can be initiated at the same time as L1 hit determination. The L2 hit determination circuit can be pipelined since it is expected that accessing the L2 takes longer than accessing the L1. The pipelined depth should not be too deep, perhaps two to five cycles. The L2 replacement unit, of course, can also be pipelined and thus should not take any additional time. If the L1 misses and the L2 hits, the data is read from the external SRAM to satisfy the request. By removing data from the processor chip and replacing it with tags, very large L2 caches that can be quickly checked for hits are achievable.

There are some disadvantages to this scheme. High-associativity requires more computation and thus more pipeline stages than a low-associativity design. In addition, this technique does require an SRAM interface implying extra pins, or at least extra pads, that would not be needed if the entire L2 cache was on-chip. If the L1 cache, however, is large and is used efficiently due to column caching, these slight penalties may not make any real performance difference. There are also other alternatives to high associativity design described the next few sections.

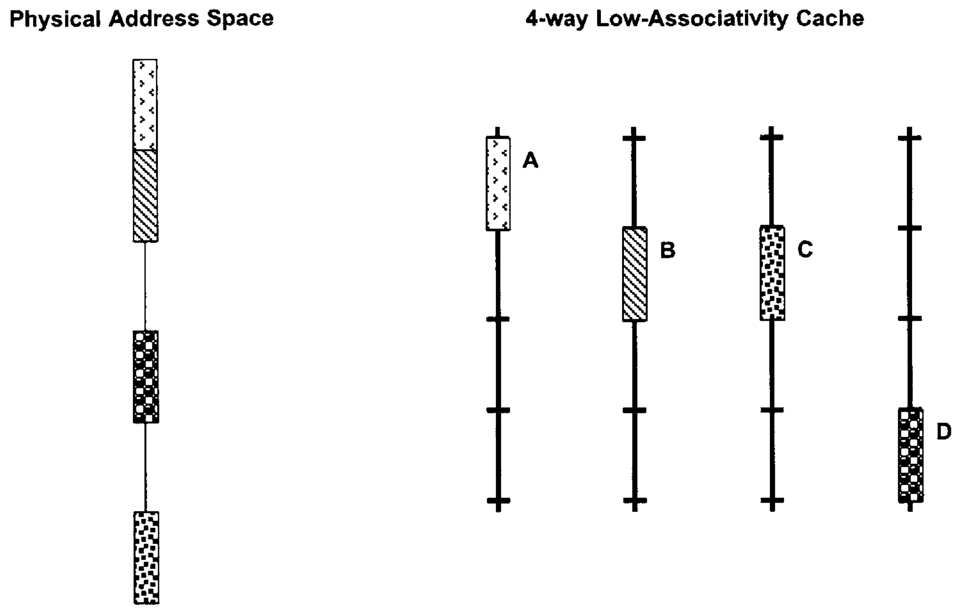


Figure 3-12: A naive low-associativity column cache. Each of the four vertical lines is a column. Four pages of memory are mapped into four cache pages, but regions A and D are permanently separated. Only B and C can potentially occupy the same region of the cache.

3.7.2 Large Columns: Low Associativity

It is desirable to have some associativity in a cache to avoid conflict misses. Modern set-associative caches are generally low in associativity, perhaps four to eight way, making each column fairly large. For example, the HP PA-8500 with a four way set-associative 1MB cache will have columns of 256MB each. Column caching techniques can still be used with low associativity caches, but the supported functionality is different than a reference column cache. With additional hardware support, however, a reasonable approximation of column caching can be provided within the context of a low-associative cache.

A naive implementation of low-associativity column caching maps page-granularity memory regions to tints that are then mapped to specific columns in the cache (Figure 3-12). When columns are larger than pages, a mapped page will consume only a cache page rather than the entire column. Only pages that map to the same sets can interfere with each other, while pages that do not map to the same sets are always completely independent of each other.

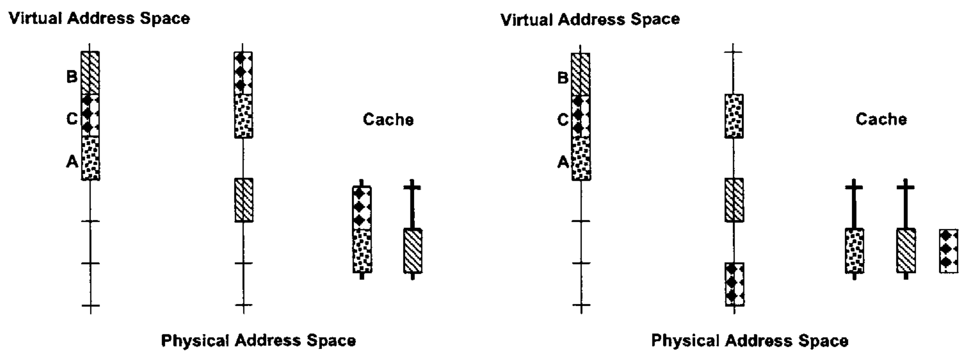


Figure 3-13: The limitations of low associativity. In the picture on the left, C cannot use the same cache space as A or B since it sits in a different sets. To fix this, C can be mapped to a physical page (via page coloring) that maps into the sets of the cache as A and B. Upon doing so, however, only 2 of A, B, and C can coexist in isolated regions of the cache since there is limited associativity. In the figure, A and B are cached, while C cannot be cached because there is not enough associativity, even though there is room in the cache.

Page coloring can improve naive low associativity column caching by allowing pages to be mapped to isolated sets of cache pages or the same set of cache pages. Column caching provides some dynamic remapping abilities, though such abilities are limited by the associativity of the cache. Figure 3-13 gives an example in a two-way set associative cache. In order for page A and page B to be able to be mapped onto the same region of the cache, they must occupy physical page frames that map to the same set of cache pages. If structure C is mapped to the same set of cache pages, however, it is impossible to isolate all three from each other and still keep them all cached because there is not enough associativity in the cache. If C is allocated to pages that do not overlap with A and B, it cannot share the same region of the cache without copying C (page coloring) to a page frame that does overlap with A and B in the cache.

Page coloring with column caching within a low-associativity cache can be thought of as several reference column caches bound together where each page is mapped to a single reference column cache. We will refer to each of the reference column caches as a *row*. Rows can be aggregated by mapping (using page coloring) a region of virtual memory to physical memory that maps to more than one row in the cache. Row aggregation increases the number of pages that can share the same cache space, but

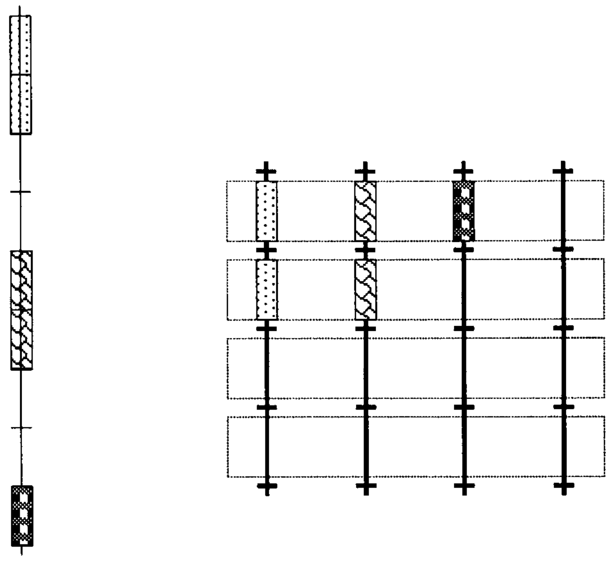


Figure 3-14: If the two regions of memory, each consisting of two pages, need to be able to interfere as well as be isolated in the cache, they might be mapped to two cache pages each, but within the same cache page sets, rather than every page to the same cache page. That way, they can coexist in the cache separated from each other as well as the third, one page region.

at the cost of increasing the minimum amount of cache space they take up. Thus, as potential cache region sharing goes up, the cache region actually increases in space, limiting benefits (Figure 3-14).

Another disadvantage of low-associative column caches is that, even with page coloring, they cannot map contiguous regions of memory to a single cache page. Even with all these restrictions, however, low-associativity column caching still improves on page coloring by providing more mapping control and thus enabling reducing potential waste of memory and cache. Low-associativity column caching enables efficient isolation in set-associative caches, eliminating the memory and cache waste that accompanies page coloring on set-associative caches. Figure 3-15 gives an example.

The limited mapping capability provided by low-associativity column caching with page coloring is sufficient in some cases, and provides benefits over plain page coloring. There may be cases, however, when more flexibility to remap regions quickly and/or the ability to map a contiguous region of memory to a single cache page is essential. In the next section, a mechanism that solves these problems is proposed.

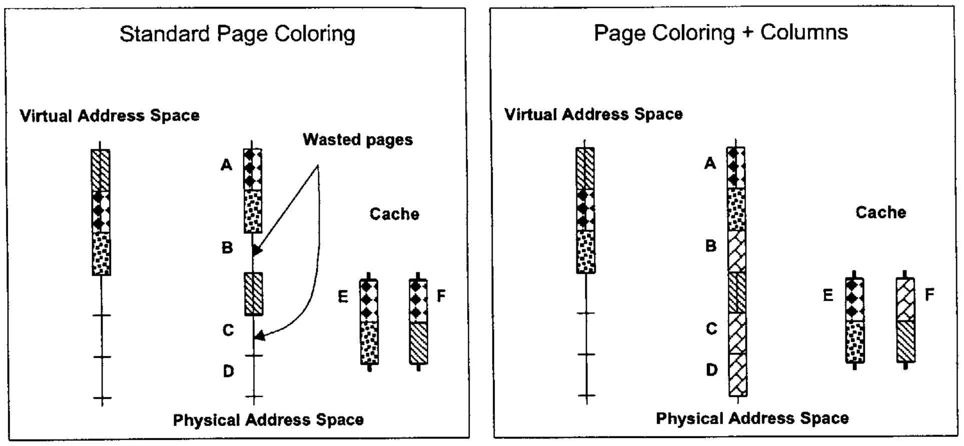


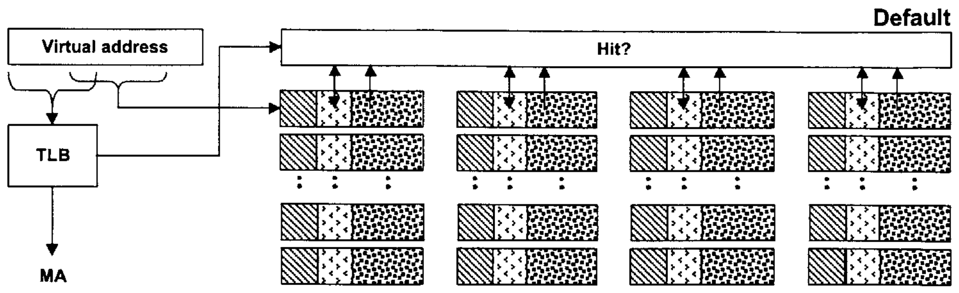
Figure 3-15: Page A should be mapped to cache page E. In a standard page coloring scheme, A cannot be isolated to only cache page E but gets replaced into either E or F. In order to avoid conflicts with A in the cache, at most one of pages B, C and D can be allocated at a time. Column caching fixes the problem by allowing A to only use cache page E and pages B, C and D to only use cache page F.

This mechanism eliminates most of the problems found in direct-mapped cache-based page coloring as well.

3.7.3 Separating Cache Addresses and Memory Addresses

Caches use memory addresses both to derive the cache set and to generate the address tag. Using the memory address as the basis of the cache address is the root of the page coloring problems described throughout this chapter as well as some of the problems with the low-associativity column caching implementation.

Rather than using the same address for cache and memory, an independent address can be generated for the cache. In fact, such a translation can be done logically at each level of the memory hierarchy. We call this separation of "global address" from a physical address used to access the current memory hierarchy *retargeting*. See Figure 3-16 and Figure 3-17 for the default and three examples of how retargeting might be implemented. In the default, the TLB generates a physical address that is used both for the next level of the memory hierarchy as well as the cache. In the first example, labeled A, the TLB stores two addresses, a physical memory frame number and a cache frame number. In the second example, labeled B, the TLB generates



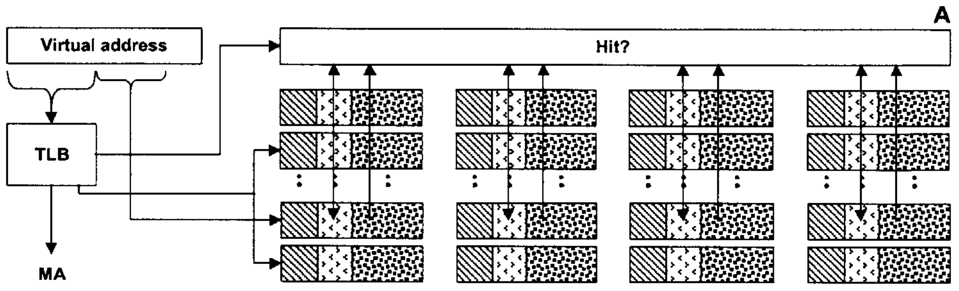


Figure 3-16: A standard cache (top) and a modified cache (bottom) that uses a Cache Address (CA) generated by the TLB to access the cache. MA stands for memory address and is the standard physical address.

the memory address for the next level of the memory hierarchy as well as a set of cache pages. The lower order bits of the address index into the cache pages. In the third example, labeled C, the functional unit takes all of the address bits as well as information from the TLB. The functional unit can ignore any of the information.

The last method is the most flexible of the solutions presented here and is the one we will refer to when discussing retargeting. One possible function is to specify separate cache pages for each column. Doing so allows full flexibility as to where a page is mapped within the cache.

If the retargeting functional units are sufficiently powerful, retargeting can even provide the ability to map regions of memory to regions of cache that are smaller than a cache page (Section 3.8.2). Flexible retargeting allows the mapping of any memory region onto any cache region. Such an ability is useful for many things including large pages and for regions of memory that need less than a page of cache.

Note that adding such a mechanism does introduce additional logic into the cache hit path that adds at least a cycle since the cache cannot be read until the cache address is generated. The logic could also impact cycle time. Low associativity,

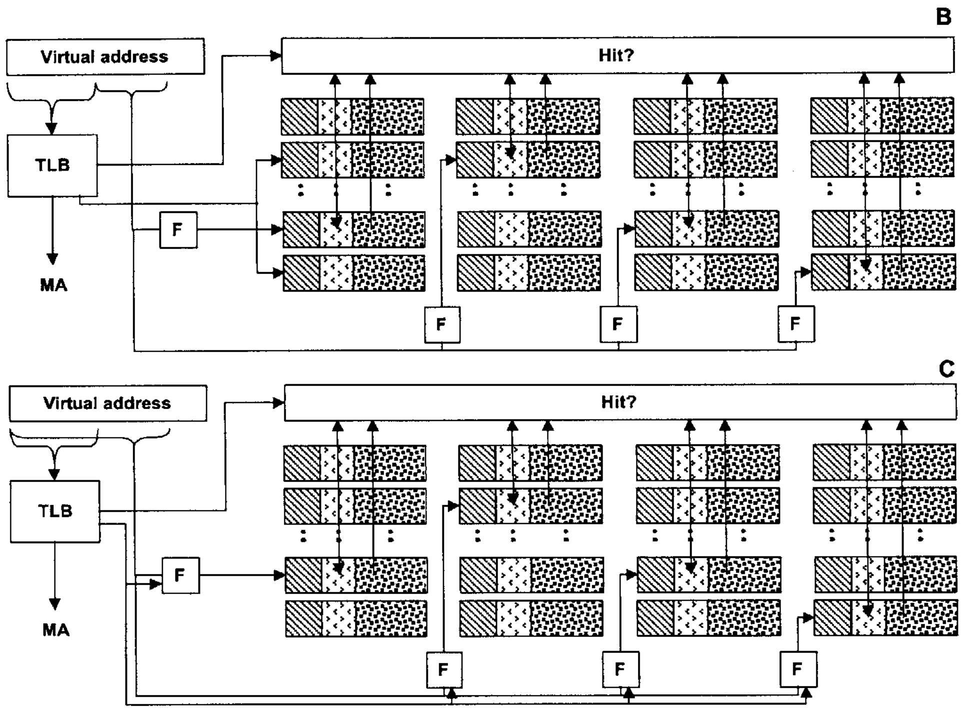


Figure 3-17: A cache (top) that uses a TLB-generated Cache Address that selects a cache page and a functional unit per column that generates the offset within that region. An obvious simplification provides a single functional unit to select the same cache page in each column. The second cache (bottom) generates the set of Cache Addresses as well as the address tag from the entire address and TLB-generated information using a functional unit per column. Any arbitrary functionality mapping can be implemented by such a scheme, assuming the functional units are sufficiently powerful.

however, is more prevalent in lower level caches. Since lower level caches are virtually all physically addressed, if retargeting can occur within the time needed for address translation, there will be no impact. Our primary design for low-associativity retargetted designs has that property.

Lower levels of cache generally can tolerate more latency anyways. Finally, initiating requests for these lower level caches early, in a similar fashion to that described to deal with high associativity (Section 3.7.1), could be used and thus potentially eliminate any cycle time impact.

Another issue raised by retargeting is the coherence problem within a single column cache (not the same as the coherence problem between multiple caches). By changing a memory page's cache page, it is possible to miss in the cache when the requested data is actually in the cache, causing another copy of the same data to be read and cached in a different position in the cache. Because the original copy of the data might be modified, the newly read data may actually be stale.

Given a standard set-associative cache structure, there is no simple solution to the coherence problem. In order to remap within the same column, the original location must either be cleared of modified data or both locations must be searched. It is possible that the clearing of the cache-lines can be done lazily, with the replacement unit tracking the cache-lines that are replaced. If it is expected that the entire cache region will be replaced, a simple counter indicating how many cache-lines new data has replaced is sufficient. More advanced tracking hardware, that keeps track of which cache-lines were replaced for example, could reduce the number of invalidations that need to be done in order to reuse the column. A block invalidation unit is another potential solution that has other uses as well.

Associativity can be achieved by sequentially cycling through multiple possible locations within a direct-mapped cache[1, 75]. Obviously, the most likely position is checked first. Such a technique could be used to alleviate the coherence problem associated with remapped data by providing the old location as a secondary location to look. Depending on the circumstances, however, it may or may not be advantageous to move data found in second or greater tries to its first try location or change the first try location if possible. If the associativity is really needed, it may be wise to leave data in the non-first position. If not, it would probably be best to move the data to avoid the successive checks.

Though such cycling will incur more latency, because the cycling starting point is specified under software control, it is likely to hit on the first try. Such emulation of set associativity, however, will require additional address tag bits, since a particular cache-line could exist in a variety of positions.

Having two copies of the same data in different columns of the cache is, however, potentially desirable. The hit algorithm, however, needs to know that this might happen and must handle it in a reasonable fashion. One way to handle multiple copies of the same data is to have a favored column for each page where data is read from, if possible.

One possible use for having multiple copies of data is a speculation buffer. Two columns are allocated as a single speculation buffer. One caches read data and the other caches write data to the same addresses. When reading from speculation buffer, the write column is favored. If the speculation turns out to be incorrect, the write column is thrown away while the read column remains. Having such an ability eliminates the need for having dedicated speculation buffers.

Retargeting makes cache position independent of the memory address, an ability that is useful for a variety of purposes including improving low-associativity column caching and making page coloring truly useful.

3.7.4 Improving Low Associativity with Retargeting

Retargeting improves low-associative column cache functionality since it enables full flexibility to map a memory page to any cache page. The footprint of a memory region in a specific column is difficult to change once it has been set due to the coherence problem. However, the footprint in each column can be different if a separate function is provided for each set. Figure 3-18 shows an example. The retargeting functions map the memory region to 4 cache pages in column 0, 2 cache pages in column 1 and 1 cache page in column 2, while column 3 is reserved and not currently mapped. Thus,

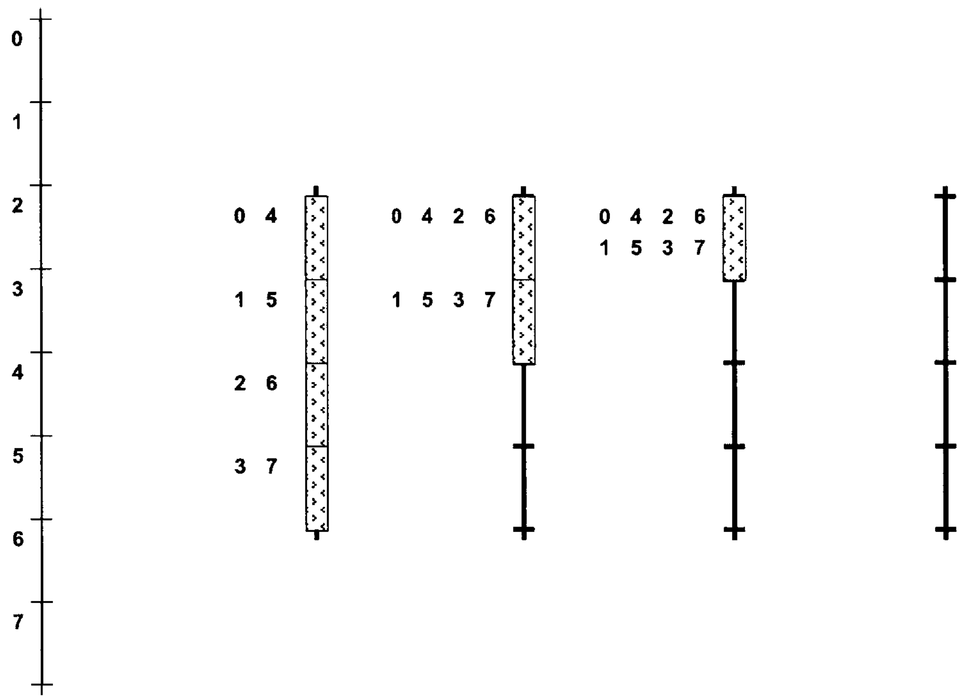


Figure 3-18: The memory range is mapped onto a different number of cache pages in each column. Such a mapping can be achieved by simple retargeting support that allows each memory page to be mapped to a distinct cache page for each column. Such support can be implemented by a set of cache page numbers in each page-table/TLB entry, avoiding the need for any special functional unit. By mapping in increasing binary sizes, any number of cache pages from 0 to 7 can be used to cache this memory region. Expansion and contraction is easy to do simply by changing the bit vector that maps the region to columns. In order to avoid coherence problems, the mapping of pages to cache pages cannot change without copying the data.

the memory region can consume between zero and 7 cache pages, with the possibility to expand to 7 + (1/4 cache-size) if the last column is mapped completely to the address region. Thus, retargeting supports fairly flexible expansion and contraction of a single memory region.

It is clear that the retargeting function required to implement such a mapping is quite simple. One way to implement this type of retargeting function is to maintain for each memory page, a cache page for each column. Again, using a level of indirection similar to tints either in the page table entries or in the TLB that is then converted to the respective page numbers simplifies remapping.

With retargeting, statically isolating two regions of memory from each other in the cache or combining two regions of memory within the cache is no different than in the naïve low-associativity cache and still trivial. Allowing two regions to dynamically combine or be isolated still depends on the associativity of the cache in order to avoid the coherence problem and is thus limited by the available associativity.

Regions that were previously not able to share the same region of the cache, however, can with retargeting support. Two cache regions, mapped to independent columns regardless of the specific cache page set within those columns can be mapped on top of each other via retargeting. Thus, they can be quickly separated or combined simply by changing their tints. The retargeting mapping must not change until the previously mapped cache pages are flushed of the retargetted data to avoid having multiple, potentially inconsistent copies of the same data in the cache.

3.7.5 Pseudo-High-Associative Caches

An alternative to a retargetted low-associative cache is a pseudo-high associative cache. In such a cache, the number of columns is large. Rather than looking in all cache-lines in a set to determine hit, however, only specify columns could be examined. Some specifier, perhaps another bit vector, must be provided to indicate which columns to search for a specific address. Though there is some overhead to providing this sort of selector for reads, the overhead will probably be less than searching all columns in a high associativity cache.

If the limit for the number of columns simultaneously accessible is reached, additional columns are unaccessible and thus non-intersecting. Thus, pseudo-high-associative caches are limited in the maximum size of the cache region that can be mapped to a specific region of memory, since there is a limit on the associativity and each column is a page large.

Of course, pseudo-high-associative caches have the coherence problem. If a memory address exists in one column, but that column is not specified as a searched column, the cache will falsely determine that the address does not exist in the cache and proceed to fetch it into the cache. If there is not enough associativity to address the problem, a memory copy is required.

Overall, it is likely this scheme is not quite as flexible as the retargeting-enabled low-associativity column cache, but could still be useful for implementing column caching.

3.7.6 Retargeting and Direct-Mapped Caches

The simplest form of retargeting, having the TLB generate a separate cache address, can easily implement some column caching-like abilities within direct-mapped caches. By making the cache address independent of the memory address, full page coloring functionality can be provided without the restriction of having to map to specific page frames. Thus, the requirement of two memory pages being an integer number of cache strides apart in order to map to a single cache page is eliminated. In addition, it is now possible to map a memory page to a unique cache page without wasting memory since the other memory pages that normally map to the unique cache page in a standard cache can be put elsewhere in the cache. It is, however, still potentially costly to remap memory pages to different cache pages.

This idea was proposed independently by other researchers[66].

3.8 Fine Granularity Mappings

The assumption so far has been that memory pages and cache were the minimum granularity of mapping. Though pages are convenient because of the virtual-to-physical translation hardware is already available, it may be desirable to map at finer granularities to capture small data structures, or may be desirable to map non-contiguous regions of memory. We discuss these alternatives and the issues involved with their use and implementation in the latter part of this section.

3.8.1 Mapping memory regions smaller than a page

There might be a need to be able to map regions of memory smaller than a page. In order to do so, the ability to associate more than one tint with each page is necessary. The simplest solution is to divide each page into some number of equally sized, aligned sub-pages. A separate tint for each sub-page is provided within the page table entries and are also cached within TLB entries. The appropriate tint corresponding to the accessed sub-page is selected during each memory access and provided to the replacement unit.

Another solution is to provide support for pages of different sizes. By doing so, memory regions that are smaller (or larger) than a page are automatically mappable. If the page sizes do not have to be powers of two, problems involving strided data accesses that all map to the same cache-line can be solved.

3.8.2 Mapping Cache Regions Smaller than a Page

It is also desirable to map to regions of cache each smaller than a single column. The simplest solution is to make smaller columns. Doing so, however, requires higher associativity.

Mapping cache regions smaller than a page requires retargeting ability to allow changing the memory addresses to cache addresses mappings. If the mapped cache region is smaller than a column, some of the same issues raised in the low-associativity column caching sections arise. By mapping memory regions to these small cache regions, certain memory regions cannot interfere with each other. The same solutions proposed for retargetted low-associativity column caching apply.

3.8.3 Non-Contiguous Mappings

There is clearly a need to map noncontiguous regions of memory to continuous regions of cache. Providing this ability will require additional translation units, since page tables and TLBs are designed specifically for pages. Retargeting provides the appropriate capability.

By providing the correct functions within the retargeting functional units, the cache is not limited to storing consecutive regions of memory in each successive set. See Figure 3-19 for an example. In this example, there is an array of structures,

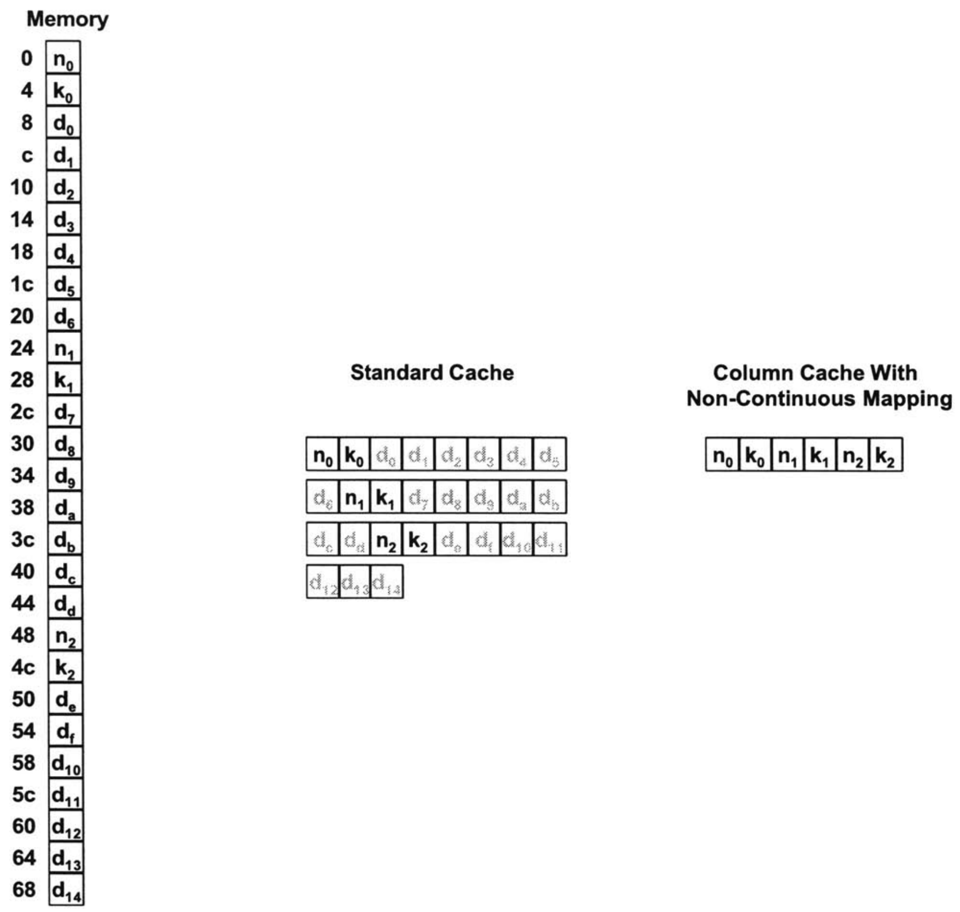


Figure 3-19: By altering the mapping between addresses and cache addresses, non-contiguous data can consume contiguous cache space and thus using the cache space much more efficiently.

each containing nine elements. The first two elements are the key and the pointer to the next structure. Most accesses to the structure are only to the key and to the next pointer and thus, only those two elements need to be cached. Using the right retargeting functional units, we can efficiently pack the key and the next pointer into a region of the cache.

Cache-lines of greater than one word are a problem, since they assume that the data within each cache-line comes from a contiguous address range. If retargeting functional units can relax that assumption, full generality of the cache becomes possible. Strided data or even irregularly spaced data can potentially be stored in contiguous regions of the cache, saving space within the cache and improving performance by better utilizing the cache.

As usual, the same consistency issues will again arise for general solutions during remapping. The same sorts of solutions, notably cache invalidation and pushout, memory copy and searching the old space, are all possible solutions.

The Impulse[17] project proposes something very similar to retargeting, but within the memory controller rather than the cache. By changing mappings within the memory controller, non-contiguous data can be packed into a contiguous region of memory before being sent over the bus, thus saving bandwidth. The modifications were proposed to be made within the memory controller in order to avoid any changes the the processor. A major disadvantage of such a scheme is that the software must be aware of the remapping, which is difficult and thus would make such remappings infrequent. In addition, memory controllers are becoming as complex or more complex than the processors they serve, since they must deal with network interfaces, a variety of peripheral interfaces, and multiple processors.

Such a memory controller, however, would potentially work well with a column cache that support non-contiguous regions of memory since the memory controller could return requested data in the correct, packed format and the cache deals with transforming requests from the processor. There would then be no need for software to know about the transformation since the transformation back would be done within the cache. The partitioning benefits of column cache would be available, along with the bandwidth benefits of the Impact memory controller.

3.9 Related Work

Conceptually, column caching provides much of the same functionality as page coloring, but eliminates the limitations of page coloring. Column caching eliminates the need to do a memory copy when remapping to a new region of the cache. Without the ability to do fast remappings, much of the potential benefits of partitioning are lost. In addition, unlike page coloring, column caching has the ability to map contiguous page frames to a single cache page. Column caching also works well with set-associative caches, where page coloring potentially wastes a significant amount of space.

Page coloring, however, can be used to assist the implementation of column caching-like functionality in low-associativity or direct-mapped caches. It is also possible that page coloring can be used in some hierarchy levels and not in others. We will discuss these possibilities later in this chapter.

Some existing and proposed architectures support a pair of caches, one for spatial locality and one for temporal locality[62, 72, 39, 9, 49, 27]. These designs statically separate the two caches in hardware, generally wasting resources since the partition is rarely exactly correct. Some rely on hardware-based algorithms that separate the reference streams into one or the other cache. Hardware algorithms may not be able to react quickly to changing reference patterns. Others keep information indicating which cache to use in the page table, allowing software to specify the mapping of memory to a specific cache.

Sun Microsystems Corporation holds a patent on a mechanism[57] very similar to column caching that allows partitioning of a cache between processes at cache column granularity. As part of a process state, a bit mask is specified that indicates which columns can be replaced by that process. The Sun technique allocates partitions to processes, rather than to address ranges, limiting its usefulness to isolating processes from each other.

3.10 Summary

Software partitioning of the cache enables application and operating system code provides the opportunity to optimize cache usage and, therefore, tradeoffs between control resources and performance. We summarize column caching and cache mapping in Figure 3-20.

ch Description	Column Caching	Associativity	Page Coloring	Retargeting	Remap	Flexible Mapping	Efficient Resources	Backward Compatible	Cycle Time	Inexpensive Implementation
PC	no	direct	yes	no	5	5	5	1	1	1
CC	yes	high	no	no	1	1	1	1	1	4?
CC + PC	yes	low	yes	no	3	5	2	1	1	3
CC + PC + RT	yes	low	yes	yes	2	1	1	1	3	4?
CC + pseudo	yes	pseudo	no	no	4	2	2	2	3	3
PC + Column Assoc	no	direct	yes	no	4	4	4	1	2	3
PC + RT	no	direct	yes	yes	5	1	1	1	3	4

Figure 3-20: A summary of the costs and benefits of some of the mapping options discussed in this chapter. The abbreviations are as follows: Column Caching (CC), Page Coloring (PC), Retargeting (RT), pseudo-set-associative (pseudo), Column Associative (Column Assoc). Difficulty is rated from 1 (easiest) to 5 (hard). In order to simplify the chart, several options were omitted or simplified. Retargeting stores a cache page per column within the page table. Column-associative is assumed to be direct-mapped. By adding additional support within page table entries and TLBs, sub-pages of memory can be mapped. By adding additional mapping structures and aggressive retargeting support, non-contiguous regions of memory can be mapped. Aggressive retargeting could also enable different cache mapping granularities.

Chapter 4

Curious Caching

Curious caching is a set of mechanisms that enable a cache to incorporate data not explicitly requested by its master, but, observed on the snooping side. Traditionally, caches incorporate new data only in reaction to actions initiated by the master side. In a curious cache, deciding whether to incorporate data is done on a per bus transaction basis based on the bus transaction as well as the current state of the snooping cache. A curious cache can insert data accessed by specific threads or processes, for example, the producer of data used by the curious cache's master. Curiosity can control the cache state associated with data being inserted. Curiosity can also prevent writes from occurring to lower levels of the memory hierarchy. Though curious caching is useful on its own, it was designed assuming some form of cache partitioning and, thus, works best in such an environment; throughout this chapter, we assume column caching support both from the master side and from the snooping side of the cache. For example, curious caching can have full retargeting ability on the snooping side.

When combined with column caching, curious caching also provides other valuable benefits such as reducing read latencies, improving message transmission and reception, emulation of buffers and RAM within the cache, improving stream buffers and enabling external prefetching. A cache supporting both column and curious caching functionality is called a Column/Curious Cache or CCC.

Curious caching was designed to address the high cost of reading data loaded or stored by another device, for example, another processor or a network interface. These reads are expensive because of large caches that cause deep memory hierarchies. This problem has been attacked repeatedly in the past, but with limited success. Most other approaches only handle one aspect of the problem such as message passing, ignore causes of the problem and often try to circumvent the cache. Curious caching, on the other hand, is a general solution to a wide range of problems introduced by large, deep cache hierarchies that uses caches rather than trying to get around them.

The next two sections discuss the problems addressed by curious caching and common extant approaches. Curious caching is then described in greater detail, compared to the extant approaches and contrasted with related work. A specific example demonstrates the power of curious caching. Specific uses and software control follow. The chapter finishes with implementation details and tradeoffs.

4.1 The Problem

Memory hierarchies are not well suited for communication. As we have argued in Chapter 1 and Chapter 2, memory hierarchies are already very deep and getting deeper. Deep hierarchies increase the distance to the memory bus, increasing latency for data that do not benefit from large caches, notably message data. Deep hierarchies are especially bad for receiving messages, since they generally must be read from the memory bus through all the memory hierarchy levels, incuring significant latency that is often difficult to mask.

Non-local reference patterns are in direct conflict with locality-based reference patterns; optimizing one within a shared mechanism will degrade the other. Local reference patterns need large caches that are best implemented with deep hierarchies. Non-local data patterns, on the other hand, work best with little or no cache and as little hierarchy as possible. If data changes often and moves between hierarchy stacks frequently, large caches are not necessary since there is little valid data to cache.

Non-local reference patterns have their revenge on caches, however, by polluting them. As soon as communication data is written, the producer will not reuse the data and therefore does not need to cache it. The same is true, though sometimes to a lesser extent, after the consumer has consumed received data. Standard replacement algorithms consider the used data most recently accessed and are thus likely to keep the data around, polluting the cache.

To minimize pollution the communication regions should be as small as possible. Small buffers, however, reduce elasticity in the communication and increase the amount of synchronization necessary to manage the buffers, potentially restricting performance since a slow consumer will hold up a faster producer. Keeping the entire communication buffer in cache will pollute the cache, reducing the amount of space available to data that should be cached. Thus, buffers should be large to minimize synchronization and maximize elasticity between producer and consumer, but the cache footprint in both the producer and consumer should be small to limit pollution.

Communication buffers can be implemented either in memory or in dedicated hardware such as a network interface. If the buffers are only accessible over the memory bus, at least two bus operations are needed to transmit/receive each cache-line of message data: one for the producer to write the data to memory and the other for the consumer to read the data from memory. Memory bandwidth can be significantly lower than SRAM bandwidth and therefore cause bottlenecks if the communication buffer is implemented in memory. Providing buffers for communication data (as is done in START-VOYAGER) in the network interface to eliminate memory bandwidth constraints, however, adds cost and complexity to the design and restrictions on buffering size, raising flow-control and synchronization issues.

4.2 Extant Approaches to the Problem

The latency issue for communication data associated with deep cache hierarchies is a significant problem that others have attacked with varying degrees of success. We discuss three approaches in this section: dedicated interface, prefetching and update protocols.

4.2.1 Dedicated Interface

Conceptually, the simplest approach is to avoid the memory hierarchy altogether by providing a dedicated communication interface on the processor[36, 59, 10, 25]. The interface might be in a special region of the physical address space in order to leverage virtual-to-physical translation mechanisms for protection. The interface would be accessed by load/store operations, their variants, or dedicated communication instructions. Buffer space provides elasticity in communication and makes the often-required software disciplines to avoid deadlock/livelock easier to implement. Providing dedicated buffer space for message composition and receive also avoids the copies required to compose and receive elsewhere.

Because a dedicated interface completely bypasses the memory hierarchy, the latencies and cache pollution problems are avoided, but there are significant disadvantages. A dedicated connection consumes processor pins, a critical resource[14]. Having the network interface share pins with a high-speed memory interface such as a back-side L2 cache interface[19] can create significant electrical design challenges since there is an additional drop on the signal lines connecting the processor and the cache RAMs. In the START-NG design[19], for example, adding the network interface reduced the speed of the L2 cache interface from 1/2 the processor clock rate to 1/3 the processor clock rate, reducing cache performance as well as network interface performance.

Another disadvantage is the inability of the dedicated interfaces to use the caches, especially for shared memory. Though a completely separate interface supporting shared memory can be provided, such a duplication of communication resources is undesirable.

Yet another disadvantage is the difficulty to migrate a communicating process from one processor to another. Outstanding messages in the network destined for the original processor must either be dropped, automatically redirected or the network must be drained prior to any migration. All such solutions are expensive to implement and/or impact performance.

Finally, there is the additional cost of buffer space. Large buffers reduce flow-control, synchronization and deadlock/livelock problems but also consume more resources.

4.2.2 Prefetching

Prefetching[56] the data into the cache before it is actually used is another approach that potentially eliminates the processor overhead of reads. In the best case, the data is available when it is needed. While the data is being prefetched, other work not dependent on the requested data, if available and economical to run, is done to hide the latency to memory.

There are hardware prefetching solutions and software prefetching solutions. Virtually all standard caches implement a form of hardware prefetching by having cache-line sizes larger than a single word that automatically prefetch the data around the requested data. More aggressive forms of hardware prefetching include stream buffers[43, 58] that automatically read ahead and adaptive prefetching that can dynamically adjust the amount of data prefetched[74].

Hardware prefetching can be quite effective, but is limited since it generally prefetches in reaction to master-initiated operations. Such prefetching tries to determine a pattern from the master's operations and prefetch the next data elements assuming that the master will continue to access memory in that pattern. If the pattern cannot be correctly determined or the master does not continue to access memory in the detected pattern, bandwidth and sometimes cache space is wasted prefetching data that will not be used. Some hardware solutions provide additional buffer space, eliminating cache pollution, but others that prefetch directly into the cache often pollute the cache.

Software prefetching inserts additional loads, sometimes in the form of special instructions, that anticipate future usage. If successful, software prefetching can reduce or eliminate the observed latency of memory reads. The technique, however, does require the issuing of additional load or prefetch instructions that in turn require bookkeeping overheads as well as additional load/store unit issue slots. To effectively prefetch, significant compiler/user analysis is necessary to determine *when* to prefetch. Since it often unknown exactly when communication data is produced, bandwidth and other processor resources may be wasted by speculative prefetches that are never used. Such failed prefetches pollute the cache as well. Prefetching without regards to cache space can displace other prefetched but not-yet-consumed data, causing ping-ponging[47].

4.2.3 Update Protocols

Rather than trying to guess when interesting data is created so that it can be prefetched, update protocols watch the bus for interesting data, then incorporate that data into their caches. Update protocols incorporate data that is already it its cache. Implemented systems with update protocols[71, 30, 44] tend to use write-through caches for data that is shared to allow the updates to occur. Such systems allow the data to be delivered to the consumer as soon as possible. A variant of the update protocol called *read snarfing* [60, 3, 24] uses a standard invalidation-based scheme, but allows updates to cache-lines with the same address tag but that were necessarily in the INVALID state. Depending on the reference patterns, however, such a scheme can reduce the effectiveness of the update protocol since it is possible that the invalid cache-lines will be replaced before the updated data is pushed out.

Update protocols can work well for communication data. If a consumer has the memory locations containing its communication buffer cached, the producer writing new data to that buffer will automatically propagate to the consumer's cache. Then, the consumer can receive the data directly out of its cache, effectively eliminating memory latency from processor overhead.

Update protocols, however, only update already-cached data. Latency-critical data, or at least their address tags, must stay in the consuming cache in order to get updated. Keeping a lot of data in the cache, however, will pollute it. Thus, pollution control, such as restricting the amount of cache used for communication buffers, conflicts with update protocols.

In addition, making writes to shared data write-through can negatively impact performance since, for many locations, there are many more writes to a location than are actually used externally. Single word writes, most common for write-through caches, often use buses inefficiently.

There are no widely-used systems that use an update protocol.

4.3 Curiosity Overview

A mechanism that addresses the communication data latency problem should have the following properties:

- Eliminate memory latency from the processor overhead of consuming data.
- Require little or no software support.
- Be compatible with the memory hierarchy, allowing its use with cache-line coherent distributed shared memory.
- Be compatible with cache pollution containment mechanisms in place.
- Minimize bandwidth requirements.
- Use sharable resources rather than dedicated resources.

Existing solutions fail to meet these goals. All optimize the communication case at the expense of the usual case. Dedicated communication interfaces either require additional pins or put an additional device on a specialized, point-to-point interface, generally slowing that interface down. Prefetching often wastes bandwidth by incorrectly guessing what data is needed and when that data should be fetched, often causing pollution.

The update protocol, on the other hand, fulfills some of the most important requirements. It eliminates the latency component of data consumption, it requires no software support, it is compatible with the memory hierarchy and it uses shared (cache and memory) resources as buffer space. It's major shortcoming is that it cannot incorporate data that is not already in the cache, making it difficult to use with pollution control mechanisms.

Curious caching, a new mechanism that allows the incorporation of snooped data *whose address need not already be cached in the snooping cache*, solves that problem.

Rather than relying only on address tags in the cache to determine what should be incorporated into the cache, curious caching provides additional hardware structures that allow software (or potentially other hardware) to specify what snooped bus transactions' data should be incorporated into the cache. A cache could be instructed to be curious about a range of memory or the memory accesses of a specific thread or process. Advanced forms might make curiosity dependent on bus operations, current cache tags, whether cached data was brought in because of curiosity, etc. If curiosity hardware indicates it is curious about a snooped bus transaction, the corresponding data is automatically incorporated into the cache. A bus transaction determined to be curious can be thought of as a read *that was initiated by another bus device.*

Thus, curious caching is a mix of prefetching and the update protocol. In prefetching, once it is decided to prefetch, the prefetch occurs, bringing the data into the cache. In an update protocol, any snooped data already in the cache is automatically brought into the cache at that point. Curiosity combines the benefits of update and prefetch by automatically incorporating data when it is inexpensive to do so (update), but not being limited to data that is already in the cache (prefetch). Curiosity betters both by specifying regions of curiosity rather than dealing with single cache-lines of data.

There are two extremes to hardware handling of curiosity directives: they can be viewed as performance hints that the hardware can ignore if desirable or necessary or they can be considered *imperatives* that the hardware must fulfill. The former makes hardware easier, the latter provides additional functionality such as memory emulation. Of course, software must be told whether to assume its specifications are hints or imperatives and act accordingly. The hardware itself is configurable to treat curiosity specification as a hint or as an imperative to enable additional functionality without forcing all regions of curiosity to be imperative.

To ensure fair resource usage, the running process, with the blessing of the operating system, controls that processor's curiosity. Often times, however, a producer knows more about what will be produced and/or what the consumer should consume, than the consumer. In these cases, the producer should be able to have limited control the consumer's curiosity. By simply exporting the ability to write to the curiosity control regions, the producer can modify curious caching structures on another processor. Each curious processor should be curious about snooped operations to its curious control region and accept commands that it snoops.

Curious caching was originally designed for a bus-based system, but can be adapted for point-to-point networks as well. We will discuss how in Section 4.3.6.

Rather than giving more functionality detail at this point, we now give an extended example that demonstrates and motivates various components of curious caching.

4.3.1 Curious Caching in Action

Consider emulated message passing between two processors on in an SMP. In a standard system, the processor overhead of transmitting a message is much lower than the processor overhead of receiving a message because transmitting fundamentally does not need to wait for a round-trip to the memory bus. Though transmits, performed by writes, do need to get permission to write either by a `read-exclusive` or a `dclaim` bus operation, aggressive systems can let writes continue while simultaneously obtaining permissions. A receive, however, cannot continue until the data is obtained, requiring a round-trip to the bus.

Assume a four-page region of shared memory that is used as a circular buffer between the producer processor and the consumer processor. The producer writes its data to the tail and updates the tail pointer to signal that data is ready. The consumer reads from the head of the buffer and updates the head to receive a message. The head and tail pointers are compared by both producer and consumer to determine whether there is more room to transmit messages in the buffer and whether there are yet-to-be-received messages within the buffer. When the end of the communication buffer is reached, the pointer wraps around to the front of the communication buffer. For this example, we ignore how the head and tail pointers are communicated between the producer and consumer.

An aggressive invalidation-based protocol will allow the transmit to proceed at speed, allowing the writes to proceed while the write permission is simultaneously obtained. Receives, however, must go to the bus since the transmit write has in-

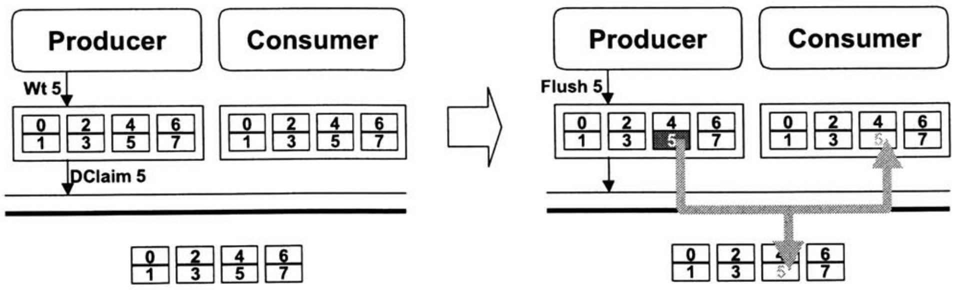


Figure 4-1: Simple Curiosity. Rather than having the consumer go to the bus to receive data, curiosity lets the data come to the consumer. When the curious cache sees software-specified "interesting" data, the curiosity inserts that data according to the replacement algorithm specified by the curiosity.

validated its cached copy of the data. Though aggressive coherence protocols allow cache-to-cache transfers, especially when the snooping cache is caching the data in MODIFIED state, the consumer must still wait for the data to return from the bus.

A simple form of curious cache can solve this problem, eliminating the processor overhead from receiving messages. Consider mapping the communication buffer to four cache pages in the producer and consumer, mirroring the entire communication buffer in both the producer's and consumer's caches (Figure 4-1). Make the consumer curious about the communication buffer region. After the producer produces data, it flushes that data out of its cache. Performing the `flush` operation is relatively easy, since the producer knows exactly when it completes a message. As the data is written back to memory, the consumer's cache snoops that data and the curiosity incorporates that data into its memory. When the consumer receives data, it always reads that data from its cache, eliminating the formerly-necessary bus transaction.

To eliminate the need for the producer to flush communication data as it is written, the producer maps the communication buffer to the minimum granularity of cache, for now a page. The consumer continues to map the communication buffer to four pages of cache and is still curious about the entire communication buffer. Assume for now that the producer is a couple of pages ahead of the consumer and that they operate at approximately the same rate (Figure 4-2).

Column caching limits the amount of cache the producer uses for the communication buffer, creating constructive interference (Section 3.4.2) that automatically

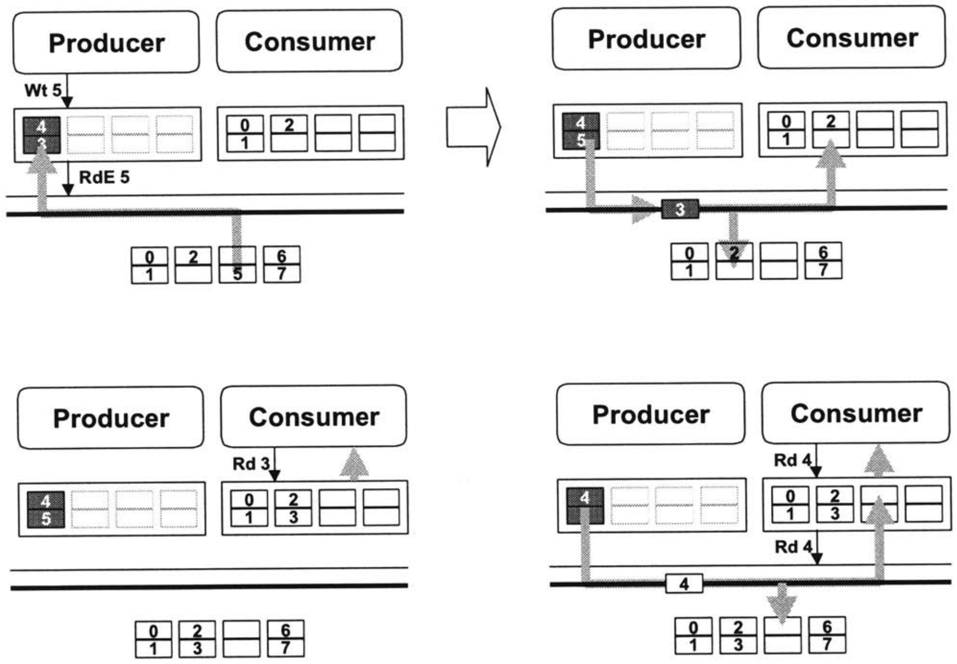


Figure 4-2: Improving Transmit with Column Caching. Rather than having the producer flush out composed data after composition, the communication buffer is mapped to a single page of cache. As the producer continues to produce within the circular buffer, new data will be brought in to be written, automatically pushing out older composed data. As that data is pushed out, it is captured by the curious cache.

pushes the produced data to the bus. Because the receiver is curious about all operations to the communication buffer, the producer's pushed out data will automatically be inserted into the consumer's cache. If the consumer maps the communication buffer to an equally-sized, unique cache partition, there will be no conflicts with other data and the entire communication buffer will fit into the receiver's cache as long as the producer is more than a page of data ahead of the consumer. Note that the produced data is still being written to memory as it is being pushed out of the producer. Thus, the consumer can always read communication data from memory, making the mechanism completely transparent, even if processes are migrated to other processors.

Thus, curious caching meets our requirements for a mechanism that eliminates the read latency from receiving processor overhead. Though the minimum latency is no shorter than the minimum latency of optimal prefetching, there is far less overhead and software bookkeeping than prefetching. In addition, there is no wasted bandwidth as is seen in most update protocols and prefetching. Curious caching not only effectively uses caches and the memory bus, it also seamlessly handles migration of consuming processes. In addition, it works well with column caching which, at the very least, allows pollution to be controlled.

4.3.2 Curiosity Parameters

A cache master explicitly configures the curiosity of its cache to ensure fair management of resources. Conceptually, curiosity is determined by a curiosity table that maps a set of *parameters* to *actions*. In the next few sections we discuss some possible parameters and actions in the context of their utility.

A simple form of curiosity allows the specification of *regions of curiosity*, regions of memory that should be brought into the cache by curiosity. When a curious cache snoops an address that falls within a region of curiosity is snooped, it is inserted into an appropriately selected replacement cache-line. The complexity of the replacement decision for such data is similar to that for a standard cache or a column cache.

Consider the case when the allocated cache space is smaller than the communica-

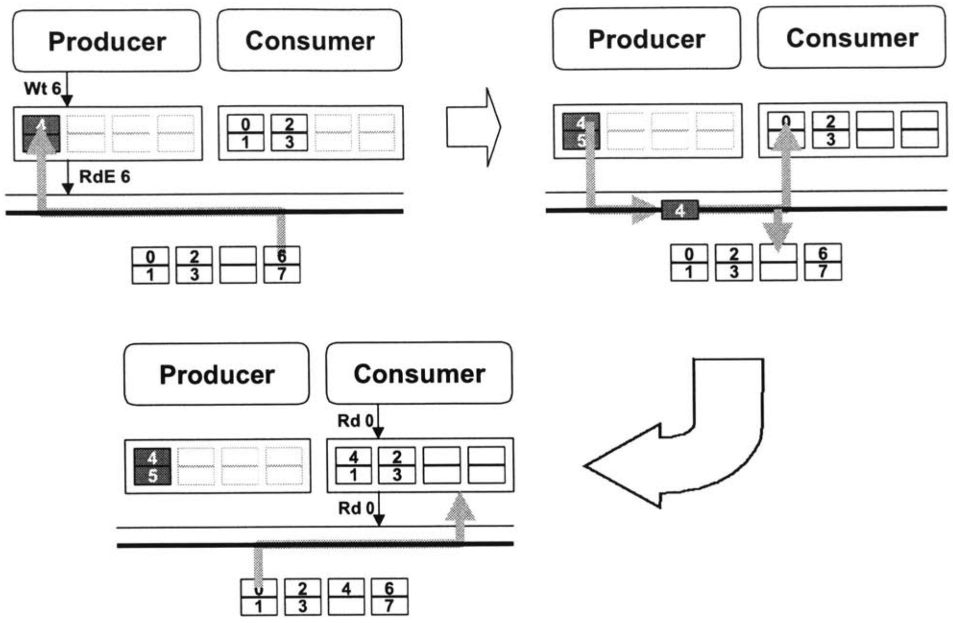


Figure 4-3: Basic Curiosity going bad. It is possible for basic curiosity (or poorly configured curiosity) to replace data that will be needed in the near future by data that will not be needed until later.

tion buffer. Assume that the consumer caches only two pages of the communication buffer (Figure 4-3). As the producer writes back data, the consumer's curiosity will bring that data into the cache that will overwrite older queue data, even if the older data has not yet been consumed. By the time the consumer reads the head of the queue, the desired data may have already been replaced by data from further in the queue.

The receiver can avoid this problem by changing the region of curiosity as data is received, being curious only about data that will be used in the future and that will fit in the allocated cache partition. This scheme, however, is clumsy and depends on curiosity granularity.

A more general solution makes the curiosity decision dependent on current cache state such as cache address tags, permission tags and LRU information. Basing replacement decisions on current cache state is done even in conventional caches that use replacement tags and permission tags to make replacement decisions. With a curious cache, however, specific cache state can result in curiosity hardware deciding to *not* be curious about a snooped bus operation, even though the bus operation address is within a curious range.

One way to avoid curious data thrashing is to make curiosity dependent on the availability of an INVALID cache-line within the allocated cache partition. Another way to avoid thrashing is to make curiosity dependent on a cache-line that does not cache a communication buffer location. In both cases, the receiver flushes each cache-line of data after it consumes it to free space for new curiosity data. Thus, two possible parameters are *permission tag state* and *address tag state*.

Under these schemes, as long as the consumer is less than two pages behind the producer and either the producer is flushing transmitted data or one page ahead of the consumer, the consumer will be able to consume out of its cache. If the consumer falls behind, the data will not be brought in to its cache via curiosity, but must be read out of memory. Thus, in the worst case, the protocol degrades to a standard system without curiosity. By careful tuning of the consumer cache space allocated to communication buffers, however, such degradation should rarely occur.

Since the producer's cache is guaranteed not to be caching addresses that it will produce, a `read-exclusive` is issued by most coherence protocols to get a full copy of the cache-line and obtain permission to write. Those read operations read data useless to the consumer and thus the consumer should not be curious about them. Rather than being curious about all bus operations issued to the communication buffer memory region, the consumer should only be curious about `pushouts`. Thus, another possible curiosity parameter is the *bus operation* of snooped bus transactions.

It may be easier for the consumer and also improve semantics to issue a *store* to the received data to indicate completion. Assume that the producer produces $n - 1$ words of data, where n is the size of the cache-line. In the first word of each cache-line of data, it writes a 1 indicating to the consumer that the data has been produced. The consumer writes a 0 to that first location after it consumes the data, indicating that the producer can reuse that address.

This "in-band" handshake between producer and consumer works very well with a column/curious cache. In this case, curiosity is dependent on both the region of curiosity and a replacement cache-line in MODIFIED state. Instead of having to perform a `flush` operation, the consumer just does his standard handshake write. After the consumer writes to indicate that it has received data, that data will automatically be pushed out by newly produced data being inserted by curiosity.

Weak memory models may allow `stores` or `flushs` to occur before the receiving reads complete, causing potential correctness problems. Such problems are common and are solved by strategically placing the memory barrier already required for handshaking.

4.3.3 Curiosity Actions

The most important curiosity action is, of course, whether the cache is curious about a bus transaction. Given that the cache is curious, there are additional actions that can improve curiosity and increase its range as a mechanism.

For example, if in-band handshaking is used it would be preferable for the receiver to get an exclusive copy of the data, rather than a shared copy that will need to be upgraded before being updated. Thus, the *permission tag* is a possible curiosity action. When curiosity inserts data into a cache, it can easily set other information such as the permission tags or LRU tags.

At first glance, making the permission tag for data inserted by curiosity dependent on the snooped bus operation seems reasonable. One simple policy is when a `pushout` is snooped, curiosity inserts that data in an exclusive state and when a `read` is snooped, curiosity inserts that data in a shared state. This naïve policy, however, is incompatible with standard coherence protocols. For example, if two caches are curious about the same region of memory and a `pushout` occurs, both caches will get exclusive copies of the data, generally considered a protocol paradox. Care in curiosity specification must be taken to avoid coherence paradoxes though the coherence protocols in systems that support curious caching should be as accommodating as possible.

Rather than hard-wiring the mappings of bus operation to new cache states, default curious caching makes states a function of the all parameters. Thus, in our example, the consumer could configure curiosity to obtain an exclusive copy upon snooped `pushout`. In other circumstances where an exclusive copy is not needed or not allowed, curiosity could be configured to obtain a shared copy.

There are benefits to having the *producer* be curious about the communication buffer. Curiosity can create constructive interference that helps push out the produced data sooner. For such a purpose and to save internal cache-bandwidth, curiosity may be configured to insert with an INVALID permission tag, allowing it to forgo inserting the data.

Another benefit to producer curiosity occurs if the producer maps the communication buffer to a large enough cache space. Then, the consumer's `pushouts` can be brought into the producer's cache so the producer's handshake read can be satisfied from the cache. The producer should be curious only about `pushouts` to the communication buffer to avoid inserting data that the consumer is reading.

Since caches are significantly faster than memory, it may be advantageous to use cache as memory to lessen the bandwidth demands on the memory and to reduce request latencies. By mapping a region of memory to the same size region of cache, the data will not be replaced once it has been entered into the cache. By making the cache curious about `pushouts` to that region of memory and having the curiosity insert a modified copy, the cache will capture all updates to the memory. If a `read` or a `read-exclusive` is issued by another processor, that cache will satisfy the read. Thus, with column and curious caching support, a cache can emulate memory available to all bus devices *that has no backing DRAM.*

Aggressive caches allow cache-to-cache transfer (called cache intervention) when one cache issues a request for data that resides in another cache in modified state, avoiding a write-back to memory .[1] If the requester issues a `read-exclusive`, the cache acting as memory sends the data to the requester in MODIFIED state, avoiding the necessity of a writeback to the sink (sink is used throughout this section to mean the next level of the memory hierarchy). If the read requested the data in shared state, the cache acting as memory will provide the data to the requester in shared

[1]Some processors such as certain MIPS R4X00 processors, allow a cache to satisfy requests if it has a clean copy of the requested data.

state while writing back to the sink.

Once the cache in question has data in shared state, however, the coherence protocol may not let it satisfy future reads. If the coherence protocol allows caches with a clean copy to satisfy a snooped request, there is no problem. This sort of protocol requires arbitration to decide which cache will satisfy the request.

To remove the arbitration requirement, an additional state, EXCLUSIVE-SHARED, indicating the cache should satisfy requests for the associated data, can be introduced. Only one cached copy of specific data can exist in that state. Curiosity insertion is the only way data can be associated with EXCLUSIVE-SHARED. The extra state eliminates the need for complex arbitration to decide which cache will satisfy a load.

Rather than encoding the EXCLUSIVE-SHARED state within each cache-line, increasing cache tag size, the information can be encoded within the curiosity state. An entire curiosity region would return data if that data is available in the cache. Thus, curiosity hardware would determine that its cache should satisfy the request and indicate that to the bus. This function logically belongs within the standard snooping hardware since it decides when to intervene and thus may be implemented there with the appropriate information from curiosity hardware.

Regardless of how a cache satisfies snooped loads, certain coherence protocols might still write to the sink during writebacks. It is often desirable to bypass the sink if possible, especially if the sink is memory. For example, the cache may implement memory regions that do not actually exist in memory.

The cache knows when it will be acting as memory and can assert a *sink-redirection-line*, called MODIFIED_P for cache intervention, to indicate to the sink that it should ignore that bus operation. Sink redirection is already supported for cache intervention, but only when one cache issues a `read-exclusive` for data cached in MODIFIED state in another cache. This same ability should be provided to curious caching. Sink redirection can be ignored by the sink but it might slightly alter curious cache memory semantics, depending on the supported cache operations.

For improved efficiency, a `clear` operation that deletes MODIFIED data from the cache without a `pushout` bus operation occurring can be provided. This instruction would be useful when remapping a region of the cache that was emulating memory. Rather than writing back values that are no longer useful, `clear` operations would clear them out.

4.3.4 Curiosity Parameters and Actions Summarized

Logically, a series of curiosity lookup tables are provided, one for each region of curiosity. Figure 4-4 gives three example curiosity tables for the consumer of our examples. In these examples, the specific table to use is associated with a region of memory. After it has been determined that the cache is potentially curious about a bus transaction due to its address, the corresponding table is applied. Table access can be implemented as a simple memory lookup operation. Since cache state may be curiosity parameters, one lookup per potential replacement cache-line may be necessary. The shown tables are simplified, giving only three parameters: bus operation, permission tag and whether the data in the cache-line is curiosity data. Only the curiosity action is specified in these tables. A C table entry indicates that the cache is curious about the data, while a blank entry indicates that curiosity should ignore the data.

The table actions are asserted only if the snooped address falls in the corresponding region of curiosity, the curiosity address is column-mapped to the cache-line and the bus transaction has not been retried. The table generates the possible action for a specific cache-line. It is likely that each cache-line meeting the initial conditions uses the same table to determine its possible course of action but different tables could be provided for different columns as well.

Obviously, additional parameters and actions make curiosity even more flexible. Every observable bus signal and current cache state is a possible parameter. Additional bus signals, such as an indicator of whether other caches are curious, may be introduced to further specify curiosity behavior.

Additional actions, such as sink-redirection, propagation to ancestor caches can also be useful to provide curious caching with more capability and flexibility. Providing separate bit vectors for curiosity data, for example, allows separation of data

1

		F	T	F	T	F	T	F	T
	current state	I	I	S	S	E	E	M	M
busop	read								
	pushout	*C*	*C*						

2

	curious?	F	T	F	T	F	T	F	T
	current state	I	I	S	S	E	E	M	M
busop	read								
	pushout	*C*		*C*		*C*		*C*	

3

	curious?	F	T	F	T	F	T	F	T
	current state	I	I	S	S	E	E	M	M
busop	read								
	pushout							*C*	*C*

Figure 4-4: Three possible curiosity tables. Table 1 indicates that the cache should be curious only if the snooped bus operation is a pushout and the current cache state is INVALID. This reaction is useful for a communication buffer where the receiver invalidates data after receiving it. Table 2 indicates the cache should be curious if the snooped bus operation is a pushout and the current cache does not contain data inserted by curiosity. Table 3 indicates the cache should be curious of the snooped bus operation is a pushout and the current cache state is MODIFIED which is useful for a communication buffer where the receiver writes the received data to indicate has completed its receive.

brought in explicitly by the master from data that is speculatively inserted due to curiosity. The separation of bit vectors for master data and snooped data are probably needed to allow parallel access anyways.

4.3.5 Pitfalls

Depending on the curiosity functionality supported, coherence paradoxes can arise. For example, a curious cache can be curious about a snooped read caching the data in modified state. This situation is useful if the reading cache is only a reader and will soon discard the data, while the curious cache will write the location after the reader discards. The situation, however, is a coherence paradox for a single-writer protocol. Of course, software should avoid paradoxical situations, but hardware should be implemented to handle any possible paradox gracefully. Thus, it is likely that either hardware protection will be provided, user-level software will be restricted to safe policies when directly accessing curiosity configuration, or software must use operating system calls that ensure safety to modify curiosity.

Cycles are possible with unrestricted curiosity (Figure 4-5). To avoid cycles, certain mappings will be disallowed. The simplest policy would be to have disjoint curiosity, that is, no two cache hierarchies can be curious about the same region of memory in the same way. Of course, the operating system can be called upon to ensure that no curiosity cycles exist.

4.3.6 Curious Caching with Memory Hierarchies

Hierarchical caches present additional challenges to curious caches. In machines where the cache hierarchy is filtered, that is a bus operation seen by a descendant cache may not be seen by an ancestor cache, additional support is needed to support curiosity in ancestor caches. Filtered cache hierarchies are always inclusive, that is, a descendant cache contains all data cached by all ancestors to maintain standard memory semantics while filtering.

Data that higher levels of the cache hierarchy are curious about should be sent

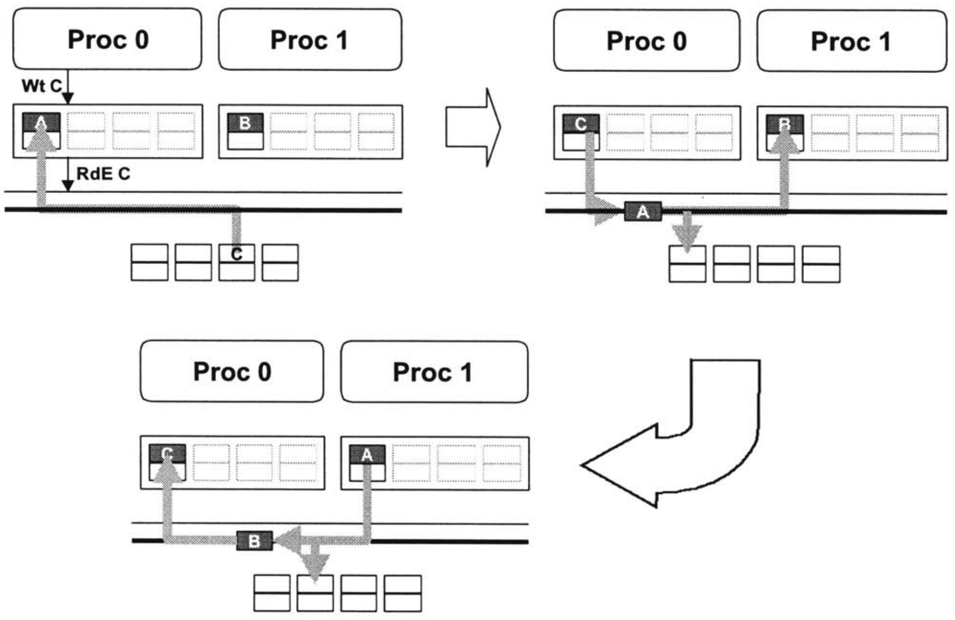


Figure 4-5: A curiosity cycle. Two caches are curious about the same region of memory and both allocate a single column to that region of memory. Assume that A, B and C are distinct addresses within the mapped region of memory that map to the same cache-line in both caches. A is cached modified in Processor 0's cache and B is cached modified in Processor 1's cache. Processor 0 writes C, forcing a `read-exclusive` of C, writing back A in the process. Processor 1's cache snoops the `pushout` of A and brings in the data, writing-back B in the process. Processor 0's cache snoops the `pushout` of B, bringing it in and writing-back C in the process.

up to those higher levels of cache. The simplest solution is to propagate all snooped operations to each higher level of the hierarchy. Doing so, however, eliminates filtering and increases bandwidth requirements.

To support filtering, a propagation bit indicating whether an ancestor is potentially curious could be associated with each curiosity table entry. The operation is propagated up only if the bit is asserted and the currently snooping cache is curious about the transaction. Since curiosity may depend on cache state, however, ancestor cache state must be duplicated in descendent caches to achieve full filtering. Because inclusive descendent caches must contain all data stored in ancestor caches and thus must have similar curiosity mappings, the fact that the descendent cache is curious with the propagation bit is likely to provide adequate filtering.

Distributed shared memory systems have a similar problem to hierarchical caches since each node will not see every bus transaction. Simple curiosity does not work on such systems, but the same sorts of solutions for hierarchical caches will work here.

In order for curiosity to work, potentially-curious bus transactions must be forwarded to a destination snoop-able by the curious cache. One way to do so is to incorporate curiosity hardware within the network interface that sits on each node's memory bus. That curiosity hardware acts as an agent for curious caches, forwarding interesting data to the appropriate curious nodes. The consumer, or an enabled producer, would specify to the curiosity hardware what to be curious about and where to send snooped data that met that criteria.

Curiosity hardware could also be placed at data home sites within a distributed shared memory system. Once data propagates back to its home site, it can be automatically sent to curious consumers. Similar solutions has been proposed independently by others[64].

4.3.7 Curiosity with Point-to-Point Data Networks

If data travels over a point-to-point network rather than a bus, the data will not normally be available to a curious cache. One possible solution is to add one more bus signal, CURIOUS_P, that indicates to the data network that a copy of the data should also be sent to the curious cache. The data network will have to support this functionality. Another possible solution is for the curious cache to issue its own memory access to get the data. Such a solution consumes both address bus bandwidth and data bus bandwidth and increases latency but can simplify hardware and automatically deal with data networks.

The point-to-point data network must also be able to handle the MODIFIED_P line as well, instructing the memory to ignore the data if the signal is asserted. Again, this signal can be ignored at the cost of a slightly different memory semantics.

Out-of-order data bus/networks are very common. Typically, bus operations in an out-of-order system will include a tag that is attached to data when it is returned. Handling out-of-order data tags for curiosity data is exactly the same as if the cache issued the bus operations itself. It must store the tags it snoops and match against those stored tags with the tag delivered with the data.

4.3.8 Related Work

A mechanism very similar to curious caching, called "cache injection" has been independently proposed by other researchers[52, 53]. This mechanism brings snooped data based on memory regions into special purpose buffers that are then incorporated into the cache if accessed. The special purpose buffers are intended to avoid pollution. Later versions of cache injection inject directly into the cache, potentially polluting the cache. Curious caching, however, uses column caching to avoid pollution as well as provide additional functionality, allowing more aggressive specification of curiosity without worrying about cache pollution. In addition, curious caching can use parameters other than just memory ranges to determine curiosity and allows the exportation of the ability to specify curiosity to others. These additional features enable a wider range of functionality than cache injection.

IBM patented[67] a mechanism called ALL-READ, a special load instruction that automatically inserts the loaded data into all snooping caches. This mechanism is similar to curious cache in that it has the ability to insert data into a snooping cache. Since all caches are loaded with the requested data, however, there is no ability to restrict pollution. In addition, other bus operations are not included.

The KSR-1[16, 31] parallel computer implemented a Poststore instruction, essentially a store that would immediately update other caches that were caching the same data. Poststore is essentially a write-through store and the automatic integration of Poststore written data is essentially update. Unlike curious caching, only the producer of data controls what data is updated in potential consumers' caches, rather than the consumer indicating what data would be useful. Also, the producer must use a special instruction to issue the write.

The original Avalanche design[18] injects data from the network interface into any level of the memory hierarchy depending on the current context being run on the processor. The design is not precisely defined in any published document and thus cannot be accurately compared to column/curious caching. This design can, however, place received data into the L1 cache, allowing very rapid access from the receiver and very low latencies. This design is actually a modification of a dedicated interface, since incoming data does not travel over the bus, but through a dedicated interface. Cache inclusion is probably difficult to maintain and wastes bandwidth, since a write to the L1 cache must be simultaneously reflected in lower levels of the cache to maintain inclusion.

4.4 Software Control

It is likely that user-level software will not use curiosity directly but instead will be given library calls that implement performance functionalities. These libraries would abstract away architectural details such as the number of hierarchy levels, the size of the caches and so on. These library calls would include calls that provide such functionalities as in-FIFO, out-FIFO, dedicated SRAM and critical shared memory. The performance, however, is not guaranteed, though additionally library calls may be provided to quantify the actual performance obtained.

A compiler can analyze programs to determine which regions of memory would benefit from curiosity. Generally, data that is actively shared between two threads or processes that may reside on different processors are good candidates for curiosity. Software writers can easily annotate explicit communication code such as message passing libraries. Examining data accessed between acquiring and releasing of locks in shared memory programs may yield memory locations that would also benefit from curiosity.

Data shared between some input I/O device and the processor is also a good candidate for curiosity. For example, making gigabit Ethernet buffers or read-buffers for a disk drive producing critical data curious could potentially improve performance dramatically. Such curiosity notations could actually be made within the device drivers themselves, avoiding the need for the compiler or the user to do the mappings.

The operating system provides arbitration and resource allocation for curiosity resources. The specification interfaces would be memory-mapped and protected by the translation mechanism, allowing the operating system to export control if desired and allowing the operating system to be backwardly-compatible with processors that do not support curious caching. Software with the right permissions could write to the curiosity tables, specifying mappings of parameters to actions for each region of curiosity. Hardware protection may be necessary to prevent software from specifying a curiosity configuration that may create a coherence paradox that cannot be handled by the hardware. An alternative to exposing curiosity configuration to software directly would be to have the operating system can provide configuration services, allowing it to check for illegal configurations in software.

When an application requests and receives direct curiosity control, the operating system returns a pointer that allows user-level manipulation of curiosity. That region will be mapped write-through in the reference implementation. If the operating system cannot allocate the curiosity resources to the application or provide that control to the application, it can notify the application of that fact. If curiosity specification is only a performance hint, the operating system can return the pointer that allows the application to specify curiosity mappings. When more resources become available, the mappings can then be used.

Like bit vector state, curiosity state is process state and must be maintained across process swaps. Doing so provides transparent (except for performance) migration from one processor to another.

4.5 Uses

Curiosity is especially useful to improve the performance of memory regions used for communication. Curiosity can also enable or enhance performance hacks. For example, a separate prefetch engine can be incorporated into the memory controller that then determines reference patterns and puts that data on the bus for the curiosity to incorporate into the cache. The memory controller is generally the bus arbitrator and thus knows when the bus is idle, allowing the prefetch to use only bus cycles that would be otherwise wasted. Another bus device can also provide external prefetching. For example, another processor could implement this functionality, executing codes that can more accurately model what will be needed in the future and prefetch those locations for the working processor. These prefetching techniques provide better performance than stream buffers since the data is brought directly into the cache while eliminating the need for the stream buffers.

As mentioned in Section 4.3.1, curiosity can also be used to force data out of a cache by being curious about bus transactions that will replace unwanted data or data that needs to be written-back as soon as possible. Rather than issuing separate `flush` operations to force out newly produced data, a producer can be curious about what the consumer was reading or pushing out. Since the consumer is reading data different from the data that the producer is currently producing, the producer will replace old produced data with newly consumed data. If the producer has sufficient cache space mapped to the communication buffer, that data brought into the producer in EXCLUSIVE state avoids the need for the producer to get permission to write that data.

Instead of moving data, curious caching enables the prefetching of the write-permission alone, allowing a device to push write-permission to another device. To do so, the producer should be curious about a `flush` operation (that must be propagated to the bus), incorporating a EXCLUSIVE state into its cache without getting the useless data.

A CCC can take advantage of many memory performance optimizations designed for distributed/parallel systems, such as memory-mapping, but can often do better than the hardware for which the optimizations were designed. For example, global memory management[29, 4] allows nodes in a distributed system to page to memory on other nodes, thus using remote memory as a disk substitute. Curiosity can do the same for caches, but more efficiently since no software involvement is necessary.

Curious caching can also provide additional functionality such as emulated memory. The SRAM/memory emulation capability of curious caching, however, is more powerful than a true explicit SRAM because (i) with appropriate modifications, it can serve as SRAM to *other* bus devices, (ii) it allows lazy swapping of the emulated SRAM and (iii) and exists in the same address space as the standard RAM. If the performance demands on that memory are reduced, the cache region can be remapped and curiosity removed without having to copy the data to standard memory as would be required with an explicit SRAM. If performance is not terribly critical, swapping in can also be done lazily rather than eagerly. Of course, if performance does matter, swapping can be done explicitly as well.

The same instruction set architecture is often shared between embedded processors and general-purpose processors, such as in the PowerPC family. A curious cache brings embedded and general-purpose processors closer together, allowing more shared development and enabling general-purpose processors to better emulate embedded processors. Such closeness can reduce time to market and give an accurate development environment for embedded processors that often appear after the general-purpose version. Putting a curious cache in an embedded processor or DSP allows better utilization of that valuable state, allowing the region to be dynamically partitioned between cache and explicit SRAM.

Processors-in-Memory systems implemented with curious caches instead of memory provide the benefits of PIMs with all of the performance and functionality benefits of curious caching. Though more costly to implement because of the cache structures, a CCC has significant advantages over a PIM. Memory can be easily migrated from one processor to another in a CCC since all the memory is implemented in cache. Communication can potentially become much faster. Multiple copies are much easier to support. Data movement is automatically managed by hardware but under software-defined policies.

4.6 Implementation

Like any other architectural mechanism, there are many ways to implement curious caching. In this section, we describe one reference implementation to demonstrate its feasibility.

Curiosity hardware transforms snooped bus transactions into a decision whether to incorporate and, if so, a set of curiosity actions. Snooped bus operations are transformed to curiosity tables using TLB-like structures for single page regions and BAT-like (Block Address Translation, essentially base and bound pointers, as provided in PowerPC architectures[51]) structures for larger contiguous address ranges. BAT structures could also be used to identify curiosity when addresses are not a parameter by specifying the full range of memory. These translation structures only determine whether a bus transaction's address is within a curious range while the corresponding curiosity tables further determine whether a cache is really curious based on other parameters. There are, of course, many actual implementations of curiosity determination.

It is important to have separate structures than the standard virtual-to-physical translation TLB/BAT for the curiosity TLB (CTLB)/curiosity BAT (CBAT) and the TLB/BAT not only for ease of implementation ease but also for additional flexibility. For example, a cache can be curious about an address that that is not yet mapped into its TLB or can have a different column bit vector (discussed earlier in Chapter 3) for data brought in via the curiosity mechanism than data brought in under explicit master memory operations.

For configuration, each CTLB and CBAT is assigned its own address regions that software reads and writes to read/write table state. To set curiosity parameters, the application simply needs to write the appropriate actions to the appropriate entries in the table. Each cache is also curious about snooped operations to the current curiosity mapping address space. In this manner snooped operations can reconfigure curiosity.

Translation between virtual addresses that user-level code would specify to the physical addresses needed by curiosity is necessary. Hardware support could be provided. Of course, operating system routines could be written that provide the necessary functionality.

The operating system manages the assignment of CTLBs and CBATs to processes. It is likely the CTLBs and CBATs will be managed in a partitioned fashion. Obviously, there may be more curiosity mappings than are possible due to limited hardware, making swapping of curiosity potentially desirable. Also, column caching (if provided) limitations on cache space and partitioning ability will probably restrict how much curiosity can be effectively used. To reduce complexity, the reference implementation of curious caching requires the operating system to perform all CTLB/CBAT swapping. When curiosity is swapped in, its curiosity mappings and the base pointer to the physical memory region that stores its curiosity mappings are restored. It is unnecessary to save curiosity mappings on a swap since they are already written to memory via configuration tables mapped as write-through. Curiosity swaps may not correspond directly to their owner process swaps, since curiosity can be useful even if its owner is swapped out.

It would be convenient to keep curiosity information within the page table entries. Unfortunately, doing so in a simple-minded way limits curiosity. Generally, there is a single page table entry for each virtual page within a process. In that case, each virtual page is associated with a single curiosity table. In the case of producer and consumer threads in the same process using the same virtual address to access the communication buffer, the same curiosity information will be used for both. Though such sharing is still useful, curiosity is more more flexible if separate structures are provided for curiosity data. There may be such a structure provided for each thread, rather than each process. That structure will be small since it will be limited to the amount of curiosity that the thread is allowed to specify.

As usual, configuring curiosity should be protected to prevent one process from overwriting another's curiosity specification. Especially important to protect is the ability to specify curiosity permission tags, redirection and the curiosity bit vectors. The same sort of protection mechanisms provided for column caching can also be provided for the specification of curiosity bit vectors. As described in Section 4.3.5, complex situations can arise if permission tags and or redirection are used incorrectly. Operating system oversight may be essential to avoid the most complex difficulties. Operating system oversight would also facilitate the building of data structures necessary to track and adjust for any demand paging activity.

Rather than using the familiar associative CTLB structure, a hashed TLB can be used instead. Since the snooped bus is slow compared to the processor, a simple hash function is affordable and could enable a much large CTLB.

4.6.1 Hardware Overview

Hardware support for curious caching consists of three basic components: determining whether the cache should be curious about a particular bus operation, informing the bus of the decision (if necessary) and incorporating the data brought in by curiosity. Curious caching can be implemented on top of existing snoopy cache mechanisms (Figure 4-6 shows standard snooping hardware). Logically, there is a curiosity table associated with each CTLB/CBAT entry. If a CTLB/CBAT determines that a snooped address is within a curious region, the other parameters are passed to the corresponding curiosity table which is accessed using those parameters to determine a curiosity action, if any.

To improve lookup speed, it is possible to split the CTLBs into CTLB$_a$s and CTLB$_b$s and the cache replacement unit, CRU, that uses CTLB units-generated information to determine replacement (Figure 4-7). In addition, the Bus Interface Unit (BIU) needs to be modified to inform the bus of its cache's curiosity and to accept curiosity data.

The two CTLB units provide information to determine if the cache is curious about a snooped bus operation. The CTLB$_a$ is accessed with the bus operation and snooped address in parallel with the cache-line state (that is read for snooping anyways) and produces a lookup-table specifier. The CTLB$_b$ uses the read cache-line state (potential replacement lines for the snooped data) to access the lookup-table specified by the CTLB$_a$, producing actions for each possible replacement cache-line.

The CTLB is split into three parts to improve performance: CTLB$_a$, CTLB$_b$ and the CRU . The lookup part of the CTLB is split into two parts, the CTLB$_a$ and the CTLB$_b$. The CTLB$_a$ is accessed right after a bus transaction is snooped with parameters from the bus transaction. The CTLB$_a$ produces a lookup table specifier that is used as a pseudo-parameter for the CTLB$_b$. Parameters that are available after the CTLB$_a$ is accessed, such as cache state, are looked up in the CTLB$_b$ when they become available. Of course, if no read state such as cache-line state is a curiosity parameter, there is no need for the CTLB$_b$. The CTLB$_b$ may need to provide a lookup table for each column if cache-line state are curiosity parameters and speed is essential. The results are combined in the CRU. Of course, the CTLB$_a$ and CTLB$_b$ may be combinable depending on the parameters and the timing[2].

The possible replacement cache-lines along with their associated actions generated by the CTLB$_a$/CTLB$_b$ are passed to the CRU that then determines whether the cache is curious and, if so, which cache-line to replace. If the CRU determines it is curious about a snooped bus operation, it notifies the BIU of that fact. The BIU notifies the bus that its cache is curious (if necessary) and coordinates moving the curious data into the cache when the data becomes available. If BIU buffer space is unavailable, the snooped bus operation can either be retried or, if curiosity is only a hint, that particular data is ignored.

Some possible parameters, such as whether data was inserted due to curiosity, could be stored in the cache rather than performing the corresponding lookups in the CTLB$_b$. Such a design can sometimes avoid extra lookups, but will also require

[2]If a low-associativity column cache is used, the cache access needed to access the CTLB$_b$ may have to wait until the CTLB$_a$ produces a cache address or series of cache addresses.

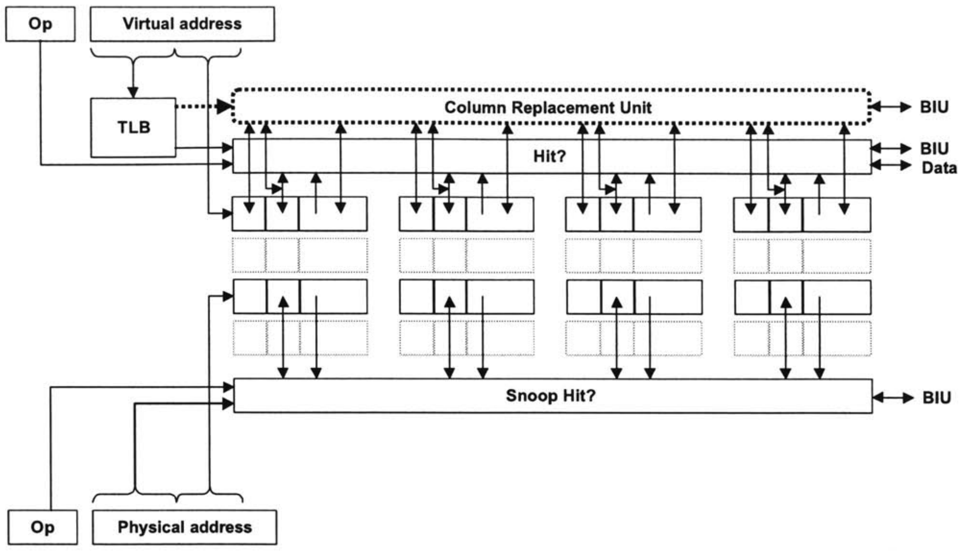


Figure 4-6: Standard Snoopy Cache Address Structure (assuming a column caching replacement unit).

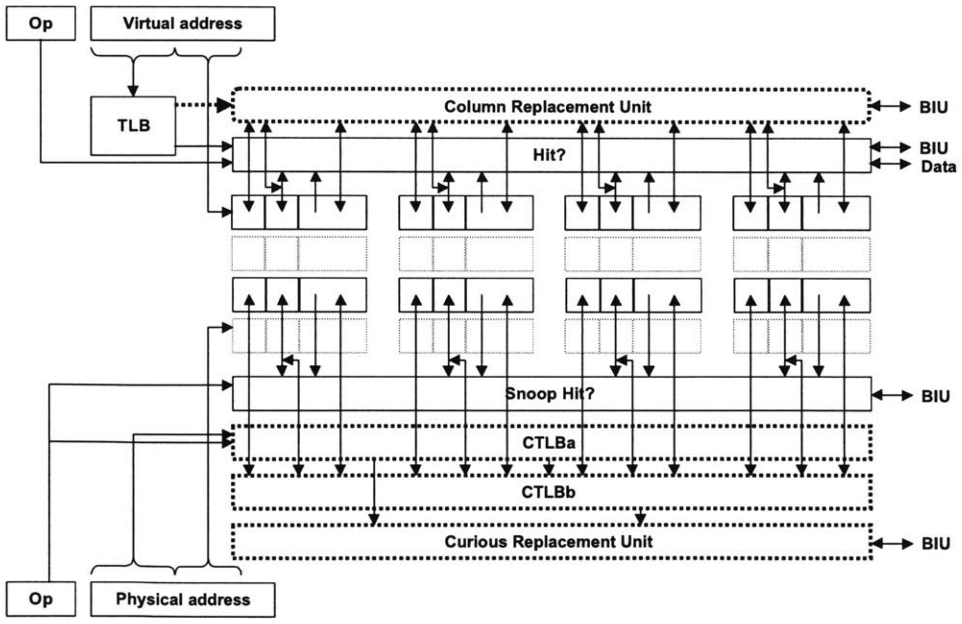


Figure 4-7: Curious Cache Address Structure

more bits in the cache and support to handle changing curiosity mappings within the cache. In addition, the CRU may become more complex to accommodate and thus may negate any savings.

There are, of course, alternate implementations. Rather than putting curiosity detection within the cache, the cache could have an input indicating whether to be curious about the current bus transaction. That input could be a single bit or a number of bits indicating how data should be brought in. Input bits may also specify a curiosity *hue* that is looked up within a simple table that specifies curiosity and curiosity actions. Such an implementation allows implementations of curiosity outside of the cache, greatly simplifying the hardware within the cache and enabling potential sharing of that hardware between several caches. For example, hues could be stored as part of the cache state, eliminating the need for CTLB/CBAT structures altogether and only requiring additional support during remapping of curiosity.

4.6.2 BIU Modifications

The BIU needs to be modified to (i) provide additional signals to the bus to convey necessary curiosity information and (ii) provide the hardware needed to handle the incoming data due to curiosity. Both are straightforward.

Our default implementation assumes a simple in-order broadcast-based data bus. We discuss designs that accommodate out-of-order, point-to-point network-based data buses in Section 4.3.7.

Bus Signals

In our default implementation, only MODIFIED_P one-bit bus signal to signal sink redirection needs to be added. Its assertion indicates that the data will reside in modified state within the cache, eliminating the need for memory or another cache to keep a copy of the data. Similar functionality is already supported in high-performance buses that support intervention such as the PowerPC 6XX. In those cases, the signal indicates that the data will be passed from a snooping cache that contains the data

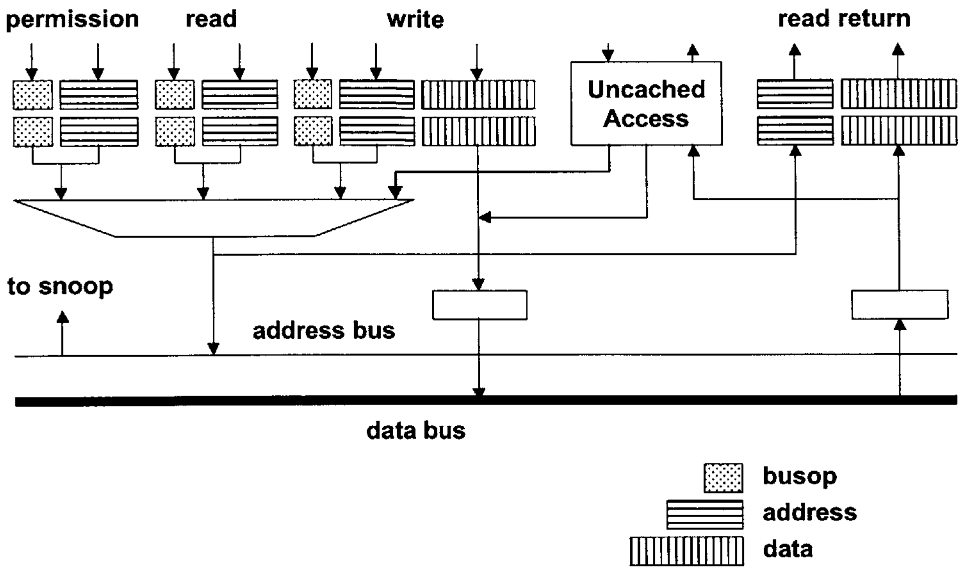


Figure 4-8: Unmodified BIU

in modified state to a cache that is requesting an exclusive copy of the cache-line. The sink, therefore, is not required to make a copy, since the requesting cache has the most up-to-date copy and will ensure it is either written back or passed to another exclusive read.

The MODIFIED_P signal can be implemented as a bussed signal that is asserted only if a cache will take a copy, otherwise it remains in tri-state. If point-to-point electrical characteristics are desired, a combining circuit can be used. Either way, the signal is an input to memory.

Handling Curious Data

A simplified, unmodified BIU is shown in Figure 4-8 while a curious cache BIU is shown in Figure 4-9.

When CRU indicates that the cache is curious about the data and passes that intention to the BIU, the BIU must allocate a buffer to store the relative information until the data returns. This buffer, which we call a curious buffer, is virtually identical to a standard read buffer. The standard collision detection mechanisms (either in the master or in the snooper, depending on the implementation) that ensure two bus operations to the same cache-line are not simultaneously outstanding conveniently

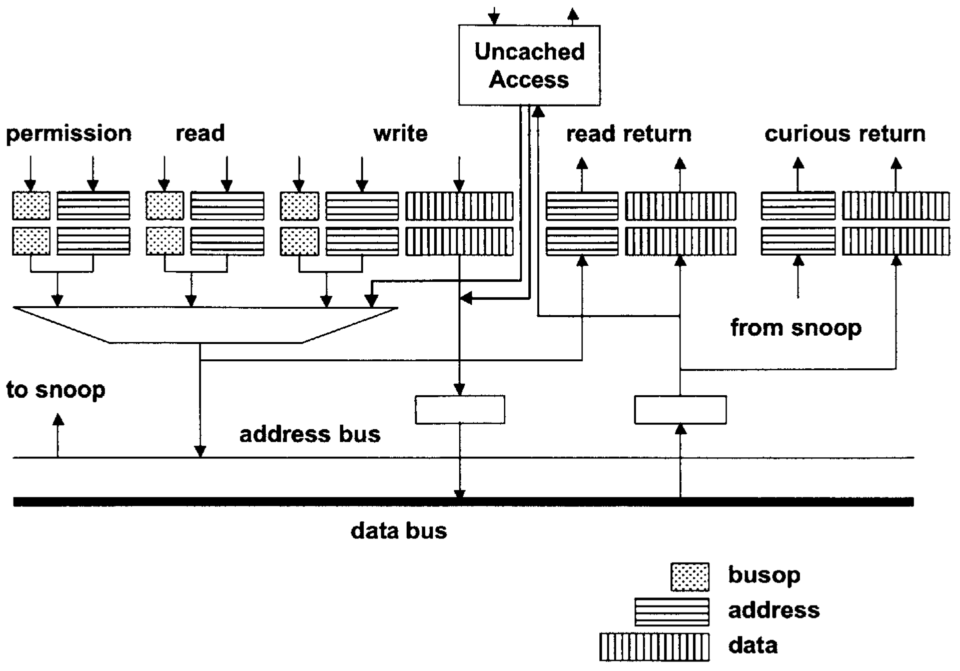


Figure 4-9: Curious Cache BIU

prevent ancestor caches from issuing loads to the same data that is being brought in via curiosity. The buffer must be able to handle a retry of the curious bus operation at a later time if the bus protocol supports it.

The only difference between the standard BIU path and the curiosity BIU path is that completion of the bus operation does not necessarily signal an outstanding load/store that it can continue since there might not be a load/store issued by a master for that data.

When the curious data appears at the cache's BIU, the BIU moves the data into the cache when possible. If the cache master had issued a memory operation to the same cache-line and the permissions are correct, it is possible to service that memory operation with the curious data as if the memory operation issued the curious bus operation.

4.6.3 Impact of Curious Caching on Clock Cycle

Curious caching snooping operations occur in parallel with standard snooping. They do, however, potentially take a bit more time than standard snooping because an additional lookup (CTLB_b) takes place before curious snooping can complete and deciding on actions is potentially a bit more complex. If the timing impact of the CTLB_b is too large, the information it generates should be made part of the cache lookup and additional support to flush or change that information when curiosity mappings change should be provided.

Since caches generally run some 2 to 5 times faster than the buses they snoop, there is generally ample time to perform such operations when the cache is snooping a bus across a chip boundary. If the cache is snooping traffic that is being generated on the same die, additional pipeline latency may occur. If the additional latency is significant, curiosity can be asserted eagerly, that is, curiosity can be indicated to the bus but the data not necessarily incorporated. During the time the data is being returned, the curious cache can positively determine if the data will be inserted into the cache. Thus, there should be no impact on the clock cycle.

4.6.4 Additional Support

Additional support can assist curious caching. Block operations, such as a block invalidation, are very useful to curious caching, especially to force a region of memory to be written back so that they can be brought into another cache via curiosity. Such operations are easy to implement and can also provide benefits to standard codes. Those block operations can also be implemented outside of the processor/cache by placing them on the memory bus.

With hardware support, a producer can start to write data before permission to write that data is obtained. In a standard coherence protocol, the permission can be obtained in parallel and the written data combined with the rest of the cache-line of data when it is available for writing within the cache. Another solution could allow a page to be specified "write without permission". The reader would be responsible to invalid its cache entries to read new data.

A variant of this approach is supported by the Commit/Reconcile/Fence (CRF) instructions[65] A store-local instruction is provided that allows the processor to store to cache; permission or the cache-line of data does not need to be obtained first. This store would take the place of the page specifier “write without permission”. This CRF instruction set is also capable of implementing most memory models with no additional hardware support.

Chapter 5

Putting It All Together

This chapter examines how column and curious caching might be used to improve performance, simplify design complexity and reduce resource requirements for a wide range of applications from standard sequential codes and multitasking/multithreaded execution to message passing in parallel systems like StarT-Voyager. We also describe how StarT-Voyager can efficiently emulate column and curious caching and provide a test-bed to try various implementation ideas.

Column/curious caching can significantly improve performance and reduce part counts. For example, curious caching reduces StarT-Voyager message-passing receive processor overhead time by almost 50% assuming a memory latency of only twenty cycles. Column/curious caching would also reduce the non-trivial part count of the StarT-Voyager network interface by 50%. We observed in some cases that column caching can achieve the same cache hit rates as an LRU cache with 40% less cache space or improve hit rates by at least 10% with the same size cache. Column caching also enables more flexible process/thread scheduling while keeping cache hit rates high. We detail each of these results in the rest of this chapter.

5.1 StarT-Voyager with Column/Curious Caches

StarT-Voyager would have been easier to design, implement and achieve better performance had it been built around processors that supported column and curious caching. This section describes StarT-VCCC, a modified StarT-Voyager that assumes such caches.

5.1.1 Simplify Design

StarT-Voyager's network interface included buffer space for cached memory-based message composition. The network interface must be prepared to buffer cache-lines that may have been prematurely evicted. Such premature eviction requires the ability to satisfy requests for the same data, making some sort of RAM essential for buffering messages. A FIFO is not sufficient. Premature eviction also eliminates the possibility of using bus transactions to signal that messages have been composed or consumed since the number of bus transactions per message is not constant.

A common solution is to buffer in memory. The network interface reads composed messages from memory, potentially getting data via intervention from the cache, and writes received messages to memory for the processor to receive. This solution eliminates the need for RAM on the network interface but requires additional bus crossings for receives and may incur additional bus crossings for transmits if messages are not read from the cache. In addition, DRAM bandwidth is consumed and DRAM latency is incurred, potentially limiting performance. Putting a cache on the network interface, an approach taken by the current Utah Avalanche[70] design, eliminates bus crossings DRAM impact in the normal case but does complicate the design considerably to include the coherent, dual-ported cache. Neither of these solutions, however, affect the possibility of premature eviction and its implications.

A column/curious cache addresses all these design/implementations issues, improving performance at the same time. Assume communication buffers backed by memory. By mapping each active communication buffer into a dedicated cache partition, the possibility of premature eviction is eliminated assuming that messages are composed/consumed sequentially and the cache partition is large enough to avoid premature eviction due to speculative memory accesses to the buffer itself. Without the possibility of premature eviction, it is possible to use a simple FIFO rather than a RAM for transitory buffering of messages within the network interface. By using a smaller cache partition than the entire communication buffer, constructive interference will push composed/received messages out as new messages are being composed/received.

If the cache partition is curious about their respective communication buffers as well, the network interface can push write permissions and received data into the producer/consumer's caches, respectively. Thus, it is possible to eliminate on-demand bus transactions to transmit/receive a message. From the processor's perspective, all such operations will be performed out of the cache, eliminating the need for aggressive write-buffers for transmits and dramatically improving receive performance.

Standard handshaking can still be used to signal the network interface aggressively to move data in/out of the cache from/to the network interface. Since there is no chance of premature eviction and the resulting extra bus transactions, however, it is possible for producers/consumers to use implicit bus transactions to indicate composition and consumption completion. In other words, producers/consumers do not need to explicitly indicate completion of their tasks; the implicit bus transactions created by continued execution of their tasks can interpreted by the network interface as a task completion signal.

For example, if the network interface sees a `read` bus operation for an element in the receive buffer, it knows that the previous messages have been received and thus can push additional messages into the cache via curiosity. The same applies for the transmit queue. Of course, such a scheme needs to be augmented to work with messages that are not continuously being transmitted or received or if there are dependencies between message transmits and receives.

Because the message queues are backed by memory, there are no correctness issues. In the worst case, performance degrades to a standard memory-based queue design. In the usual case, however, performance will be significantly better, design/implementation complexity significantly reduced, and new lower-overhead message mechanisms such as the reduced-handshake scheme described about can be supported. We discuss the first two points in the next two sections.

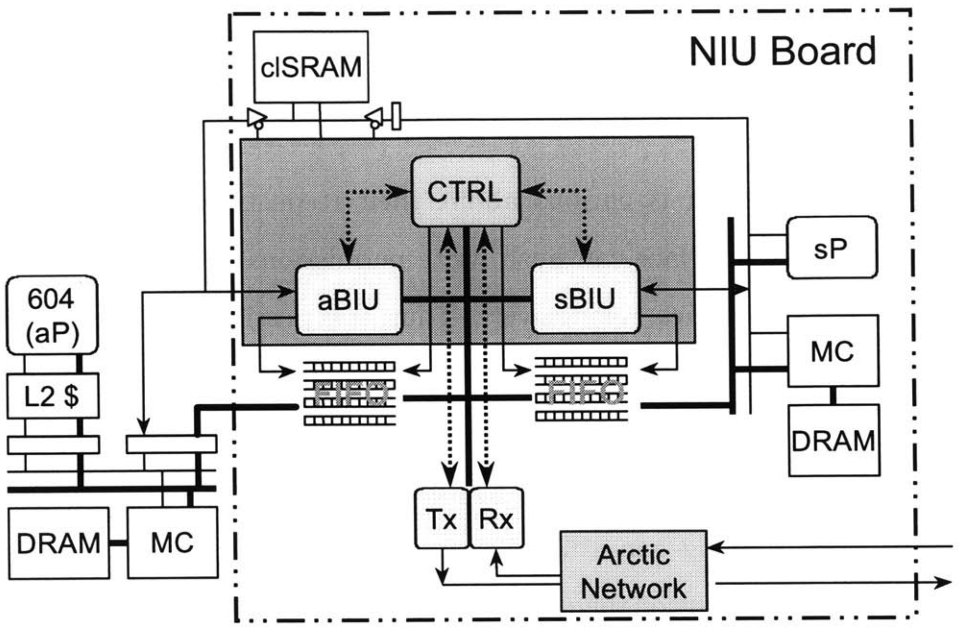


Figure 5-1: StarT-VCCC

5.1.2 Actual Differences in StarT-Voyager

StarT-VCCC is shown in Figure 5-1. The biggest visible difference is the part count reduction by going from dual-ported SRAMs to FIFOs. StarT-VCCC does not require dual-ported SRAMs. A pair of transmit FIFOs and a pair of receive FIFOs would be sufficient, allowing high priority bypass of low priority messages. An additional pair of transmit address FIFOs needs to reside in each of the BIUs. Another pair of FIFOs between the aP and the sP would be desirable to minimize latency between the two processors, though is not necessary as the high priority buffers could be used. These FIFOs could make the Tx-Rx FIFOs redundant, especially if the clock boundary crossing is done in the added FIFOs.

Another benefit is reduced BIU state and complexity. The BIUs do not have to track SRAM addresses for each of the supported queues. For transmits, the BIU simply needs to know a range of physical addresses that correspond to valid transmit queues and a transformation from the physical address to the queue number. If it sees a write to one of those addresses and the queue is active, it enqueues the written data into the appropriate transmit FIFO. Depending on the type of transmit queue, the BIU passes either the address and bus operation to CTRL or some condensed information via the appropriate address FIFO.

For receives, the BIU simply writes the data to where CTRL tells it to write the data. This write is identical to an shared memory write of data back to DRAM. If the receive queue is active and in the cache, the data will be brought into the cache via the curiosity mechanism.

A third benefit would be hardware support of many more queues than are currently possible on StarT-Voyager. Buffer space and to a lesser extent BIU state, were the limiting factors for number of queues and queue size in StarT-Voyager. By moving buffer space to DRAM, many more queues can be efficiently supported. Queue state within CTRL is implemented as RAM and could easily scale to support significantly more queues. Caches are configured to store the "hot" queues while the others operate like a standard DRAM-buffer-based machine.

Since the number of supportable queues can grow a significant amount, non-resident queue support may not be be necessary. If they are, however, hot swapping queues becomes much more elegant since resident and non-resident queues are handled almost exactly the same way. Rather than swapping queues to/from the dual-ported SRAMs, only the memory region recognized as a transmit queue must be changed. If pushouts signal that a message has been composed, the sP should be notified of such an event for a non-resident queue. Transmitting to and receiving from non-resident queues proceeds as with the original StarT-Voyager.

To summarize, the benefits of column and curious caching to the design and implementation of a parallel machine like StarT-Voyager are as follows.

- Elimination of premature eviction enables
  - FIFOs replace RAM for fast buffering.
  - Processor-to-network-interface signaling to be done implicitly in bus transactions.
- Reduction of control complexity.
- Increase in number of queues.
- Simplify non-resident queue handling.

5.1.3 Improving StarT-Voyager Message Passing Performance

Processor overhead on StarT-VCCC is dramatically better. Consider StarT-Voyager's Basic message passing mechanism. A single message transmit requires

$$43 + 2 * num_{words} + 5 * num_{CL} + 1\ store_{uncached}$$

instructions to execute while a receive requires

$$27 + 2 * num_{words} + 5 * num_{CL} + 1\ store_{uncached}$$

instructions. The numbers are not exact because we do not count code, such as aggregated pointer reads, that are not executed for every message library call. Of course, since these are instruction counts and not cycle counts, cache miss penalties are not included.

To generate numbers comparing StarT-Voyager and StarT-VCCC performance, we made the following assumptions: (i) one instruction is issued per cycle, except for memory operations, (ii) there is sufficient buffering to avoid blocking on writes and (iii) no memory barrier instructions are required. Under these assumptions, transmit overheads and receive overheads for standard caches (20 cycle and 100 cycle hierarchy latencies) and a curious cache are given (Figure 5-2). The StarT-Voyager system has about a 30 cycle memory latency. Current processors have a 100 cycle memory latency. We do not account for a performance advantage of StarT-VCCC, notably that the network interface pushes pointer updates to the processor via curiosity eliminating the memory latency component of pointer reads.

In particular, note that with curious caches, message receives are faster than the corresponding message transmits, reducing the chance of overflowing queues and blocking the network or invoking flow-control or dropping and retransmitting messages. In addition, since StarT-Voyager's message passing bandwidth is limited by processor overhead[5], StarT-VCCC's significantly lower processor overheads make substantially higher bandwidth possible (Figure 5-3).

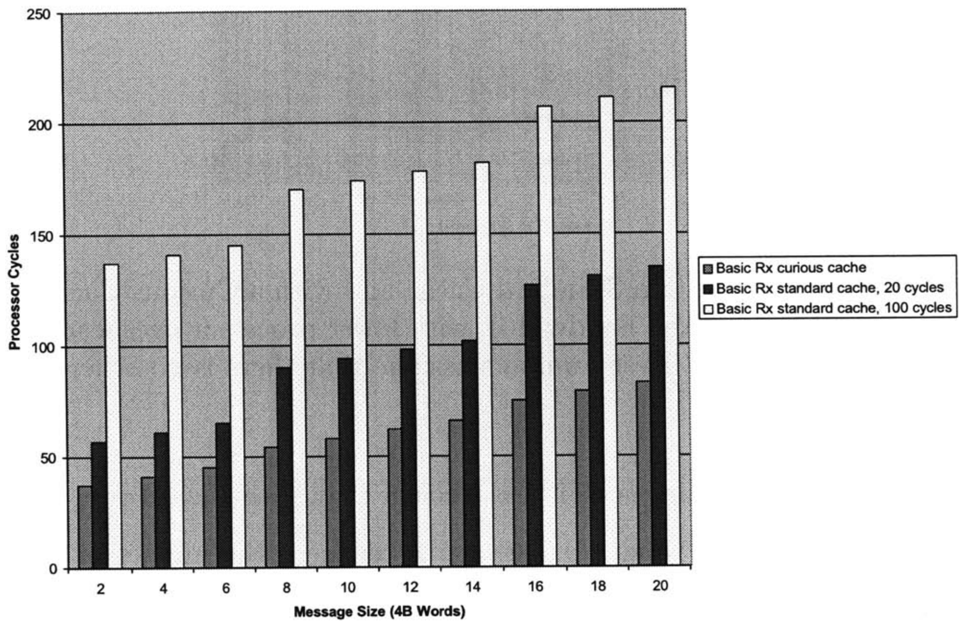


Figure 5-2: Receive Overheads with a Standard Cache and with a Curious Cache. The numbers presented here are generated using the instruction counts and an estimate (generated from the real system) of a 20 cycle penalty for a cache miss. A cache penalty of 100 cycles is also presented. It is assumed that the cache is aggressive and can issue multiple overlapping loads, allowing subsequent cache-lines to be accessed 16 processor cycles (4 bus cycles per operation, 4 processor cycles per bus cycle). A curious cache allows messages to be pushed directly into the cache, avoiding cache misses for receives and thus achieving a much lower receive processor overhead.

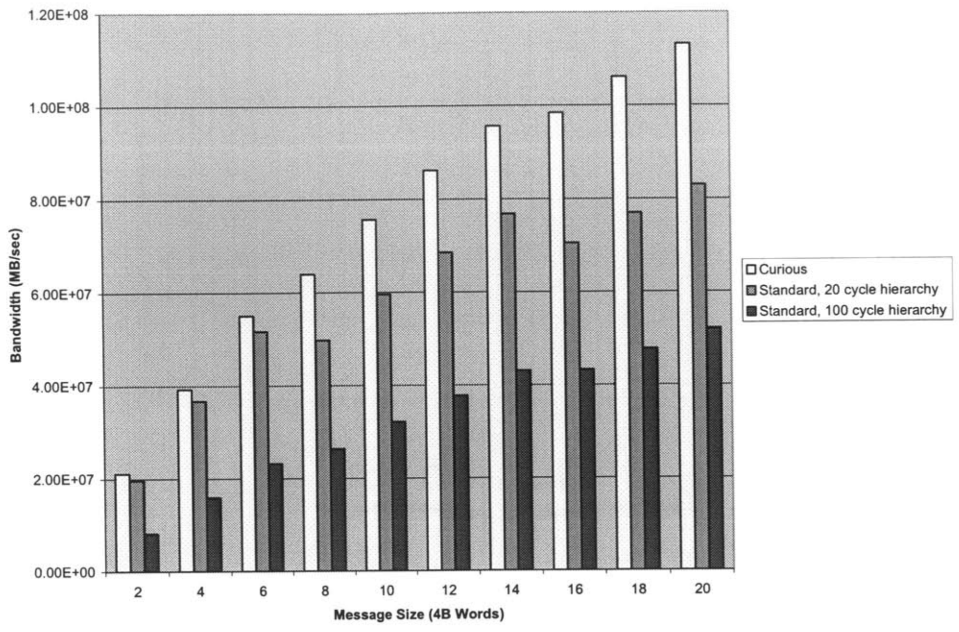


Figure 5-3: Bandwidth with a Standard Cache and with a Curious Cache. Since processor overhead bottlenecked bandwidth, with lower processor overheads, bandwidth correspondingly goes up. These numbers assume that there is no other bottleneck in the system.

General message handling code is less efficient than specialized code. If the number of instructions needed to receive a message is reduced, the impact of cache latency will increase since there memory latency will consume a larger fraction of the overall overhead and there is less opportunity to overlap computation with the memory latency, further increasing the benefit of curiosity compared to standard caches.

If bus transactions are used to signal handshaking events as described in Section 5.1.1, synchronization can be eliminated. Even if handshaking is explicit, synchronization can be minimized or eliminated by exploiting the limited speculation and reordering inherent in superscalar processors. These techniques can further reduce processor overheads. Of course, they increase latency and are thus not appropriate for certain forms of message passing.

Column caching can restrict cache pollution, further improving performance by improving hit rates. The actual performance improvements are dependent on what else is running with the message passing routines.

5.2 Emulating Column/Curious Caching on StarT-Voyager

Ideally, processors and memory hierarchies should allow programmability of their communication and storage components. A program should be able to decide what data gets cached, using what policy in what part of the cache, and what information goes out onto the bus when. Column and curious caching are mechanisms that are easily supported by such hardware. Unfortunately, such control does not currently exist in current commercial hardware.

The StarT-Voyager network interface connects to the MIT Arctic network[13], giving it the ability to communicate with other StarT-Voyager sites. With the network capability emulating a bus or a network and each StarT-Voyager site emulating a single processor (see Figure 5-4), StarT-Voyager is an excellent platform to research advanced processor/cache mechanisms such as column and curious caching, or other storage and communication mechanisms.

StarT-Voyager emulates smart memories by watching all bus operations and performing programmable actions based on those observed bus operations. For example, StarT-Voyager has the ability to implement an S-COMA cache, a large L3 cache whose data exists in memory and whose tags are maintained in clsSRAM.

Using this S-COMA support, we can accurately emulate column caching by slight changes in the coherence protocol currently implemented by the sP (Figure 5-5). In normal S-COMA, pages are the unit of allocation, but cache-lines are the unit of coherence. Thus, each page being accessed in S-COMA space is allocated a physical page frame in local memory. Replacement decisions are only made when a new page is accessed and needs to be mapped into local memory. Then, an old page needs to be vacated to make space for the new page.

By having the coherence protocol artificially limit the number of valid cache-lines within a specific region of memory to the number of columns where that region of memory is mapped, column caching is accurately emulate column caching. Of course, this limits the amount of space in the S-COMA cache, potentially significantly. It

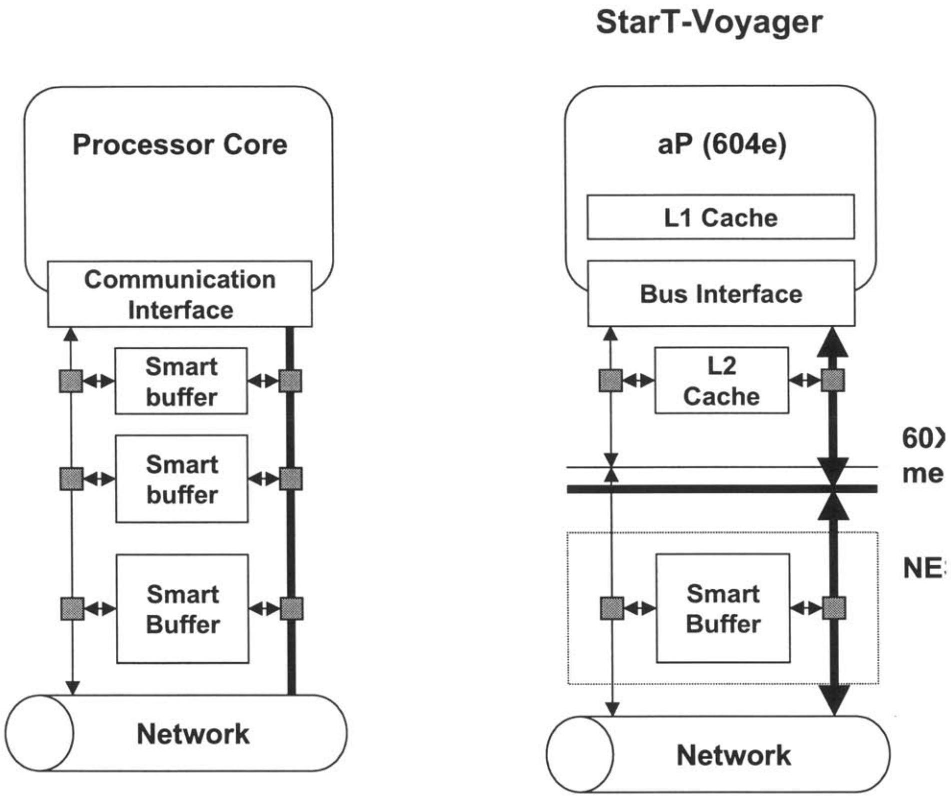


Figure 5-4: StarT-Voyager site seen as a single processor with flexible caches and bus interface unit. FPGAs provide hardware programmability, while an embedded processor provides firmware programmability. With its programmable ability to interpret and respond to any bus operation in an arbitrary way, StarT-Voyager is able to emulate storage and communication control. StarT-Voyager could easily emulate curious caching. Using the Arctic network to emulate a bus, each StarT-Voyager site is now a processor sitting on a bus. Using firmware, data that the site is curious about can be brought into the system and written to local DRAM. StarT-Voyager is also able to effectively emulate column caching with a specialized replacement algorithm and leveraging existing S-COMA support.

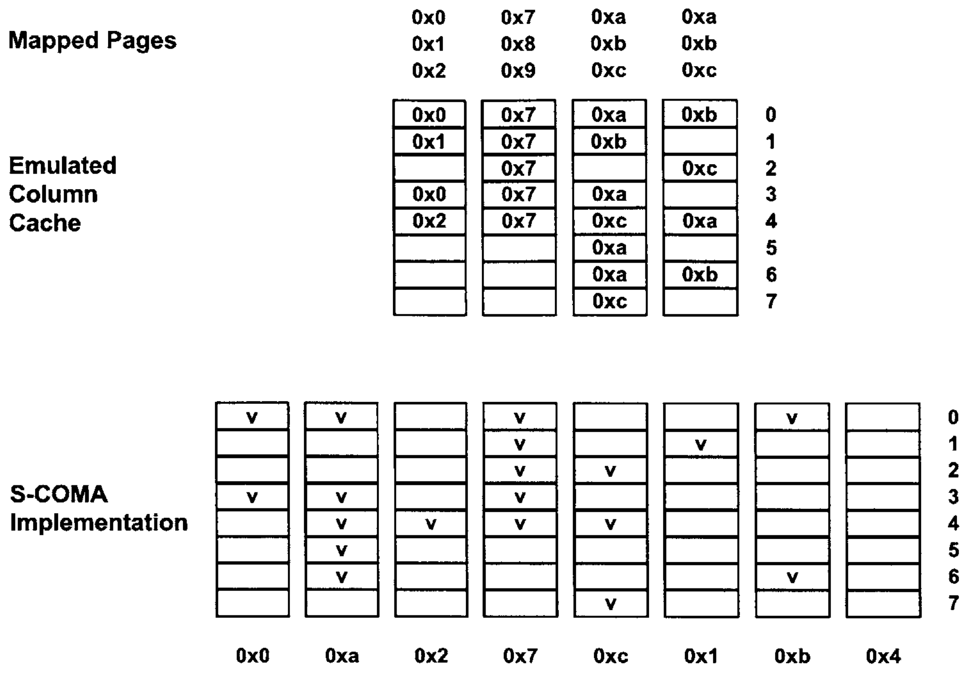


Figure 5-5: START-VOYAGER emulating column caching by artificially restricting the number of valid cache-lines in each set to n where n is the associativity and selecting cache victims using column mapping information. Note that for a particular region of memory (0xa-0xc for example), the number of valid cache-lines in a given set is no greater than the number of columns in the assigned partition. For example, if cache-line 4 of page 0xb should be brought into the cache, either cache-line 4 of page 0xa or cache-line 4 of page 0xc must be invalidated.

does, however, enable accesses that hit in the cache to proceed at a normal pace and accesses that miss incur approximately the same miss penalty as a normal remote miss, since the additional invalidation to emulate the limited resources of column caching is not on the critical path and thus can occur later. The timing characteristics of such an emulation are very similar to that of a real L3 column cache implemented with DRAM.

START-VOYAGER can also implement a distributed bus protocol over its network, emulating a bus or a full network or anything in between. Since both firmware and configurable hardware can access the network, the emulation can be made quite efficient.

Curiosity can also be easily emulated by START-VOYAGER. For simplicity's sake, however, assume that START-VOYAGER is emulating a bus over its network, letting every network interface sees all requests. The network interface can then easily implement curiosity. The site's processor is provided with an interface that allows it specify the curiosity table and the network interface, either the sP, the FPGAs or both in conjunction, can then bring in snooped data and incorporate that data into the emulated column cache.

As described before (Section 4.3.6), StarT-Voyager can emulate distributed curiosity. The aBIU and sP have the ability to see writebacks to S-COMA space. If a writeback occurs that another cache is curious about, that writeback can be forwarded to the curious cache. The producer of the data needs to know which caches are curious to avoid a broadcast. Thus, the curiosity information is pushed to the producer which then sends the desired information as it is produced.

5.3 Column Caching Evaluation

Column and curious caching provide control over storage and communication resources. Control can provide two potential benefits: reducing necessary resources and improving performance. Though we have focused on improving performance rather than reducing resources, in reality they are highly related.

By dynamically mapping regions of memory to regions of the cache in a fashion not unlike overlays which were popular before demand paging, it is possible for software to improve performance with a given set of resources. By judicious mapping of memory regions to cache regions, data that should be kept but would have been replaced by a fixed replacement algorithm can be kept and data that should not be kept will be replaced instead.

5.3.1 Simulation Tools

We use several tools to evaluate column caching. A trace-driven approach was selected for several reasons including the ability to quickly rerun experiments with different parameters, to run experiments on different machines than the trace-generation machines, to improve performance, to allow ideal cache simulation, and to facilitate multithreaded/multitasking experimentation. Instruction and data reference traces are generated by SimICS[50], a fast and accurate instruction-level simulator. SimICS was augmented to produce traces in the PDATS format[41], a trace-knowledgeable compressed format. The traces are further compressed with the Gnu compression utility gzip[32] and stored in a gzip'ed pdt format.

The traces are processed by our cache simulator `hiercache`. Multiple cache sizes with constant column sizes and increasing associativity (Figure 5-6) are simulated simultaneously, immediately illustrating the benefit of additional columns[68]. Multiple cache levels can also be simulated.

Two replacement algorithms are used for general experimentation: LRU and pseudo-ideal. The multiple-cache simulation technique automatically simulates the LRU replacement algorithm. Pseudo-ideal replacement is done by looking forward within the traces and choosing the cache-line that will be accessed furthest in the future for replacement[11]. Despite its name, it is not truly ideal since since it (i) it does not consider the cost of cache misses as replacing pushing out modified data is more costly than clean data and (ii) it models a set-associative cache rather than a full-associative cache.

`hiercache` also simulates a column cache with LRU weighting by allowing the specification of a bit vector of replacement columns for any page of memory. Columns are not all equal, since lower-numbered columns are "more-recently-used" and higher-numbered columns are "less-recently-used". Thus, column specification in `hiercache` also specifies LRU weighting that can make a difference during remapping.

The pseudo-ideal replacement algorithm adds lookahead information to the LRU simulation. Rather than always moving the accessed address to the first column as is done in the standard LRU cache, the pseudo-ideal replacement looks ahead in the reference stream to choose the cache-line that contains data that will be used furtherest 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

Initial State

0	1	2	3	4	5	6	7
0x3	0x5	0x1	0x0	0xa	0x2	0x4	0x6

Access address tag 0xa

0	1	2	3	4	5	6	7
0x3	0x5	0x1	0x0	0xa	0x2	0x4	0x6

Record hit in column 4, shift address tags right up to column 4

0	1	2	3	4	5	6	7
	0x3	0x5	0x1	0x0	0x2	0x4	0x6

Write accessed address tag to column 0

0	1	2	3	4	5	6	7
0xa	0x3	0x5	0x1	0x0	0x2	0x4	0x6

Figure 5-6: `hiercache` simulates multiple cache sizes simultaneously in the following way. A memory reference is first checked to see if it exists in the correct set of the cache. If it does, a hit is recorded, otherwise a miss is recorded. Then, the addresses in the set are each moved down one column. If the referenced address was found in the cache, the movement stops at the column where it was found. The referenced address is then placed in the first column. The position of an address within its set indicates how recently it was accessed relative to the other addresses in the set. Hit rates for multiple cache sizes, from one column to n columns, where n is the number of columns, can be generated from such a simulator simultaneously.

the cache-line that the original address was located in is reached, where the currently comparison data will be inserted. This algorithm will produce optimal replacement statistics for a range of cache configurations, just like the LRU simulator.

Our numbers are entirely in terms of hit rates. Because of the variable costs to service a miss, hit rates cannot give a completely accurate picture of the performance benefits of a mechanism. Different processor architectures can vary the actual benefits, making it impossible to say that one mechanism is always better than another. As research in this area continues, we hope eventually to be able to account for such effects and give much more accurate evaluation of these mechanisms.

In a similar vein, large cache-lines deceptively raise cache hit ratios. For example, given a cache-line size of eight and a sequential pattern of reference, an 87.5% hit rate (7/8) will be achieved. However, if the additional references to the data in the cache-line will follow one right after the other, those loads may still pay almost the same latency as the first access that fills the cache-line. Thus, the 87.5% hit rate achieved may be very different than eight accesses to data in different cache-lines, where one cache-line of data is uncached and the other seven are cached.

5.3.2 Column Caching with a Single Program

It is possible make applications cache-aware, improving their performance within a cached system. Many benchmark applications are already somewhat optimized in this regard, making it difficult to improve their performance with column caching. It is interesting, however, to examine these applications and understand whether cache mapping can improve performance at all. In this study, we examine a set of standard applications, such as gzip and a few synthetic benchmarks.

Column caches can be viewed as a superset of separated spatial/temporal caches and thus can achieve all of the benefits of such caches. A simple example demonstrating the benefits of such a cache reads data from an input, lookups up the data in a table and outputs the result. The stream reads and the writes pollute the cache, displacing the lookup table data.

Assume the input and output streams are larger than the cache, and a uniformly

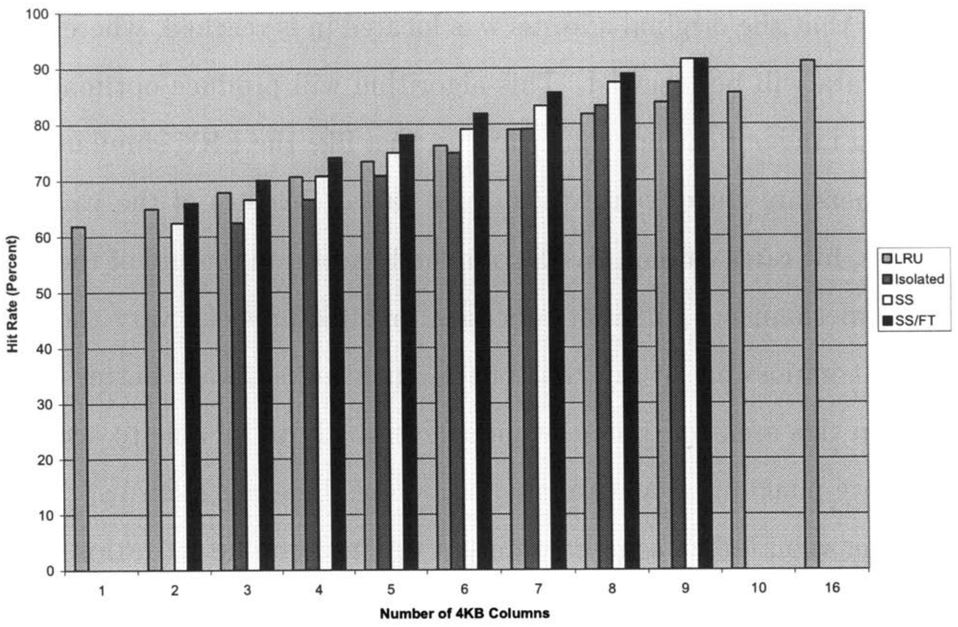


Figure 5-7: LRU verses a Column Cache. The various mapping strategies are (i) Isolated where each stream is allocated one column and the lookup table the rest of the cache (ii) Single Stream column (SS) where the two streams share the same column and the lookup table takes the rest of the cache and (iii) SS/Full Temporal, where the streams share a column and the lookup table uses the entire cache. Note that a separate spatial/temporal cache where the temporal cache is n columns in size will perform like an SS cache of $n + 1$ columns.

accessed 32KB lookup table. The hit rates for a perfect LRU cache, along with four and eight column caches with different column allocation policies are given in Figure 5-7. The column allocation policies are (i) separate each region of memory, (ii) separate stream from temporal regions and (iii) isolate streams into a single column but allow temporal regions to also use that column.

The best policy depends on the amount of available cache space compared to the working sets of the various regions of locality. It is generally better, however, to let the temporal region use the entire cache, but restrict the streams to a single column. This policy is fairly logical; the temporal region of memory can use cache-lines in the sets that are not currently being used by the streams. If stream usage is very high, the LRU algorithm will effectively keep temporal data out of the streams' column while if the stream usage is low, the temporal data will use that column.

A separate spatial/temporal cache is very similar to a column cache where streams are isolated from temporal data. Because of the ability of column caches for temporal data to use columns where stream data is isolated, column caching often provides better performance than split spatial/temporal caches.

If cache space is not an issue, every region of memory can be isolated into its own set of columns. In that case, column caching achieves an pseudo-ideal cache hit rate between nine or ten columns, since there are no replacement conflicts at all. The standard LRU cache is only able to achieve 85.49% hit rate at the same size and needs 16 columns (64K of cache) to get over 91% hit rates. Thus, a mapped cache improves hit rates by almost 6% at the same size cache or requires more than 6 columns or 24KB less cache space to achieve the same hit rates.

For all programs, we started by looking at L1 hit rates. Because of the high hit rates and the modest sizes of current L1 caches, however, little advantage can be gained from standard programs from column caching in the L1 cache. As the L1 cache grows, however, or if the additional functionalities that column caching can provide are desired, column caching may become useful within the L1 cache.

Hit rates within the L2 cache, on the other hand, are not nearly so uniform. In many cases, pseudo-ideal L2 hit rate and LRU L2 hit rate are significantly different. There is significant pollution within the L2 cache making it fertile ground for column caching. In addition, low-associativity designs that rely on some translation of the address almost force column caching to be implemented after the TLB or some other form of translation unless a virtual L1 cache is used.

We examined the current version (1.2.4) of gzip in some detail. We compiled gzip with the standard arguments that included -O optimization. We ran gzip on SimICS, compressing three equal-sized files: a completely random file (not much compression possible) called HARD, a portion of this thesis text (medium amount of compression possible) called MEDIUM, and a single repeated character (very high compression possible) called EASY. We identified the most important variables from the source code, aligned them[1] so that they could be mapped on a page granularity

[1]Previous checks showed that alignment did not negatively affect hit rates; sometimes, they even marginally improved hit rates.

and determined the cache hit rate had each been given a dedicated cache. Using this information off-line, we determined preliminary cache partitions, statically mapping variables to sets of unique cache columns.

We examined both L1 and L2 cache performance for gzip. As expected, the L1 was difficult to partition without very small column sizes. Benefits were seen at 4 cache-line columns, but such column sizes are impractical. We had better luck, however, with the L2 cache.

We expected the hard case the perform the best under our simple form of cache partitioning, since the access patterns are predictable and fairly static but perform poorly under an LRU replacement algorithm because the reference patterns cause an sub-optimal division of the cache (see Section 3.1). By separating regions that would otherwise conflict, we were able to achieve significant improvements (Figure 5-8) in the L2 cache.

On the other extreme, the EASY trace had very high hit rates in an LRU L1 cache and thus had very few accesses to the L2 cache. This behavior was also expected since the hash table storing found patterns was minimally sized due to the single pattern. Because it was trivial to find the pattern, the input window was effectively not retraversed, turning it into a stream. The frequent accesses to the single entry in the pattern hash table prevented it from being replaced by the polluting input stream. The output stream was so infrequently used, it's impact was negligible.

For the MEDIUM trace, however, our performance was virtually the same as a standard LRU cache (within a tenth of a percentage point for 16 columns) using a mapping almost identical to the mapping for the hard trace. By relaxing the constraint that memory regions never share columns, we were able to improve the medium case to perform marginally better than an LRU cache by about 0.5% for either an 8 or 16 column cache. We believe, however, that dynamic partitioning and better sharing off cache between regions of memory can significantly better these numbers.

The strong dependence between data and cache behavior for programs like gzip argues for the ability to dynamically change cache partitioning. For example, if the

Columns	LRU Hit Rate	Column Hit Rate
8	17.29%	26.27%
16	36.81%	41.28%
32	70.43%	73.21%

Figure 5-8: L2 Hit Rate Comparison: LRU verses Column Caching on hard trace.

file is non-uniformly compressible, it is desirable to change the amount of cache space as the demands on the cache change. Software knows the compression ratios and thus can adjust the mappings accordingly. The high cost of remapping for approaches like page coloring probably make such approaches ineffective for this class of problems.

As compilers get better, the number of load/store operations will reduce, increasing column caching impact on hit rates. Extraneous memory operations to recently accessed locations improve hit rates, reducing the proportional improvements of partitioning the cache. We noticed this effect with our initial traces that were unintentionally under-optimized.

Keeping important data within the cache may not make a noticeable difference within the hit rate, but could dramatically improve performance depending on the instruction level parallelism that is "guarded" by a load that might otherwise have missed. In some cases, hit rate may actually be lower for a mapped program but performance may be higher because the misses that are incurred may incur less of a penalty. Modern superscalar processors do not have constant miss penalties, reducing the ability to use hit rate as a measure of performance. This effect is even more pronounced in a parallel system. For example, remote data that may take hundreds or thousands of cycles to access could be allocated to dedicated partitions to avoid their being replaced by local data that is accessed more frequently but has much lower miss penalties. Column caching provides the ability for software to guide hardware in these cases, with huge potential payoffs.

5.3.3 Partitioning Between Multiple Programs

As processors become faster and more capable, being able to issue, process, and retire more instructions per cycle, users want to do more and more with them. Since more is done in each cycle and each cycle is getting shorter, fewer cycles are required between threads switches and more threads can be supported. In addition, recent architectural work[26] has shown the potential performance advantages of fine-grain, hardware-supported multithreading. Having multiple threads time sharing or even simultaneously running on a processor can potentially put significant pressure on the cache.

We have run a set of experiments demonstrating the effect of rapid context switching. Not surprisingly, if there is sufficient room within the cache and the contexts are switched rapidly enough, there is no detrimental effect on the unpartitioned cache. Each thread is run sufficiently often, keeping its cache footprint warm and preventing it from being replaced.

There are schedules of well-behaved (in terms of caching behavior assuming a dedicated cache) programs, however, that perform sub-optimally in the cache. In these cases, the overall footprint is larger than the entire cache, but each job's footprint is no bigger than the entire cache. Many scheduling quantums in a non-partitioned cache will result in the cache being cold when a job is swapped back in.

By partitioning the cache between different processes, this problem can be alleviated. We ran a simple experiment with 10 synthetic jobs, one large one with a 1 MB footprint and 9 small jobs with 64KB footprints. The L1 cache is 64KB large (16-way set-associative) and unmapped, while the L2 cache is 512KB large (128-way set associative) and is mappable at 64KB granularity. We varied scheduling time quantums between 1 and 1,000,000 cycles but eliminated combinations where there was more than a factor of 100 difference in scheduling time since such policies are unlikely to occur.

The cache performance of a standard LRU cache was determined for each option, along with the cache performance of a statically-mapped cache. The two policies

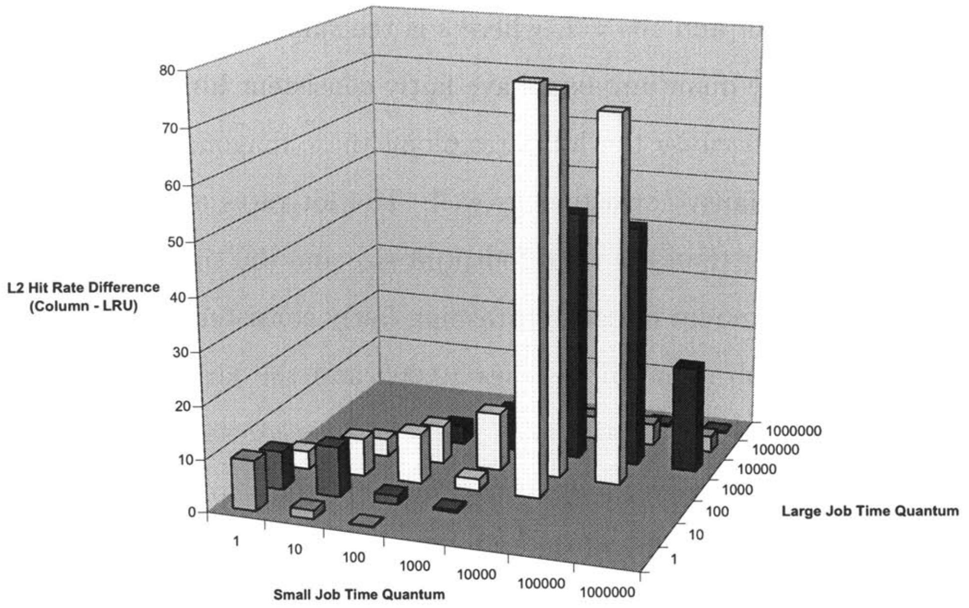


Figure 5-9: This figure shows the difference in L2 hit rate between a column cache and a standard LRU cache over a number of different time quantums for each small job and the large job. The small jobs are all scheduled for the same time quantum. Round-robin scheduling is used.

allowed for the static cache were (i) all of L2 allocated to the big job and (ii) all of L2 allocated partitioned between eight of the smaller jobs. Determining the allocation strategy from process working sets and scheduling quantums is quite simple since hit rates given a certain size cache can be estimated from working sets.

Column caches enables operating system/multithreading scheduling flexibility. There is no case when column caches perform worse than a standard LRU algorithm and there are many cases were a column cache does substantially better (Figure 5-9). By mapping specific processes into the L2 cache, while providing sufficient cache space to those programs that cannot be maintained in the cache across context switches, certain processes state can be saved, improving hit rate. Doing so improves system throughput and can significantly decrease latency of a few critical jobs.

There are a few trends in Figure 5-9 worth noting. One general trend is that when one job runs for a long time compared to its footprint, leveraging the locality in the reference stream, LRU hit rates are similar to column caching hit rates.

Another trend is for hit rate differences to be fairly constant along diagonals. For example $s = l$, $s = 10l$ and $10s = l$, where s is the small job time quantum axis and l is the large job time quantum axis, have fairly consistent hit rate differences. This phenomenon occurs because the hit rates along those diagonals, for both LRU and column caching, are fairly consistent as well. The hit rates are consistent since the cache sizes are smaller than the total footprint size and the time quantums are small enough to keep the amount of cache thrashing fairly consistent.

Once the small job quantum reaches 10,000 and the large job quantum is less than that, however, the vast majority of the references are satisfied by the L1 cache. The few references that go to the L2 cache invariably miss for LRU. Because column caching pins the footprints of 7 of the 9 small jobs into the L2 cache, column caching still achieves reasonable hit rates and thus produce significantly higher hit rates than LRU.

One may question the actual performance impact of L2 hits, given that the context switch time is 10,000 cycles for 9 of 10 jobs. Even at time quantums of 10,000, such pinning can impact performance. There are 2048 cache-lines (assuming a 32B cache-line) in a 64KB footprint. Given 10,000 references, it is likely that all cache-lines will be touched and thus need to be loaded. If each incurs an average of only only 5 cycles of memory latency, the access time will be doubled over always hitting within the L1 cache.

Of course, latency is mask-able to some extent. For example, a 10 cycle latency to an L2 cache may be quite mask-able by modern superscalar architectures, but a 100 cycle latency to memory might not be. There is likely to be a performance "cliff" that hitting in L2 avoids but missing in L2 excites. Thus, actual run-time impact may be significantly more than what might be predicted assuming a linear penalty.

Thus, column caching greatly expands the potential for multitasking/multithreading by reducing or eliminating the cache impact of switching processes/threads. Correct mapping does require knowledge of the cache footprint of the running threads, but is quite easy to do once that information is known.

Chapter 6

Conclusions

Caches need to be very large in order to compensate for non-ideal replacement algorithms and thus are often under-utilized. As multi-tasking, multi-threading and communication become more important, and processors are powerful enough to support many tasks, better use of the cache can translate to better performance at a lower cost. Column and curious caching open the door to a wide range of new ways to use caches better.

Partitioning a cache can achieve significant performance improvements by isolating regions of locality that would otherwise conflict in the cache. Static replacement algorithms perform better in a well-partitioned cache because the reference behavior is more regular within each partition. Column caching and its variants provide partitioning ability and are straightforward to implement.

Partitioning the cache can also minimize cache footprint while maintaining performance, allowing more jobs to run well on the same machine, reduce bandwidth requirements by avoiding poor replacements. Minimizing cache usage can improve processor throughput without reducing the latency of any given application.

Configurable partitioning allows the tuning of the resource-to-performance ratio. If a specific process is more critical than the overall system throughput, that process can be allocated more resources at the expense of throughput. As a process's requirements change over time, the cache can be quickly and efficiently repartitioned to account for those changes.

"Pull-only" caches ignore the possibility that another device may know more about what a master wants than it knows itself, and assume that addresses will be often reused and are unlikely to be modified by anyone besides the pulling master. Today, these assumptions are becoming less and less valid. For example, most communication completely breaks these assumptions.

The ability to push data into a cache in a protected fashion immediately enables significant performance and functionality benefits. Read latencies can be effectively eliminated and bandwidth requirements reduced, while SRAMs and dedicated buffers with specific functionality can be provided within cache space. Curious caching is fairly easy to implement and can be fully backwardly-compatible, allowing processor/caches to use existing unmodified system infrastructure.

Our designs for high-speed communicating systems using a variety of approaches provide a road map of possible design tradeoffs. StarT-Voyager demonstrates that fairly fast systems can be built from completely stock hardware. The modified design that assumes column and curious caching, however, produces significantly better performance with much less effort and resources, demonstrating their potential in the realm of high-speed computing.

As with most research work, there is always more to do. Though users can annotate code, either directly or via library calls, to exploit column and curious caching, it would be preferable to have compilers be able to do some of that work automatically. In addition, tuning mapping strategies for specific multiple-issue, out-of-order processors can probably further improve performance.

Appendix A

StarT-Voyager Shared Memory

Because global memory is simply another memory hierarchy level, cache-coherent distributed shared memory is a better match for standard memory hierarchies than message passing. Getting good performance from shared memory, however, is still quite difficult. Shared memory performance depends on two things: miss rate and miss penalty. Network topologies, degrees of pipelining, amount of buffering, etc. all can have a substantial effect on global miss penalty. Miss rates are strongly affected by cache size, replacement policies, and the ability to pre-fetch or push data.

Unfortunately, reducing miss rate often conflicts with reducing miss penalty. Reducing miss rate relies heavily on software, as cache sizes have gotten quite large. We believe that miss rates can be dramatically reduced if software is allowed to control the replacement algorithms. Software control of replacement algorithms, however, requires flexibility that can potentially increased miss penalty. Minimizing miss penalty implies keeping all miss handling functionality implemented in simple hardware and eliminating any software flexibility.

We designed StarT-Voyager for flexibility rather than absolute minimal latency. Achieving minimum latency requires specialized hardware support to move shared memory requests and replies as quickly as possible. Interpretation should be kept to a minimum, buffers optimized for flow-through and pipelining minimized. StarT-Voyager is not as heavily optimized for shared memory as it could be, though it does have significant support for high performance shared memory.

Choosing flexibility over absolute minimum latency, however, might not be as bad a decision for shared memory as it might seem. The flexibility we provide allows hardware and firmware to be customized for the specific application being run. Thus, we can support a specialized coherence protocol for each application that could significantly improve performance, even though latency is longer than the absolute minimum.

START-VOYAGER implements basic shared memory within its interpreted 1GB region of physical address space that are handled in firmware by the sP. A handled bus operation is retried on the AP bus while the sP is notified of the bus operation for interpretation and handling (see Figure A-1). The sP first determines dispatches on the address, since multiple functionalities may be supported within service space. If the interpreted bus operation is a shared memory operation, the sP then determines which node is the home site of the desired data and sends the request to that home site.

The home site sP receives the message and determines the appropriate action. In a standard shared memory protocol, the home site may be able to satisfy a request on its own, or may need to issue a read to a third node and/or one or more invalidations to one or more other nodes to maintain coherence first. We assume the simple case where the sP simply issues a read to its local AP's DRAM, updates its coherence tables and returns the data to the requesting sP via a Tagon message. Hardware support is provided to ensure that sP commands to a special queue (that support issuing bus operations to the local AP bus and message sends among other things) proceed in exact FIFO order, allowing an sP to issue a block of commands that complete an entire task without polling for completions of sub tasks.

After the requesting sP receives the message and determines it is the requested data, it issues a command that stops the retrying of the AP bus operation that started the whole process. The sP is then out of the loop of returning the data. The next time the AP issues the bus operation, it receives the returned data automatically.

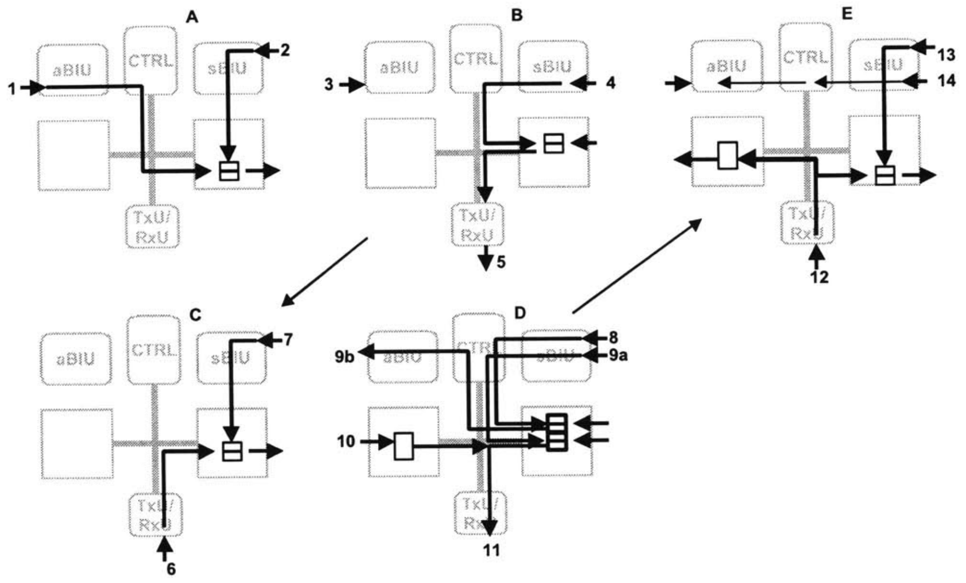


Figure A-1: A simple but slow implementation of shared memory. The requester starts the process by issuing a memory operation that misses in its cache creating a bus operation that is enqueued into a special sP queue (A.1). The sP polls that queue, finding the request (A.2.). The requester's sP issues a request to the home site using an Express message (B.4, B.5) as the requester is continuously retried (B.3.). The request is received by the home site (C.6) and received by the home site sP (C.7.). The home site determines that a clean copy exists locally and issues two commands to the command queue; the first reads the desired data (D.8) and the second sends a reply back to the requester using an Express Tagon message (D.9a). While the second command is being issued, the read could be taking place (D.9b.). As soon as the read completes (D.10), the reply is sent back to the requester (D.11). The reply is received by the requester NIU (E.12), then polled by the requester sP (E.13) who then indicates to the aBIU that the operation should be allowed to continue with the returned data (E.14.).

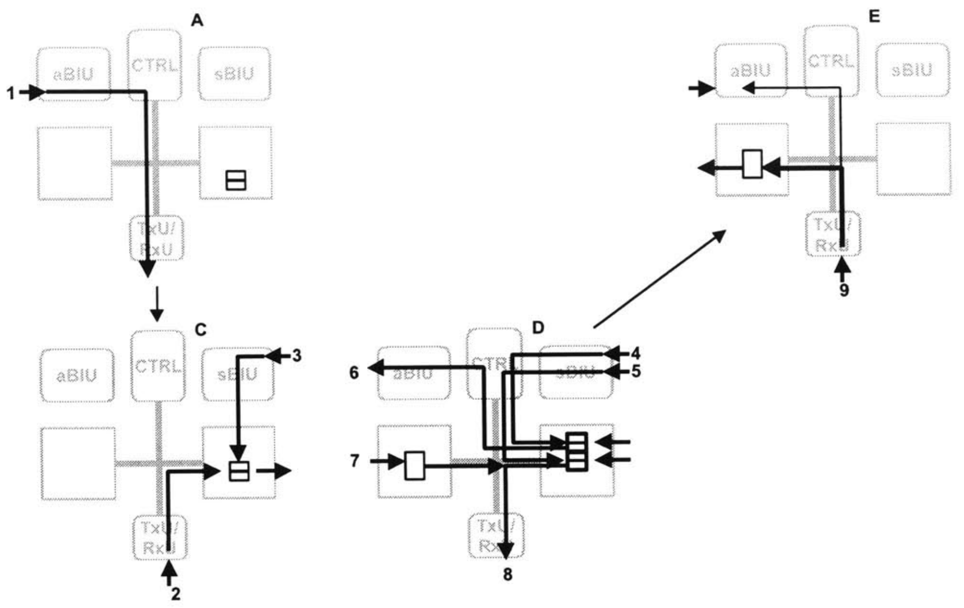


Figure A-2: Eliminating the requester sP from global cache miss servicing.

A.1 Removing the sP from Miss Handling

The simple shared memory described in the previous section is slow, mainly due to sP involvement. The sP wastes time polling and reacting to events. Even though the sP uses the low-latency Express messages to perform these tasks, it is still virtually impossible to compete with fully hardware implementations in terms of round-trip times.

Because of StarT-Voyager's flexibility, however, performance can be significantly improved. The simplest improvement is to remove requester sP involvement. An automatic request forwarding mechanism that automatically issues request messages can be implemented within the aBIU. The remote memory queue already supported by StarT-Voyager can automatically handle the return of the requested data (Figure A.1). With these two mechanisms, the requester sP does not need to take any action to service a cache miss, saving a significant amount of time.

Similar tricks can be used on the home site. Rather than having the home sP issue the bus request to fetch the data, the incoming request can prefetch that data by targeting the remote memory queue (Figure A.1). The request can specify that the home sP is notified when the prefetch is complete. When that happens, if the

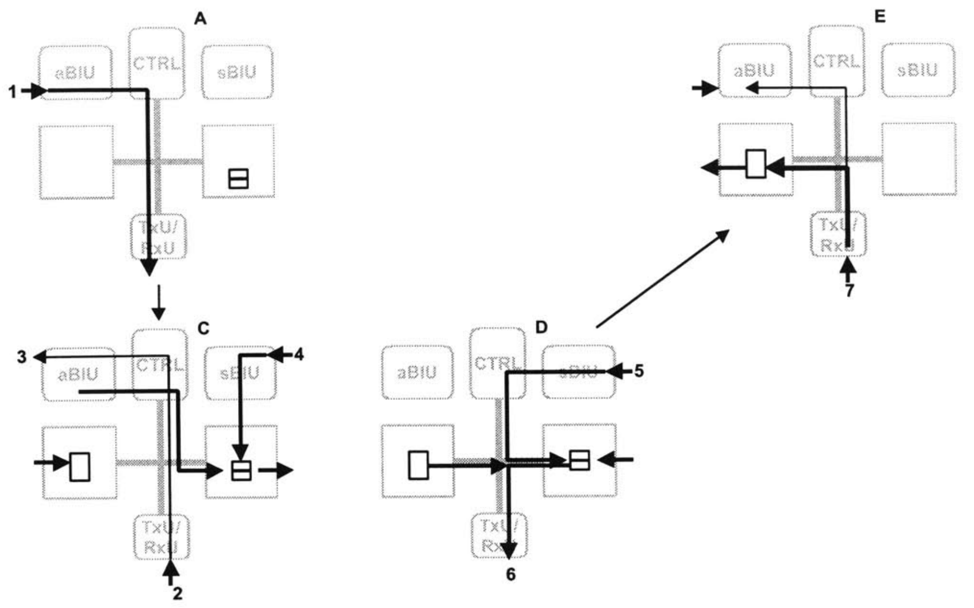


Figure A-3: Prefetching data at the home site.

sP determines it is safe to, it sends an Express Tagon message, returning the data to the requester.

If hardware is thus configured, StarT-Voyager can service a round-trip remote request in about 150 processor cycles. Such latencies are actually quite similar to current memory latencies of standard processors. The Alpha, for example, takes close to 100 processor cycles to access local memory.

This technique only save a little time, since the home sP must explicitly send back the requested data. The clsSRAM that provides 4 bits of state for each cache-line region of data enables an automatic reply (Figure A.1). Data should not be returned unless the coherence state is correct. When the request message is received, it is "mirrored" into an identical queue for the sP, allowing the sP to see the request as soon as it arrives. When the prefetch is automatically issued onto the home's bus, the aBIU reads the clsSRAM state associated with that cache-line of data. If the state is acceptable for the request, the aBIU creates an Express Tagon message itself and sends back the request, otherwise it notifies the home sP. Simultaneously, the aBIU changes the clsSRAM state for that address to be a transient state and notifies the sP that it returned the data who updates its directory state and resets the clsSRAM state correctly.

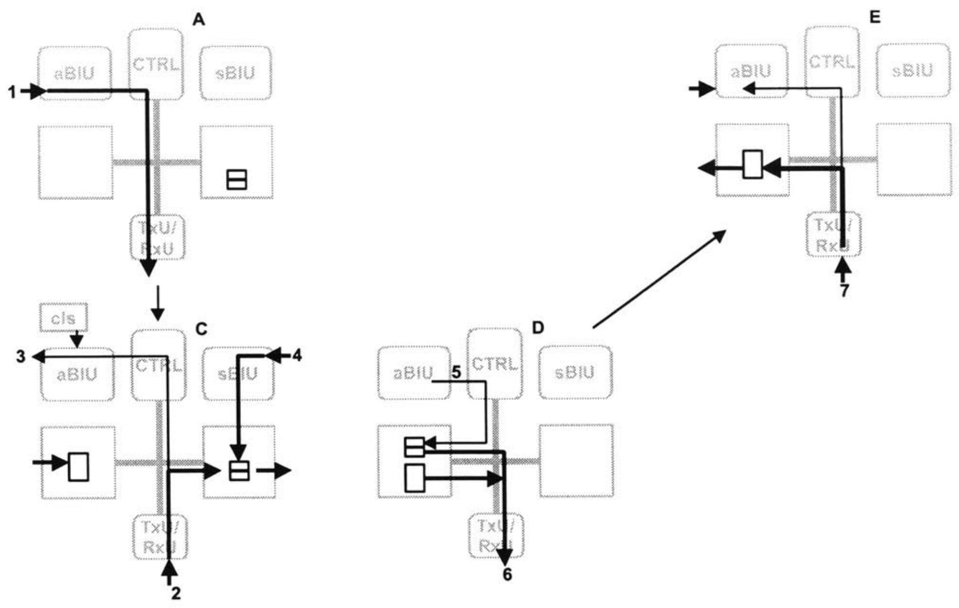


Figure A-4: Removing the sP from the critical path altogether.

In this last case, the critical path of a clean global access is handled completely in hardware. Though StarT-Voyager's datapath is not absolutely optimal for ease of implementation, it will still perform quite well with such a configuration. There are issues of buffer management and deadlock avoidance, but both can be handled by a combination of firmware and configurable hardware.

A.2 Making Shared Memory Faster

The clsSRAM can also be used as cache tags to implement S-COMA. 64MB of DRAM can be used as an S-COMA cache. If that space is well managed, hit rates will generally be very low.

In addition, replacement policies are implemented by firmware code allowing the ultimate flexibility. Applications can communicate with firmware to specify automatic prefetching, memory usage durations so that the replacement policy can yank data back after a specified time, completion of an entire region of memory, or messages to indicate that data has arrived. In fact, applications can even download firmware code to the firmware engine if required. This flexibility allows the user to minimize the miss rate of his application.

Although synchronization is still a problem in shared memory, we can use message passing for synchronization thus reducing the shared memory synchronization problem to the message passing synchronization problem. The `sync` probably still needs to be used for shared memory synchronization but is less costly relative to the miss penalty than for message passing.

Pushing and prefetching data will become more and more important as processors become faster relative to networks and memory. Currently, caches relying solely on "pulling", that is, they require an explicit read from their processor before data is brought into the cache. Pushing and prefetching can dramatically reduce latencies over pulling, since these techniques attempt to keep data close to the consumer rather than forcing the consumer to fetch at high latency the data right when it needs it. StarT-Voyager's configurable hardware and firmware can easily implement pushing, potentially reducing latencies significantly.

Bibliography

[1] A. Agarwal and S. Pudar. Column-Associative Caches: a Technique for Reducing the Miss Rate of Direct-Mapped Caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 179–190, 1993.

[2] AMD. *AMD-K6-III Processor Data Sheet*, Feb. 1999.

[3] C. Anderson and J.-L. Baer. Two Techniques for Improving Performance on Bus-based Multiprocessors. In *First IEEE Symposium on High-Performance Computer Architecture, Raleigh, North Carolina*, pages 264–275, Jan. 1995.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, Feb. 1996.

[5] B. S. Ang. *Design and Implementation of a Multi-purpose Cluster System Network Interface Unit*. PhD thesis, Massachusetts Institute of Technology, Feb. 1999.

[6] B. S. Ang, D. Chiou, D. Rosenband, M. Ehrlich, L. Rudolph, and Arvind. StarT-Voyager: A Flexible Platform for Exploring Scalable SMP Issues. In *Proceedings of SC'98, Orlando, Florida*, Nov. 1998.

[7] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. Message Passing Support in StarT-Voyager. In *HiPC98*, Dec. 1998.

[8] B. S. Ang, D. Chiou, L. Rudolph, and Arvind. The StarT-Voyager Parallel System. In *Proceedings of PACT'98, Paris, France*, Oct. 1998.

[9] K. Asanovic. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, May 1998.

[10] P. Bannon. Alpha 21364: A Scalable Single-chip SMP. http://www.digital.com/alphaoem/present/sld001.htm.

[11] L. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[12] B. K. Bershad, B. J. Chen, D. Lee, and T. H. Romer. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *ASPLOS VI*, 1994.

[13] G. A. Boughton. Arctic Routing Chip. In *Parallel Computer Routing and Communication: Proceedings of the First International Workshop, PCRCW '94*, volume 853 of *Lecture Notes in Computer Science*, pages 310–317. Springer-Verlag, May 1994.

[14] D. Burger, J. Goodman, and A. Kagi. Limited Bandwidth to Affect Processor Design. In *IEEE Micro*, November/December 1997.

[15] D. Burger, A. Kägi, and J. R. Goodman. The Declining Effectiveness of Dynamic Caching for General Purpose Microprocessors. Technical Report 1261, Computer Sciences Department, University of Wisconsin, Madision, WI, Jan. 1995.

[16] H. Burkhardt III et al. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, Feb. 1992.

[17] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *Fifth International Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.

[18] J. B. Carter, A. Davis, R. Kuramkote, C.-C. Kuo, L. B. Stoller, and M. Swanson. Avalance: A Communication and Memory Architecture for Scalable Parallel Computing. Technical Report UUCS-95-022, Computer Systems Laboratory, University of Utah, 1995. Original Avalanche.

[19] D. Chiou, B. S. Ang, R. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. E. Hicks, and A. Boughton. StarT-NG: Delivering Seamless Parallel Computing. In *Conference Proceedings of the First International EURO-PAR Conference, Stockholm, Sweden*, pages 101 – 116, Aug. 1995.

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

[21] D. E. Culler and A. Singh, Jaswinder Pal with Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1998.

[22] Cyrix. *Cyrix 6X86MX Processor*, May 1998.

[23] Cyrix. *Cyrix MII Data Book*, Feb. 1999.

[24] F. Dahgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *The 22th Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy*, pages 60–69, June 1995.

[25] W. J. Dally et al. Architecture of a Message-Driven Processor. *IEEE Micro*, 12(2):23–39, 1992.

[26] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, , and D. M. Tullsen. Simultaneous Multithreading: A Foundation for Next-generation Processors. *IEEE Micro*, pages 12–18, September/October 1997.

[27] G. Faanes. A CMOS Vector Processor with a Custom Streaming Cache. In *Hot Chips 10*, August 1998.

[28] E. Farquhar and P. Bunce. *The Mips Programmers Handbook*, 1993.

[29] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing Global Memory Mangement in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Dec. 1995.

[30] J.-M. Frailong, C. M., P. Sindhu, J. Gastinel, M. Splain, J. Price, and A. Singhal. The next-generation SPARC multiprocessing system architecture. In *COMPCON Spring '93*, pages 475–80, 1993.

[31] S. Frank, H. Burkhardt III, and D. J. Rothnie. The KSR1: Bridging the Gap Between Shared Memory and MPPs. In *COMPCON 93*, Feb. 1993.

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

[33] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache Performance of the SPEC92 Benchmark Suite. *IEEE Micro*, August 1993.

[34] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 2nd edition, 1996.

[35] P.-Y.-T. Hsu. Design of the R8000 Microprocessor. *IEEE Micro*, 1993.

[36] M. Hull, D. Crookes, and P. Sweeney, editors. *Parallel processing. The Transputer and its Applications.* Addison-Wesley, 1994.

[37] Intel. *IA-64 Application Developer's Architecture Guide*, May 1999.

[38] Intel, http://developer.intel.com/design/pentiumii/manuals/243191.htm. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*, 1999.

[39] Intel. *Intel StrongARM SA-1100 Microprocessor*, April 1999.

[40] D. Johnson. Techniques for Mitigating Memory Latency Effects in the PA-8500 Processor. In *Hot Chips 10*, August 1998.

[41] E. E. Johnson and J. 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.

[42] T. L. Johnson and W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24st Annual International Symposium on Computer Architecture (ISCA)*, June 1997.

[43] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Full-Associative Cache and Prefetch Buffers. In *The 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[44] A. W. W. Jr., R. P. L. Jr., R. J. Ionta, R. P. Valentino, B. Hu, P. R. Breton, and P. Lau. Update propagation in the galactica net distributed shared memory architecture. CHPC TR 93-007, Center for High Performance Computing, Worcester Polytechnique Institute, 293 Boston Post Road West, Marlborough, MA 01752, 1993. (Also in IPPS '94).

[45] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application Performance and Flexibility on Exokernel Systems. In *16th Symposium on Operating Systems Principles, Saint-Malo, France*, October 1997.

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

[47] J. Kubiatowicz, D. Chaiken, and A. Agarwal. Closing the Window of Vulnerability in Multiphase Memory Transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 274–284, Oct. 1992.

[48] C. Law. A new competitive analysis for randomized caching. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1999.

[49] B. Lynch and G. Lauterbach. UltraSPARC III: A 600 MHz 64-bit Superscalar Processor for 1000-Way Scalable Systems. In *Hot Chips 10*, 1998.

[50] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. In *Usenix Annual Technical Conference, New Orleans, Lousiana*, pages 101 – 116, June 1998.

[51] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* Morgan Kaufmann Publishers, Inc., 1994.

[52] A. Milenković and V. Milutinović. Cache Injection on Bus Based Multiprocessors. In *Workshop on Advances in Parallel and Distributed Systems, West Lafayette, Indiana*, Oct. 1998.

[53] V. Milutinović, A. Milenković, and G. Sheaffer. The Cache Injection/Cofetch Architecture: Initial Performance Analysis. In *MASCOTS-97*, Jan. 1997.

[54] MIPS Technologies, Inc. *R10000 Microprocessor User's Manual*, December 1996.

[55] Motorola. *MPC8240 Integrated Processor User's Manual*, July 1999.

[56] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching.* PhD thesis, Stanford University, Mar. 1994.

[57] B. Nayfeh and Y. A. Khalidi. Us patent 5584014: Apparatus and method to preserve data in a set associative memory device, Dec. 1996.

[58] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 24–33, April 1994.

[59] G. M. Papadopoulos et al. *T: Integrated Building Blocks for Parallel Computing. In *Proceedings of Supercomputing '93, Portland, Oregon*, pages 624–635, Nov. 1993.

[60] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Processors. In *The 11th Annual International Symposium on Computer Architecture, Ann Arbor, Michigan*, pages 340–347, June 1984.

[61] R. M. Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[62] F. Sánchez, A. González, and M. Valero. Software Management of Selective and Dual Data Caches. In *IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, pages 3–10, Mar. 1997.

[63] SGI. *R8000 Microprocessor Product Information.*

[64] X. Shen, Arvind, and L. Rudolph. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, Rhodes, Greece*, June 1999.

[65] X. Shen, Arvind, and L. Rudolph. Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. In *Proceedings of the 26th International Symposium On Computer Architecture, Atlanta*, May 1999.

[66] T. Sherwood, B. Calder, and J. Emer. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proceedings of the International Conference on Supercomputing, Rhodes, Greece*, June 1999.

[67] S. Shimizu and M. Ohara. Method and apparatus to maintain cache coherency in a multiprocessor system with each processor's private cache updating or invalidating its contents based upon set activity. United States Patent 5,228,136, July 1993.

[68] R. A. Sugumar and S. G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 24–35, May 1993.

[69] Sun Microsystems. *UltraSparc User's Manual*, July 1997.

[70] M. R. Swanson, R. Kuramkote, L. B. Stoller, and T. Tateyama. Message Passing Support in the Avalanche Widget. Technical Report UUCS-96-002, Department of Computer Science, University of Utah, Mar. 1996. Current Avanlanche design.

[71] C. Thacker, L. Steward, and E. Satterthwaite, Jr. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, Aug. 1988. Also SRC Research Report 23.

[72] M. Tomasko, S. Hadjiyiannis, and W. Najjar. Experimental Evaluation of Array Caches. In *IEEE Computer Society Technical Committee on Computer Architecture: Special Issue on Distributed Shared Memory and Related Issues*, pages 11–16, Mar. 1997.

[73] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture Ann Arbor, MI*, November/December 1995.

[74] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting Cache Line Size to Application Behavior. In *International Conference on Supercomputing*, June 1999.

[75] C. Zhang, X. Zhang, and Y. Yan. Two Fast and High-Associativity Cache Schemes. *IEEE Micro*, pages 40–49, September/October 1997.