Hydra: A Chip Multiprocessor with Support for Speculative Thread-Level Parallelization

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Lance Stirling Hammond

August 2001

© Copyright by Lance Hammond 2001

All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyeunle Olukotun (Principal Advisor)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the University Committee on Graduate Studies:

Abstract

This thesis describes the design and provides a detailed analysis of Hydra, a chip multiprocessor (CMP) made up of four normal MIPS cores, each with their own pairs of primary data caches. The cores are connected to each other, a shared, on-chip secondary cache, and a high-speed off-chip DRAM interface by a pair of wide, pipelined buses that are specialized to support reads and writes, with relatively simple cache coherency protocols. The basic design supports interprocessor communication latencies on the order of 10 cycles, using the shared secondary cache, allowing a much wider variety of programs to be parallelized than is possible with a conventional, multichip multiprocessor. Our simulations show that such a design allows excellent speedup on most matrix-intensive floating point and multiprogrammed applications, but achieves speedup only comparable to a superscalar processor of similar area, at best, on the large family of integer applications that can really take advantage of the low communication latencies provided.

In order to make execution of integer programs easier on Hydra, we examined the possibility of adding thread-level speculation (TLS) support to Hydra. This is a mechanism in which processors are enhanced so that they can attempt to execute threads from a sequential program in parallel without knowing in advance whether the threads are parallel or not. The speculation hardware then monitors data produced and consumed by the different threads to ensure that no thread attempts to use data too early, before it is actually produced. If such an attempt is made, the offending thread is restarted. In this manner, threads may be generated from existing program constructs such as loops or subroutines almost automatically. Such support can be added to Hydra simply, with a few extra bits attached to the primary caches and some speculation buffers attached to the shared secondary cache. In practice, we found that most of our integer applications could be sped up to a level comparable to or better than an equal-area superscalar processor or our hand-parallelized benchmarks — and with very little programmer effort. However, we usually had to apply several manual optimization techniques to the code to achieve this speedup.

Acknowledgements

<< PUT IN AT THE END >>

Table of Contents

List of Tables

List of Figures

1 Introduction

At the dawn of the twenty-first century, the potential for further microprocessor performance enhancement using conventional uniprocessor designs is quickly dwindling away. New designs that simply add refinements to the existing well-studied art of conventional wide-issue superscalar processor cores are able to achieve higher performance, but the design costs are becoming more and more prohibitive even as the performance gains realizable from this design effort are becoming increasingly tiny. This is leading processor designers towards the use of new design paradigms that may unlock the way to higher performance processors of the future. One of the most promising of these design paradigms is explored in this thesis: the chip multiprocessor, or CMP.

1.1 Chip Multiprocessors: The Future of Microprocessors

As time passes, there is one reliable trend in electrical engineering: process engineers will continue to pack larger numbers of faster transistors on chips that can be mass-produced economically. This trend, embodied in Moore's Law, allows designers to use the enlarged transistor budgets to build larger and more complex microprocessors. One of the largest jobs of computer architects is to stay on top of this progress and design microprocessor architectures that utilize the current transistor budgets and leading-edge circuit design styles in ways that extract the most performance possible from economically sized-chips [15].

As this trend continues, design problems caused by the exceedingly large number of transistors are forcing computer architects to look at different alternatives to the conventional single, monolithic processor on a chip. Table 1-1 examines five leading contenders to succeed the current generations of small-scale superscalar uniprocessors that dominate the market today: large-scale superscalar uniprocessors [16], VLIW uniprocessors [17], simultaneously multithreaded processor cores [18], chip multiprocessors, and memory-processor integration [19].

To address these issues, we have proposed [6][7] that future processor design break away from the default option of simply making progressively larger uniprocessors and instead implement more than one processor on each chip, adopting the chip multiprocessor as the new architecture of choice. The key design issues that led us to this decision are discussed in the in the remainder of this section, and our proposed design is described in Part I of this thesis.

Table 1-1: An overview of five potential future microprocessor architectures.

Architecture	Pros	Cons
Large-scale Superscalar Uniprocessors	Current software does not need to be modified to take advantage of increasing parallelism, since it is extracted at the instruction level.	Circuitry complexity and design effort is exponentially increasing. Only a limited amount of ILP is available.
VLIW Uniprocessors	Control complexity is moved to the compiler, which can schedule instructions intelligently.	Datapath complexity is still increasing. Programs must be recompiled for each CPU version.
Simultaneous Multithreading	Allows multiple threads of execution to make optimal use of a large superscalar core.	Circuit complexity and design effort is still exponentially increasing, since the core is still superscalar.
Hydra: Small-Scale MP on a Chip	Allows reuse of small CPU designs. Can look like a slower uniprocessor or a fast MP, as software allows.	Software must be parallelized, or a small speed penalty will occur due to simple uniprocessors.
Processor-Memory Integration	Allows reuse of simple CPU designs at a higher performance point, since memory delays are reduced. Makes the design of the memory hierarchy much easier.	Software must be parallelized into small chunks that fit on each CPU. Memory configuration is inflexible. Putting memory & logic together compromises performance of both.

1.1.1 Wire Delays

As CMOS gates become faster and chips become physically larger, the delay caused by interconnect between gates is becoming more significant [20]. We are just reaching the point at which it is not possible to transmit signals across entire chips within a clock cycle in leading-edge processor designs. A long wire in a delay path can force a designer of one of these chips to limit the amount of logic in that path. These limitations will make the design of large uniprocessors that occupy entire chips extremely difficult in the near future, since the chips will have to be carefully floorplanned to keep units that need to communicate quickly close together. When this is not possible, many long wires will have to be buffered to work over multiple clock cycles in order to avoid lengthening the cycle time. However, a CMP can be designed so that each of its small processors takes up only a relatively small area on a large processor chip, minimizing the length of its wires and simplifying the design of critical paths. Only the more infrequently-used, and therefore less critical, wires connecting between the processors need to be long.

1.1.2 Fine-Grain Thread-Level Parallelism Extraction

Large-scale and small-scale multiprocessors currently dominate the market in handling scientific computing applications. These systems have been able to obtain significant speedups with large scientific programs that contain a large amount of inherent parallelism. This parallelism can often be found and exploited by vectorizing or parallelizing compilers such as the SUIF compiler [21] developed here at Stanford. These codes work efficiently on current MP systems by breaking down the execution into a number of threads that can work fairly autonomously, communicating only every few thousand instructions over the duration of the program. The current MPs are also good at exploiting parallelism at the OS level, when different processes are being scheduled to run simultaneously on different processors. In this case, the only communication between threads of control is occurring in the OS itself.

As Figure 1-1 indicates, however, there is a large amount of parallelism that cannot be extracted by current MPs due to their long communication delays. Current MPs typically must either run programs with only fine-grained parallelism on a single processor, thereby eliminating the advantage of having multiple CPUs. When one tries to use their processors together to exploit fine-grained parallelism, they typically exhibit a *slowdown*, instead of a speedup, due to the excessive amount of time wasted synchronizing the processors. A CMP, on the other hand, can make the interprocessor synchronization process much more quick and efficient, so less time is wasted on each synchronization and more time can be spent on computation. Having all processors on a single chip lowers the synchronization times so that communication even every 10-100 instructions in parallel code will be feasible while still obtaining a speedup, due to the relatively fast interprocessor communication times illustrated in Figure 1-2. This vastly increases the opportunity for finding thread-level parallelism, even on programs where it was once thought impossible.

This ability to extract parallelism using much smaller threads can allow a CMP to use thread-level parallelism in new ways. On conventional superscalar processors, additional transistors on chips are primarily used to extract more low-level parallelism from programs. While some transistors are used to build wider or more specialized datapath logic (i.e. switching from 32 to 64 bits or adding special multimedia instructions), most are used to allow processors to extract greater amounts of instruction level parallelism (ILP) by finding non-dependent instructions that occur near each other in the original program code. Unfortunately, there is only a finite amount of ILP present in any particular sequence of instructions that is executed by the processor, since instructions from the same sequence are typically closely related. As a result, processors that use this technique are seeing diminishing returns as they attempt to execute larger numbers of instructions per clock cycle, even as the logic required to process multiple instructions per clock cycle increases quadratically. By using low-level thread-level parallelism instead of ILP, a CMP avoids this limitation by extracting a completely different type of parallelism, but on a similar scale. Of course, a CMP may also attempt to extract small amounts of ILP within each of its individual processors, since some ILP is just too fine to be extracted by threads.

1.1.3 Economy and Ease of Design

A side effect of the larger uniprocessors and multiprocessors being built today is that the design complexity is increasing exponentially. Designing and verifying a modern commercial microprocessor is an exceedingly complex task. As a result, designers are now looking for ways to simplify their designs while still exploiting the full performance offered by the increasingly large dies and

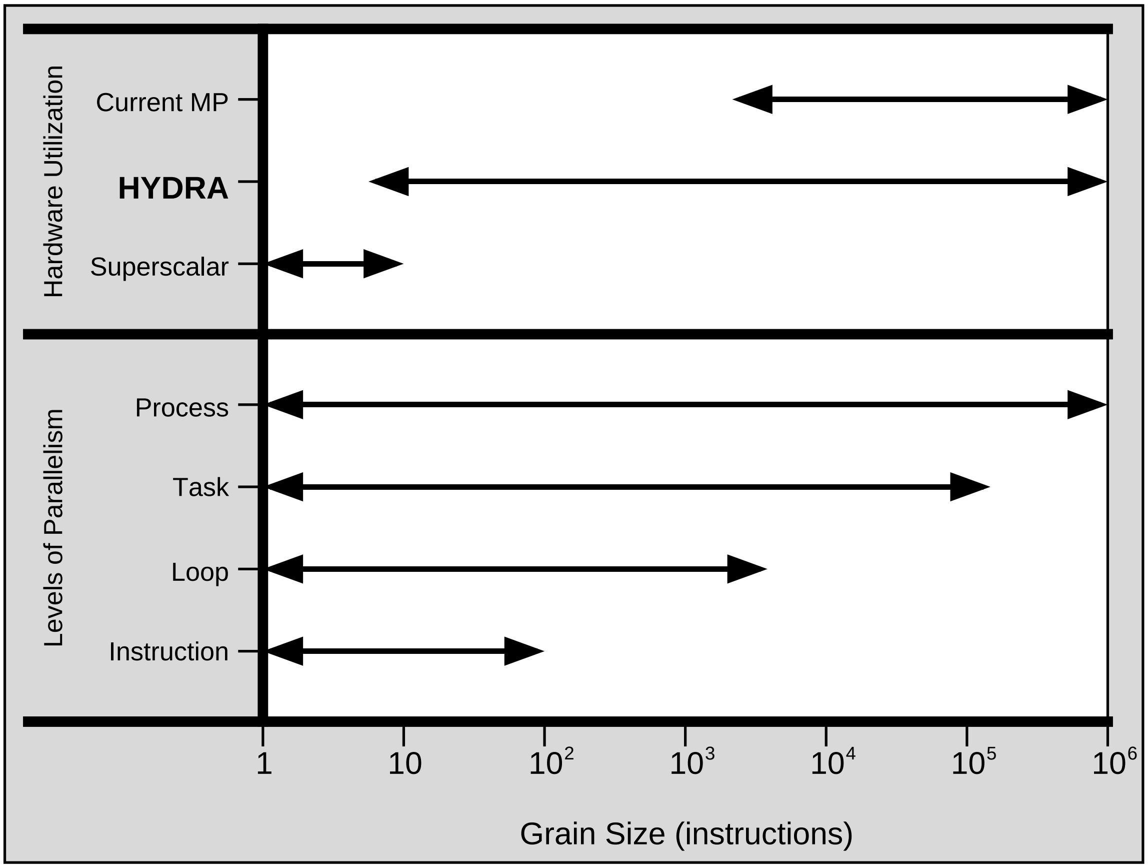


Figure 1-1: Levels of parallelism that can be exploited in applications.

more efficient MP communication that is available. CMPs attack this problem in two ways: simpler core designs and a simpler interconnection hardware.

While current microprocessors are being designed that utilize 200-400 mm^2 of silicon area just for the processor, each of a CMP's processors can be designed to be an order of magnitude smaller — and thereby much simpler. Each of the individual processors is only comparable to a uniprocessor from a generation or two ago, but while this means that these processors are easy to design, it does not mean they are necessarily slow — even if they are just old cores pulled out of past designs. Instead, increased gate speeds allow these small, simple processors to run at very high clock rates without excessive amounts of design effort. In fact, they can often be clocked faster than a large, complex processor in a similar process technology [61]. This clocking advantage can tend to off-

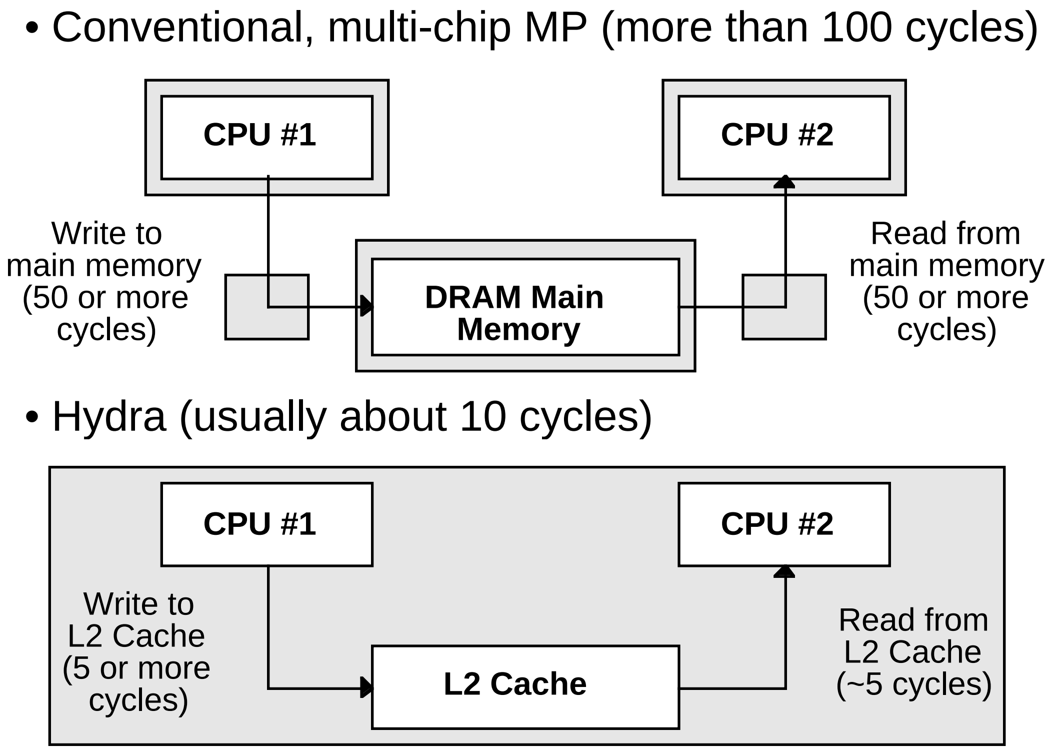

• Conventional, multi-chip MP (more than 100 cycles)
CPU #1
CPU #2
Write to main memory (50 or more cycles)
DRAM Main Memory
Read from main memory (50 or more cycles)
• Hydra (usually about 10 cycles)
CPU #1
CPU #2
Write to L2 Cache (5 or more cycles)
L2 Cache
Read from L2 Cache (~5 cycles)

Figure 1-2: Comparison of interprocessor communication latency in a traditional multiprocessor and a CMP.

set much of the advantage offered by ILP extraction in a complex processor with critical paths that are harder to optimize. On top of the basic design size advantage, design costs can be further amortized in a CMP by using the same cores for a longer period of time. As technology scales, the same cores can be re-used over again, but in larger numbers.

Similarly, large uniprocessors are taxing the ability of circuit designers to actually build circuits that are necessary to extract parallelism from the instruction stream in the increasingly short cycle times. Central components of many of these designs are vastly multiported register files that are becoming more and more difficult to design as uniprocessors attempt to issue many more instructions at once. In addition, building large, multiported L1 caches that can be accessed quickly has resulted in responses ranging from giving up and making a faster memory hierarchy [22] to vastly convoluted circuit design [23]. CMPs can easily avoid both of these problems by keeping the individual processors very simple, with only small, single-ported L1 caches. By only attempting to find parallelism between the CPUs, instead of within them, the number of complex, high-speed functional unit interconnection structures is reduced considerably. Instead, the processors executing threads in parallel on a CMP can communicate using only simple, relatively slow bus structures. Such structures are much easier to design and implement, which helps reduce the implementation time of a CMP.

1.2 Thread-Level Speculation: A Helpful Addition

Since a CMP addresses all of these potential problems in a straightforward, scalable manner, why aren't CMPs already common? Only a couple of major processor designs, the IBM Power4 [24] and Sun MAJC [25] have adopted these concepts in a wholehearted fashion. One reason is that integration densities are just reaching levels where these problems are becoming significant enough to consider a paradigm shift in processor design. The primary reason, however, is because it is very difficult to convert today's important uniprocessor programs into multiprocessor ones.

Conventional multiprocessor programming techniques typically require careful data layout in memory to avoid conflicts between processors, minimization of data communication between processors, and explicit synchronization at any point in a program where processors may actively share data. A CMP is much less sensitive to poor data layout and poor communication management, since the interprocessor communication latencies are lower and bandwidths are higher. However, sequential programs must still be explicitly broken into threads and synchronized properly. Parallelizing compilers have only been partially successful at automatically handling these tasks for programmers [26]. As a result, acceptance of multiprocessors has been slowed because only a limited number of programmers have mastered the necessary programming techniques. To fully utilize a CMP, however, there needs to be a simple and effective way to parallelize even these applications. Hence, we explored the possibility of adding hardware support to our basic CMP design to ease the parallelization process.

A survey of some of the leading research in the field suggested that thread-level speculation might help. Tom Knight first proposed this process as a way of parallelizing high-level languages in the 1980's [27]. His scheme proposed a very high-level system that could parallelize lisp code using a technique similar to the subroutine speculation described in Section 5.1.2. While impractical, it set the theoretical groundwork for what was to follow. At the same time, other researchers were proposing designs of transactional memory systems to replace standard multiprocessor synchronization techniques or for use in transactional server applications, such as the design in [45]. These systems added techniques for adding an "undo" function to memory systems for use when multiple processors wrote to the same area of memory at once over a period of time, without resorting to conventional explicit synchronization and critical code regions. In the early 1990's the Sohi group at Wisconsin combined these ideas together into the first fully realized thread-level speculative design, the Multiscalar processor [28].

Thread-level speculation takes the sequence of instructions run during an existing uniprocessor program and arbitrarily breaks it into a sequenced group of threads that may be run in parallel on a multiprocessor. In order to ensure that correct program execution is maintained, hardware must track all inter-thread dependencies. When a true dependence violation is detected, the hardware must ensure that the "later" thread in the sequence executes with the proper data by dynamically discarding threads (or portions of threads) that have speculatively executed with the wrong data. This is a considerably different mechanism from the one used to enforce dependencies on conventional multiprocessors. There, synchronization is inserted into threads that stalls a thread reading data until after the correct value has been written.

Speculation allows parallelization of a program into threads even without prior knowledge of where true dependencies between threads may occur. All threads simply run in parallel until a true dependency is detected as the program is executing. This greatly aids the parallelization of programs because it eliminates the need for human programmers or compilers to find synchronization points in programs statically at compile time or earlier, because all places where synchronization might have been required are simply found dynamically when true dependencies actually occur. As a result of this advantage, uniprocessor programs may be *obliviously* parallelized in a speculative system. While conventional parallel programmers must constantly worry about maintaining program correctness, programmers parallelizing code for a speculative system can focus solely on achieving maximum performance, since the speculative hardware will ensure that the parallelized threads always perform the same computation as the original program.

Since parallelization by speculation dynamically finds parallelism among program threads at runtime, it does not need to be as conservative as conventional parallel code. In many programs, there are points within potential threads that *may occasionally* modify the data that other threads are using and cause true dependencies, if certain conditions are met at runtime, but where dependencies seldom if ever actually occur. A speculative system may attempt to run these sorts of threads in parallel anyway, and only back up the "later" thread and execute serially if and when a dependency actually occurs. On the other hand, a system dependent on synchronization must always synchronize at any point where a dependency *might* occur, based on a static analysis of the program, whether or not the dependency actually ever occurs at runtime. Code that modifies data objects through pointers in C programs are a frequent source of this problem within many integer applications. In these programs, a compiler (and sometimes even a programmer performing hand parallelization) will typically have to assume that any later pointer reads may be dependent on the latest write of data using a pointer, even if that is rarely or never the case. As a result, a significant amount of thread-level parallelism can be hidden by the way the uniprocessor code was written, and therefore wasted as a compiler conservatively parallelizes a program.

It should be noted that speculation and synchronization are not mutually exclusive. A program with speculative threads can still perform synchronization around uses of dependent data, but this synchronization is *optional*. As a result, it is still possible for a programmer or feedback-driven compiler to add synchronization into a speculatively parallelized program *if* that helps the program execute faster.

1.3 Thesis Organization

Here at Stanford we have designed and tested the Hydra design, a prototype 4-core CMP design using a simple, bus-based interprocessor communication scheme on the chip. The design of Hydra and its software, and studies done with them are discussed throughout the remainder of this thesis. Chapter 2 is a brief description of the LESS simulator, our simplified version of the SimOS machine simulator that we developed to test the design of Hydra, and particularly the design of speculation. The remainder of the thesis is grouped into two large sections.

The first part of this thesis, describing the “baseline” version of Hydra without speculation, consists of Chapter 3 through Chapter 4. Chapter 3 gives an introduction to the design of Hydra followed by a full specification of the Hydra design. After all of the parts of the design have been fully described, we conclude this part with an analysis of simulations that tested key parts of the baseline design in Chapter 4.

The second half of the thesis describes the enhancements to the Hydra design necessary to add thread-level speculation support and the software APIs that must be used with this hardware. The beginning of Chapter 5 gives an in-depth introduction to the concepts necessary for understanding thread-level speculation implementations. Once the reader understands the needs of these implementations, the remainder of Chapter 5 can be read to get a full understanding of how speculation has been implemented in Hydra. Chapter 6 gives a full description of how to write speculative software for Hydra and then optimize it. Finally, this part of the thesis concludes with a detailed performance evaluation of speculative threads on Hydra, showing both the strengths and weaknesses of the design.

The thesis ends with a detailed set of conclusions in Chapter 8.

1.4 Thesis Contributions

This research has provided insight into the art of CMP design and practical implementations of thread-level speculation. The highlights are summarized below:

- Definition of the Hydra CMP design, a simple bus-based, quad MIPS CPU CMP. Analysis of this design has shown that the many design tradeoffs made to keep it inexpensive have generally been good choices.

- Definition of additions to the Hydra CMP design to support speculative thread-level speculation with only a very small amount of additional hardware.

- Definition of two entire sets of software control handlers used to interface between user code and the Hydra CMP while executing under speculation.

- Definition of a full speculative thread API with suggestions for how to enhance the performance of many programs.

- Characterization of the performance of speculative threads on the Hydra CMP with different software handlers and programming techniques used.

Finally, significant progress has been made towards the development of an FPGA-based prototype of the Hydra chip that can be used for further hardware and software design efforts by future members of the Hydra group.

2 The LESS Simulator

As with previous efforts to develop fundamentally new ways to attach processors together in unique ways, evaluation of the Hydra hardware design required significant simulation efforts to test the newly-designed hardware structures before they could actually be implemented in hardware — or even in an RTL. Many simulators have been made in the past both at Stanford and elsewhere. Two prominent examples at Stanford are Tango-Lite, developed for the DASH project [1], and SimOS [2], developed for the FLASH project. A couple of prominent external simulators are MINT [3] and SimpleScalar [4]. After surveying the field, we took some features from the existing simulation environments and built a Hydra simulation environment that just contained the features we needed to simulate the parts of the hardware interesting to us in an execution-driven manner. Because of the distillation process that occurred, the simulator was dubbed "LESS" (for "Lance's Extremely Simple Simulator). All results shown in later chapters of this thesis have been generated with various generations of LESS.

Early research that led to Hydra [5][6][7][8] was performed using modified versions of both MINT and SimOS. However, as we continued to add more sophisticated hardware structures to Hydra, the modifications became increasingly difficult to maintain. In particular, simulations with SimOS required that we have our modifications be fully IRIX OS-compatible, and this compatibility proved to be very difficult to achieve in practice. As a result, it was much simpler to maintain a system composed of the MIPSY processor core from SimOS surrounded by our own, simplified memory system harness. For our benchmarks of interest, system code execution time was not a primary consideration, so LESS makes some compromise in this area in order to keep the simulator itself simple while still being able to execute code containing the occasional `SYSCALL` instruction. The remainder of this section describes the decisions made during the design of LESS to keep it as simple as possible.

2.1 How LESS Runs

A programmer can use LESS with virtually any IRIX [9] program simply by changing the "`main`" routine of the program to "`mainX`" and then linking the simulator in with the targeted program. The resulting single, large binary starts up a single simulated processor with its program counter pointed at the beginning of the `mainX` routine. Other simulated processors are maintained in a dormant state until the program being simulated invokes them with special versions of the `fork` system call. Because the simulated program and the simulator are compiled together into a single binary, they both share the same memory space, a feature unique to LESS. A rough memory map of the combined simulator / program combination running under IRIX 5.3 is given in Figure 2-1.

LESS simulates the execution of user code on one or several processors simultaneously by running all stages of the MIPSY pipeline through one cycle's worth of work for each processor, in turn. After all active processors have been given a chance to execute for a single cycle, the hardware in memory system is simulated for a cycle. When this is done, the cycle repeats. This alternating execution-memory pattern results in a simple pipeline being formed, as illustrated in Figure 2-2. As is indicated in the figure, this pipeline takes advantage of the branch-delay slot required by the MIPS architecture, as branches are not resolved until the end of the EX stage of the pipeline. In the meantime, the instruction fetch of the instruction in the branch delay slot will already have been initiated. On the other hand, the structure of the pipeline makes load results available at the "beginning" of the next instruction's EX cycle, so the MIPS load delay slot is not absolutely necessary.

2.2 System Call Handling

The primary reason why LESS uses its unusual combination memory map is to ease the handling of system calls. While LESS can execute all user-level code quite easily, including the code in shared libraries such as `printf` and other `libc` routines, it is impossible for the user-level code

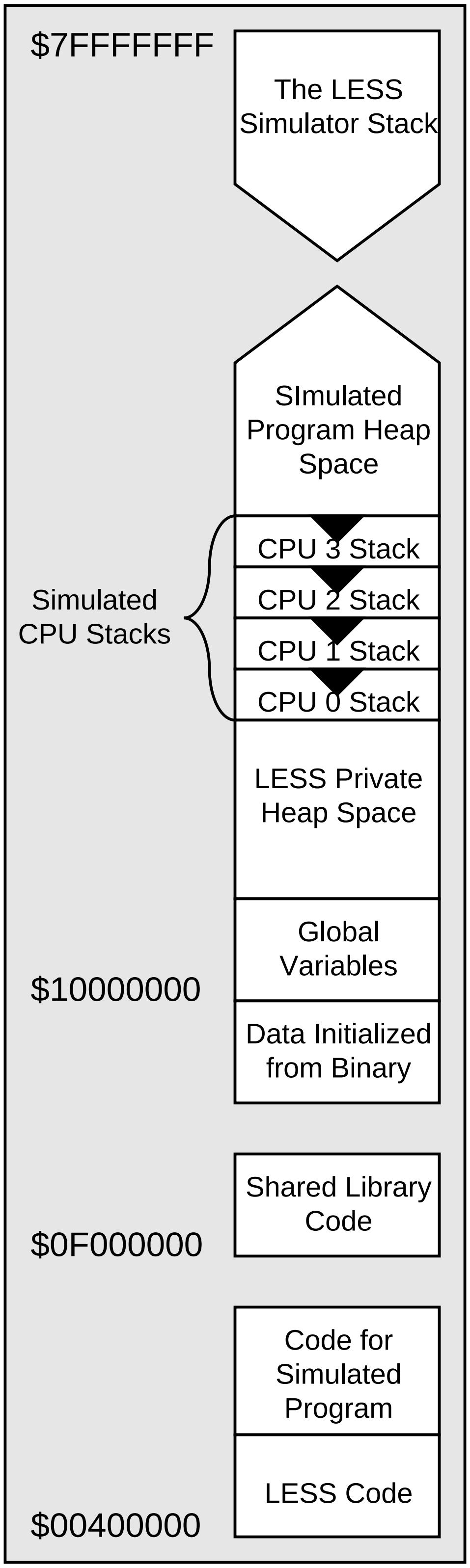


Figure 2-1: The LESS memory map

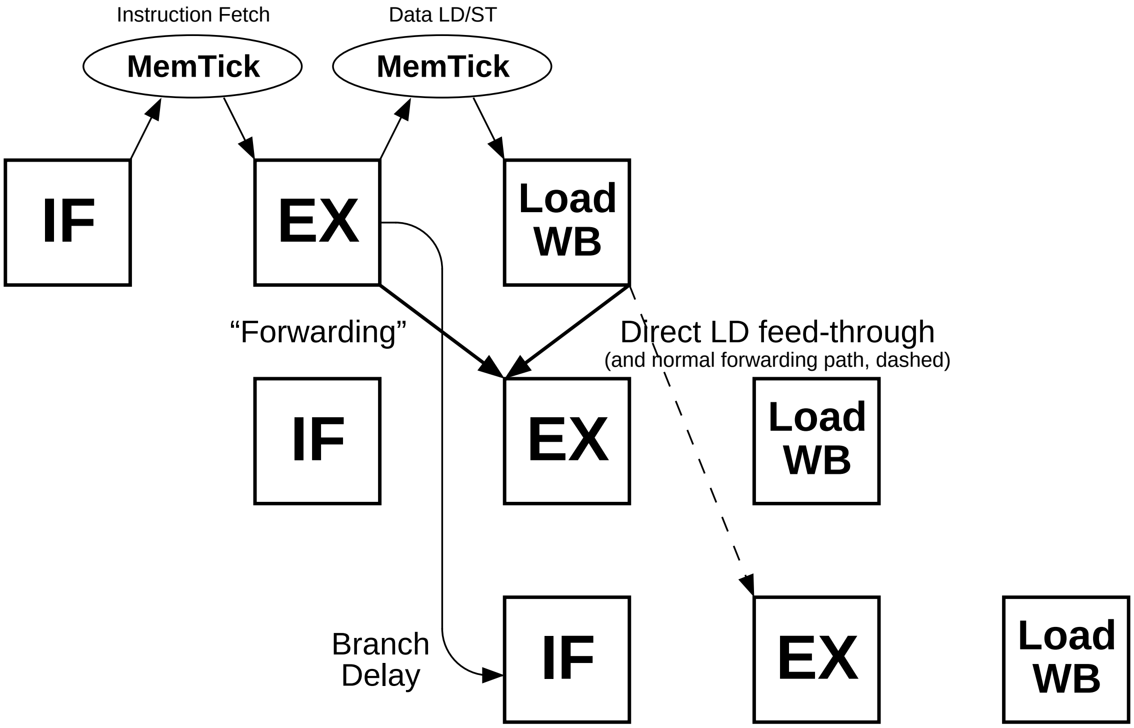


Figure 2-2: The LESS Pipeline

in LESS itself to fully emulate the instructions within a system call without simulating the entire IRIX system, as is done in SimOS [2], or at least all key system calls, as in MINT [3] or SimOS-Solo [2]. To avoid the complexity of significant system call emulation code, LESS simply performs a context switch when a `SYSCALL` instruction is encountered by saving the LESS stack and register set out to memory while bringing the simulated process' register state and stack pointer into the real processor's registers. The LESS simulator process is thereby swapped out, and the simulated process is temporarily allowed to run natively for the duration of the SYSCALL. As a result, all I/O and other system operations may be executed by the simulated process just as if they had occurred during the normal execution of the simulated program. To the user, it looks just as if the program was executing normally, albeit at a somewhat slower pace.

While this system is very flexible, and allows the simple simulation of a wide variety of system calls, it does have its limitations. First and foremost, no timing information about the time spent in the system call may normally be obtained. As a result, the entire system call appears to happen "instantaneously" in a single cycle to the simulated program, as far as statistics are concerned. Hence, any program that involves a large amount of system time in its critical regions, such as a server application, should probably not be simulated on LESS. Since the benchmarks of interest to us during most of this research had most of their system activity limited to their startup and shutdown phases, we did not consider this to be a major liability. Another problem is that a few critical system calls like `fork` cannot be simulated at all. A `fork` would normally just create a second copy of the simulator, instead of a second process *within* the simulator, working on another of its simulated processors. As a result, code which uses this or similar UNIX system calls must be adjusted to use LESS's own version of `fork` or different code altogether. We found that finding and adjusting these calls was usually a fairly trivial task. Finally, we occasionally found that the simulator and simulated program could interfere with each other, a problem that cropped up periodically throughout our research.

2.3 Using the Real Machine to "Fast Forward"

Another use of LESS's ability to context switch between the simulator and one of its simulated processes is to speed execution of the simulated program. We frequently found places in our benchmarks where startup code that we were not interested in measuring took significant amounts of time to simulate. As execution of these regions was necessary before we could reach the critical, and therefore interesting, portions of the programs, we found it helpful to execute the startup code on the real machine, in a manner analogous to how LESS already handled system calls.

While this process sounds complex, it is actually remarkably simple to use. One just has to insert a pair of special LESS "system calls" to tell the system when to drop into fast forward mode (`sys_goto_real()`, or `spec_goto_real()` if we happen to be running speculatively) and when to come back out again (`sys_goto_less(runningCPU, savedRegs)`, where the parameters are pointers to a pair of globals that LESS uses to restore itself to operation). Between these function calls, all code in the program will be executed on the real machine. They just need to be carefully located in *pairs* wherever real-machine operation is desired, or runaway real-machine operation may occur.

Like the execution of system calls, fast forward operation has some drawbacks. Once again, it is impossible to get simulation statistics during fast forwarding. While the lack of timing statistics is usually not a problem, since normally one only fast-forwards through uninteresting regions of the program, the fact that the caches are not simulated can create some timing variations later during simulation. This is because the caches will not be as “warmed up” as they normally would. Code that takes advantage of special LESS functionality obviously cannot be sped up in this manner, either. This is also not much of a limitation because such code will probably be in the heart of the critical region where simulation is desired. The only real drawback is the fact that only one simulated processor may be sped up at a time. Any other simulated processors are effectively suspended while one LESS process is running on the real machine. As a result, if the process running on the real machine ever requests data from one of the other simulated processors without switching back to LESS itself, the whole simulation will deadlock. We avoided this hazard by carefully checking our code to ensure that no multiprocessor synchronization routines were present within any region under consideration for fast forward operation.

3 The Base Hydra Design

This part of the thesis describes the baseline Hydra chip multiprocessor, a combination of four MIPS processor cores on a single chip with a shared L2 cache array. After a brief overview of the Hydra design itself, this chapter offers a detailed specification of the baseline Hydra design. The chapter is organized so that descriptions of the various parts of the Hydra chip move outwards from the modifications immediately surrounding the IDT cores to the new external interfaces defined to flow off of the Hydra chip itself.

Because most of the complexity in the Hydra system is centered on the central buses and the controllers that sequence accesses to those buses, this portion of the chapter is by far the largest. This part of the Hydra chip contains much of the custom design work, and it is what gives Hydra the unique features that may make it a good alternative for future processor development — a low-latency but fairly simple communication medium between the four CPUs and memory that can satisfy the bandwidth requirements of four small but hungry processor cores. Due to the fact that all four of the CPUs are located on a single chip, it has been designed to much different specifications than a conventional multiprocessor bus controller. For example, a central arbitration mechanism is used to arbitrate for several key chip resources on a cycle-by-cycle basis. In contrast, most CPU buses are arbitrated for on a much coarser "all or nothing" basis every few cycles.

The portions of the chapter that discuss the design in detail are roughly divided up into four parts: datapath, control, caches, and off-chip interfaces. Section 3.4.1 and Section 3.4.2 are the "datapath" sections, and cover the data buses that are responsible for transporting user data, and the address of that data around the chip, respectively. Section 3.4.3 and Section 3.4.4 are the "control" sections. The former discusses the structures used by the interconnection system to keep the datapaths running smoothly, while the latter goes over the functionality of the units during "typical" memory accesses. The caches are covered before and after the discussion of the memory interconnection system, in Section 3.3 and Section 3.5. Finally, the off-chip interfaces are discussed in Section 3.6. Also, since this is a research vehicle, many of the elements in this system will be instrumented to facilitate performance measurements and analyses. These features are discussed periodically throughout the chapter.

3.1 Hydra Design Overview

This section of the thesis gives a brief introduction to the design of the Hydra chip, as depicted in Figure 3-1.

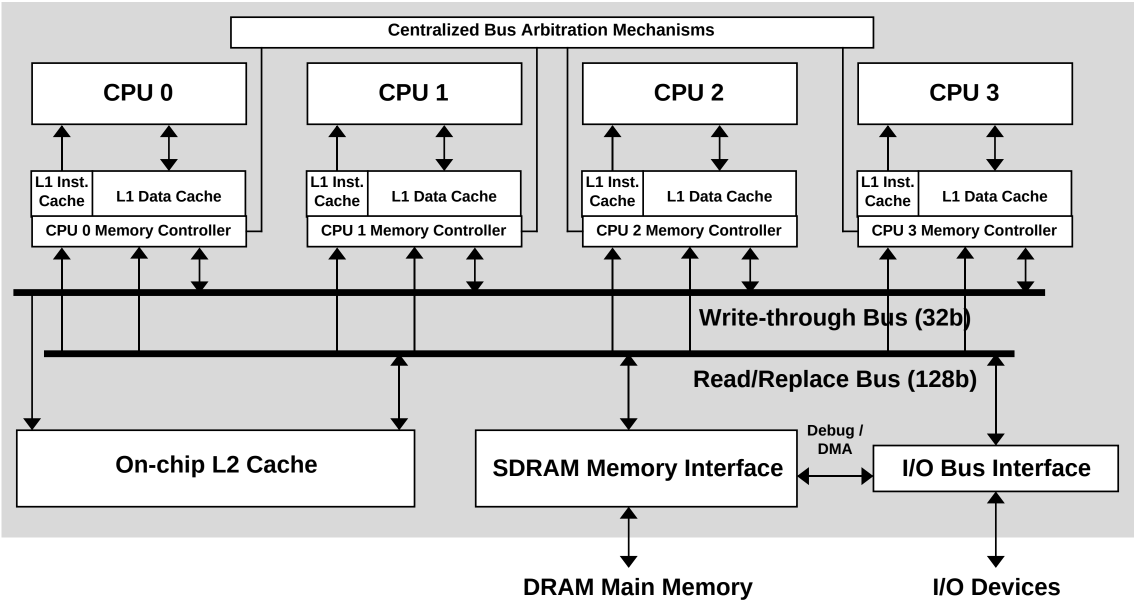


Figure 3-1: An overview diagram showing the major portions of Hydra and how they interconnect through the read and write buses.

3.1.1 CPUs and MMUs

The core of the system is the array of four IDT RC32364 MIPS cores. Each tiny core is a 200 MHz single-issue MIPS processor core with its own built-in MMU. Our cores differ from the commercially sold processors at the external interface, which has been removed to allow a direct connection to the Hydra memory system. Each core is similar to a conventional MIPS R3000 processor. The cores run all 32-bit MIPS-I (R3000) code with some extensions from the MIPS-IV

(R10000) set, plus a few extensions designed for embedded use and memory-management instructions optimized to work in this particular implementation.

The core includes the CPU, MMU, and most of the existing L1 cache controller from the commercial IDT part with only minor modifications. When memory accesses fail to hit in the L1 cache, however, they are passed to the Hydra memory system instead of the normal bus interface unit included in IDT's chips. A few other minor modifications have been made to allow external intervention into the L1 data cache tags and the attachment of speculation support resources. It should be noted that the IDT cores themselves are not described anywhere in this chapter. Readers looking for a detailed description of these cores should look to the IDT reference manual [33] for full specifications.

3.1.2 Separate L1 Instruction and Data Caches

The L1 data caches in Hydra consist of separate 8K (increased from the default 2K), 2-way associative, 256 sets (increased from 64) with 16 bytes/line, write-through, no-allocate-on-write caches for all four processors. The instruction caches are 8K, 2-way associative caches with 512 16-byte lines each. These have been integrated directly into the CPU cores by IDT. The size and design of the caches have been optimized so that all eight of the L1 caches may operate in parallel each cycle to provide each processor with a 32-bit instruction per cycle and a single 32-bit word of data each cycle. Due to their small size, the design of each cache is fairly simple. While not sufficient for typical workstation processors, these cache sizes are fairly typical for embedded-application cores.

Attached to the caches is a special cache-invalidation port that the memory system uses to invalidate lines from the caches externally. The memory system has its own collection of "shadow" tags for each of the L1 data caches so that invalidations (which tie up the L1 cache tags for 1–2 cycles) will only occur when a line that is actually in an L1 cache is modified by another processor. These additional shadow tags also include several bits used to control the contents of the data caches while a processor is executing a speculatively parallelized thread.

3.1.3 Memory Controllers

Pairs of “state machine” controllers attached to each processor are responsible for controlling all aspects of L1 cache misses and write-throughs. A ninth controller is responsible for controlling on-chip resources after data returns from main memory. These controllers sequence the use of the Hydra chip’s resources by individual accesses. In order to prevent the different controllers from interfering with each other, two central resource arbiters are included in the system. One prevents multiple controllers from using the same resources simultaneously, while the other prevents multiple accesses to the same address from being processed by the memory system simultaneously.

3.1.4 Memory Communication Buses

The core of the Hydra system is the Read/Replace (or just “read”) and Write-through (or just “write”) buses the run through the center of Figure 3-1. These two buses carry all of the data between the four processor / L1 cache combinations and the lower levels of the memory hierarchy. In addition, they are the primary points of synchronization and communication in the system. Since all major parts of Hydra have to communicate through these buses, the control system for these buses is quite sophisticated and is the topic of Section 3.4, the largest section of this chapter.

The write bus is a word-wide (32 bit) bus that carries every write made by any of the processors to their standard caches. The order that this bus is allocated imposes a distinct sequencing on the writes between the different processors. Only when a write has actually passed over this bus has it actually “occurred,” as far as the other processors in the system are concerned. If a different processor reads an address before a write has passed over this bus, it will get the old data, since only the processor that wrote the data may immediately read it before it is transmitted over the write bus. As the key synchronization point, the write bus is responsible for invalidating all older data in the L1 caches of any other processors that may be sharing the cache line containing the data, so it has a direct connection to the shadow tag system of each L1 data cache. On the other side, it has a direct connection to the L2 cache, which it must update with every word of written-through data — a task which keeps it busy constantly.

The read bus is an L2-line-wide (32-byte, 256-bit) bus that carries cache lines into the processors on demand (using half of the bus) and is also used to move lines around between the lower levels of the memory hierarchy, because it is not as heavily utilized as the write bus. While the write bus has a single, specialized job, the read bus is the workhorse that does everything else.

These two buses work as a team to allow the four processors to work together quickly and efficiently in the Hydra system.

3.1.5 Shared, On-Chip L2 Cache

The remaining die area is occupied by the shared L2 cache. This cache is designed to absorb the write bandwidth coming through the write-through L1 caches and to allow sharing between the four processors without tying up the off-chip memory port. It will necessarily be a writeback, write-allocate cache to properly handle these tasks. On a reasonably small die, the size of this cache should be about 64K - 128K, preferably with at least 4-way associativity. This would obviously be easy to enlarge using a larger die, however, so for simulation purposes we frequently enlarged the cache. The line size is set at 32 bytes/line, twice the size of the L1 cache lines.

3.1.6 Main Memory and I/O Interface

For references that do not hit in any of the on-chip caches, Hydra will recover the necessary data from main memory. The interface to the main memory and a fairly simple memory-mapped I/O port has changed significantly over the lifetime of the Hydra project. For the most part, it has been conceived as a direct connection to a bank of DRAM using various interfaces (Rambus, most frequently). However, by the final implementation the main memory interface had largely transformed into a bus bridge to the I/O interface. The I/O interface also includes features to load benchmarks into the Hydra system, perform basic I/O operations, and check results after test completion.

3.2 Some Related Work

Other than an early effort by Texas Instruments to make a DSP processor using a very simple CMP [29], the Hydra research group at Stanford was generally at the forefront of the call for chip multiprocessing. While most other research groups were still looking at techniques for stretching the performance of conventional uniprocessors or multi-chip multiprocessors, here at Stanford research efforts looked at the various aspects of processor design that would tend to lead one to consider a chip multiprocessor and then to a series of publications and a thesis that conclusively showed the advantages of such a design [6][7][8]. This work led directly to the design of Hydra. On the other hand, since the Hydra design was first publicized we have seen many industry efforts to adopt some of its ideas.

Looking towards the server market, the IBM Power4 [24] has directly adopted the idea of Hydra-style chip multiprocessing. Each processor chip consists of two processor cores, each with their own private L1 caches, and a shared L2 cache. Moreover, the Power4 has been designed so that up to four processor chips may be ganged together into essentially an 8-way “CMP” like design using a high-performance MCM substrate. The Compaq Piranha project [30] has also adopted the use of an 8-way CMP and a shared L2 cache on a chip. While the Power4 uses very high-performance cores, the Piranha is centered around a group of fairly simple cores, very much like the Hydra design, because the Piranha project is focused on commercial server applications and there is a considerable amount of evidence that shows server workloads benefit little from superscalar processing.

At the other end of the spectrum, there is even more CMP activity ongoing as this thesis is being written. Numerous asymmetric CMPs have been built to provide single-chip systems with the advantages present in more than one different processor core. The most frequent combination is that of a microcontroller and a DSP processor [31]. In the multimedia market, even a few symmetric CMPs have been designed to handle the massive parallel-processing jobs found there. The Sun MAJC chip [25], targeting the Java-based multimedia market, uses two processors sharing the same L1 data cache to make a multiprocessor system that can share even more closely than the processors on Hydra can. Meanwhile, Sony has produced an MPEG-processing chip [32] that looks like a direct implementation of Hydra.

3.3 The L1 Data Cache Tags

In order to support the frequent inter-processor invalidations that will be required of each CPU, the L1 data cache tags must be duplicated, as depicted in Figure 3-2. While one set of tags is used to handle normal cache traffic from the CPU, the second set snoops on the address bus associated with the write that is currently being processed on the write bus. If the second set finds a tag within the cache that matches the address on the write bus, an invalidation cycle is initiated to eliminate the line from the cache by clearing its valid bit.

The primary complexity in this system is due to the fact that tag updates may be generated by both the primary and the “shadow” tags simultaneously, yet there is only one write port to the tag arrays. In these rare collisions, the CPU is stalled for a cycle while the invalidation is allowed to continue.

One additional bit of added complexity is introduced by the ability of the core to issue a single non-blocking load. When this occurs, the cache tag is updated immediately to reflect the fact that the line will soon be in the L1 cache. However, the address of the load-in-progress is maintained in a side buffer at the tag comparison logic, where it can be compared with addresses of further memory reference instructions sent by the processor. Any “hits” to the line that may occur while it is still being fetched from the rest of the memory system are cancelled until after the line has been successfully returned from memory.

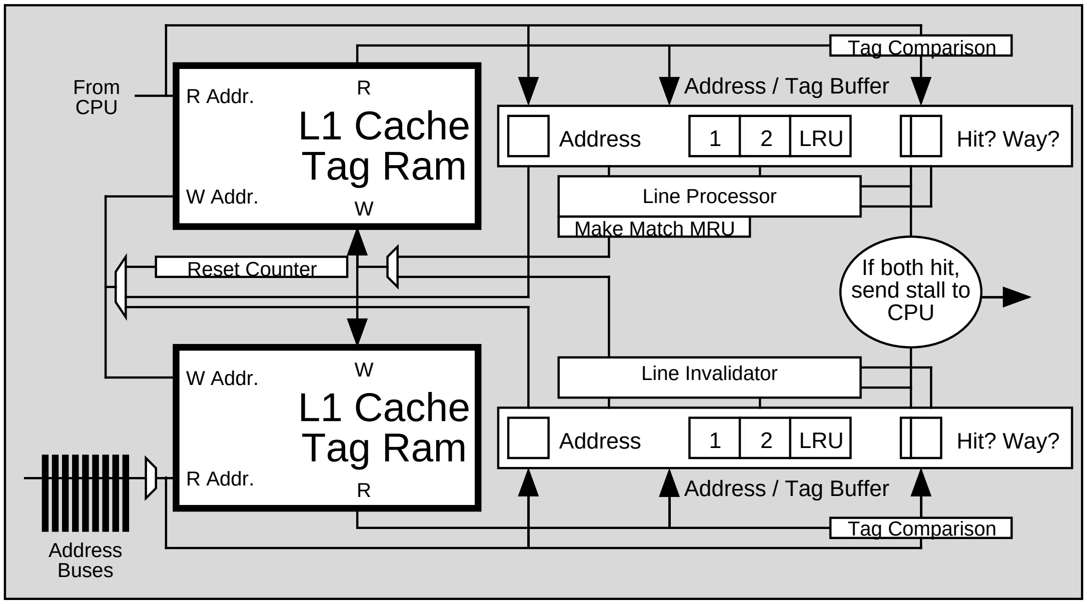


Figure 3-2: The enhanced L1 data cache tag system.

3.4 Memory Interconnection System Overview

A basic overview of the memory interconnection system is given in Figure 3-3. All of the high-speed computational and memory resources on the Hydra chip (the CPUs and their L1 caches) must use a set of buses that run down the center of the Hydra chip to talk with the lower-speed memory and I/O resources (L2 cache and the external interface). In the course of performing transactions, the lower levels of the memory hierarchy use the buses to send data and addresses between each other, as well. All of the parts are insulated from each other by a system of buffers, for the data buses, and state machines, that synchronize and sequence the interacting units.

The buses at the center of the diagram are all located in the center wiring channel of the chip, where they interconnect all of the major units of Hydra. There are 12 main buses: 9 32-bit address buses associated with each of the 9 state machines that control the memory interconnection network, a 32-bit data bus that exclusively handles writebacks from the CPUs to the caches (the writeback or “write” bus), and a 256/128-bit data bus that handles all other data transactions between units in the system. Each state machine is responsible for maintaining its own address bus in the correct state, but the shared data buses are arbitrated for through the central arbitration scheme between the different state machines.

State machines are located at the interface to all of the “active” elements of the chip — the CPUs and the external memory interface, which are capable of initiating their own accesses to the other units. As the different elements initiate accesses, the state machines for the units translate each access request into the proper sequence of arbitrations for and accesses to other units. These accesses to other units are then controlled by the buffers that connect each unit to the central buses only when such connections are actually required. Working in tandem, the ring of state machines and buffers manages to keep everything flowing through the interconnection network in a smooth manner.

3.4.1 Datapath Resources

The data-carrying resources in the Hydra chip are summarized in Figure 3-4. This figure shows the two primary buses — the read and write buses — and the ring of buffers that act as gates to control access to and from the buses and the other functional units on the chip. The following two sections first discuss the functionality of the two data buses, and then of the individual buffers that allow access to the buses.

3.4.1.1 Data Buses

The data buses are responsible for carrying data between all of the major sections of the Hydra chip: the CPUs, the L2 cache, and the external interface. The two buses can be written by several

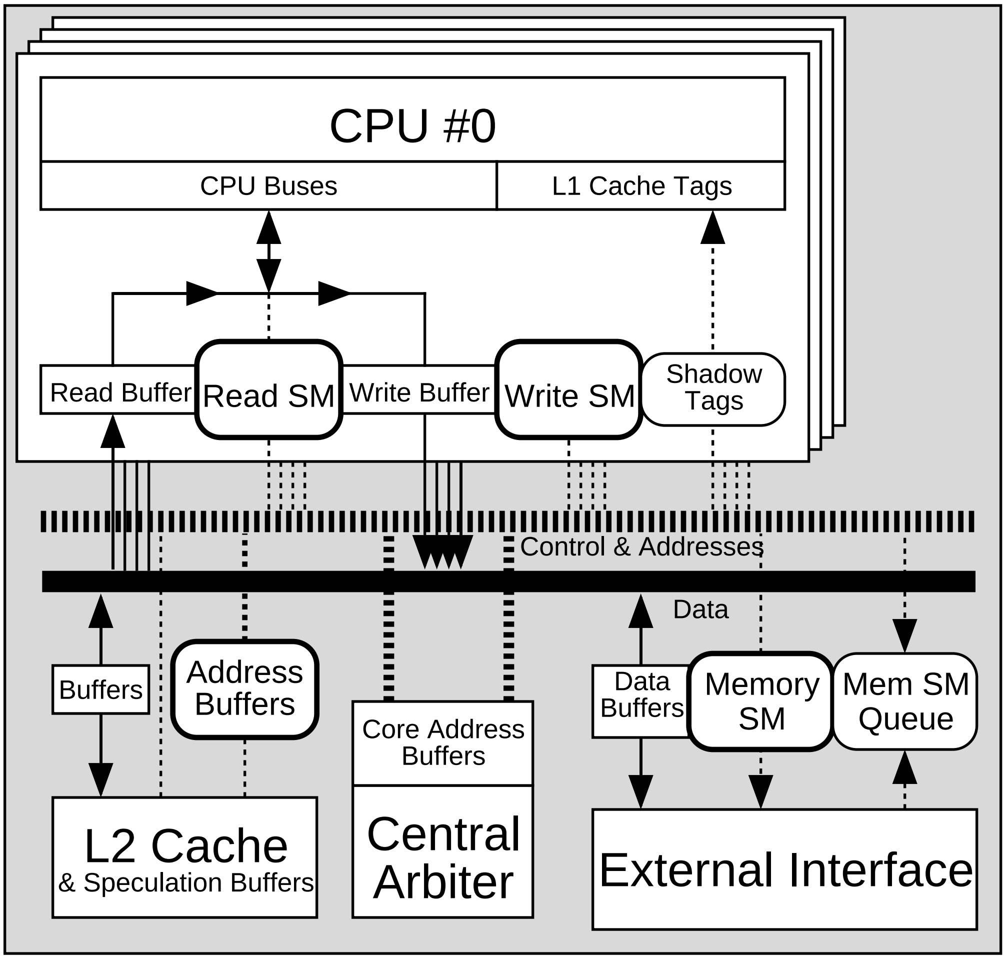


Figure 3-3: An overview of the Hydra memory interconnection system.

different drivers, so access is arbitrated on a cycle-by-cycle basis in the central arbitration mechanism (see Section 3.4.3.3). In most other respects, the two buses differ widely.

Read Bus: (256/128 bits wide) The "read" bus is responsible for carrying most of the traffic between the units on the Hydra chip. Half of the bus is used for carrying 16-byte L1 cache lines, while the full bus is used to carry 32-byte L2 cache lines to and from the external interface. Physically, the portion of the bus between the L2 cache and the external interface is twice as wide as the rest of the bus. Much of its work involves shuttling entire cache lines of data between the different units, so it is a fairly wide bus. Sometimes, critical words of data may be sent back to the CPUs before the rest of the line. In these cases, the data on only part

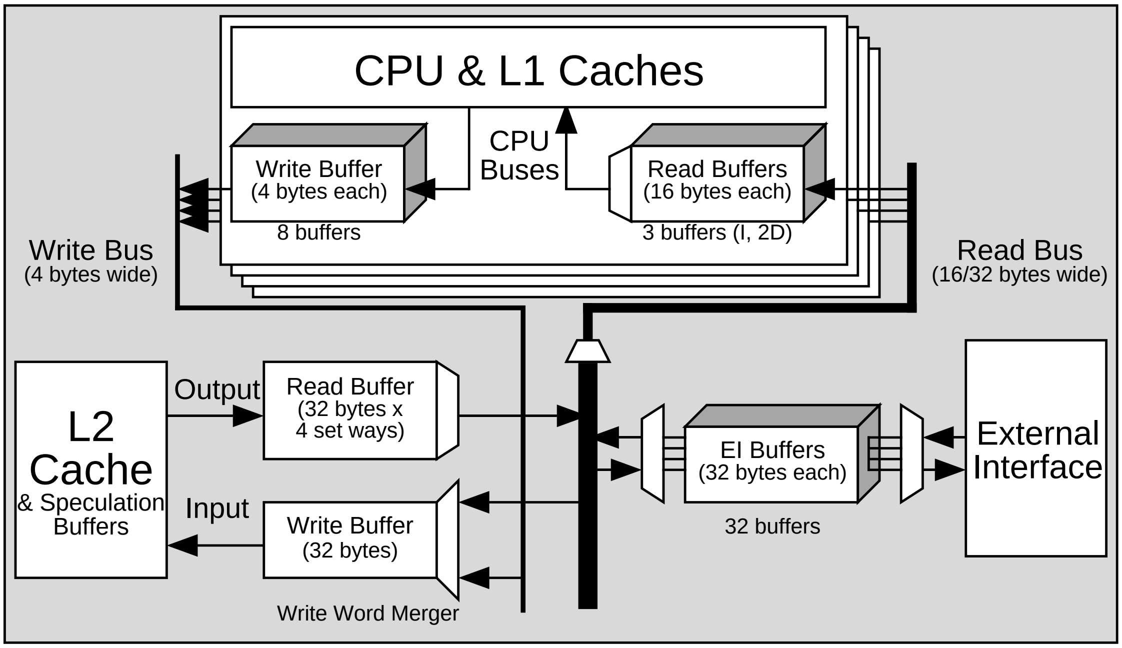

CPU & L1 Caches
CPU Buses
Write Buffer (4 bytes each)
8 buffers
Read Buffers (16 bytes each)
3 buffers (I, 2D)
Write Bus (4 bytes wide)
Read Bus (16/32 bytes wide)
L2 Cache & Speculation Buffers
Output
Input
Read Buffer (32 bytes x 4 set ways)
Write Buffer (32 bytes)
Write Word Merger
EI Buffers (32 bytes each)
32 buffers
External Interface

Figure 3-4: A summary of the data buses that connect the parts of Hydra together.

of the bus may be valid — but the controller will know which portion of the bus is valid on any particular cycle, and will be able to handle the partially-active bus correctly.

The sections of the bus interact through a series of muxes. When the external interface "broadcasts" the line coming back from memory, it puts the entire 32 bytes on the wide section of the bus. The L2 reads the full-width bus directly, thus filling its line in a single cycle. On the other hand, only half of the broadcast data is needed by the L1, so the mux passes just the critical half of the cache line onto the narrow portion of the read bus. This mechanism enables two sizes of lines to be delivered to different parts of the system in a single cycle.

Write Bus: (32 bits wide + support lines) The "write" bus is responsible for only a single task: handling the write-through data traffic from the CPUs to the L2 cache. Because the L1 caches do not filter out these accesses, a large number of them must be passed out through the bus at all times. To prevent the read bus from being constantly tied up by these transfers, they are carried on the special write bus. The data portion of the write bus is only 32 bits wide, since that is the maximum data size that any particular MIPS-I store instruction can write at once. Each write is allocated a cycle on the write bus to transfer its data. Other fields are also associated with the write bus, as described in Table 3-1.

Since different writes can carry differing amounts of data, a line is associated with each byte-lane that tells the L2 cache whether or not that particular byte is valid. The L2 cache can then take the valid bytes and store them away. A critical function of the write bus is to serialize the writes, so that writes from different processors appear to occur in a specific order. When each write's cycle on the write bus occurs, all other CPUs may see those new contents of the address for the first time. At this point, all older versions of the data in the L1 caches of the 3 other CPUs are invalidated, using the special invalidation port on the CPUs described in Section 3.3. This forces the other CPUs that may be sharing the data to read in the new data from the L2 cache.

After the speculation support described in Part II is added, further lines are required to control the speculative state of each access. First, a single bit line is added just to notify receivers that the access is indeed speculative, since speculative processors will typically mix and match speculative and non-speculative references during execution. Second, a 2-bit CPU identifier is sent along with each write so that other processors may determine how they should process the write during snoop accesses. Finally, the store PC is broadcast on a parallel bus so that violation statistics may be maintained by the processors. The write bus also gets a slightly enhanced protocol when invalidations are triggered by writes passing through it. Speculative CPUs that read incorrect data will be restarted after seeing writes of new data broadcast on the write bus.

Table 3-1: Write Bus Fields

Bus Segment	Size	Function
Write Data	32 bits	Carries write data to the L2 cache
Write Bus Valid	1 bit	Strobe indicating that the bus is active
Write Bus EIB #	5 bits	EIB number of the access that's driving the write bus
Active-byte controls	4 bits	Strobes that indicate which bytes of data are valid written data
Write Address Indicator	4 bits (or 9 if not encoded)	A set of lines that indicate which state machine is using the write bus, so that snooping CPUs will access the proper address bus. More likely, an entire 30-bit address bus can be placed here, to avoid muxing from the address buses at every fanout point.
Speculative Write Indicator	1 bit	Strobe indicating that this write is speculative, so invalidations should be processed accordingly and the L2 write buffers should handle this access
CPU Identifier	2 bits	Indicate the CPU that produced the write
Store PC Address	30 bits	Address of the store that made this write

3.4.1.2 Buffers

The ring of buffers around the central data buses are a set of flip-flop based registers that are responsible for latching data from the buses at the end of every cycle or driving data onto the buses whenever necessary. Most of the buffers are wider than the buses, so multiplexers are usually located at the inputs and outputs of the buses to properly align data as it is transmitted. In general, the multiplexers are used so that data moving around the chip are always aligned according to the low-order bits of the actual addresses of the data, for the sake of consistency. Moving around Figure 3-4, this section discusses the function of each of the buffers in turn.

L1 Write Buffers: (1 word each, 8 entries) The write buffers are critical in presenting a consistent view of memory to the processors, as well as facilitating synchronization. The full contents of each buffer are summarized in Table 3-2. The write buffers each hold a single word from a particular CPU's latest write operations. The write data is latched here until the controlling state machine manages to acquire the write bus for an L2 write cycle. If these buffers fill, then the processor must be stalled before it is allowed to perform another store operation.

While the write sits in the write buffer, it may be forwarded back to the CPU that just wrote it as part of a cache read, transparently — but *only* to the CPU that actually performed the write. To facilitate this operation, the write buffers associated with a particular processor are attached to a 16-byte wide branch off of the read bus that goes into the processor. At the fork there is a mux, controlled by the write buffer. When a line is broadcast back to this CPU over the read bus, all the write buffers snoop their contents, and set their muxes to merge the most recent data onto the CPU's branch of the read bus. The number of parallel compares required to control this muxing is the most important factor limiting the depth of the write buffer. Needless to say, the logic required to do the parallel snoop among the different write elements and then perform the necessary multiplexing is the most difficult bit of logic in the entire buffer system.

When the controlling state machine acquires the write bus, the buffer dumps one entry onto the write bus. At this time, the write becomes globally available to all four of the CPUs, and is cleared out of the buffer that was holding it. In addition to buffering the write data, the write buffer remembers if the write it is storing is the result of a store-conditional. If so, special handling is necessary to ensure that a failed store-conditional never gets a chance to commit. The write bus is the single mechanism that serializes writes, so the write buffers must continuously observe all other write transactions on the write bus. If they detect another write to the same address, then any write buffer entries holding store-conditional writes must invalidate itself. When an invalidated store conditional is given a chance to commit, the write does not pass over the write bus, and the CPU that made the request is notified of the failure. Otherwise, the SC commits normally and the processor is notified of the success.

L1 Read Buffers: (2 for data fetches, 1 for instruction fetches) Each L1 data-instruction cache pair has a designated read buffer that holds all read requests issued by the cache to lower memory. To service the instruction cache, the instruction read buffer latches the L1 instruction

Table 3-2: Write Buffer Fields

Field	Size	Function
Write Data	32 bits	Holds data being written
Write Address	32 bits	Holds address being written
External Interface & Jam Buffer Number	5 bits	Number of the external interface and address check buffer assigned to this access
Uncached Write Control Bit	1 bit	Bit that is set if this access is uncached
SC Bit	1 bit	Bit set if this is a Store Conditional reference
Store PC Address	30 bits	Address of the store that made this write (for use with speculation only)
Speculative Reference	1 bit	Bit indicating that this reference is speculative (speculation only)

request, and puts it on the request port to its designated state machine. Since the L1 instruction cache blocks on a miss, a one entry buffer will never overflow. The two data read buffers provide equivalent service for the L1 data cache. There are two buffers due to the fact that the R32364 CPU will not block until two cache misses are outstanding. If accesses are sent to both the instruction and data buffers simultaneously, the data access will be given higher priority because it presumably came from an older instruction in the instruction stream.

After initiating an access to lower memory using the proper state machine, these buffers snoop the read bus to latch the data into the appropriate L1 cache, acting as miss status handling registers (MSHRs). Since more than one data request can be in flight at any time, the buffers must do a full parallel compare of the returning access with all functioning entries that have issued to decide which one is being broadcast over the read bus on any particular cycle. The full list of fields necessary to fulfill this functionality is given in Table 3-3.

L2 Cache Read Buffer: (8 words for each of 4 cache ways) This buffer latches the output from all 8 ways of the L2 cache simultaneously, so any one of them may be accessed on the next cycle.

Table 3-3: Read Buffer Fields

Field	Size	Function
Read Address	32 bits	Holds address being read
External Interface & Jam Buffer Number	5 bits	Number of the external interface and address check buffer assigned to this access
Instruction Fetch Bit	1 bit	Notes if this is an instruction fetch
Uncached Read Control Bit	1 bit	Bit that is set if this access is uncached
LL Bit	1 bit	Bit set if this is a Load Locked reference
Speculative Reference	1 bit	Bit indicating that this reference is speculative (speculation only)

All 32 bytes of the entire cache line from each way are latched, so once the way is selected the entire cache line may be put on the read bus at once.

L2 Cache Write Buffer: (8 words) The L2 write buffer accepts one entire 32-byte cache line from the read bus at once or a single byte-to-doubleword write from the write bus for writing into the L2 cache. This buffer is always used, no matter which bus the data comes from. The output is written into the L2 cache on the next cycle, no matter what. Partial-line writes from the write bus are written directly into existing cache lines without writing over the rest of the line, since byte-lane selects are honored for writing into the L2 cache. If a write is to a line not in the L2 cache, then the write state machine controlling the write will make sure that the write appears on the write bus at the same time that the rest of the line is read from main memory on the read bus. On that cycle, the write buffer will latch the data from both into a new, composite cache line that can be written into the L2 cache all at once.

External Interface Buffers: (32 buffers of 32-64 + bytes each) The external interface acts as the “translator” between the external system bus protocol, and the protocol enforced on the internal read and write busses. These buffers must combine data from both the processor and the L2 cache into a format necessary for accessing main memory. In order to do this, several of the fields must be allocated during the L2 access, in order to collect bits of data as they become available. These “control” fields are summarized in Table 3-4. Fields without an asterisk after the name are loaded as the access starts accessing the L2 cache, from the state machine’s address bus, while the marked fields are loaded through the special writeback path from the L2, indicated on Figure 3-5, following the L2 cache access made for the reference. The address jamming mechanism eliminates the need for sophisticated disambiguation logic among these buffers, which would otherwise be necessary.

Table 3-4: External Interface Buffer Fields, control portion

Field	Size	Function
Writeback Address*	27 bits	Holds address of the line being written back to main memory, if any
Writeback Required*	1 bit	Signals that a writeback is occurring during this main memory access (set after L2 miss with dirty writeback)
Read Address	27 bits	Holds address of the new line being read into memory, if any
Read Required*	1 bit	Signals that the main memory and memory SM need to process this reference (set after L2 miss)
Memory Block*	1 bit	This is set to temporarily hold up a main memory read when there is no room in the L2 to receive the returned data (i.e. all lines in the set are locked by speculative writes)
Read/Write Buffer Number	4 bits	Number of the original CPU read or write buffer associated with this access
CPU Number	2 bits	CPU associated with this access
Uncached Access Control Bit	1 bit	Bit that is set if this access is uncached
Control-to-Data Pointer**	3? bits	Pointer from EIB control buffer to data buffer, if they are separate

The second part of the external interface buffers buffer the actual data going to and from main memory. The interface must buffer L2 write-backs until they can be committed to lower memory, and data coming back from main memory. Each buffer is capable of latching a 32-byte line. In the initial version of Hydra, these data buffers are directly attached to the control buffers, but in a more sophisticated design they could be separately arbitrated for at the start of the main memory access stage of the read and write state machines. With such a design, only a small number of physical data buffers would be necessary (probably 4-8, but this may need to vary depending upon the speed of the memory interface). In the latter case, the optional field pointing from each control buffer to its corresponding data buffer is required, as indicated by the double-asterisked entry. These buffers are loaded are allocated during the RSM/WSM main memory arbitration, and later loaded with data upon a writeback or when a line returns from main memory. The full list of fields is summarized in Table 3-5.

Table 3-5: External Interface Buffer Fields, data portion

Field	Size	Function
Writeback Data	32 bytes	Holds data being written back to main memory
Read Data	32 bytes	Holds data being read from main memory (if the memory interface timing allows, the buffers may be combined into one)

Each of the buffers may be controlled by any applicable state machines — the central arbitration mechanism for the read and write buses ensures that no two state machines attempt to use the same buffer at once.

3.4.2 Address Resources

Figure 3-5 gives an overview of the address-handling system used to transmit addresses among the different portions of the Hydra chip. It is centered around a group of 9 address buses — one controlled by each of the 9 state machines that synchronize the interconnection system. Since each state machine controls a different address bus, address-bus arbitration is not necessary.

Many units can drive signals onto the buses: CPUs (for most loads and stores), the external interface (to invalidate lines), and the caches (to write back lines from those caches). All drivers are dedicated to particular address buses, as is indicated by drive-in lines on Figure 3-5. In the figure, the buses are read 0, write 0, read 1, . . . , write 3, and memory. On the other hand, many units can

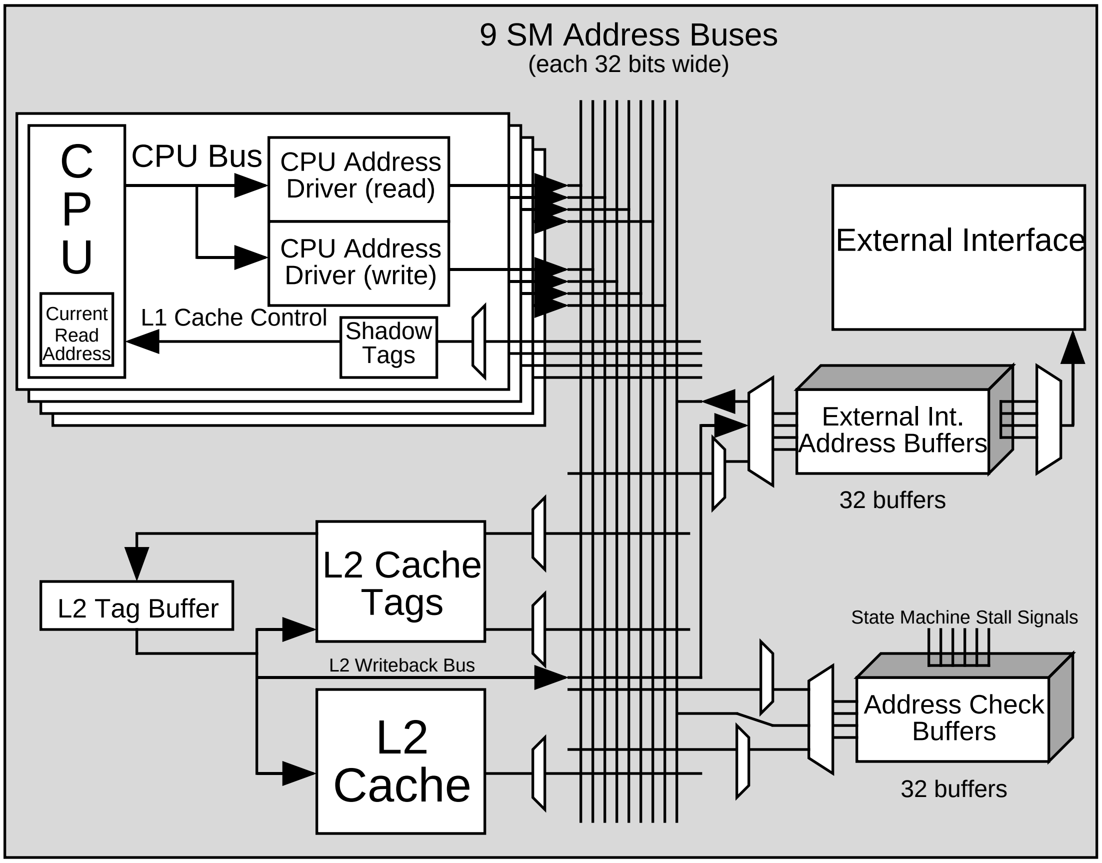


Figure 3-5: A summary of the address resources in Hydra

read addresses from any of the different buses. Lines out to “multiplexer” trapezoids indicate a port out of the buses that can read from any one of the address buses.

Details of most of the address resources are given in the chapters associated with the different portions of the Hydra chip. However, common resources that are shared by the units are noted below.

3.4.2.1 Address Format

The CPUs are designed to work with 32-bit virtual and physical addresses, so the entire Hydra memory system is built around 32-bit address buses. Virtual addresses are only used within the CPU / MMU core, so refer to the CPU core documentation [33] for more on their interpretation and translation. Figure 3-6 shows the different ways that the 32-bit physical addresses are broken down and used by the different portions of the Hydra chip. Here is a summary of the different formats, from the top:

CPU: The CPU produces the full 32-bit address after translating the virtual address to a physical one. Hydra makes no distinctions among the different portions of the address, although the CPU’s MMU may, depending upon what address translation modes it is currently using.

L1 Caches: At the L1 caches, the lower 4 bits are used as a block offset within a 16-byte cache block, while the next 8 bits are used to select one of the 256 sets in the 8K, 2-way associative caches. The upper 20 bits are then all used for the tags associated with each set.

L2 Cache: The L2 cache is considerably larger than the L1 caches, and therefore has a smaller tag. In a 128K, 4-way configuration 17 bits are allocated for the tag, 10 are used for the index (1024 sets), and 5 bits are needed for the block offset within the 32-byte blocks used in the L2 cache.

Memory: The main memory interprets the address as one big index, except for the 5 lower bits that form the block offset within a particular cache line. In the case of uncached memory references, even this distinction is moot, as these references are handled individually.

3.4.2.2 Address Buses

As previously mentioned, each state machine is associated with its own address bus, in order to eliminate a point of contention in the system and keep the arbitration mechanism simpler. Another critical reason for this breakdown is speed. By not requiring arbitration for the address buses, each state machine may put the address it wants to use on the address bus at the very beginning of a memory access, while arbitration for other resources is occurring. Since these are at most 32-bit

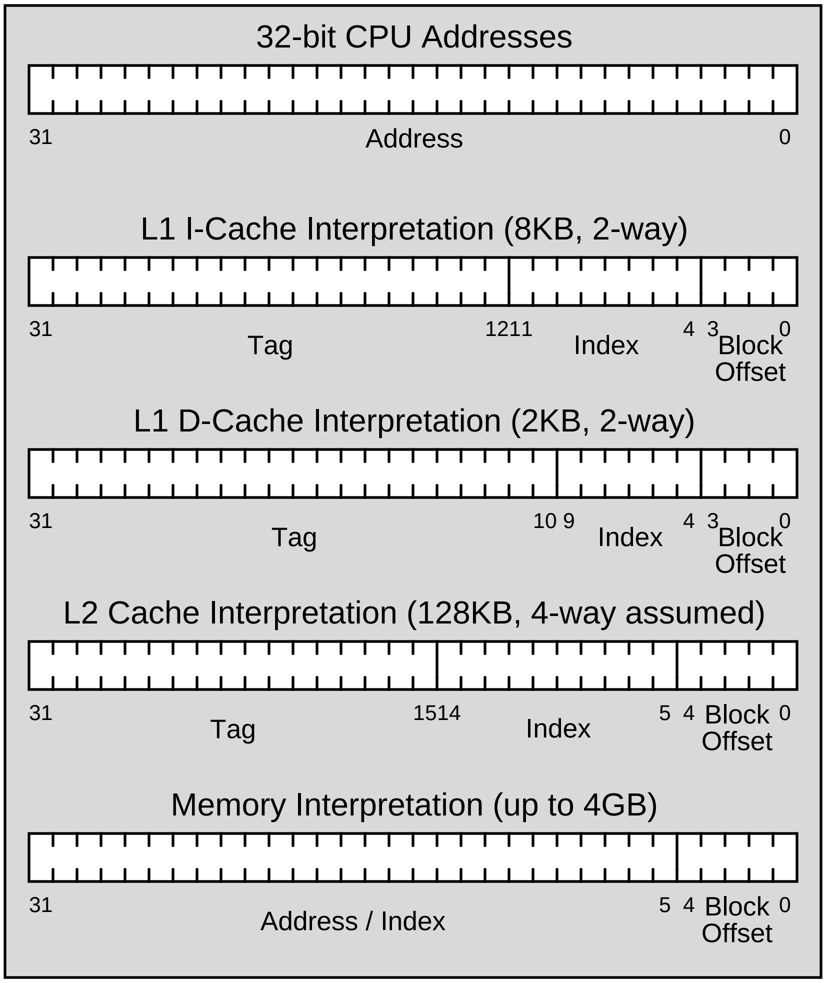

32-bit CPU Addresses
31
Address
0
L1 I-Cache Interpretation (8KB, 2-way)
31
Tag
1211
Index
4
3
Block Offset
0
L1 D-Cache Interpretation (2KB, 2-way)
31
Tag
10 9
Index
4
3
Block Offset
0
L2 Cache Interpretation (128KB, 4-way assumed)
31
Tag
1514
Index
5
4
Block Offset
0
Memory Interpretation (up to 4GB)
31
Address / Index
5
4
Block Offset
0

Figure 3-6: Address formats at different levels of the Hydra cache hierarchy

buses, running several of them for long distances is not as wasteful as making multiple copies of the data buses.

These buses are simple sets of wires, for the most part, which connect the pairs of memory controllers associated with the CPUs to the other CPUs (to allow them to snoop write operations), the L2 cache, and the external interface. Table 3-6 summarizes the wires included in each bus. The single controller associated with the external interface has a similar bus connecting it to the L2 cache and the CPUs. Finally, there is a small address bus connecting the L2 cache to the external interface which the L2 cache uses to pass writeback addresses to the external interface before writing data back to main memory.

Table 3-6: Address Bus Wires

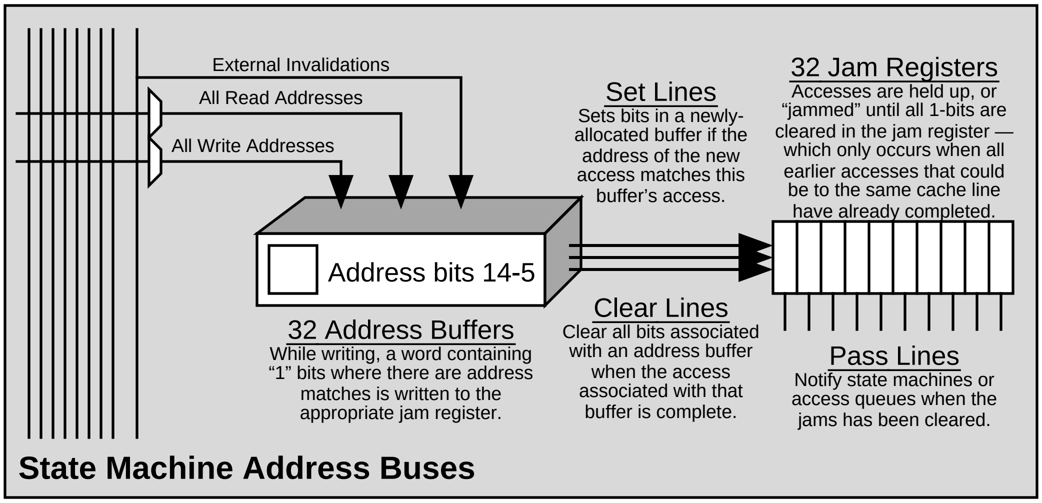
Field	Size	Function
Address	30 bits	Holds address of the line (or uncached word) being read or written
Read/Write Buffer Number	3 bits	Number of the read or write buffer associated with this access
External Interface & Jam Number	5 bits	Number of the external interface and address check buffers associated with this access
CPU Number	2 bits	CPU associated with this access
Uncached Access Control Bit	1 bit	Bit that is set if this access is uncached

3.4.2.3 Address Check Buffers

One possible problem that could occur in the system would be for 2 different CPUs to access the same address at once and cause a correctness problem due to an illegal interaction between the two state machines. For example, one could be writing back a line from a cache while the other reads a stale copy of the same line into its cache. To eliminate this possible problem, among others, the address check buffers provide a simple system for delaying accesses that may “collide” in the memory system. By delaying possible collisions, accesses that could cause correctness problems if they were allowed to run at the same time are serialized and the problems are inherently eliminated. It should be noted that this system is inherently conservative, and a less restrictive but more complex scheme might be able to avoid some of the “false sharing” that this scheme will serialize. A “possible” collision is defined as addresses that may have the same index bits during any cache access. With the address interpretation layout in Figure 3-6, bits 14–5 must match in order to force an access to block, allowing 1024 different addresses to be active at once. If desired, this could be increased in alternative implementations by decreasing L2 associativity and/or increasing L2 size, thereby increasing the number of L2 index bits, with corresponding increases in the number of possible active addresses.

The address check buffers are consulted in parallel with L2 tag checks. When each access misses in the L2 cache, it locks its address into an address buffer, depicted in Figure 3-5 and Figure 3-7. If the access hits in the L2 cache, the address is not permanently locked. One lock port is for reads, one for writes, and the third dedicated port is for the external interface controller. There are a total of 32 buffers, which are allocated using a free-buffer list that tracks the buffers which may still be allocated. This number was chosen to be equal to the number of memory buffers, so both could be arbitrated for with one arbitration.

As the address is locked into the buffer, a parallel compare with the other address buffers and any higher-priority locking addresses coming in at the same time (prioritized read, write, and finally memory) is performed. The results of all the compares come out and are stored in the “jam register” for that access. Usually, the compares will miss, and a zero-detector associated with each jam register will signal that the state machine is clear to continue. However, if any of the compares hit, then that access will “jam” and be queued until all earlier misses to the same address have been processed. Each access completion causes the “clear line” associated with the just-completed access to be raised, and all jam register bits that were set due to comparisons with that access’ address buffer are cleared out. When the jam register associated with a state machine is cleared, then the access is returned to the “new access” queue in the state machine and reprocessed as soon as possible. If an address is very popular, and has several state machines trying to use it at once, then a queue will effectively be formed by the collection of jam registers. This occurs because accesses that start later will have bits set in their jam registers for *all* of the preceding accesses to that address, and can therefore only occur after all of those accesses have completed. Because they are so common, writes that hit in the L2 cache must be sure to release their jam registers very quickly, to prevent queues from forming unnecessarily just because everyone is writing to the same cache line.



External Invalidations
All Read Addresses
All Write Addresses
Set Lines
Sets bits in a newly-allocated buffer if the address of the new access matches this buffer's access.
32 Jam Registers
Accesses are held up, or "jammed" until all 1-bits are cleared in the jam register — which only occurs when all earlier accesses that could be to the same cache line have already completed.
Address bits 14-5
32 Address Buffers
While writing, a word containing "1" bits where there are address matches is written to the appropriate jam register.
Clear Lines
Clear all bits associated with an address buffer when the access associated with that buffer is complete.
Pass Lines
Notify state machines or access queues when the jams has been cleared.
State Machine Address Buses

Figure 3-7: The address-check buffers in the Hydra chip

3.4.3 Control: The State Machines

Ten state machines provide control for the interconnection system. They are located where accesses may be initiated: 2 at each CPU (1 to handle reads, and 1 to handle write-throughs) and 1 at the memory interface (to control returning data from main memory and external intervention accesses). Each "state machine" is actually made up of several smaller state machines, each of which can take an access through a portion of the steps needed to handle that access. The different small state machines may each be handling a different access simultaneously. If a particular access requires the services of several different small state machines, it is passed between them using queues inside of each individual memory controller.

3.4.3.1 State Machine Structure

The structure of an interconnection system state machine is pictured in Figure 3-8. The core of the state machine (the darker section in the center of the figure) is essentially just a textbook design. Due to speed considerations, the next-state and output logic will probably have to be "hard-wired" logic. There is one slightly deceptive feature about this portion of the diagram, however. On the real chip, the next-state and output logic will actually be distributed throughout the controlled elements, since otherwise numerous control lines will have to be run across much of the chip, instead of just the encoded state and a few feedback lines. The other parts of Figure 3-8 are described in Section 3.4.3.2 for the FIFOs at the top, and Section 3.4.5.1 for the state counters at the bottom.

The state machines sequence actions primarily by controlling the various buffers that surround the data buses, locking data into the buffers and enabling the outputs of the buffers at appropriate times. In the caches, they also control some internal sequencing signals.

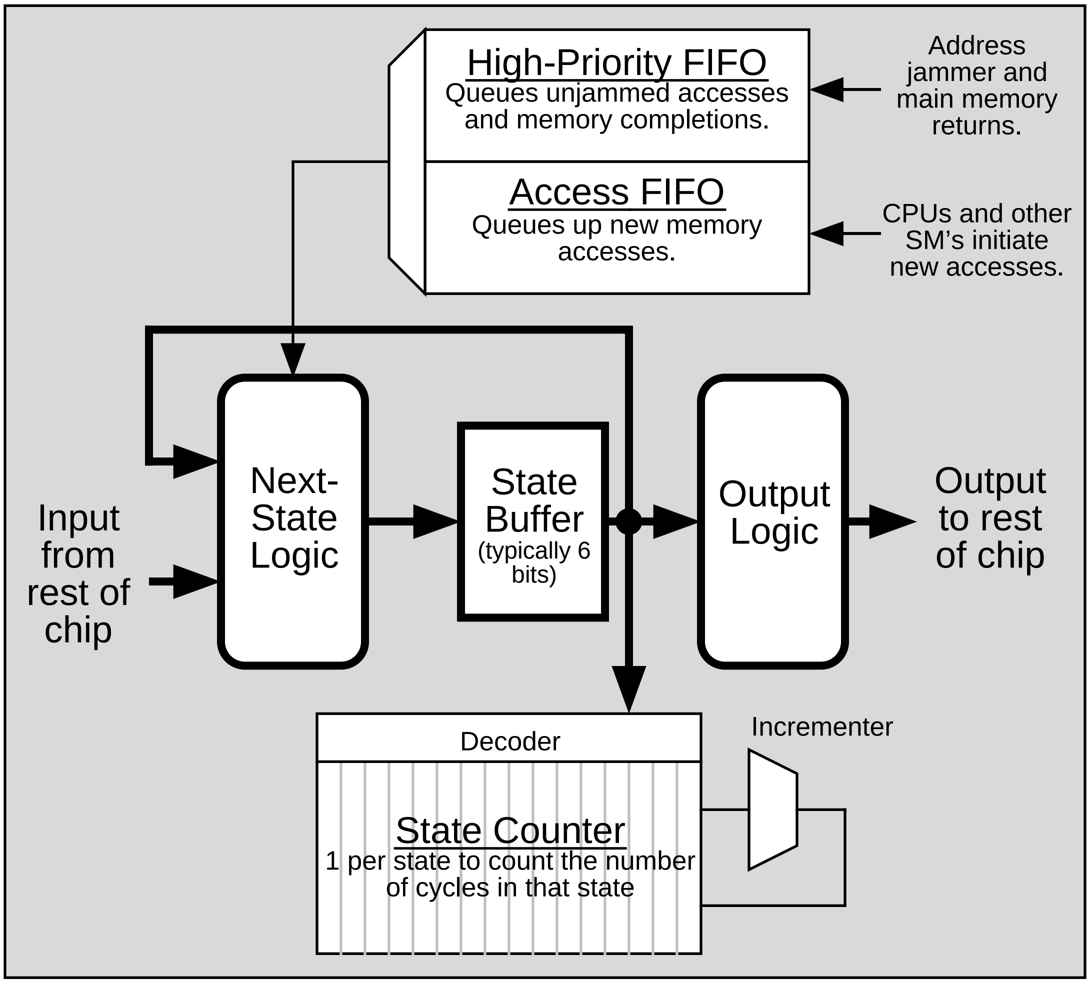

High-Priority FIFO
Queues unjammed accesses and memory completions.
Address jammer and main memory returns.
Access FIFO
Queues up new memory accesses.
CPUs and other SM's initiate new accesses.
Input from rest of chip
Next-State Logic
State Buffer
(typically 6 bits)
Output Logic
Output to rest of chip
Decoder
Incrementer
State Counter
1 per state to count the number of cycles in that state

Figure 3-8: A typical state machine that controls the memory system

3.4.3.2 Event FIFOs

All of the state machines possess an “idle” state in their state encodings, where they hold when they are not in use. Most of the state machines wait for a particular event before leaving the idle state: read SMs wait for reads either directly from their associated CPU or from the address jamming mechanism, write SMs wait for writes from their CPU or address jammer, and the memory SM handles many different requests. Therefore, these SMs need queues to collect all of these events and feed them to the state machine one at a time. In the actual system, 2 queues will be used — one for high priority work (unjammed accesses for read and write state machines or unjammed external interventions for the MSM) and one for lower priority requests (new accesses from the CPUs for read and write SMs, or all other requests for the memory state machine). In addition to assigning a priority to the events, each queue can be simpler, since fewer write ports will be necessary on both queues. Inside each large SM, smaller but similar FIFOs with no priority control are also included to buffer accesses moving from one sub-SM to the next within a particular memory controller.

The queues themselves are fairly simple circular buffers, with head and tail pointers. Whenever the SM hits the “idle” state, it tries to grab an event off of the head of the high-priority queue, or the low-priority queue if the high-priority queue is empty. New events are written on at the tail of the queues. Read SMs need 2 low-priority ports, since a CPU can only start 2 reads per cycle (instruction and data). However, they need 3 high-priority ports to handle the case when all three read accesses from a CPU are unjammed at once. Write SMs need 1 and 8 ports, for similar reasons. For the memory SM, only a single write port can write the tail of the high-priority queue each cycle, since only the external interface can put entries on this queue. The low-priority queue also has a single port, where accesses that miss in the L2 cache can place read and writeback requests. Since only one other state machine may use the L2 cache at a time, only a single port is necessary here, also.

Each entry in the queues for the read and write state machines consists of a pointer to an entry in the L1 read or write buffers (4.2.1.2) associated with the access being queued. A pointer to each entry will always be somewhere in the system, feeding the access represented by the buffer into the appropriate portion of the read or write state machines. Table 3-7 summarizes the places where read or write buffer pointers may be found as they flow around in a continuous cycle from the buffer free list and through the system.

Table 3-7: Buffer Pointer Queues

Location	Function
Free List	Holds pointers to buffer entries that are not being used.
R/W State Machine Low-Priority Queue	Holds pointers to entries that have just been read or written by a CPU, and are waiting for processing by a read or write state machine.
R/W State Machine High-Priority Queue	Holds pointers to entries that have just been unjammed, and are waiting for processing by a read or write state machine
R/W State Machine Pointer Register, L2 Access Stage	Holding register for pointers being processed by the first half of the read or write state machines.
R/W State Machine L2-to-Memory Queue	Holding queue for pointers moving from the first half of the read or write state machines into the second half. This queue may be removed, with some loss in performance, if an additional state is added to the L2 access stage
R/W State Machine Pointer Register, Memory Access Stage	Holding register for pointers being processed by the second half of the read or write state machines.
External Interface Buffers	Pointers are held here while references are working their way through the memory system.

Simultaneously, External Interface Buffer pointers flow around the system in a much less cyclic path in order to properly control allocation of the EIBs to various accesses being processed by the system. The memory state machine and the external main memory interface itself use these pointers to control how they process references. The locations where these pointers are used is summarized in Table 3-8.

These queues completely sequence accesses through the memory state machine (MSM) and main memory. Figure 3-9 shows the basic flow of an access through these paths.

Table 3-8: External Interface Buffer Pointer Queue

Location	Function
Free List	Holds pointers to empty External Interface Buffers. EIB pointers are dumped here when they are eliminated from L1 R/W buffers, after L2 accesses hit or following main memory references.
L1 R/W Buffers	During the L2 access stage of each reference, EIBs are stored here only. During main memory references, the copy is maintained here so that data values returning from the main memory may be associated with the proper reference.
Outgoing-to-Memory Queue	Holds EIB pointers to entries that are waiting for their turn to use the main memory interface.
In Memory Buffers	Holds EIB pointers to the entries that are currently being processed by the memory system. These buffers are contained in the memory controller itself, and pipeline EIB pointers around along with the reference through the memory system.
Back-to-SM Queue	Holds EIB pointers that are going back to the memory state machine, so it may process each EIB at the correct time. This may actually hold the same EIB more than once, since entries are put here for each section of a line coming back from memory, as it arrives.

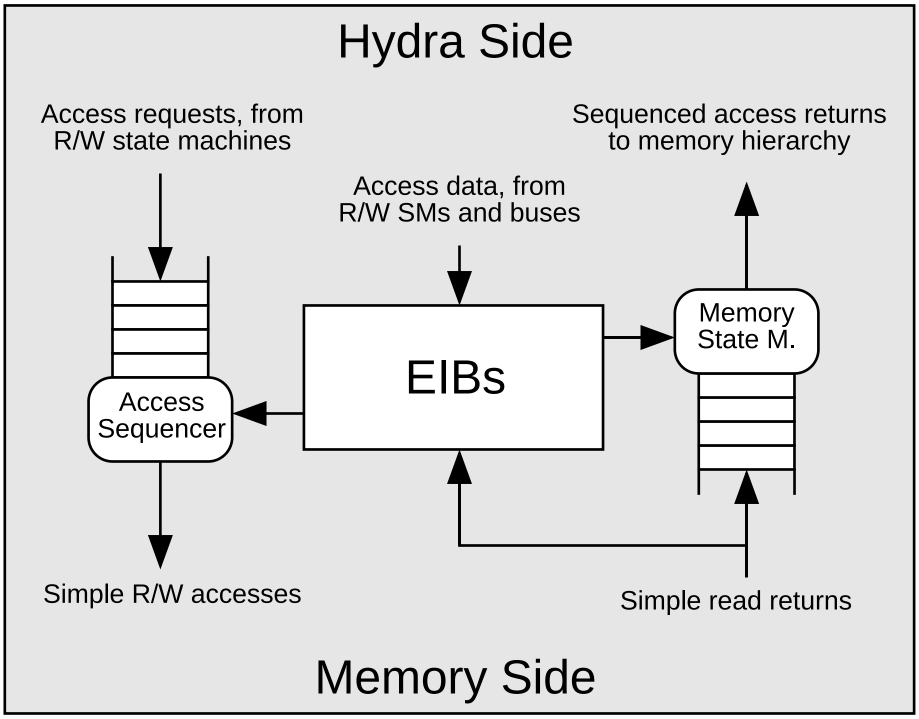

Hydra Side
Access requests, from R/W state machines
Sequenced access returns to memory hierarchy
Access data, from R/W SMs and buses
Memory State M.
EIBs
Access Sequencer
Simple R/W accesses
Simple read returns
Memory Side

Figure 3-9: Flow of memory references through the main memory interface.

3.4.3.3 The Central Arbiter

The arbiter is a single circuit that centralizes the decision-making needed to ensure that multiple state machines do not attempt to use shared resources in the same cycle. With nine different memory controllers containing 18 different state machines all attempting to perform accesses at once, there will often be collisions between different accesses that need the same resources. The central arbiter resolves these conflicts, while enforcing strict priority ordering among requests. The central arbiter watches for any "requests" made by the different state machines. When a state machine needs resources, it activates the appropriate request line, which the arbiter understands to correspond to a specific set of resources needed on specific set of cycles in the near future. The arbiter prioritizes and compares all requests being made on this cycle, and then sends "continue" signals to as many requestors as possible who do not happen to need the same resources at the same time. The current request priority, described below, is used to break any ties. Requestors that are denied access to their desired resources must simply try again until they finally succeed at getting the desired arbitration.

The arbiter controls access to seven shared resources, which are listed and described in Table 3-9.

Table 3-9: Arbiter-controlled Resources

Resource	Function
ReadBus	The 128/256-bit bus that carries all the read data, as well as the L2 write backs.
WriteBus	The 32-bit write bus that broadcasts the CPU writes and speculation commands so that all CPUs may see the writes and synchronize on them.
L2TagRead & Write	The L2 tag read port (and the write port, 2 cycles later). Latency = 2 cycles. Throughput = 1/cycle. This also controls access to the address jammer, which is always accessed simultaneously.
L2DataReadWrite	The L2 array read/write port. Latency = 2 cycles. Throughput = 1/cycle.
Mem Address Port	The single port to the external interface buffers, used to initiate requests to main memory
MemBuffer	Counter used to keep track of the number of available address check buffers and writeback buffers in the external interface. If a reduced number of EIB data buffers is desired, the data registers must be arbitrated for separately.

The “MemBuffer” resource is different from the others, in that it is not allocated on a cycle-by-cycle basis. Instead, all arbitrations requiring access to a memory buffer are not allowed while no memory buffers are left remaining. They are forced to retry their arbitration until a buffer is free. The other resources need to be allocated ahead on a cycle-by-cycle basis, because accesses use them in a pre-defined sequence. To control access to those resources, the arbiter maintains a small shift register for each one. The bits in the registers correspond to cycles, so the arbiter has a short look-ahead capability. The cycle following the arbitration is bit “zero” of each register, and all the registers are shifted one bit on each cycle. A “one” in the shift register signifies that the resource has been allocated on that particular cycle. The shift registers are best seen as a small array, in which the arbiter tries to find specific patterns of “holes” of unused resources over the next few cycles that match the requested allocation pattern. Where such a “hole” is found, it is filled by recording which resources are now allocated on which cycles, and subsequently by notifying the requestor that their arbitration has been granted.

Table 3-10 shows the various arbitrations that may occur, and the resources needed to satisfy them. The numbers in the table show for which cycles ahead of the current one a resource must be reserved. A “-” signifies a resource that is not needed for the particular arbitration.

Requests coming from the state machines may conflict on any given cycle, in which case the central arbiter must resort to priority ordering among state machines to resolve the conflict. Any request will fail if it requires reallocation of a previously allocated resource. If two simultaneous requests attempt to use the same resource that is presently free, then the one with the higher priority will win the allocation. The arbiter orders the state machine priorities as follows: memory, read, and write. This overall order is fixed, since requests coming back from the main memory should occur as quickly as possible, and after that quick completion of reads is generally more important than write completion. However, there are also 4 read and 4 write state machines, one for each CPU, which can produce identical accesses. To order these, the arbiter keeps track of a

Table 3-10: Arbitrations and Allocations

Resource Arbitration	Read Bus	Write Bus	L2 Tag Read Write	L2 Data Read Write	Mem Address Port	Mem Buffer
MemRB	0	-	-	-	-	-
MemRBL2	0	-	-	1	-	-
MemRWBL2	0	0	-	1	-	-
MemEI	2,3	2,3	0,1	0,1	0,1	Get 1
ReadL2	2	-	0	0	-	Get 1
WriteL2	-	1	0	2	-	Get 1
Read/WriteMemClean	-	-	-	-	0	Get D*
Read/WriteMemDirty	2	0,1	-	0	0	Get D*

round-robin priority scheme, so that each will get a turn at being the high-priority request owner about a quarter of the time.

The required resource allocations within three different groups overlap for certain groups of arbitrations. Conflicts caused by these overlaps are resolved through enforcement of priorities. If any arbitration from a given group is granted, the others will stall, while two arbitrations from different groups may be granted on the same cycle. The three groups of arbitrations are shown in Table 3-11. The priority enforced within each group is the order in which the arbitrations are listed. Only the specialized MemEI arbitration appears in more than one group, as it performs the jobs of arbitrations in more than one other group.

Table 3-11: Groups of arbitrations that may conflict

Group I	MemEI, MemRB, MemRBL2, MemRWBL2
Group II	MemEI, ReadMemClean, WriteMemClean, ReadMemDirty, WriteMemDirty
Group III	MemEI, ReadL2/WriteL2, WriteL2/ReadL2, MemEI

There is one occasional exception to the “reads before writes” priority rule. The priorities within Group III are not completely fixed. Instead, they are based on occupancy of the various buffers. Normally the reads are allowed to bypass writes, so they are given higher priority. But when any of the write buffers in the system begins to fill up, the writes acquire higher priority in order to clear out the buffer before it fills up and stalls the system. This causes the write buffers to empty quickly, which restores the normal priorities.

The arbiter is the home of one of the trickiest circuit design issues on the entire chip. Most of our complex circuitry can be distributed and simplified so that it may work at the required clock frequencies — the primary problem is just driving long wires from one end of the chip to the other. In the arbiter, however, it is necessary to have requests coming in from all of the state machines, then to check all of the resources in parallel, and then to notify the state machines whether or not they may continue — all in a single cycle. Due to the large numbers of inputs and outputs, it is impossible to avoid several large-fanin gate structures (that will actually probably be trees of smaller gates in the final circuits).

3.4.4 The Operation of the System

This section gives an overview of how the different state machines coordinate the series of steps responsible for all of the different types of memory accesses. Each of the four subsections describes the operation of one of the three types of state machines in detail, by fully describing its key access pattern. Table 3-12 through Table 3-16 describe full state summaries for the state machines. These are provided to allow the reader to examine key details of the control on a state-by-state level. It should be noted that the memory state machine controls all responses from memory, so the other three flowcharts do not cover the final stages of an access that must recover data from main memory.

3.4.4.1 The Read State Machines

Each CPU has a designated “read” state machine, which is responsible for handling that CPU’s read accesses to the L2 cache and initiating access to lower memory. The read state machine is roughly pipelined into two multicycle pipe stages to allow high throughput and minimum occupancy of resources shared among the processors using only a relatively simple controller. The first pipeline stage controls the L2 access, while the second controls the transfer of the access to the external interface so it may be sent to main memory. If an access hits in the L2, the access completes without ever proceeding to memory access pipeline stage. Otherwise, the read continues on into the second pipeline stage in order to be sent to memory. If the read is to an uncached line, the L2 cache stage is bypassed, and the read proceeds directly to main memory. This mechanism allows for up to two read accesses to be in progress in any state machine at any time.

The pipelined state machine is best viewed as two cooperating state machines, each with a designated “idle” state that watches for conditions that will start it on its way. A state machine stage takes a read access, and moves it through a series of arbitrations and associated stalls that sequence through the protocol required by the resources controlled by that stage. As the state machine stage sequences though its states, it drives control lines that direct the read access through the data path of the controlled resource. If the read access hits in the L2, the state machine completes the access. Otherwise, the access passes to the memory interface and its associated state machine, and the read state machine starts processing the next access.

The state machine is best understood by enumerating the possible states, and describing the function of each state. The complete state transition table is shown for each stage of the state machine after the stage has been described. The state transitions described are summarized in Table 3-12. Note that the input and output names in that table are not formal names of signals, but rather abbreviated statements of circuit conditions. The following list fully describes the states of the first stage of the state machine:

RL2Idle: This is the “standby” state of the first stage. The machine monitors the read buffer and the jam buffer, looking for an access that needs processing. If an access is found in one of the queues, the state machine fires an “Output Enable” line going to that queue. The jam queue is given priority over the read queue because the accesses residing there are presumably older. The OE signal that goes to each queue only facilitates book keeping. In reality, the queue is always driving the next access on its port, waiting for the state machine to grab it. When the OE signal goes high, the access is latched into the state machine, and the queue can deallocate an entry and drive the next access onto its port. Meanwhile, the state machine puts the address of the access that it is beginning to process onto its L2 address bus. If that access is cacheable, the machine proceeds to “RL2ReadArb.” Otherwise, it goes to “RL2MissStall.”

RL2ReadArb: The state machine raises the arbitration request line that tells the arbiter that it intends to read the L2 cache. The machine remains in this state, with the request line raised, until the arbiter responds with a corresponding grant signal. When the grant signal for this state machine goes high, it causes many things to happen in the system. The address that the state machine has been driving on its designated L2 address bus is muxed to the L2 address port. The L2-address-valid strobe fires. The same address is muxed to the input of the address jammer, and the jam-address-valid strobe fires. These two events initiate the L2 access and an address jam check. The machine then goes on to “RL2ReadA.”

RL2ReadA: This is the first cycle of the L2 read access. The next state is “RL2ReadB.”

RL2ReadB: In this cycle, the tag compare of the L2 is complete, and the L2 asserts the L2-hit line if any of the lines in the cache hit. The data from all four sets of the L2 are available, and are latched in the internal data buffer at the end of the cycle. If the L2-hit line is high at the end of the cycle, the next state is “RL2Hit.” If the access is a cache miss, then the address jammer, which is accessed in parallel with the L2, is consulted to see which state is next. If the address jammer jams the access, the machine returns to “RL2Idle,” temporarily forgetting the current access. The dropped access will be added to the jam queue after its jam clears, which will allow it to be retried. If the address is unjammed (the common case), the machine proceeds to “RL2Miss.”

RL2Hit: In this cycle, the state machine drives the L2-buffer-enable line, which causes the L2 state latched in the previous cycle to update the L2 tag array. It also drives the L2-data-output-enable line, which causes the contents of the L2 line to be gated onto the 512-bit section of the read-bus. The data is grabbed off the bus by the resources that need it, and the L2 access is complete. Meanwhile, the external interface buffer reserved for this access is released. The state machine returns to “RL2Idle.”

RL2Miss: In this cycle, the state machine drives the L2-buffer-enable line, which causes the L2 state latched in the previous cycle to update the L2 tag array. While the state machine has the read bus for this cycle, the reservation is wasted. If the state machine sees that a writeback is generated, it sets a bit that makes it remember that the current L2 access generated a write-back. This bit is used in the following state machine stage. The state machine also checks to see whether the next pipe stage (the memory read) is ready. If ready, the address is handed over to the memory read stage, and this stage goes to “RL2Idle.” Otherwise, the next state is “RL2MissStall.”

RL2MissStall: This “state” is not really a state at all, but instead a small queue in which the access can wait for the memory read stage to complete its current access. When the memory stage is ready, the address is handed over, and this stage goes back to “RL2Idle.”

Table 3-12: State Transitions — L2 Read Stage

Current State	Input	Next State	Output
RL2Idle	nothing-valid	RL2Idle	
	queue-valid & cacheable	RL2ReadArb	
	queue-valid & non-cacheable	RL2MissStall	
RL2ReadArb	!L2-rd-arb-grant	RL2ReadArb	L2-rd-arb-request
	L2-rd-arb-grant	RL2ReadA	L2-rd-arb-request
RL2ReadA		RL2ReadB	
RL2ReadB	L2-hit	RL2Hit	L2-buffer-latch
	!jammed AND !L2-hit	RL2Miss	L2-buffer-latch
	jammed AND !L2-hit	RL2Idle	
RL2Hit		RL2Idle	L2-buffer-drive, L2-data-OE
RL2Miss	mem-stage-idle	RL2Idle	L2-buffer-drive
	!mem-stage-idle	RL2MissStall	
RL2MissStall	!mem-stage-idle	RL2MissStall	
	mem-stage-idle	RL2Idle	

The second stage (memory read) of the state machine has an idle state during which it checks if the first stage has an access ready for it. An access becomes ready when the first stage processes an L2 miss, or when it finds an uncached access in one of its queues. The memory read stage takes the access from the first stage, and then the first stage is free to return to its “idle” state. The following is a listing of the memory stage states, with a brief description of each one. Each is summarized in Table 3-13, also.

RMemIdle: The memory stage is idle, waiting for the L2 read stage to pass down an access. While in this state, the state machine monitors the L2-write-back strobe, as well as the dirty-bit set by preceding stage. If an access comes down while either of those is high, the next state is “RMemDirtyArb.” Otherwise it is “RMemCleanArb.” The address of the access is driven out onto the state machine’s memory address bus.

RMemCleanArb: The state machine must acquire access to the external memory address bus. It raises an arbitration request line, and waits for the arbiter to return a corresponding grant. The next state is “RMemClean.”

RMemClean: The state machine memory address (from its memory address bus) is issued to the external interface through the memory address port. The next state is “RMemIdle.”

RMemDirtyArb: This state is entered whenever the state machine is writing a memory line that will take place of a dirty line in the L2. The L2 must write back its dirty data to the external interface buffer, and the arbiter, which controls the accesses to that buffer, must be notified of the write-back. The state machine notifies the arbiter that a write back has taken place by raising this arbitration line. The machine waits in this state until the corresponding grant line is raised by the arbiter. The next state is “RMemDirtyInv1.”

RMemDirtyInv1: The state machine memory address (from its memory address bus) and the writeback address (from the writeback address bus) are issued to the external interface through the memory address port. Meanwhile, an invalidation to the first half of the dirty line being removed is broadcast over the write bus to ensure that L1 inclusion in the L2 is maintained. The system also starts a data read of the dirty line (see RMemDirtyInv2) during this state. The next state is “RMemDirtyInv2.”

RMemDirtyInv2: This state completes the invalidation process begun on the previous cycle. Since every L2 line contains two L1 lines, two invalidations must be generated for every L2 write back. A single bit is flipped in the memory address, and the second invalidation is broadcast. The system also concludes its data read of the dirty line during this state. The next state is “RMemDirtyWB.”

RMemDirtyWB: During this stage, the dirty line is written to the external interface buffer using the read bus. The next state is "RMemIdle."

Table 3-13: State Transitions — Memory Read Stage

Current State	Input	Next State	Output
RMemIdle	nothing-valid	RMemIdle	
	Access_ready & !dirty	RMemCleanArb	
	Access_ready & dirty	RMemDirtyArb	
RMemCleanArb	!Clean-grant	RMemCleanArb	Clean-arb-request
	Clean-grant	RMemClean	Clean-arb-request
RMemClean		RMemIdle	
RMemDirtyArb	!Dirty-grant	RMemDirtyArb	Dirty-arb-request
	Dirty-grant	RMemDirtyInv1	Dirty-arb-request
RMemDirtyInv1		RMemDirtyInv2	
RMemDirtyInv2		RMemDirtyWB	
RMemDirtyWB		RMemIdle	

3.4.4.2 The Write State Machines

Each CPU also has a designated "write" state machine, which is responsible for handling that CPU's writes to the L2 and lower memory. The write state machine is very similar to the read state machine in its implementation. There are minor differences in the L2 cache access, since a write operation is performed to merge data into the L2 cache instead of reading a line. The second stage, which initiates accesses to lower memory, is virtually identical to the read state machine. Both just initiate the read of a cache line from lower memory. For a write, the system just has to remember to perform the write after the line is returned from main memory, but this does not affect the structure of the write state machine since that part of the access is handled by the memory state machine.

If the write is to an uncached line, the L2 cache stage is bypassed, and the write proceeds directly to memory under the control of the external interface.

The following list, summarized in Table 3-14, describes the states of the first stage of the state machine:

WL2Idle: This is the "standby" state of the first stage. When speculation has been added, it is forced to sit idle by a control line from the L2 if the L2 buffers are not ready to accept more writes during speculation. Otherwise, the machine monitors the write buffer and the jam buffer, looking for an access that needs processing. If an access is found in one of the queues, the state machine fires an "Output Enable" line going to that queue. The jam queue is given priority over the write queue because the accesses residing there are presumably older. The OE signal that goes to each queue only facilitates book keeping. In reality, the queue is always driving the next access on its port, waiting for the state machine to grab it. When the OE signal goes high, the access is latched into the state machine, and the queue can deallocate an entry and drive the next access onto its port. Meanwhile, the state machine puts the address of the access that it's beginning to process onto its L2 address bus. If that access is cacheable, the machine proceeds to "WL2WriteArb." Otherwise, it goes to "WL2MissStall."

WL2WriteArb: The state machine raises the arbitration request line that tells the arbiter that it intends to write the L2 cache. The machine remains in this state, with the request line raised, until the arbiter responds with a corresponding grant signal. When the grant signal for this state machine goes high, it causes many things to happen in the system. The address that the state machine has been driving on its designated L2 address bus is muxed to the L2 address port. The L2-address-valid strobe fires. The same address is muxed to the input of the address jammer, and the jam-address-valid strobe fires. These events initiate the L2 access and an address jam check. The machine goes on to "WL2TagCheckA."

WL2TagCheckA: This is the first cycle of the L2 tag read access. The next state is "WL2TagCheckB."

WL2TagCheckB: In this cycle, the tag compare of the L2 is complete, and the L2 asserts the L2-hit line if any of the lines in the cache hit. The state machine drives its L2-buffer-select line high. If the L2-hit line remains low, the L2 access is a miss, and the address jammer, which is accessed in parallel with the L2, is consulted to see which state is next. If the address jammer jams the access, the machine returns to "WL2Idle," temporarily forgetting the current access. The dropped access will be added to the jam queue after its jam clears, which will allow it to be retried. If the address is not jammed (the common case), the machine proceeds to "WL2Miss." If the access is a hit, the L2-hit signal causes the data from the write queue to be put on the write bus, which makes the write visible to the system. The L1 caches see it, and begin any necessary invalidations. The write bus data in aligned and latched into the L2 input buffer. The write happens during the next state, which is "WL2WriteToCache."

WL2WriteToCache: In this cycle, the state machine drives the L2-buffer-enable line, which causes the L2 state latched in the previous cycle to update the L2 tag array. The write that was latched into the input buffer is put on the internal L2 write bus, and the appropriate byte-select lines fire, causing the data to be merged "in-place" into the L2 array. Meanwhile, the external interface buffer reserved for this access is released. At this point the write is considered complete, and the state machine returns to "WL2Idle."

WL2Miss: In this cycle, the state machine drives the L2-buffer-enable line, which causes the L2 state latched in the previous cycle to update the L2 tag array. If the state machine sees that a writeback is generated, it sets a bit that makes it remember that the current L2 access generated a writeback. This bit is used in the following state machine stage. The state machine checks whether the next pipe stage (the memory read) is ready. If ready, the address is handed over to the memory read stage, and this stage goes to “WL2Idle.” Otherwise, the next state is “WL2MissStall.”

WL2MissStall: This “state” is not really a state at all, but instead a small queue in which the access can wait for the memory read stage to complete its current access. When the memory stage is ready, the address is handed over, and this stage goes back to “WL2Idle.”

Table 3-14: State Transitions — L2 Write Stage

Current State	Input	Next State	Output
WL2Idle	nothing-valid	WL2Idle	
	queue-valid & cacheable	WL2WriteArb	
	queue-valid & non-cacheable	WL2MissStall	
WL2WriteArb	!L2-wr-arb-grant	WL2WriteArb	L2-wr-arb-request
	L2-wr-arb-grant	WL2TagCheckA	L2-wr-arb-request
WL2TagCheckA		WL2TagCheckB	
WL2TagCheckB	L2-hit	WL2WriteToCache	L2-buffer-latch
	!jammed AND !L2-hit	WL2Miss	L2-buffer-latch
	jammed AND !L2-hit	WL2Idle	
WL2WriteToCache		WL2Idle	L2-buffer-drive
WL2Miss	mem-stage-idle	WL2Idle	L2-buffer-drive
	!mem-stage-idle	WL2MissStall	L2-buffer-drive
WL2MissStall	!mem-stage-idle	WL2MissStall	
	mem-stage-idle	WL2Idle	

The second stage (memory write) of the state machine has an idle state during which it checks if the first stage has an access ready for it. An access becomes ready when the first stage processes an L2 miss, or when it finds an uncached access in one of its queues. The memory write stage takes the access from the first stage, and then the first stage is free to return to its “idle” state. It should be noted that this half of the write state machine is *identical* to the one for the read state machine, since they're performing the same jobs, except that it sends "Write…" requests to the arbiter instead of "Read…" requests, since it is a part of the Write SM. The following, along with Table 3-15, lists the states:

WMemIdle: The memory stage is idle, waiting for the L2 write stage to pass down an access. While in this state, the state machine monitors the L2-write-back strobe, as well as the dirty-bit set by preceding stage. If an access comes down while either of those is high, the next state is "WMemDirtyArb." Otherwise it is "WMemCleanArb." The address of the access is driven out onto the state machine's memory address bus.

WMemCleanArb: The state machine must acquire access to the external memory address bus. It raises an arbitration request line, and waits for the arbiter to return a corresponding grant. The next state is "WMemClean."

WMemClean: The state machine memory address (from its memory address bus) is issued to the external interface through the memory address port. The next state is "WMemIdle."

WMemDirtyArb: This state is entered whenever the state machine is writing a memory line that will take place of a dirty line in the L2. The L2 must write back its dirty data to the external interface buffer, and the arbiter, which controls the accesses to that buffer, must be notified of the write-back. The state machine notifies the arbiter that a write back has taken place by raising this arbitration line. The machine waits in this state until the corresponding grant line is raised by the arbiter. The next state is "WMemDirtyInv1."

WMemDirtyInv1: The state machine memory address (from its memory address bus) and the writeback address (from the writeback address bus) are issued to the external interface through the memory address port. Meanwhile, an invalidation to the first half of the dirty line being removed is broadcast over the write bus to ensure that L1 inclusion in the L2 is maintained. The system also starts a data read of the dirty line (see WMemDirtyInv2) during this state. The next state is “WMemDirtyInv2.”

WMemDirtyInv2: This state completes the invalidation process begun on the previous cycle. Since every L2 line contains two L1 lines, two invalidations must be generated for every L2 write back. A single bit is flipped in the memory address, and the second invalidation is broadcast. The system also concludes its data read of the dirty line during this state. The next state is “WMemDirtyWB.”

WMemDirtyWB: During this stage, the dirty line is written to the external interface buffer using the read bus. The next state is “WMemIdle.”

Table 3-15: State Transitions — Memory Write Stage

Current State	Input	Next State	Output
WMemIdle	nothing-valid	WMemIdle	
	Access_ready & !dirty	WMemCleanArb	
	Access_ready & dirty	WMemDirtyArb	
WMemCleanArb	!Clean-grant	WMemCleanArb	Clean-arb-request
	Clean-grant	WMemClean	Clean-arb-request
WMemClean		WMemIdle	
WMemDirtyArb	!Dirty-grant	WMemDirtyArb	Dirty-arb-request
	Dirty-grant	WmemDirtyInv1	Dirty-arb-request
WmemDirtyInv1		WmemDirtyInv2	
WmemDirtyInv2		WMemDirtyWB	
WMemDirtyWB		WMemIdle	

3.4.4.3 The Memory State Machine

The memory state machine is responsible for handling the last stages of any memory access that goes to main memory. Since the delay through main memory will be tens to hundreds of cycles, it makes little sense to have the read or write state machines assigned to each CPU tied up waiting for the response from memory. Instead, they are allowed to continue operating on other accesses during the main memory access. When the access completes, the memory state machine, located in the memory controller, is responsible for completing the duties performed by the read and write state machines on other accesses — controlling the caches and internal chip buses.

Unlike the other state machines, the memory state machine does not have multiple, separate state machines to handle more than one reference at once. This is because each event handled can usually be handled by the state machine in only a few processor clock cycles, while events can only be supplied to the state machine at the much slower speed of the off-chip memory interface.

All states of the memory state machine are summarized in Table 3-16.

MemIdle: The memory state machine is idle, waiting for an access to return from memory. While in this state, the state machine monitors the external interface. Depending upon the type of a returning access (line load after read, line load after write, or just an uncached word being read) and the number of words from the request that have been returned, the state machine may be advanced to “MemReadFirstA” (critical word read on load-after-read, or uncached read), “MemReadL1A” (16 bytes read on load-after-read), “MemReadL2A” (32 bytes read on load-after-read), or “MemWriteA” (32 bytes read on load-after-write, or uncached write). Should one exist, any external intervention requests that might happen to be pending are allowed to use the state machine before normal memory reference returns. If one is found, the state machine transfers to the “EIArb” state. Any external interventions that are jammed by the address arbiter are given priority over ones coming in from off-chip, if multiple interven-

Table 3-16: State Transitions — Memory State Machine

Current State	Input	Next State	Output
MemIdle	nothing-valid	MemIdle	
	!EI & Access_ready & Read & 4_bytes	MemReadFirstA	
	!EI & Access_ready & Read & 16_bytes	MemReadL1A	
	!EI & Access_ready & Read & 32_bytes	MemReadL2A	
	!EI & Access_ready & Write & 32_bytes	MemWriteA	
	External_intervention (highest priority)	EIArb	
MemReadFirstA	!MemRB-grant	MemReadFirstA	Read Bus-request
	MemRB-grant	MemReadFirstB	Read Bus-request
MemReadFirstB		MemIdle	Mem buffer OE
MemReadL1A	!MemRB-grant	MemReadL1A	Read Bus-request
	MemRB-grant	MemReadL1B	Read Bus-request
MemReadL1B		MemIdle	Mem buffer OE
MemReadL2A	!MemRBL2-grant	MemReadL2A	MemWB-request
	MemRBL2-grant	MemReadL2B	MemWB-request
MemReadL2B		MemReadL2C	Mem buffer OE
MemReadL2C		MemIdle	
MemWriteA	!MemRWBL2-grant	MemWriteA	MemWBW-request
	MemRWBL2-grant	MemWriteB	MemWBW-request
MemWriteB		MemWriteC	Mem buffer OE, Write buffer OE
MemWriteC		MemIdle	
EIArb	!MemEI-grant	EIArb	MemEI-request
	MemEI-grant	EIL2ReadA	MemEI-request
EIL2ReadA		MemEIL2ReadB	L2-address-strobe
EIL2ReadB	!jammed & L2-hit	EIL2HitC	L2-buffer-latch
	!jammed & L2-miss	EIL2MissC	
	jammed	MemIdle	

Table 3-16: State Transitions — Memory State Machine

Current State	Input	Next State	Output
EIL2HitC	L2-hit (second)	EIL2HitD	L2-buffer-latch, L2-buffer-drive, L2-data-OE
	L2-miss (second)	EIL2MissD	L2-buffer-drive,L2-data-OE
EIL2MissC	L2-hit (second)	EIL2HitD	L2-buffer-latch
	L2-miss (second)	EIL2MissD	
EIL2HitD	L2-hit (second)	EIL2HitD	L2-buffer-drive,L2-data-OE
EIL2MissD	L2-miss (second)	EIL2MissD	

tions are pending. If none of these conditions are met, the state machine just stays in the "MemIdle" state.

The "Read First" group of states is responsible for returning the critical word from a newly-loaded cache line as soon as possible. These states are only used for cached accesses if critical-word-first memory operation is enabled. They are also used to return uncached access words, which are always the "critical" words of uncached accesses.

MemReadFirstA: The state machine must acquire access to the read bus. It raises an arbitration request line, and waits for the arbiter to return a corresponding grant. When the grant line is strobed, the state machine is allowed to progress into the "MemReadFirstB" state. Otherwise, the state machine busy-waits at this state.

MemReadFirstB: In this state, the state machine puts the critical first word of the returned cache line on the read bus for early pickup by the CPU, if possible. The word from memory is put on the read bus in the position that it would normally occupy if the entire cache line were passed over the bus, and latched directly through the receiving CPU into its pipeline. The state machine then returns to the "MemIdle" state.

The “Read L1” group of states is responsible for returning the critical 16 byte line of a cache reload access for return to the L1 caches of the reading processor. These states should only be used if the memory system is capable of returning the correct 16-byte half of a cache line first.

MemReadL1A: The state machine must acquire access to the read bus. It raises an arbitration request line, and waits for the arbiter to return a corresponding grant. When the grant line is strobed, the state machine is allowed to progress into the “MemReadL1B” state. Otherwise, the state machine busy-waits at this state.

MemReadL1B: In this state, the state machine puts the first half of the returned cache line on the read bus for pickup by the CPU and L2 buffers. The line from memory is put on the read bus and latched directly through the receiving CPU into its L1 cache and pipeline. It is simultaneously latched into the L2 input buffers. The state machine returns to the “MemIdle” state.

The “Read L2” group of states is responsible for returning the final, completed line to the L2 cache and writing it there permanently for future use.

MemReadL2A: The state machine must acquire access to the read bus and data arrays of the L2 cache. It raises an arbitration request line, and waits for the arbiter to return a corresponding grant. When the grant line is strobed, the state machine is allowed to progress into the “MemReadL2B” state. Otherwise, the state machine busy-waits at this state.

MemReadL2B: In this state, the state machine puts the critical half of the returned cache line on the read bus for pickup by the CPU, if necessary (when the “Read L1” group of states is not used). The entire 256-bit line from memory is put on the read bus and latched into the L2 input buffers, in preparation for driving it into the L2 cache on the next cycle. The state machine hops ahead to the “MemReadL2C” state.

MemReadL2C: In this state, the state machine initiates the write of the newly read cache line from the L2 cache control buffers into the L2 cache data arrays. Meanwhile, the external interface buffer reserved for this access is released. The state machine returns to the “MemIdle” state.

The “Write L2” group of states is identical to the “Read L2” group, except for the fact that it also enables the write bus to send the written word of data from the CPU so that it may be merged in with the newly read line immediately.

MemWriteL2A: The state machine must acquire access to the read bus, write bus, and data arrays of the L2 cache. It raises an arbitration request line, and waits for the arbiter to return a corresponding grant. When the grant line is strobed, the state machine is allowed to progress into the “MemReadL2B” state. Otherwise, the state machine busy-waits at this state.

MemWriteL2B: In this state, the state machine puts the entire returned cache line on the full 256-bit read bus for pickup by the L2 input buffers, in preparation for driving it into the L2 cache on the next cycle. Meanwhile, the data that was originally written by the CPU to this cache line is sent over the write bus to be merged with the newly read line in the L2 input buffers. The state machine hops ahead to the “MemReadL2C” state.

MemWriteL2C: In this state, the state machine initiates the write of the newly read cache line, plus the written data from a CPU, from the L2 cache control buffers into the L2 cache data arrays. Meanwhile, the external interface buffer reserved for this access is released. The state machine returns to the “MemIdle” state.

Finally, the “EI” group of states is devoted to processing external intervention requests from the off-chip interfaces. These requests are designed to knock cache lines out of all caches on the chip, so that they may be accessed by the off-chip interface. While Hydra currently only uses this function for I/O purposes, it could eventually be used as the basis for a multi-chip Hydra in the future, since it allows external invalidations of lines in the on-chip cache.

EIArb: The state machine must acquire access to the read bus, write bus, and data arrays of the L2 cache for two cycles each — 1 for each half of the line being knocked out of the chip. It raises an arbitration request line, and waits for the arbiter to return a corresponding grant. When the grant line is strobed, the state machine is allowed to progress into the "EIL2ReadA" state. Also, an L2 access is initiated. The address that the state machine has been driving on its L2 address bus (for the first half of the line being evicted) is muxed to the L2 address port. The L2-address-valid strobe fires. The same address is muxed to the input of the address jammer, and the jam-address-valid strobe fires. These two events initiate the L2 access and an address jam check. Otherwise, the state machine busy-waits at this state.

EIL2ReadA: This is the first cycle of the first L2 read access. In a pipelined manner, the second access is initated. The address that the state machine has been driving on its L2 address bus (for the second half of the line being evicted) is muxed to the L2 address port. The L2-address-valid strobe fires. The next state is "EIL2ReadB."

EIL2ReadB: If the address jammer jams the access, the machine returns to "MemIdle," temporarily forgetting the current access. The dropped access will be added to the jam queue after its jam clears, which will allow it to be retried. If the address is unjammed (the common case), the machine examines the results returned by cache. In this cycle, the tag compare of the first L2 access is complete, and the L2 asserts the L2-hit line if any of the lines in the cache hit. The data from all four sets of the L2 are available, and are latched in the internal data buffer at the end of the cycle. If the L2-hit line is high at the end of the cycle, the next state is "EIL2HitC." If the access is a cache miss, then the next state is "EIL2MissC."

EIL2HitC: In this cycle, the state machine drives the L2-buffer-enable line, which causes the L2 state latched in the previous cycle to update the L2 tag array, invalidating the first half of the line. It also drives the L2-data-output-enable line, which causes the current contents of the L2 line to be gated onto the 512-bit section of the read-bus. The data is then grabbed off the bus by the EIB assigned to this external intervention. Meanwhile, an invalidation to the first half of the dirty line being removed is broadcast over the write bus to ensure that L1 inclusion in the L2 is maintained. Finally, the tag compare of the second L2 access is complete, and the L2 asserts the L2-hit line if any of the lines in the cache hit. The data from all four sets of the L2 are available, and are latched in the internal data buffer at the end of the cycle. If the L2-hit line is high at the end of the cycle, the next state is "EIL2HitD." If the access is a cache miss, then the next state is "EIL2MissD."

EIL2MissC: The tag compare of the second L2 access is complete, and the L2 asserts the L2-hit line if any of the lines in the cache hit. The data from all four sets of the L2 are available, and are latched in the internal data buffer at the end of the cycle. If the L2-hit line is high at the end of the cycle, the next state is "EIL2HitD." If the access is a cache miss, then the next state is "EIL2MissD."

EIL2HitD: In this cycle, the state machine drives the L2-buffer-enable line, which causes the L2 state latched in the previous cycle to update the L2 tag array, invalidating the second half of the line. It also drives the L2-data-output-enable line, which causes the current contents of the L2 line to be gated onto the 512-bit section of the read-bus. The data is then grabbed off the bus by the EIB assigned to this external intervention. Meanwhile, an invalidation to the second half of the dirty line being removed is broadcast over the write bus to ensure that L1 inclusion in the L2 is maintained. When this cycle is complete, the system returns to the "MemIdle" state.

EIL2MissD: Nothing happens in this cycle, but we have reserved some resources that we will not be using and this cycle holds the system during their "use." It may be possible to just return to the "MemIdle" state, instead, with some small system optimizations. When this cycle is complete, the system returns to the "MemIdle" state.

Unlike all other memory accesses that use an EIB, no state machine step releases the EIB when it is no longer needed after an external intervention *or* an uncached write. Instead, the external I/O interface itself is responsible for releasing any EIB used during either of these processes.

3.4.5 Instrumentation for Research Purposes

Here are a few techniques that can be used to instrument the memory system in order to gain insight about the operation of the different parts of the system. In our simulation or FPGA-based implementations of Hydra, these may be added or deleted as needs demand.

3.4.5.1 State Counters on the State Machines

The "state counter" at the bottom of Figure 3-8 is an instrumentation mechanism. With this mechanism hanging off of the state machines, it is possible to see exactly where the state machines are spending their time — and by determining this it will be possible to see where the bottlenecks in the memory system really are for a tested benchmark. Not all states need to be instrumented — just ones where delays may occur and the first state of each path after a "decision point" in the state diagrams, to see the usage frequency of various "basic blocks" of states.

3.4.5.2 Total Latency Counters in the Address Check Buffers

In Figure 3-7, when a new address buffer is allocated the start time of the access is also recorded in the address buffer. When an event is done, its start time is subtracted from the current time and the difference is added to the total memory latency register (or one of several sub-registers). Meanwhile the total number of events is incremented by one. When desired, both may be read, and the total average memory latency may be obtained by division. This is necessary even with the state machine instrumentation described in Section 3.4.5.1, since delays caused by main memory accesses will not be noted by the state machine counters. This is the case because no state machine is cycling while the system waits for main memory access, since the access is just an entry in the memory system queues.

3.4.5.3 Occupancy Accumulators on the Arbiter

For any particular resource in the arbiter, there is a shift register that records the cycles during which the resource has been reserved. As bits are shifted out of these registers, the contents will be TRUE or FALSE depending upon whether or not the resource is to be used on that cycle. By using this signal to control a counter, it will be possible to determine the occupancy of a particular resource. In this manner, resources that are acting as bottlenecks to the system or underutilized resources will become apparent.

3.4.5.4 Arbiter Success and Attempt Counters

The success percentage of various arbitrations could be determined from the state machine counters described previously. However, if those cannot be built then it may be desirable to implement small counters attached to the arbiter that just count the number of attempts that are made on each arbitration request line, and also the number that are successful, so that arbitration success statistics may be calculated.

3.4.5.5 Bus Monitors

In order to get very raw data it may be necessary to instrument the buses themselves with sets of registers to record the contents of the buses at predetermined times, or spans of time. These “snapshots” or “movies” of the bus activity may then be pulled out of the system at a low speed by the user. In essence, the monitors would act like a small logic analyzer built into the hardware.

3.4.5.6 Event Queue Counters

In addition to the total latency counters (Section 3.4.5.2) as a means to determine the full access time of events, it may be possible to put counters on the event queues to the memory state machine that will indicate the total time each type of event sits in the queue. Much like the total latency counters, a running total and an event count may then be maintained in order to determine an average queue latency at any time.

3.5 The Level 2 Cache

This section describes the structure and interface of the Level-2 (L2) cache. This large, on-chip cache is responsible for acting as a large yet high-speed secondary cache for all four CPUs on the chip. In this respect, it is much like any of the other on-chip level-2 caches that have been used on commercial systems in the past few years (such as the Alpha 21164 [22] and several Intel, AMD, and IBM PC processors [34]). However, our L2 cache also performs a critical function in accepting all of the write-through traffic from the CPUs, preventing them from forcing performance-limiting off-chip accesses. In this function, it acts as the repository for the chip's permanent state.

3.5.1 Address and Tag Buffering System

Figure 3-10 gives an overview of the L2 cache subsystem. The top portion of the diagram focuses on the address/tag resources, while the bottom part of the diagram details the connections between the data memory and the data buses in the memory interconnection system.

3.5.1.1 Tag Processing Loop

On each L2 access, the set of four tags selected by the index bit field in the address sent from the CPU is read out of the tag array, using the dedicated read port in the tag memory, and latched into the tag buffer (a short pipeline of physical buffers). As it is latched into the tag buffer, the tags are compared with the actual address, to determine whether or not there is a cache hit. The way within the associative set that hits is also recorded. At this point, the controlling state machine can determine whether or not there has been a hit in the cache, and make its next state selection accordingly.

After the tag is processed in the tag buffer, it is updated and written back into the tag array using the write port into the tag RAM. Any hit line within the set will have its “age” bits set to indicate that it is the most recently used line within the set, as the other lines within the set are appropriately aged. The output line processor also performs other modifications to the line, as is appropriate for the access:

Read Hit: No special processing occurs.

Write Hit: The dirty bit for the hit line is set.

Read Miss: The read address is substituted for the LRU one, while the old LRU address (if valid) is thrown out to the external interface over the L2 LRU address bus. The line is validated and marked as clean.

Write Miss: The write address is substituted for the LRU one, while the old LRU address (if valid) is thrown out to the external interface over the L2 LRU address bus. The line is validated and marked as dirty.

After all appropriate processing, the updated set of tags is written back into the tag RAM and the system continues with the next access. It should be noted that due to the multicycle nature of this read-modify-write sequence, several tag updates will often be occurring at once in a pipelined fashion. However, these will never interfere with each other, since the address arbitration mechanism will always ensure that only accesses to different sets within the L2 cache are allowed to proceed simultaneously.

When the system is reset, all valid bits in the L2 cache tag RAM are cleared using a special reset counter that loops through all sets in the cache while writing lines with cleared valid bits. Normal cache functionality only starts after this clearing has occurred.

Table 3-17 summarizes the bits in the L2 tag field associated with each cache line and their functions. In each set there are 4 complete sets of these bits, for the assumed 4-way L2 cache.

Table 3-17: L2 Cache Tag Fields

Field	Size	Function
Tag Address	17* bits	Holds the actual tags for the 4 cache lines in this set. *assumes 128K, 4-way cache
Valid Bit	1 bit	Signals that this entry is valid
Dirty Bit	1 bit	Signals that a writeback is occurring during this main memory access
Age Bits	2* bits	Holds the "age" of this line (most recently to least recently used), relative to the others in the set. The most recently used line has an age of 00, while the least recently used is 11. *assumes a 4-way cache
Speculative Lock Bits	8* bits	Locks this line in the cache, regardless of LRU age status, if the speculative write buffers contain data to write to this line *the number of bits = the number of L2 write buffers

3.5.1.2 Data Processing Loop

For read accesses, the L2 data array is accessed in parallel with the read of the tag array. The address to the data array is also the index field from the address sent by the CPU. Following the read of all lines within a set, the correct line within the set is put on the read bus, as selected by the hit way (for hits), the LRU way (for misses that discard dirty lines, which will be written back to the external interface), or nothing at all (for misses that discard clean or invalid lines).

For write accesses, the L2 data array is accessed in parallel with the write of the tag array. The specific byte, halfword, or word written by a processor is selectively written into the cache array. The address to the data array is composed of the index from the state machine’s address to select the proper set, the hit way information from the tag buffer, the offset within the state machine’s address to locate the correct position of the write within the line, and the byte-lane selects from the write bus to activate writing on a byte-by-byte basis so the data can be merged into the cache “in place.” If a miss occurs and a dirty line must be discarded, the writeback address and the way of the LRU line being discarded is recorded in a side buffer, as noted in Table 3-18. There are 32 of these buffers, which are assigned to accesses along with EIB entries. During the memory stage of the write access, the LRU way is then used to read out the dirty line and place it on the write bus, exactly as it is done during a read access.

Table 3-18: L2 Side Buffer Format

Field	Size	Function
Writeback Index	10 bits	Holds index of the line being written back to main memory, if any
L2 Way of Miss	2 bits	This remembers the way of the line being written out to memory, and the location where the line's replacement will go

When a missed line is returned from the external interface, the index and the LRU way information are also returned from the buffers at the external interface. These are used to select the correct place in the cache to write the line coming in from main memory. In the case of write accesses, the write merging mux slips the data from the original write into the middle of the new line being written to the cache, using the offset bits from the cache and the byte selects on the write bus to properly place the new data. No further tag access is necessary during this stage, since the tags will have been properly updated during the original L2 cache access.

3.6 External Memory and Testing Interface

The final portion of the core Hydra chip is the unit that connects all of the on-chip resources to the outside world — the external memory and test interface. An overview of this portion of the chip is given in Figure 3-11. This portion of the chip is responsible for buffering data as it passes between the zones controlled by the high-speed on-chip bus protocols and the slower, narrower off-chip data communication protocols. To accomplish this job, the interface actually connects four different interfaces together through a mass of logic. The following paragraphs describe the four interfaces:

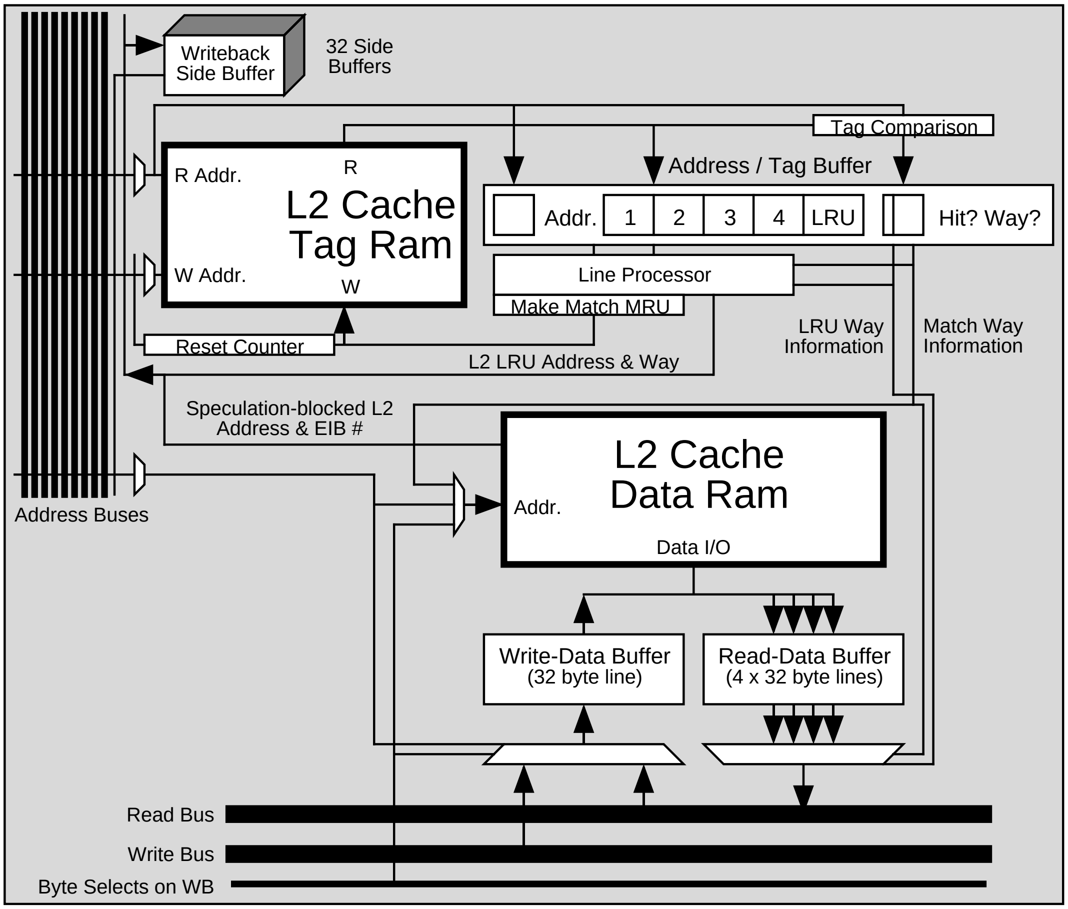


Figure 3-10: The structure of the Hydra L2 cache

Read and Address Bus Interface: The CPUs connect to the outside world using the external interface, through the on-chip address and read buses. The external interface buffers (EIBs, see Section 3.4.1.2) are responsible for buffering address and data as they pass between the off-chip and on-chip domains. Data return from main memory to the central core of the chip is controlled by the memory state machine, described in Section 3.4.4.3. The read bus is used both to write lines back from the L2 cache and read lines in from the external interface.

SDRAM Main Memory Interface: Main memory for the Hydra system is attached directly via an SDRAM interface located right on the chip. With the current Hydra configuration, a 32-bit, 133 MHz interface would be a good target size and speed (see Chapter 4 for more details). This interface is only responsible for the timing and signaling control required by the DRAM chip interfaces. As a result, it could easily be replaced with a different type of physical DRAM interface, such as Rambus,, in order to attach those types of chips instead. The I/O interface and CPUs are attached to this interface using the memory access arbiter/sequencer block. This piece of hardware is responsible for selecting the next access that will be allowed to use the main memory DRAM from the two interfaces, and then for sequencing the DRAM access. Meanwhile, any necessary completion signals are passed to the output queues of the appropriate interface following each access. On the CPU side, the EIB numbers of completed memory read accesses are passed back to the memory SM to notify it when the accesses are ready to be returned to the requesting CPU and caches. On the I/O side, returned data is queued to be sent out through the parallel interface. Finally, the block may also act as a "short circuit" between the CPU and I/O interfaces. If an uncached write is made to a small, preset address range, the write is passed to the I/O interface's output queue as if it was data returning from memory. This function allows the Hydra chip to "push" out I/O results when necessary.

Parallel I/O and Debug Interface: The secondary off-chip interface is logically a simple, bidirectional parallel port used to connect to a host workstation to the Hydra chip during testing. Like the main memory interface, it could be transformed into another physical interface with additional hardware prior to long-distance transmission without affecting the basic specification of the interface. The use of this interface is described below.

Debug Registers: This "register file" is actually an interface to all of the on-chip debug and statistics resources that happen to be included on any particular revision of the Hydra chip. It may be accessed only through reads and writes through the debugging interface. Each "register" in the "file" is actually an I/O port for a specific bit of on-chip hardware. As the portions of the chip visible using this interface may vary depending upon the chip revision, especially while the chip is emulated on FPGAs, this document will not discuss the various debugging resources accessible through this mechanism in detail. However, all implementations of

> Hydra must have “register” #0 be an interrupt trigger. Bits 0–3 send interrupts to their corresponding processor when they are written with a 1. This can be used to send interrupts from a host workstation to emulate an I/O device.

While the bridge between the CPU and the SDRAM is the most important connection, all of the lower-bandwidth connections to the I/O and debugging ports are critical to the overall functioning of the Hydra system. The I/O interface is complex enough to warrant its own subsection.

3.6.1 I/O Interface Command Description

The I/O interface is a simple, byte-wide parallel interface that teams up an 8-bit command word (sent first) with one or two optional 32-bit data words following it. If the physical medium is narrower than 8 bits, the returning command word may be shorter, as it comes in only four varieties. On input, the command word is primarily used to allow the user to send different kinds of accesses: debug reads, debug writes, DRAM reads, DRAM writes, and external intervention requests. On the way back, the command word allows read returns to be differentiated, because returns from different sources (DRAM vs. debug registers) may come back interleaved differently from the order in which the requests were sent, as debug registers may be read faster than DRAM. The return command also allows the processors on the Hydra chip to write data directly to the I/O port using uncached writes. The full interpretation of all command words and any data words that may follow them is described in Table 3-19. The current command word allows up to 64 debug registers to be addressed, although this could be increased by adding commands which address debug registers using an address encoded in a data word, instead of command word bits. Finally, should link synchronization ever be lost, it should be noted that the interface may be re-synchronized at any time by sending 9 or more “no command” commands in a row.

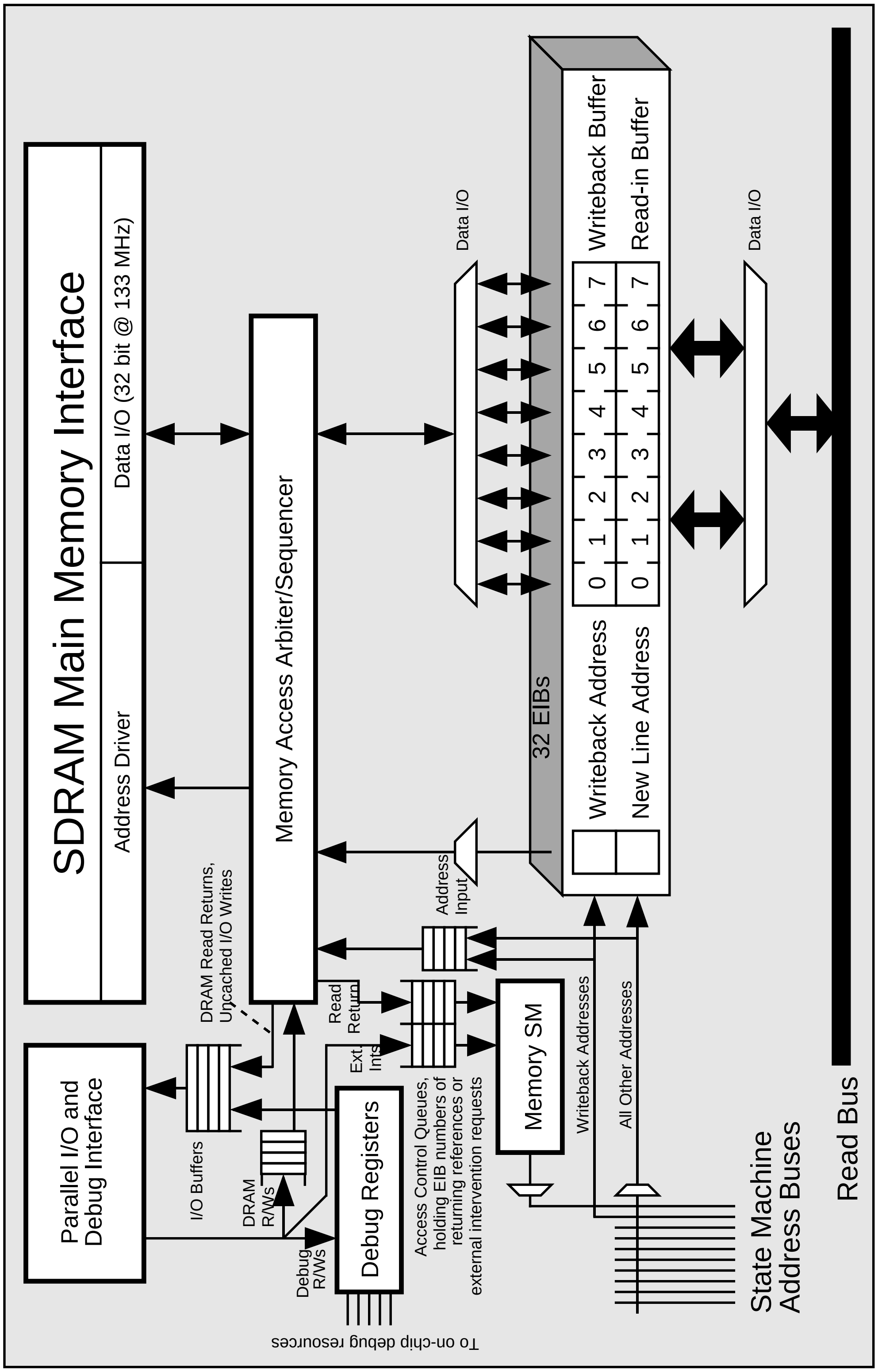

SDRAM Main Memory Interface
Address Driver
Data I/O (32 bit @ 133 MHz)
Parallel I/O and Debug Interface
DRAM Read Returns, Uncached I/O Writes
Memory Access Arbiter/Sequencer
I/O Buffers
DRAM R/Ws
Debug R/Ws
Debug Registers
To on-chip debug resources
Read Return
Ext. Ints
Access Control Queues, holding EIB numbers of returning references or external intervention requests
Memory SM
Address Input
Data I/O
32 EIBs
Writeback Address
New Line Address
0
1
2
3
4
5
6
7
Writeback Buffer
Read-in Buffer
Data I/O
Writeback Addresses
All Other Addresses
State Machine Address Buses
Read Bus

Figure 3-11: Overview of the external memory and testing interface.

Table 3-19: The list of commands that may be used with Hydra's parallel I/O port.

Command	Send / Return	Encoding	Data Word 0	Data Word 1
No Command	S / R	0x00	—	—
External Intervention Request	S	0x01	Invalidation Address	—
DRAM Read	S	0x02	Read Address	—
DRAM Write	S	0x03	Write Address	Data to Write
Send Reserved	S	0x04 to 0x7F	—	—
Debug Read	S	0x80 to 0xBF	—	—
Debug Write	S	0xC0 to 0xFF	Data to Write	—
Core-to-I/O Write	R	0x01	Address Written	Data to Write
Debug Return	R	0x02	Data to Return	—
DRAM Return	R	0x03	Data to Return	—
Return Reserved	R	0x04 to 0xFF	—	—

3.7 Design Analysis

Probably the most beautiful part of the Hydra design is how it eliminates the need for any form of fancy cache or memory protocols. With the judicious application of a pair of cross-chip buses, all coherence traffic is simply broadcast to all processors and handled using simple, immediate invalidations. The L2 cache maintains the current state of the machine at all times, so no transfers between L1 caches ever need to even be considered. Complex protocols such as MESI [60] [55] — or worse — are simply eliminated.

Hydra almost offers as good a memory coherence protocol. The basic design was targeted to provide a Total Store Ordering (TSO, [58] [59]) memory coherence protocol, that allows most interprocessor communication to work without any explicit synchronization. The atomicity of write propagations provided by the shared write bus makes this relatively simple coherence model a possibility in Hydra. Unfortunately, the "jamming" of references caused by the address arbiter breaks this model whenever multiple references to the same address collide in the memory system, as later writes from the processor that produced the "jammed" reference are allowed to pass the jam.

As a result, the end product is a Partial Store Ordering (PSO, [58] [59]) protocol that can be forced to act as a TSO one with a SYNC instruction. While this isn't a bad memory model, TSO is better — but probably not better enough to warrant stalling during every store jam.

Most of the specific parts of the design itself came out fairly well. With the benefit of hindsight, however, it is clear that there were an assortment of minor details that could have been done somewhat better.

First, the number of buffers scattered around the chip, as listed in Section 3.4.1.2, probably could have been reduced somewhat with more design effort. The large number of tables describing all of the different kinds of buffers demonstrates how these multiplied rapidly throughout the chip. However, most of these are necessary for one reason or another, given the current design. One particular set of buffers that needs to be cut down is the collection of EIBs. Currently, there are a fairly large number of these because each access must allocate an EIB when it starts and keep it until it completes. This can be a fairly long time for an access that must get a reference from main memory. This is a problem because EIBs are fairly large buffers, with data sections sized to hold entire cache lines, so they take up a non-trivial amount of die area. An optimization to reduce the dependence on large numbers of EIBs would be very helpful.

In Section 3.4.3.1, the state machines are probably a little too centralized. While this made the memory system design easier to conceptualize, it ended up resulting in too many signals that originated from a very small piece of control logic. While we were able to distribute many control lines by decoding state signals into control signals remotely, closer to where they were used, this really only partially solved the problem, since there were still far too many wires to control. For example, as described in Section 3.4.2.2, each state machine must quickly control address drivers onto an address bus dedicated to it, so that it can get its current address across the chip while it is still arbitrating for on-chip resources. This saves a cycle or so from each reference, but results in a mass of buses and lines that have to be controlled.

The worst part of the system is definitely the central arbiter, described in Section 3.4.3.3. Unfortunately, all systems like Hydra must have some sort of bus arbitration system. However, it is unlikely that they have such a complex one. While it was possible to build the arbiter in a fairly reasonable fashion using a few ROM tables and a large variety of other logic, the central arbiter is ultimately just way too complex because of the way it tries to interleave arbitrations that use different resources. Aside from the obvious circuit design issues that occur because all of that logic must actually work in a cycle, the arbiter itself often caused more protocol trouble than it was worth. Most of the trouble comes from the fact that the design actually goes back to the days when Hydra had an L3 cache. With just an L2 cache, it should be possible to make all accesses "look alike" enough so that the arbiter can just arbitrate for cycle-long timeslices of all chip resources, instead of trying to match collections of resource requests with resources. With the L3, however, the wide variety of utterly different access types made this totally impossible — so the resource-by-resource arbiter was born. Since then, the design has been plagued by problems like accidental priority inversion — when a high-priority request that needs a lot of resources keeps getting put off by lots of low-priority ones that only need a few — and other odd behavior that has required adding and removing more than one "special case" situation to the arbiter over the years. At the very least, we should have switched to a simple, one-pattern arbiter when we eliminated the L3 cache. Going back even further, we should have probably put the original L3 cache on its own bus with its own "secondary arbiter" and moved its resource needs away from conflict with the L2 accesses.

Other than these particular regions that, for one reason or another, got stuck with a less-than-optimal design, most of the design turned out quite well. The cache memory interfaces, address arbiter, read and write buses, and the off-chip interface control all turned out fairly well and could definitely be considered models for building parts of future designs.

4 The Performance of the Base Hydra

We have performed extensive simulation to evaluate the potential performance of the Hydra design. Once we had the basic design of Hydra established, we simulated it using LESS and SimOS [2] (in the early stages of development). This simulation regimen was used to pursue two different broad categories of evaluation. The first family of simulations attempted to execute a fairly large family of benchmarks on Hydra in order to determine approximately how well it would speed up code in general. The other group of simulations was a set of "stress tests" performed on key parts of the Hydra system with a pair of our most demanding benchmarks. This allowed us to evaluate some of the specific design tradeoffs made in the course of finalizing several parts of the Hydra memory system design. These two families of simulations are covered in the following two sections.

4.1 Overall Performance

Using a model with the memory hierarchy summarized in Table 4-2, we compared the performance of a single Hydra processor to the performance of all four processors working together. This process allowed us to evaluate to what degree the additional processors were actually able to contribute usefully to the execution of a single application. While any multiprocessor can run multiple separate applications, the ability to "team up" the processors in Hydra and run single applications across the entire chip in a relatively easy fashion, thanks to the low-latency interprocessor communication, is an important characteristic of our design. We used the ten benchmarks summarized in Table 4-1, primarily SPEC benchmarks, to generate the results presented in Figure 4-1. The binaries were generated using GCC 2.7.2 with optimization level `-O2` on an SGI workstation running IRIX 5.3.

The results indicate that on multiprogrammed workloads and highly parallel benchmarks, such as large, dense matrix-based floating point or multimedia applications, we can obtain nearly linear speedup by using multiple Hydra processors working together. These speedups typically are much greater than those that can be obtained simply by making a single large ILP processor occupying the same area as the four Hydra processors [6]. Hence, with workloads like this a CMP like Hydra would be an excellent replacement for a traditional ILP processor in the center of a system. In addition, these types of programs require little or no effort on the part of the programmer to obtain such excellent results. Multiprogrammed workloads such as databases are inherently parallel, while today's compilers can automatically parallelize most dense matrix Fortran applications [21]. Only the MPEG benchmark took a large amount of work to parallelize, including making some fundamental adjustments to the algorithm [44]. On the other hand, both of these categories of applications also tend to work well on traditional multiprocessors [55], so these achievements are not necessarily unique to a *chip* multiprocessor like Hydra. Instead, they are a function of multiprocessors in general.

However, there is still a large category of less parallel applications, primarily integer ones that are not easily parallelized (`eqntott`, `m88ksim`, and `apsi`, on this graph). The speedups we obtained with Hydra on these applications would be difficult or impossible to achieve on a conventional multiprocessor, due to the long interprocessor communication latencies required by a multichip design. However, even with Hydra's vastly improved communication latencies, the speed improvement obtained after weeks or months of hand parallelization effort is just barely comparable to that obtainable with a similarly sized ILP processor with absolutely no programmer effort [6]. Even more troubling, `compress` represents a large group of applications that cannot be parallelized at all using conventional techniques. Before a CMP can really become a viable replacement for conventional processors, these sorts of programs must be sped up **and** sped up in a way that does not require undue amounts of programmer intervention. This observation led us to the research that will be discussed in part II of this document.

Table 4-1: A summary of the application benchmarks used during the evaluation of Hydra .

	Application	Source	How Parallelized
General Integer	compress	SPEC95 [48]	Not possible using conventional means
	eqntott	SPEC92 [53]	On inner bit vector comparison loop
	m88ksim	SPEC95 [48]	Simulated CPU is pipelined across processors
	apsi	SPEC95 [48]	Automatically by SUIF [21]
Multimedia and Matrix FP	mpeg-2 (decoding)	MediaBench [46]	"Slices" in the input bitstream are distributed among processors
	applu	SPEC95 [48]	Automatically by SUIF [21]
	swim	SPEC95 [48]	Automatically by SUIF [21]
	tomcatv	SPEC95 [48]	Automatically by SUIF [21]
Multiprogram Workloads	OLTP	TPC-B [54]	Different transactions execute in parallel
	pmake	UNIX command	Compilations of different files execute in parallel

Table 4-2:A summary of the cache hierarchy simulated during the evaluation.

	L1 Cache	L2 Cache	Main Memory
Configuration	Separate I & D SRAM cache pairs for each CPU	Shared, on-chip SRAM cache	Off-chip DRAM
Capacity	16KB each	2 MB	128 MB
Bus Width	32-bit connection to CPU	256-bit read bus + 32-bit write buses	64-bit bus at half of the CPU speed
Access Time	1 CPU cycle	5 CPU cycles	at least 50 cycles
Associativity	4-way	4-way	N/A
Line Size	32 bytes	64 bytes	4 KB pages
Write Policy	Writethrough, no allocate on writes	Writeback, allocate on writes	"Writeback" (virtual memory)
Inclusion	N/A	Inclusion enforced by L2 on L1 caches	Includes all cached data

4.2 Analysis of Sections of the Design

This section focuses on several key parts of the Hydra design and evaluates them quantitatively. Some key numbers from the `swim` and `tomcatv` applications are used throughout the remainder of the chapter to illustrate the rationale for key design features of Hydra. The numbers of interest are plotted in Figure 4-2 and Figure 4-3. These two applications were chosen as targets for these

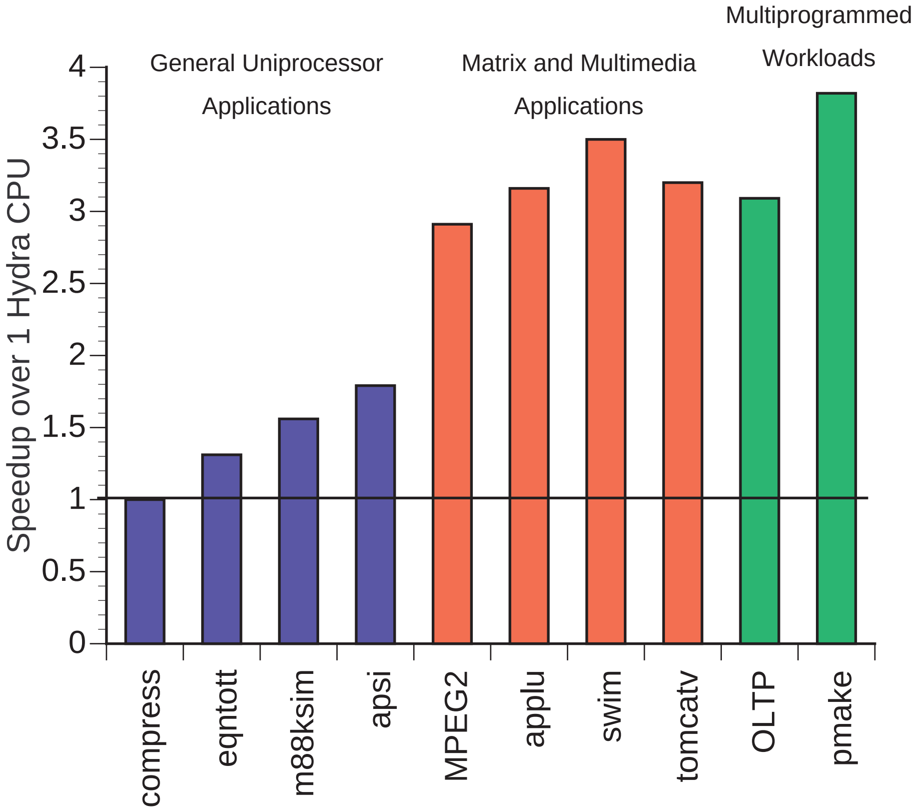


Figure 4-1: Performance of Hydra on a variety of applications from different domains.

focused tests because they stress the memory system with large numbers of accesses. In contrast, the Hydra memory system's cache hierarchy easily handles the small data sets of the other SPEC benchmarks.

4.2.1 The Caches

Hydra's L1 caches allow each processor to have its own small and fast L1 cache that processes a single access every cycle. Such caches can be easily optimized to return data within a single cycle. Our measurements have shown that in typical applications significantly more than 90% of loads hit in these L1 caches and do not need to progress further down the memory hierarchy.

We found that in practice the large, on-chip L2 cache serves several functions. First, it acts as a larger on-chip cache to back up the small L1 caches with a nearby memory an order of magnitude

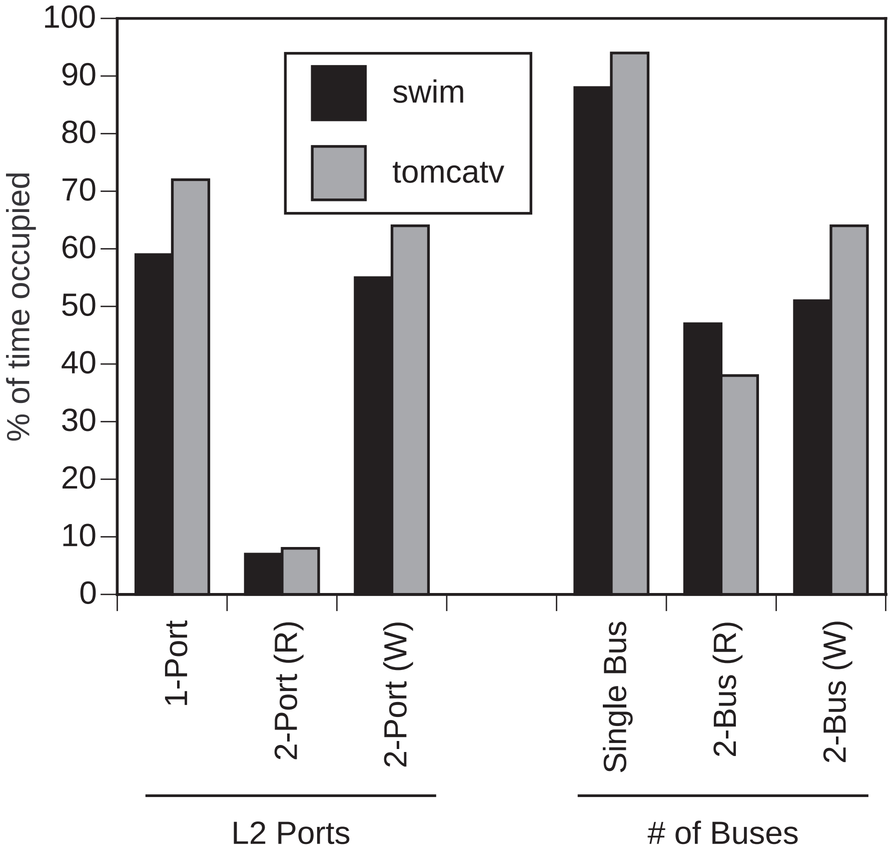


Figure 4-2: The occupancy of L2 cache ports and buses during runs.

larger, but five or more cycles slower. In practice, the L2's access latency is so short that it has little effect on the overall execution time. With most applications, the L1 caches have already exploited much of the locality present in memory accesses, so unless the application's entire data set fits into the L2 the local cache hit rate in the L2 tends to be poor — usually well under 50%, and sometimes under 20%. More importantly, the L2 cache serves as a sort of write buffer between the processors and the outside world. Since the L1 caches are write-through, the write bandwidth generated by the four processors would easily overwhelm off-chip buses. Using numbers from [15], a typical program can be expected to write about once every ten instructions, or about once every five cycles from a 2-way processor executing at its peak rate. With four processors executing at peak throughput, about 80% of cycles will produce a write from one of the processors, on average. Our simulations showed that after CPU stalls were considered, there were

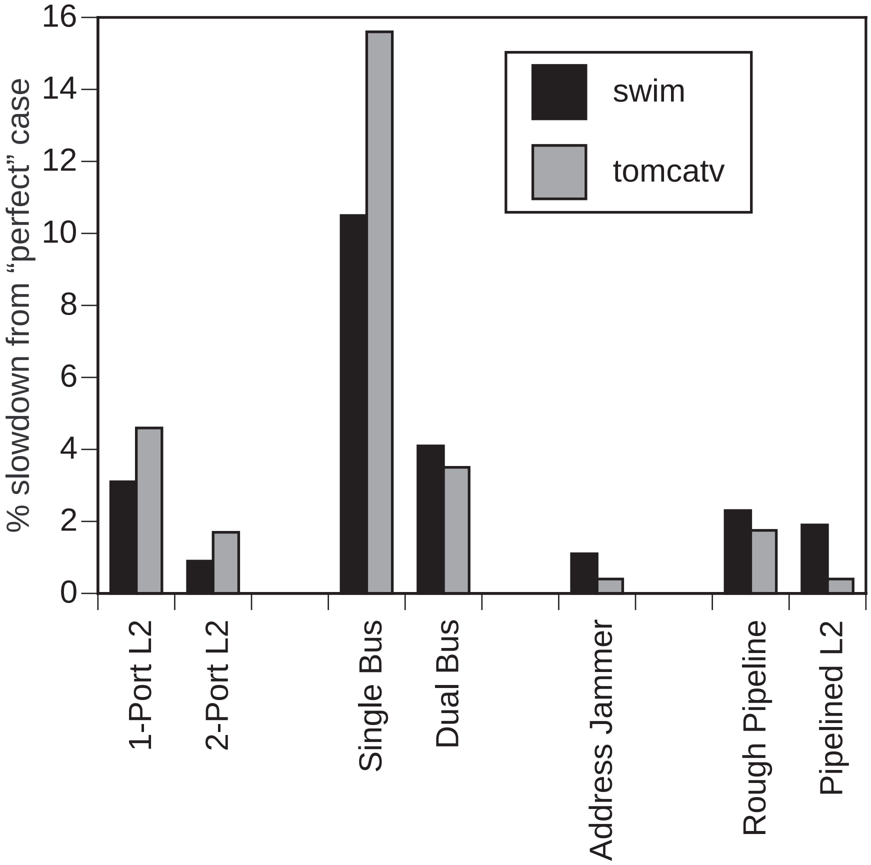


Figure 4-3: The % increase in execution time seen as a result of making several realistic architectural decisions, over "perfect" scenarios.

writes in 40-60% of cycles, typically, supporting these estimates. The L2 cache captures all writes and collects them into dirty cache lines before passing them down to the lower levels of memory via its writeback data handling protocol. By doing this, it reduces the off-chip bandwidth caused by writes to a manageable level. The L2 also acts as a communication medium through which the four processors can communicate using shared-memory mechanisms. Since it always contains the most up-to-date state for any line, it can supply shared data immediately to any processors that need it.

Figure 4-2 shows the occupancies of the cache ports when the L2 cache is single-ported (the normal case) or a dual-ported variant, with dedicated read and write ports. The occupancies from the 2-port case clearly show that most of the bandwidth in the L2 cache is used to absorb writes from the L1, while only a small percentage of the cycles are used to handle read accesses (L1 cache refills). Comparing caches of equal capacity, Figure 4-3 shows that the performance loss incurred by using a single-ported L2 cache is at most about 4% over an infinitely-ported cache — or 2% over a dual-ported cache — largely due to the fact that the small number of read accesses does not disturb the stream of writes into the L2 much. In reality, a single-ported design could have a larger capacity in a given area, since it could be made using simpler, more compact SRAM cells. Due to the negligible performance loss, we have chosen a single-ported implementation for Hydra.

4.2.2 The Buses

While the read bus performs many tasks, we have found that the bus is typically occupied less than 50% of the time, as shown in Figure 4-2, even with our most memory-intensive applications. With other applications it is even better. Contention for this particular resource is thus not a problem. As Figure 4-3 indicates, contention for *both* the read bus and the write bus slows performance by only a few percent over a perfect crossbar, even in the worst cases.

Since the L1 caches do not filter write bandwidth, write bus traffic can be quite high — sometimes exceeding 60% full, as is shown in Figure 4-2, even though each access only reserves the bus for a single cycle. It is even possible to saturate the bus at times of high write traffic. Reasonably deep write buffers are needed between the processors and the write bus to collect and feed stores onto the bus one by one while not stalling the processors, as bursts of store instructions are not uncommon. Merging this bus with the read bus would typically result in a completely saturated bus and significantly reduced performance, as Figure 4-2 and Figure 4-3 indicate.

4.2.3 The Control Mechanisms

In our simulations, most read accesses are handled by one of the L1 caches. Thus, a single read pipeline is sufficient to process the occasional cache misses. On the other hand, due to the writethrough L1 cache policy, every write made by each processor must pass through the write pipeline. On average, typically a store will happen only about every 5-10 cycles [15], even with the processor running at peak efficiency. However, bursts of stores do appear frequently in real code. Hydra only includes one write pipeline, but this is usually sufficient because the bursts of stores may simply be buffered without stalling the processor. We have found that the store buffer and write pipeline combination can be saturated by store-intensive code, but generally other system resources such as the write bus or the L2 cache port are also approaching the saturation point at these times, so the pipeline is not much of a bottleneck. Therefore, short of making a much more complex and expensive system, the single pair of control pipelines on each CPU appear to form a reasonable design.

In the process of tuning Hydra, the “state machine” controllers were adjusted extensively to improve performance. Initially, the controllers had no pipelining at all — instead, they were just state machines, that could only process a single access at a time. The performance of this system was surprisingly good. On applications other than `swim` and `tomcatv`, the limited number of accesses that could be processed at once was never a problem. These simple controllers became a tremendous performance drag with the more memory-intensive applications, however. We next evaluated the opposite alternative — fully pipelined controllers that could start processing a new access every cycle, no matter what. As shown in Figure 4-3, performance improved, but not as much as we had imagined. In a later experiment, we reduced the degree of pipelining by eliminating the cycle-by-cycle pipelining in either just the memory half or both halves of the controller. This allowed only a single access to be handled in each of these unpipelined stages at a time. We found that performance was normally almost unchanged from the fully pipelined case, eve n with the most memory-stressful applications. Handling more than two accesses per controller, therefore, turned out to be unnecessary — the coarser stages were sufficient. We have discovered one situation, however, where this is not the case. Part of the reason that deeper pipelining in the controllers is not effective is because four processors are normally trying to use the buses simultaneously. Of course, if the user decides to only use a single processor at a time, then the buses will become much less occupied. In this case, the L2 access pipeline of the WSM can become a limiting factor during code performing large bursts of writes, such as memory copies, unless it is pipelined to accept a new access every cycle, due to the large number of writes that can be produced by a single processor.

We were also concerned about possible performance implications of the address arbiter, which stops accesses from progressing to the L2 cache if they will access the same set as a reference that is always in flight. While this mechanism might appear to be rather heavy-handed, in reality it has a negligible impact on performance, as is noted in Figure 4-3. There are three basic reasons why it is not a problem. First, the mechanism is not invoked by reads that hit in the L1 cache, so the number of accesses “checking in” to the arbiter is limited. Because many more writes use the arbiter, it is necessary for them to release address arbitrations quickly on L2 hits, in case all of the processors are writing data to the same cache line. However, this is not a problem because they can be easily designed to release their address arbitrations after a single cycle. Second, even 10-bit indices offer 1,024 different possible values. Since there are typically nowhere near that many active references reserving addresses at once in Hydra, odds of a random collision are reasonably small. Finally, the mechanism allows accesses to the same address to work together more efficiently. When multiple processors attempt to read the same cache line from the L2 cache and miss, an event that occurs surprisingly often when processors are teamed up to work on the same code, the address arbiter will allow only the first pipeline controller to actually see and process the miss. The others are stalled by the address arbiter, and not using any resources in the meantime, until after the first controller reads the line into the L2 cache. Any trailing processors then just hit in the L2 cache, effectively riding on the coattails of the first access.

4.3 Conclusions

Chip multiprocesssors (CMPs) such as Hydra are an excellent way to build a microprocessor, even today — unless your primary goal is to allow speedup on hard-to-parallelize integer benchmarks.

On those programs, Hydra's performance will probably not be competitive, although in some cases it may be able to achieve parity with a focused effort at parallelizing code. With all other applications, chip multiprocessors offer at least competitive performance, and can often be far superior to other architectures by achieving nearly linear speedup with the four processor cores in the design.

Hydra is an example of a CMP which balances complexity and performance very well. Some results of our experiments show that while we have added resources to Hydra where they can do some real good — such as pairs of buses running across the chip. On the other hand, we have also avoided the temptation to add too many features that can make the system harder to design, yet provide little or no actual performance benefit. Ideas for a dual-ported L2 cache, fully pipelined controllers in all stages, and other essentially unnecessary frills were examined and discarded in the course of developing the final, fairly svelte Hydra design.

There are a couple of possible weak points in the Hydra design that can become problems in very unbalanced code that performs extremely large bursts of stores on a single processor. While the baseline system is good at dealing with average cases, its performance can drop significantly when faced with these situations. The first potential problem comes from the controller. The write controller for each processor (Section 3.4.4.2) is pipelined into two rough stages. The first stage handles L2 accesses, while the second hands off cache misses to main memory. While these stages are short enough (about 5 cycles) that a "normal" application rarely saturates them, one that performs many stores in a row can saturate the first pipeline stage quite easily. This situation results in a noticeable slowdown on such code. The other potential problem is the write bus itself. It was not a bottleneck for most of our benchmarks, but write buffers leading to it were often reasonably full as they "smoothed out" the store traffic flowing to the write bus. If the performance of the Hydra system is scaled up, then the write bus will probably become the system bottleneck at almost all times unless it is transformed into an alternate structure, such as multiple buses that can perform stores simultaneously. These potential weaknesses would probably have to be addressed before a commercial chip could be implemented so that programs that do occasional store-intensive tasks like memory copies would still work well.

5 The Design of Speculation Support for Hydra

A chip multiprocessor is clearly an excellent platform for executing code that has already been parallelized into a form consisting of several explicit threads of execution. Many types of computation tasks already lend themselves to this model, such as dense matrix scientific code and commercial server applications. However, the majority of computer application programs are not already conveniently divided into parallel threads. Unfortunately, without parallel programs, there is no really compelling reason to implement a chip multiprocessor. Hence, this is clearly a critical area that must be addressed if chip multiprocessors may be used to replace conventional uniprocessors in many sectors of the commercial marketplace.

As described in Chapter 1, speculative thread-level parallelism is a promising technique for taking existing uniprocessor programs and dynamically splitting them into threads that can run on a multiprocessor with very little programmer intervention. Therefore, this technique can be used to address the lack of parallel software in the marketplace. However, it is only practical if it does not add significant complexity to the hardware implementation of the Hydra design.

Most of this chapter details the additions to the baseline Hydra design and the low-level software protocol handlers that allow Hydra to handle the speculative thread based parallelism. This chapter is largely divided into four major sections. The chapter starts by describing what speculative thread-level parallelization is, and what kinds of hardware support are required to make it work. The second part details the hardware additions to the Hydra memory system that are necessary to properly track and buffer speculative memory references while speculative threads are executing on the processor cores. The third section describes the speculative coprocessor that has been added to Hydra's cores using the MIPS CP2 coprocessor interface port in order to interface between the speculative memory system hardware and the software described in the fourth portion of the chapter. This largely assembly language-level software is responsible for sequencing the speculative threads and controls how they interact with each other and other system resources.

5.1 What Additions Need to Be Made?

Hydra is a good architecture for the extraction of speculative TLP because it consists of a group of four very tightly coupled processors already, with low interprocessor communication latencies. This is the single most important architectural feature for the extraction of speculative TLP because speculative threads are all generally taken from the same uniprocessor program, and will therefore usually exhibit a large amount of poorly scheduled interprocessor communication. While this requirement is the most critical one, several others must also be met. Figure 5-1 shows a high-level view of the hardware that must be added to Hydra in order to successfully execute speculative threads. This additional hardware, discussed in more detail in the remainder of this chapter, must perform several jobs to correctly execute speculative threads.

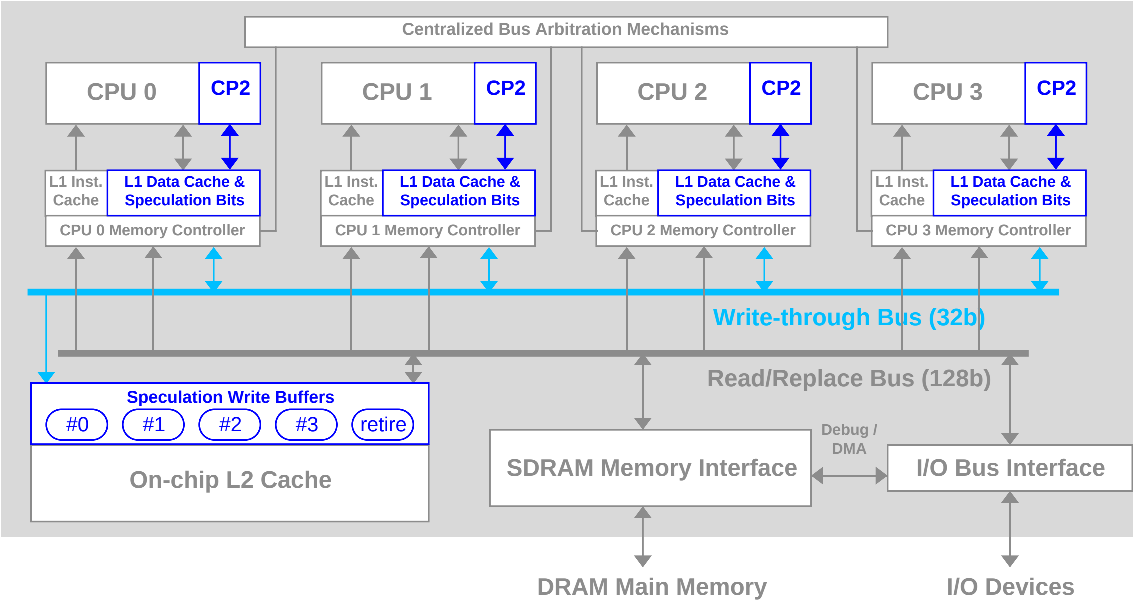


Figure 5-1: An overview diagram of Hydra, with additions required to support speculative threads highlighted.

5.1.1 Speculative Memory Support

In order to support speculation, special coherency hardware is required to monitor data shared by the threads. This hardware must fulfill five basic requirements. These requirements are illustrated in Figure 5-2, which show some typical data access patterns in two threads, i and i+1. Figure 5-2 (a) shows how data flows through these accesses when the threads are run sequentially on a normal uniprocessor. Figure 5-2 (b-e) show how the hardware must handle the special memory system requirements necessary to support speculative threads:

1) **Forward data between parallel threads:** While good thread selection can minimize the data shared among threads, there is typically a significant amount of sharing required simply because the threads are generally very small. Also, these threads were generated from a program in which minimizing data sharing among the "threads" was not an original design consideration — since the threads did not even exist when the program was first written. As a result, a speculative system must be able to forward shared data quickly and efficiently from an "earlier" thread running on one processor to a "later" thread running on another. This is depicted in Figure 5-2 (b).
2) **Detect when reads occur too early:** The speculative hardware must provide a mechanism for tracking reads and writes to the shared data memory so that if a data value is read by a "later" thread and subsequently written by an "earlier" thread, then the hardware will notice that the earlier read retrieved incorrect data, since a true dependence violation has occurred. Violation detection allows the system to determine when threads are not actually parallel, so that the violating thread can be re-executed with the correct data values. This is also depicted in Figure 5-2 (b).
3) **Safely discard bad state after violations:** As is depicted in Figure 5-2 (c), speculative memory must have a mechanism allowing it to be reset after a violation. All speculative changes to the machine state must be discarded after a violation, and no permanent machine state may be lost in the process.

4) **Retire speculative writes in the correct order:** Once speculative threads have completed successfully, their state must be added to the permanent state of the machine in the correct program order, considering the original sequencing of the threads. This may require the hardware to delay writes from "later" threads that actually occur before writes from "earlier" threads in the sequence, as Figure 5-2 (d) illustrates.
5) **Maintain multiple "views" of memory via renaming:** Figure 5-2 (e) depicts the case when an "earlier" thread reads an address after it has already been written by a "later" processor. The speculative hardware must ensure that the older thread cannot "see" any changes made by later threads, as these would not have occurred yet in the original sequential program.

Viewed from another perspective, requirements 1, 2, and 5 can be illustrated by examining how two memory references to the same address from two different processors running speculatively will interact. Table 5-1 (a) and (b) illustrate the effect of having these two memory references occur in the correct order (a) or "backwards" from the original program execution (b). Depending upon the combination of references, forwarding, a RAW violation, or memory renaming can all occur.

In some proposed speculative hardware, the logic enforcing these requirements monitors both the processor registers and the memory hierarchy [28][10]. However, for implementation cost and complexity reasons we chose to have the hardware in Hydra only enforce speculative coherence on the memory system while register-level coherence is handled by software.

5.1.2 Speculative Thread Control

In addition to speculative memory support, any system supporting speculative threads must have a way to break up an existing program into threads and a mechanism for controlling and sequencing those threads across multiple processors at runtime. This generally consists of a combination of

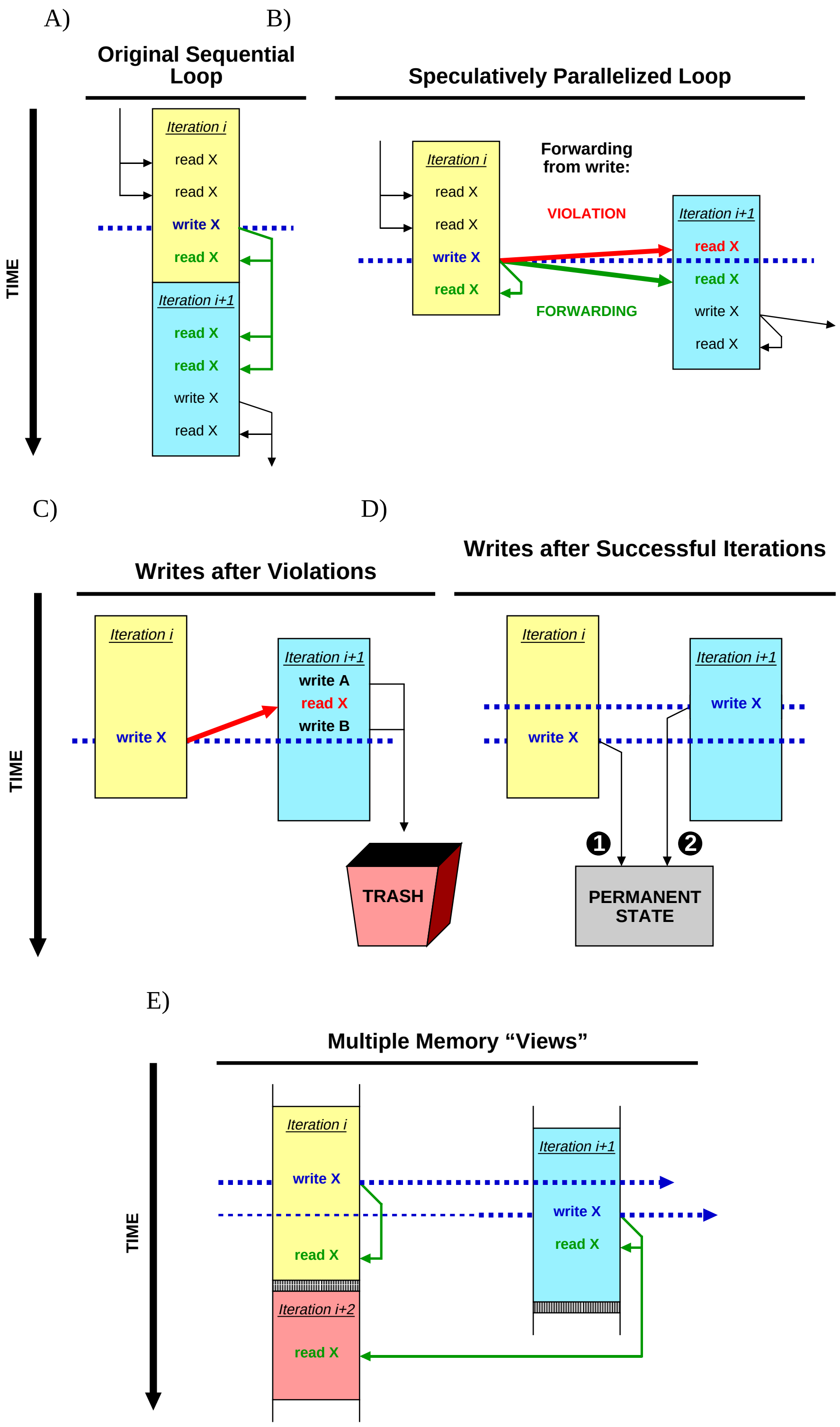


Figure 5-2: Diagrams illustrating the five critical requirements that must handled in a speculatively parallel memory system using a sample thread that can be broken into two speculative threads (a): b) Forwarding and violation detection on each memory reference, c) discarding of speculative state following a violation, d) reordering of writes following thread commits, and e) memory renaming among the threads.

Table 5-1: Summary of what happens when references to the same address from two speculative threads occur: (A) If the accesses occur in the original program order, or (B) if the accesses occur “backwards,” in the opposite order from the original sequential program.

A)			**B)**		
First: CPU *i*	**Then: CPU *i+1***	**Action**	**First: CPU *i+1***	**Then: CPU *i***	**Action**
Read	Read	—	Read	Read	—
Read	Write	Rename in *i+1*	Read	Write	Rename in *i+1*
Write	Read	Forward written data from *i* to *i+1*	Write	Read	RAW Hazard Violation: *i+1* must restart
Write	Write	Forward occurs, but then *i+1* overwrites its copy of the data	Write	Write	Rename in *i+1*, so eventual forwarding from *i* is just ignored

hardware and software that finds good places in a program to create new, speculative threads and then sends these threads off to be processed by the other processors in the CMP.

While in theory a program may be speculatively divided into threads in a completely arbitrary manner, in practice one is limited by the fact that initial program counter positions (and, for speculative-memory-only architectures like Hydra, register states) must be generated when threads are started. Without some sort of oracle to determine the future state of a program in several places, we must rely on heuristic methods to estimate future states of the program. Two of the most reasonable ways to divide a program into threads are at loops and subroutine calls. For loops, several iterations of the loop body can be started speculatively on multiple processors, as is depicted in Figure 5-3. As long as the number of loop-carried dependencies is reasonably small, the execution of the loop bodies on different processors can be overlapped to achieve speedup. With subroutines, it is possible to start a new thread to run the code following the subroutine call while the original thread actually executes the subroutine itself (or vice-versa, if desired), as is depicted in Figure 5-4. As long as the return value from the subroutine is predictable (most frequently, when there is no return value at all) and side effects of the subroutine are not used immediately, then the two threads will be able to run in parallel. In general, achieving speedup with this technique is more difficult because thread sequencing and load balancing among the processors is *significantly* more complicated with subroutines than loops. Some of this relative complexity can be visualized simply by comparing Figure 5-3 to Figure 5-4. While the loop iterations can be evenly distributed among the processors and clearly pipeline fairly well on individual processors, the varying sizes and unpredictable start times of subroutine iterations occupy the processors only in a very uneven manner.

Once threads have been created, the speculation runtime system takes up the job of selecting the four least speculative threads available and allocating them to the four processors in Hydra. The system that actually performs the thread allocation among processors and controls proper handling of all of the special cases in Hydra is a set of software routines that controls the Hydra speculative memory system extensions using a special speculation coprocessor. This system is fairly complicated, and is described in considerable detail in the remainder of this chapter.

The design of the speculation runtime system is significantly complicated by the frequent violations that may occur and also by special events like system calls. Many of these special cases are handled by taking advantage of a simple observation. It should be noted that *the* least speculative, or "head," thread is special. This thread is actually not speculative at all, since all older threads that could have caused it to violate have already completed. As a result, it will never violate — and will therefore always make forward progress — and it has the ability to handle events that cannot normally be handled speculatively such as operating system calls, instruction exceptions, etc. Since all threads eventually become the "head" thread, processing of these events can simply be delayed until a processor becomes the head.

5.2 Earlier Work on Speculation

The speculation work described in this thesis is a direct descendant of the pioneering work on speculative threads at Stanford by a collaboration between the SUIF and Hydra groups and

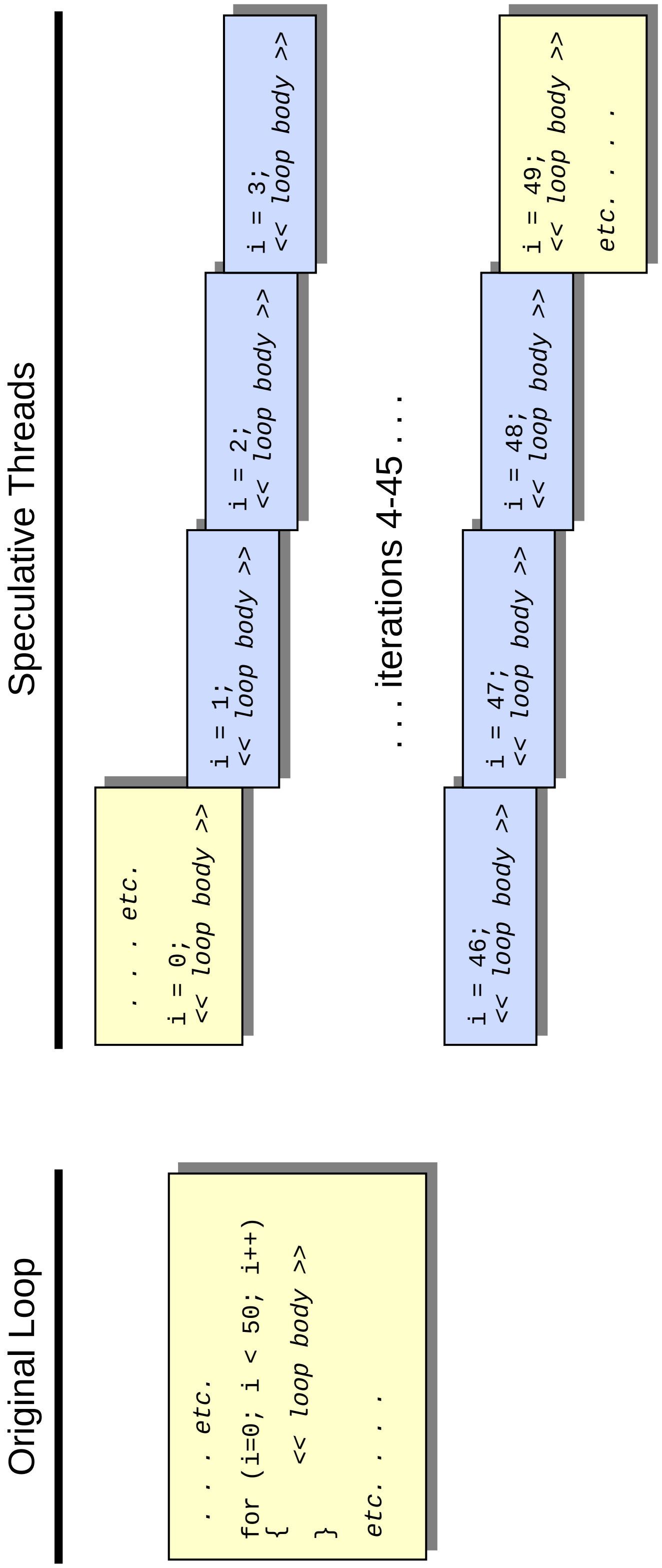
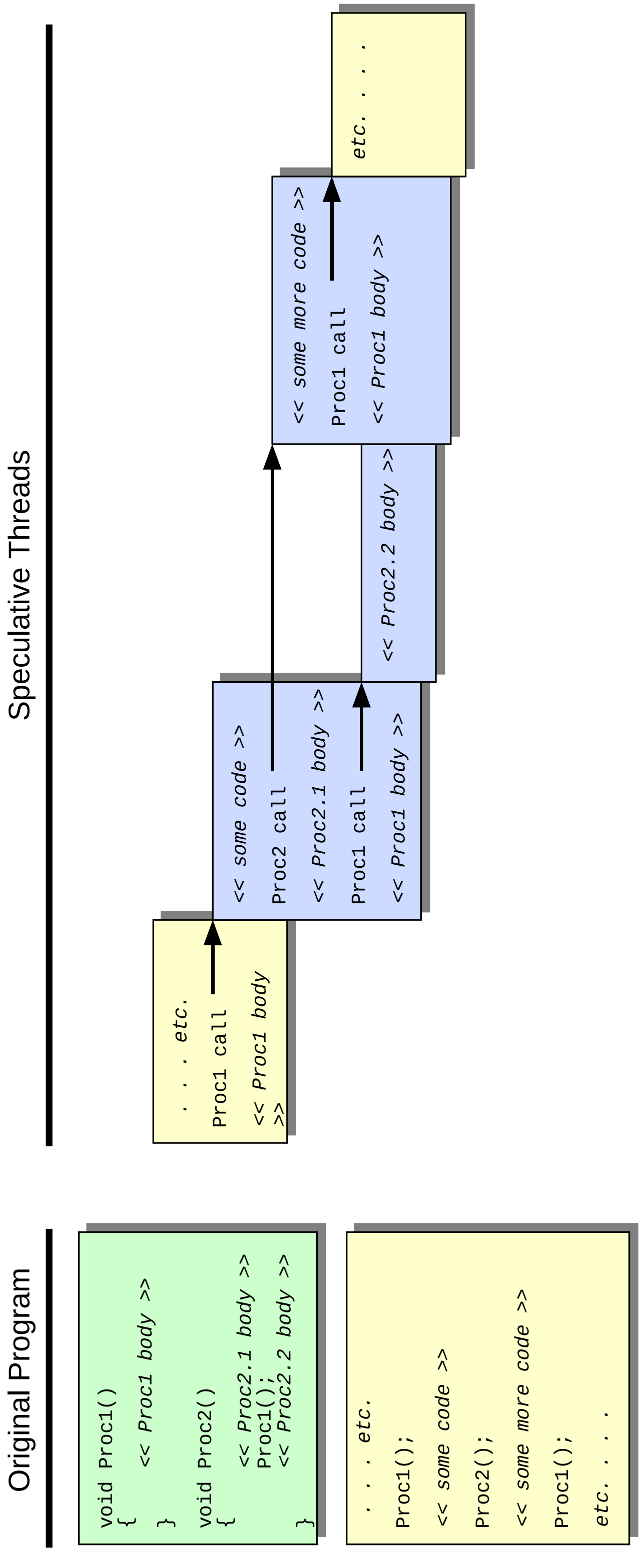


Figure 5-3: A graphic example of loop iteration thread-level speculation in action.



```
void Proc1()
{
    << Proc1 body >>
}
void Proc2()
{
        << Proc2.1 body >>
        Proc1();
        << Proc2.2 body >>
}
```

```
 . . . etc.
 Proc1();
 << some code >>
 Proc2();
 << some more code >>
 Proc1();
 etc. . . .
```

Figure 5-4: An example of post-subroutine-call thread-level speculation in action.

described in [43]. In that work, the SUIF compiler was used to slice target programs into speculative threads using both of the techniques described above. The threads were then fed into an "oracle" simulator to determine the maximum possible parallelism obtainable by overlapping speculative threads and a more realistic, execution-driven simulator that was able to see how much of this performance was actually obtainable. A more detailed version of the latter study appeared in [62]. The promising results seen in this study sparked the subsequent speculative thread revolution in the Hydra project, which had just been investigating traditional multiprocessing aspects of CMPs up to that time. Outside of Stanford and the Hydra project, three main groups have been working on speculative thread-level parallelism over the course of the past decade.

The first and most influential of these architectures was the Wisconsin Multiscalar project [28]. It proposed a CMP consisting of a dedicated ring of processors that distribute speculative threads about the size of basic blocks around the ring. Register-to-register communication and synchronization was achieved using the high-performance interprocessor ring connection. While this interconnection offered high interprocessor communication bandwidth and a runtime system almost entirely built in hardware, it was also expensive and essentially *dedicated* the system to finding speculative thread-level parallelism, at the expense of discarding more traditional but still useful mechanisms. To support this speculative processor core design, two different speculative memory systems were proposed. The first was the ARB [35]. This was simply an L1 data cache shared among all processing elements that had additional hardware to track speculative memory references within the cache. While a reasonable first concept, its centralized structure and the way it added complex logic to an already difficult-to-implement multiported L1 cache made it an implementation nightmare. More recently, the Multiscalar group introduced the speculative versioning cache [36], a set of separate, writeback L1 caches distributed among the Multiscalar processor cores, that maintain their speculative state within the caches using a complex and sophisticated cache coherence protocol. While it still has one centralized resource — the L1 cache tag control unit — its hardware is much more distributed and therefore a much more realistic design.

Concurrently with Hydra, the TLDS project at the University of Toronto and Carnegie-Mellon proposed a considerably different scheme [14]. Instead of a complicated hardware core designed to handle very small speculative threads, they chose to design a more Hydra-like CMP that had virtually all of its speculation control runtime system in software. As a result, they were able to work with much larger speculative threads, but were much less tolerant of threads experiencing many violations. To combat this, they proposed using sophisticated compiler technology to optimize the code being executed on the TLDS machine in order to make it more parallel. In the memory system, they also chose to keep their protocol much simpler, at the expense of performance-limiting bursts of bus traffic at the end of every speculative "epoch" and an inability to forward data from one speculative iteration to another. Instead, they added a special shadow memory for critical values that required early forwarding between epochs. Once again, this relied upon sophisticated compiler technology to pinpoint key memory locations that needed shadowing. More recently, they have optimized their hardware design somewhat [37] and extended the design to work in a much larger-scale system consisting of up to 32 processors [38]. While effective, systems such as this are useful only with a small collection of *very* parallel applications that cannot already be parallelized using traditional parallelizing compilers.

The third major thread-level speculation project has occurred at the University of Illinois. They originally started out with a Multiscalar-like design that used a special bus to communicate register values among processors [10]. A unique innovation used by their processor design was to exploit existing uniprocessor code binaries, without any modification whatsoever, in a thread-speculative manner by defining loop-iteration threads when backward branches were taken in the code. Such tight and unplanned thread-level speculation required a very closely-coupled architecture that maintained register coherency as well as memory coherency among their speculating processors, much like the Multiscalar design. Since then, their design has moved to a much looser structure, more closely resembling the TLDS design [39]. This led to competitive large-scale system design proposal at virtually the same time as the TLDS group [40].

Unlike CMP technology, which is starting to achieve widespread acceptance in industry as this thesis is being written, thread-level speculation is still largely confined to the domain of academia. The only prominent commercial design using even a limited form of thread-level speculation is the Sun MAJC architecture [25]. This architecture uses speculation in a way first proposed by our group [41] to accelerate the processing of Java method calls, a variation of subroutine speculation that works much better than subroutine speculation with most other computer languages, due to the constraints imposed on programmers by Java itself. However, this architecture uses a simplified version of our original protocol that takes advantage of Java's memory usage characteristics to implement most parts of the design in software — only a fast interrupt control mechanism is implemented in hardware. Other than this very limited beachhead, the use of thread-level speculation in industry will probably only occur when microprocessor manufacturers start trying to sell CMPs to markets other than the server and embedded multimedia markets, where programs are already so multithreaded that thread-level speculation is largely unneccessary.

5.3 Hardware to Buffer Memory References

We took advantage of Hydra's writethrough L1 caches and bus system to build a protocol that should be easily implementable without impacting the CMP's basic cycle time, as efficient as the SVC [36], and requiring only relatively simple coherence protocols. To add this speculation support, several key hardware elements have been added to the existing Hydra design. Figure 5-1 showed the global placement of most of these additions. Figure 5-5 expands upon some of the details that are too small to see in the previous figure, such as the location of the interrupt screening mechanism and the L2 cache buffer controller. The remainder of this section summarizes these additions.

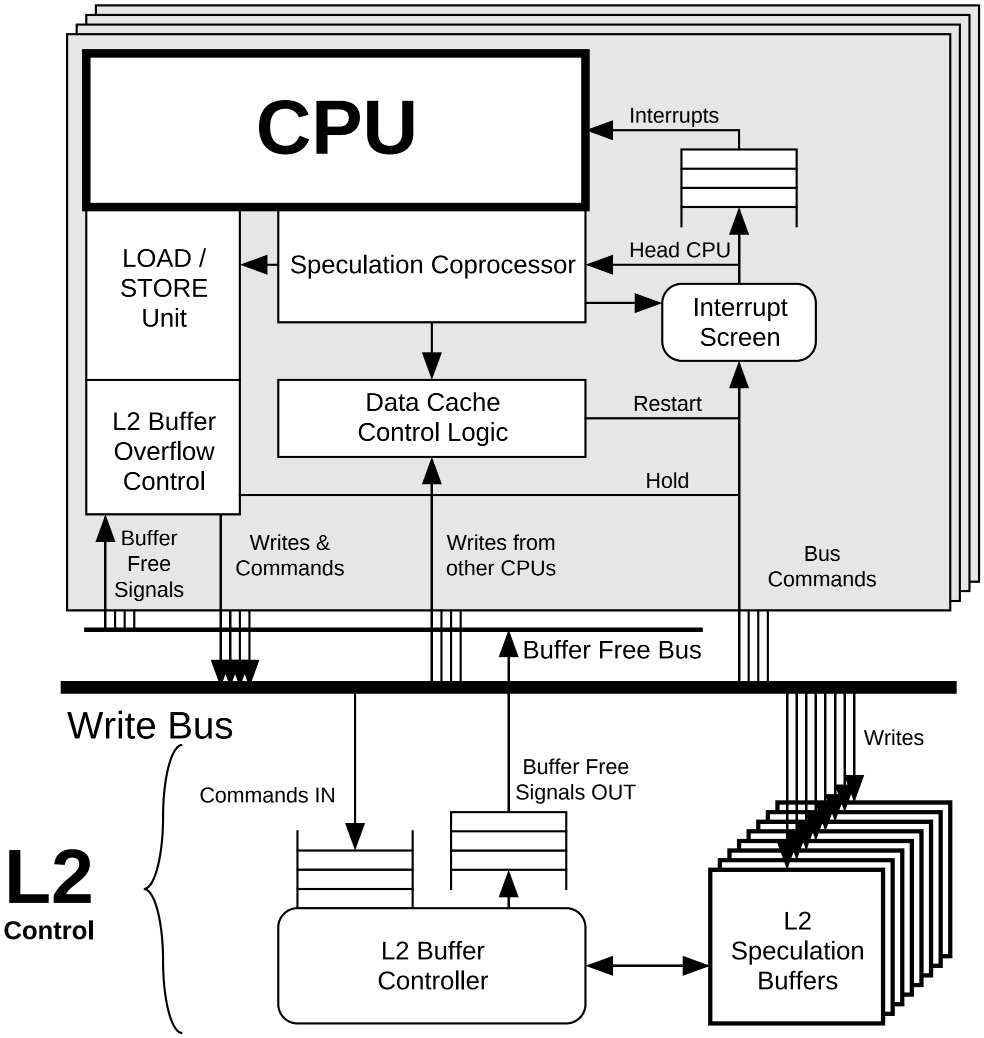


Figure 5-5: An overview of the speculation hardware added to Hydra

5.3.1 L1 cache modifications

Ten bits must be added to the existing L1 cache state in order to properly maintain the state of the cache during speculation and to detect RAW violations that force the processor to restart after a load speculatively reads erroneous data. These bits and their functionality are summarized in the following text and Figure 5-6. Unlike other L1 cache bits, these bits must be designed to be gang-clearable when threads complete or are restarted. This functionality can be obtained through specialized tag SRAM cells or by manipulating the power lines to the cell array.

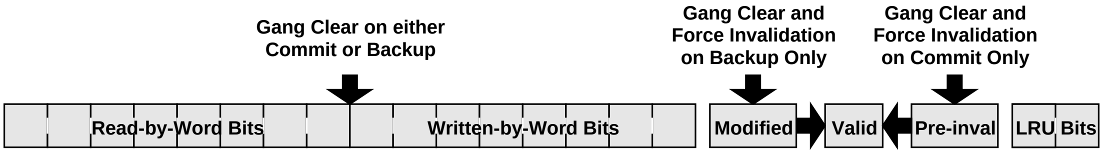

Gang Clear on either Commit or Backup
Gang Clear and Force Invalidation on Backup Only
Gang Clear and Force Invalidation on Commit Only
Read-by-Word Bits
Written-by-Word Bits
Modified
Valid
Pre-inval
LRU Bits

Figure 5-6: Speculation control bits supplementing the L1 cache tags.

The first two additional bits are responsible for modifying the basic cache coherency scheme that invalidates an L1 cache line only when a write to that line from another processor is seen on the write bus.

- **Modified bit:** This bit acts like a "dirty" bit in a writeback cache. If any changes are written to the line during speculation, this bit is set. These changes may come from stores by this processor or because a line is read in that includes speculative data from active L2 buffers. If a thread needs to be backed up and restarted on this processor, then all lines with the modified bit set are gang-invalidated at once.

- **Pre-invalidate bit:** This bit is set whenever another processor writes to the line, but it is running a more speculative thread than this processor. In this case, it marks a "future invalidation" that we will need to take eventually, but not right away. It is also set if a line is read from the L2 cache when more speculative processors' L2 buffers already include writes to the line. In this case "future" changes to the line have already been made by more speculative processors, yet these changes will not be included in the line sent back from the L2 cache. The pre-invalidation bit must be set so that the line will be properly updated after this CPU completes its current thread. Since writes are only propagated back to more speculative processors, we are able to safely avoid invalidating the line in either case until a different, more speculative thread is assigned to this processor — which will inevitably happen after the current thread on the processor is completed and the processor needs more work to do. Thus, this bit acts as the opposite of the modified bit — it invalidates its cache line when the processor completes a thread. Another way of looking at this bit is as the “renamed” bit. When set, it indicates that a later processor has generated a renamed version of this line which can be ignored in the short term, but must be updated after this thread completes. Again, all lines must be designed for gang-invalidation.

The other two sets of bits allow the L1 cache to detect true dependence violations using the write bus mechanism. They must be designed to allow gang-clearing of the bits when a speculative region is either restarted *or* completed.

- **Read bit(s):** These 4 bits, one for each 32-bit word in the line, are set whenever the CPU reads from *any part* of the word they represent within the cache line, unless that region’s written bit is set. If a write from a less speculative thread, seen on the write bus, hits an address in an L1 cache with a set read bit, then a true dependence violation has occurred between the two processors. A violation exception is then initiated to handle the problem. Subsequent stores will not activate the written bit for this line, since the potential for a violation has been established.

- **Written bits:** These 4 bits, one for each 32-bit word in the line, prevent unnecessary violations by renaming the memory locations used by multiple threads in different ways. If a processor writes to an *entire* word, then the written bit for that word is set, indicating that this thread now has a locally generated version of the address. Subsequent loads will not set the read bit for this section of the cache line, and therefore cannot cause violations. Unlike the other L1 cache bits, these bits are technically optional, because all they do is prevent unnecessary violations, but we found that they are almost essential for performance reasons.

It should be noted that all read bits set during the life of a thread must be maintained until that thread becomes the head, when it can no longer cause violations. Even if a cache line must be removed from the cache due to a cache conflict, the line may still cause a speculation violation. Thus, if the L1 cache attempts to throw out a cache line with read bits set it must instead halt the processor by invoking a processor hold runtime exception until the thread becomes the head, is restarted, or is completely discarded. This problem can largely be eliminated by adding a small victim buffer [42] to the L1 cache. This victim buffer only needs to record the address of the line and the read bits in order to prevent processor halts until the victim cache is full. For simplicity, our simulations have always assumed that an infinite-size victim buffer, containing only read bits and addresses, is attached to each L1 cache.

It should be noted that the read and written bits only need to be included in the L1 cache tag array that is actively processing invalidations from the write bus. They are unnecessary and redundant in the array used only for local hit detection.

5.3.2 Local L2 Buffer Tags

Each processor also keeps a duplicate copy of the CAM-array tags associated with its currently active L2 memory buffer, as described in Section 5.3.3. Only the tags are necessary, and not the auxiliary fields associated with the actual L2 memory buffer, because the local copy maintained by each processor is simply to keep track of which lines are being kept in the speculative buffer.

The primary purpose of these local buffers is to notify the processor core when it will run out of space in the L2 memory buffers during the actual store that would overflow the buffers. When an overflow is detected, the coprocessor immediately generates a processor hold exception for the store. This exception works much like any other exception that can be caused by an illegal memory reference, such as a page fault, except that it runs the speculative hold handler, which waits until the processor is the nonspeculative “head” processor before letting the offending store continue.

A secondary purpose of the local buffers is to allow stores to be selectively converted into non-speculative writes after the processor becomes the head processor, as described in Section 5.4.4.

5.3.3 Speculation Memory Buffers

Buffering of data stored by a speculative region to memory is handled by a set of buffers attached to the L2 cache data array that collect speculative writes and hold them until they may be safely committed to the permanent state of the machine in the L2 cache. In Figure 3-10, most of the structures described within this section are actually inside of the L2 Cache Data RAM block, along with the SRAM banks themselves. During non-speculative execution, writes on the write bus always write their data directly into the L2 cache. During speculation, however, each processor has an L2 buffer assigned to it by the L2 buffer controller, using a simple command sent over the write bus. This buffer collects all writes made by that processor during a particular speculative thread. If the thread is restarted or flushed from the system entirely, then the contents of the buffer are discarded. If the thread completes successfully, then the contents are permanently written into the L2 cache. Since threads may only complete in order, the buffers therefore act as a sort of reorder buffer for memory references. This section describes the necessary additions.

5.3.3.1 LRU Enhancements

One enhancement must be made to the existing tag control bits in the L2 cache itself in order to avoid a potential deadlock situation. This is the addition of a set of “line locking” bits to each cache tag that is only used during speculation. Much like the additional L1 cache control bits, the added bits must be gang-clearable in hardware when a thread completes execution.

Since the L2 cache cannot initiate accesses to read main memory lines itself, all L2 lines that are to receive data waiting in the speculation buffers must be “locked down” so they will not be thrown out on a cache line replacement. If this requirement is not met, the system may become deadlocked when the data actually tries to commit to a line that is no longer in the cache. To control this locking, 8 bits (one per memory buffer in 7.1.3.2) must be added to the LRU information for each line in the cache. When a speculative buffer allocates space for a new dirty line within itself, it locks down the permanent state of the line by setting the lock bit associated with itself and the appropriate cache line.

In the event that a speculative memory buffer is cleared, or if it finishes committing its contents back into the L2 data RAM array permanently, it gang-clears all of its locking bits at once, thereby preparing the locking bits for the next iteration in a single machine cycle.

A write or read that misses in the L2 cache and attempts a replacement when all 4 lines within the set are locked cannot be allowed to proceed, for it will throw out a line that is necessary for successful retiring of a speculation memory buffer. In this case, the EIB number and the address of the access are stored away in the speculative memory buffer entry associated with the least speculative reference within the set (i.e. the one that will commit first). Meanwhile, the memory block bit within the EIB for the current access is set, and the L2 LRU address & way are put in the L2 side buffer for this access. This will effectively suspend the overflowing access while keeping the critical information necessary to revive it present in the L2 cache system. Later, when the buffer entry is cleared following a commit or clearing of the buffer, the system must wait until it can use a free cycle on the L2 tag array to re-execute the tag processing for the access. Simultaneously, the access is sent to the main memory by clearing the block bit within its EIB with a remote signal, and the access is freed to continue.

5.3.3.2 Speculative Data Buffers

Attached to the L2 data arrays are the speculation memory buffers, the largest single hardware addition required to support speculation. These buffers hold the data from speculative writes made by the processors until the threads that made the writes complete and are able to safely commit their data. There are 8 speculative memory buffers in the system, with a pair of buffers permanently assigned to each processor. In a more optimal system, this number of buffers could be reduced to 4 or 5 buffers, with no extra buffers or only a single extra “floating” buffer, but for simplicity we chose to implement the fully double-buffered version.

The two halves of a buffer pair act to double-buffer each other. When a processor commits or discards a buffer, execution switches to the other buffer, allowing the first to commit or discard its contents without stalling the speculative processor. If fewer buffers had been implemented, the processor would have to wait or switch and pick up the extra “floating” buffer before continuing, but with full double-buffering it can simply switch to the other half of the pair of buffers. Going back from each buffer to the write state machine associated with the buffer’s processor is a signal line indicating that a buffer is committing or draining. If the processor requires a new buffer while its backup buffer is still draining, then speculative writes from the processor will temporarily be halted. In this case, if the head of the queues into the write state machine for this CPU is a *speculative* write, then the state machine will stop operating temporarily until the needed buffer drains out and is ready for action.

Each individual buffer consists of an array of entries that hold a 32-byte cache line of data, control fields associated with it (as summarized in Table 5-2, and including a byte-by-byte valid mask for the entire line), and a tail pointer to indicate how full the buffer is (there is effectively also a “head” pointer, but it is always 0). A block diagram of one of these buffers is given in Figure 5-7. The primary function of these buffers is to intercept writes to the L2 cache and reads from the L2 cache during speculation in such a way that speculatively written state may be stored, yet may also be safely discarded at any time. It should be noted, however, that nonspeculative reads and writes may bypass these buffers at any time, even during speculation.

Table 5-2: L2 Speculation Buffer entry structure

Field	Size	Function
Tag-Index Address	27 bits	Holds the tag and index for the cache line held in this buffer entry. This is a CAM entry, so that it may be addressed directly by later accesses.
Data Field	32 bytes	Holds speculative write data
Valid Bits	32 bits	Indicate which bytes in the data field contain valid written data, so that only these bytes will be merged into the L2 cache
Lock used?	1 bit	Indicates that there is a locked access that should be freed when this entry is cleared or committed
Lock EIB Entry	5 bits	Holds the external interface buffer number associated with an L2 cache access that is stalled since its set is locked up by previous speculative writes
Lock Tag-Index Address	17* bits	Holds the tag associated with the locked access, if any *assumes 128K, 4-way cache

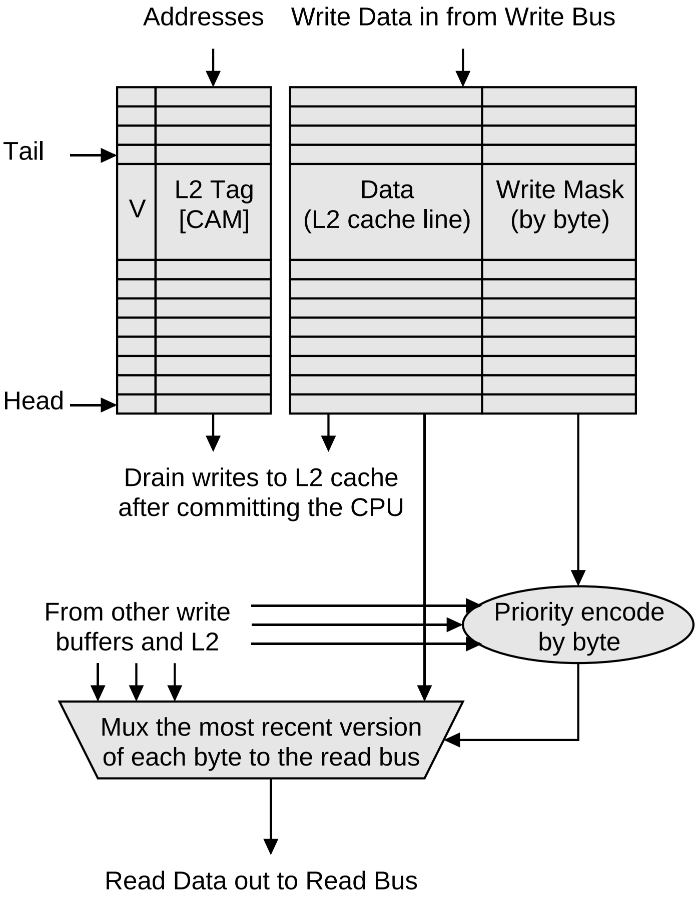

Addresses
Write Data in from Write Bus
Tail
Head
V
L2 Tag [CAM]
Data (L2 cache line)
Write Mask (by byte)
Drain writes to L2 cache after committing the CPU
From other write buffers and L2
Priority encode by byte
Mux the most recent version of each byte to the read bus
Read Data out to Read Bus

Figure 5-7: Block diagram of a single write buffer (1 of 8)

Write operations occur for each speculative store coming from the processors. For each write, the address array in the write buffer is checked for a hit. On a hit, the active data bytes for the write are written directly into the appropriate portion of the line already allocated in the buffer, overwriting any data previously written there. Meanwhile, the bits corresponding to the location of the new write are set in the write mask for the buffered line. On a miss, these same updates occur at the buffer entry selected by the tail pointer. Along with the usual updates, the tail buffer's address entry is set to the address of the write for future reference. In parallel, the tail pointer itself is incremented to point to the next entry in the buffer in preparation for the next miss. The buffer can never overflow thanks to the duplicate CAM arrays maintained in each processor's speculative coprocessor that cause the processor to stall first, as described in Section 5.3.2.

The buffers also help read operations for any cached load coming from the processors. Each read (speculative or not) causes all speculative buffers to attempt to read the line from within themselves in parallel with the L2 data array read. When these reads are complete, the most recently written version of each *byte* in the line must be selected and returned to the processor. 16 5-input priority encoders working in parallel, one for each byte in the 16-byte L1 cache lines, perform the selection using the byte masks stored with each L2 buffer entry. The lowest priority input to each encoder is set to 1 automatically, since it represents the default — using the data returned from the L2 cache as a data source. The next lowest priority belongs to the buffer for the head, nonspeculative processor. The middle priority belongs to the first speculative processor, and the pattern continues up to and including the processor actually making the read. At the same time, the byte mask inputs from all processors less speculative than the processor that initiated the read are ORed together and sent back along with the data to set the "speculatively modified" bit there correctly, as was described in Section 5.3.1. This is done by setting the inputs for processors more speculative than the one making the read are set to 0, since more speculative data cannot travel "upstream" to a less speculative processor. However, the inputs from these more speculative processors are also ORed together and sent back to the reading processor so that it may set its "pre-invalidation" bit correctly. Once the most recently written data for each byte is chosen by the priority encoders, 16 byte-by-byte muxes select the most recent data and put all 16 bytes out on the read bus in parallel. While this prioritization, byte assembly, and control bit generation process is reasonably complex, it may be done in parallel with each L2 cache read, which is normally a multicycle operation already.

5.3.3.3 Speculative Data Buffer Control

The speculative write bus commands described below in Section 5.4.2 also cause effects among the L2 cache buffers. These effects are the way that the speculation software handlers control the buffers. A small central control unit among the L2 buffers snoops the write bus at all times for accesses to 0xAFFF0000. When such a write is seen, the command in the 4 low-order bits of the data is extracted and the L2 buffers respond accordingly, as is described in Table 5-3.

Two basic things can happen in response to these commands. First, the L2 buffer associated with the processor issuing the command may initiate a clearing or committing cycle. In these cases, the controller switches the active buffer from among the processor's pair of buffers while the other one commits or clears, as described in Section 5.3.3.4. Second, the sequencing of the 4 processors may be rearranged. The control unit has to track the processor sequencing so that it may properly merge lines from the different buffers together on reads of the L2 cache, as described in Section 5.3.3.2. It keeps a simple 0 (least speculative) to 3 (most speculative) sequence of the buffers at all times, that corresponds to the order listing of the speculation states in the various processors (see Section 5.4.2).

5.3.3.4 Speculative Buffer Clearing and Committing

When a processor finishes using a buffer, it sends a command to the L2 buffer controller that initiates either a clearing operation, discarding the contents of the buffer, or a committing operation, causing the contents of the buffer to be permanently merged into the L2 data array. Both of these

Table 5-3: L2 buffer effects caused by speculative write bus commands

#	Command	Buffer Response	Sequence Change
0	Clear Buffer	—	Sequence is reset to 0, 1, 2, 3 (same ordering as processor IDs)
1	Kill Buffer	Clear L2 buffer	—
2	Commit Buffer	Commit L2 buffer	—
3 – 4	N/A	—	—
5	Kill	Clear L2 buffer	—
6	Kill and Halt	Commit L2 buffer	—
7	Commit and Head	Commit L2 buffer	Sender is moved to end of sequence (#3), and others are rotated down one position
8	Kill Others	—	—
9	Fork	—	Nothing if the sender is at end of sequence (#3). Otherwise, the #3 buffer moves down to the sequence number of the sender + 1. Any buffers originally from sender + 1 to #2 are incremented up in position by 1.
10	Start Loop	—	—
11 – 14	N/A	—	—
15	Free	—	—

operations require the buffer to cycle through all of its current entries and process them. To perform this, each pair of buffers is supplied with a counter that can iterate from the head to the tail of the speculative buffer that is clearing or committing. When an L2 command initiates a clear or commit operation, this counter is cleared to 0 and then runs through the buffers up to but not including the one pointed to by the tail pointer. At this point, the tail pointer is reset to 0 and the buffer is ready for re-use. Also, the processor associated with the pair of buffers is notified that the alternate buffer in its pair is ready for use (as described in Section 5.3.3.2).

When clearing a buffer, generally each entry is ignored and discarded. However, if the entry has an EIB and address locked into it (as described in Section 5.3.3.1) because the L2 set associated with the line has filled up completely, then the clearing processor waits for a cycle when the tag array is not being used. At this point, it re-runs the formerly blocked write through the tag processor, so that the tag replacement may be properly handled. Simultaneously, the EIB number of the blocked access is sent to the external interface unit, freeing the EIB entry and allowing it to continue on its way through main memory.

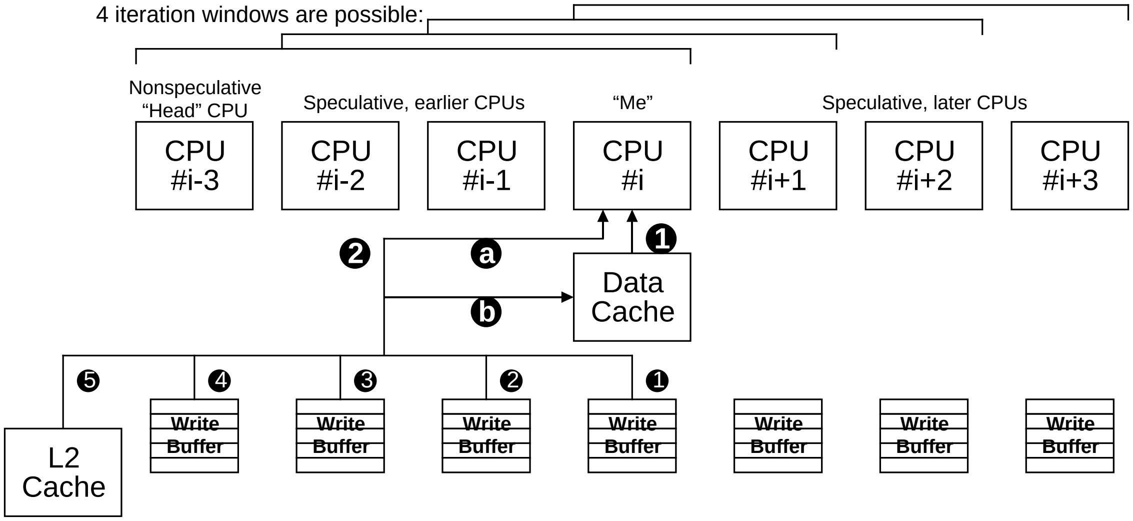
When committing a buffer, processing of stalled tag/EIB pairs may also occur. These occur just as they do while clearing a buffer. In addition, however, the committing processor also looks for cycles when the data array is free so that the now-committed data may be properly written into the L2 cache array. It should be noted that this is not as difficult as it sounds, since any speculative writes occurring at the time (the most common form of reference that the L2 will be processing during speculation) are not allowed to use the L2 cache data array directly. The committing processor just needs to use a few of the many cycles that avoid reads and non-speculative writes, which always need to use the L2 data array directly.

5.3.4 An Overall View of the Speculative Memory System

To end this section, a brief example of how the various buffers and bits actually work together is helpful. Figure 5-8 and Figure 5-9 illustrate the operation of the various components in the speculative memory system during the execution of a speculative load or store, respectively.

5.4 Speculative Thread Control Hardware

The speculative coprocessor, attached to each CPU as MIPS coprocessor CP2, is the key hardware mechanism for controlling the operation of speculative memory when the Hydra system is used to execute speculatively parallelized threads of control. It performs four basic functions: it provides direct control of the speculation features of the L1 caches, it controls and screens speculative interrupts caused by messages from other processors, it holds key data to speed up speculative software handlers, and it provides hardware support for maintaining runtime statistics with speculation.



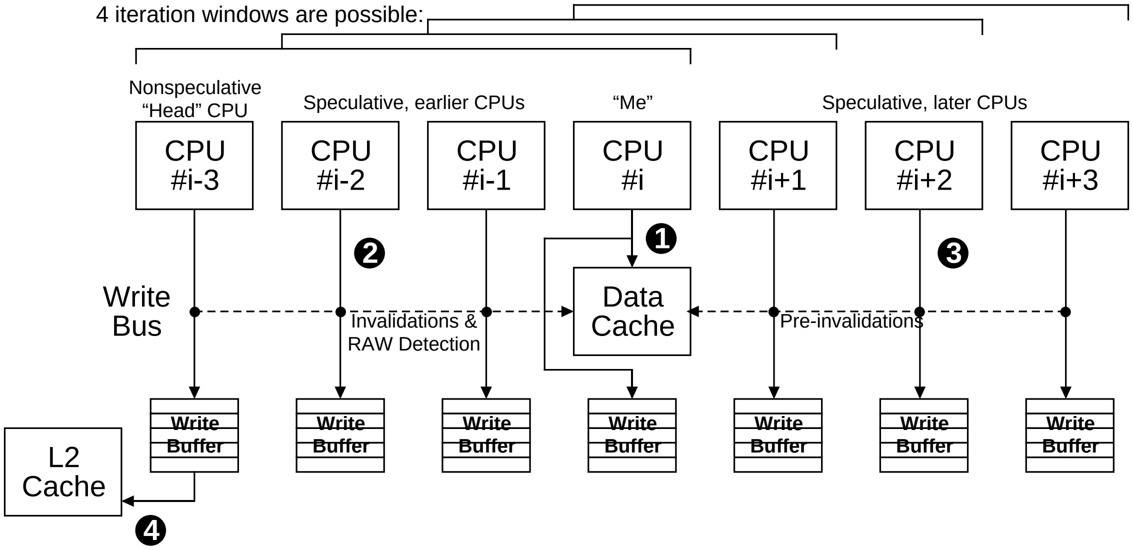
1. A CPU first reads from its data cache. The read bit for the word is set, if the written bit for the word does not indicate that it is already a local copy.
2. In the event of an data cache miss, the L2 cache and write buffers are all checked in parallel. The newest bytes written to a line are pulled in by pri-ority encoders on each byte, according to the indicated 1–5 priorities (1 = highest priority, 5 = lowest). This line is then returned to the CPU using the read bus. The requested word is delivered to the CPU (a), while the line is delivered to the data cache (b). The read bits for the word just read and the modified bits are set. A possible optimization would be to not set the modi-fied bit if the line only came from the L2 cache, without any speculative additions from the buffers, but we chose not to implement this.

Figure 5-8: The operation of speculative loads in Hydra

5.4.1 The CP2 Registers

In order to allow relatively easy assembly-language coding of the speculative software control routines, CP2 is designed so that it can be used with MTC2 (move to coprocessor 2) and MFC2 (move from coprocessor 2) instructions, which any MIPS assembler should be able to generate. Table 5-4 summarizes the functions of the important CP2 registers. Some registers are read-only or write-only, while others are read-write. A few are just normal registers, with no special hardware functionality. These are used by the software speculation handlers for long-term storage of key values that would otherwise have to be constantly loaded to and from main memory.

The command register, register 0, is designed to allow several commands to be issued to the speculative coprocessor at once, based on which bits in the word moved to the register happen to be set. Table 5-5 summarizes the function of the various command bits.



1. On a store, each CPU writes to its data cache, if the line is present there, and its assigned write buffer, using the write bus. The modified bit of any hit lines in the data cache are set. If the read bit of the word stored to is cleared, then the written bit is set to indicate that this word is now a local copy. The data from the store is recorded in the store buffer in a newly-allocated line or included in an existing line.
2. Earlier CPUs invalidate data cache lines directly, if they write to a cache line present in the data cache. Also, these writes cause dependence checks. If they write to a location in the data cache or victim buffer with the read bit set, a true dependence violation has been detected, and the pro-cessor is forced to restart.
3. Later CPUs just cause the pre-invalidate bits in our data cache lines to be set, so that the lines will be invalidated when a new thread is allocated to this CPU.
4. When the contents of a write buffer are no longer speculative, the buffer is allowed to drain out into the L2 cache on free cycles.

Figure 5-9: The operation of speculative stores in Hydra

The state statistics control register, number 19, performs various functions associated with statistics measurements. While this register could be merged with the command register, we currently have left it separate. Table 5-6 summarizes the function of the various command bits.

5.4.2 Speculative State Tracking

A key feature of Hydra's speculative execution is that it forces sister threads to behave as if they had been executed serially, even when they are actually run in parallel. In order to do this, it needs to keep track of the current ordering of the threads currently running on the different processors in hardware. To maintain the ordering, each processor has a hardware "speculative state" counter that records two things: whether or not it is running a speculative thread, and its current ordering in the list of processors. This state is accessible to software using CP2 register #1, and its interpretation is given in Table 5-7.

While it is possible to set a processor's speculative state using direct writes to CP2 register #1, in practice this is only used when starting up or ending speculation. During speculation, special memory writes made by the speculative software handlers cause the speculative state in all 4 processors to be updated simultaneously, as the writes pass out over the write bus. These special writes are to address 0xAFFF0000, out in the middle of uncached kernel memory. When these writes pass over the write bus, the CP2s attached to all four of Hydra's processors snoop the data sent with the write and interpret the lower 4 bits of the data as a command. The L2 cache controller also snoops for this address, as some of these commands are directed at it also, as was explained in Section 5.3.3.3. If multiple simultaneous speculative processes are allowed to execute simultaneously, the upper 28 bits of the word are the PID of the speculative process that the command applies to, so that processors running different speculative processes are able to ignore messages that don't apply to them. Table 5-8 summarizes these commands and the state changes that these commands may cause in the various processors. "Order," a term used in the table, is equal to the ((CPU state – 1) % 4) + 1, so that free and running processors together are numbered from 1 to 4.

In practice, each CP2 must know the state of the other processors in order to properly handle the state transitions described in the previous table. This can be accommodated by either having redundant copies of the state machines for all 4 processors in each CP2 or by having the state machines in all 4 processors cross-wired to each other. Either way is acceptable, but the redundant technique is probably easier to build.

Table 5-4: The CP2 Registers

#	Name	Use	Function
0	Command	W	Causes various coprocessor operations to be initiated if bits are written to it (see below)
1	State	R/W	Allows the current state of this CPU’s speculation state machine to be read or modified
2	Master Pointer	Reg.	Storage place for pointer to the global speculation data structures
3	Speculation PID	R/W	Storage place for the OS process identifier that "owns" this speculative thread. If we allow multiple speculative contexts, this is used to differentiate between them at runtime. The lower 4 bits MUST be 0.
4	Head Flag	R	Flag that is 1 when we’re the "head" nonspeculative processor, 0 otherwise
5	CPU Pointer	Reg.	Storage place for pointer to this processor’s local speculation data structures
6	Current Vector	R	Following an interrupt, the interrupt vector that should be taken is read out of here by the CPU and JR’ed to [NOTE: This isn’t present in LESS, since it’s not necessary.]
7	Restart Vector	Reg.	Sets the speculative restart vector at the beginning of the currently executing region of code
8	Interrupt Vector	W	Sets the address of the normal OS interrupt handler [NOTE: This is used for an internal function by LESS.]
9	Violation Vector	W	Sets the location of the speculative violation handler, called when the L1 cache detects a speculative RAW hazard
10	Hold Vector	W	Sets the location of the speculative hold handler, called when a line with read bits set is about to be thrown out of the L1 or if the L2 buffers for this CPU fill up
11	Exception Vector	W	Sets the location of the routine called when exceptions are generated by code executing speculatively
12	Total Kill Vector	W	Sets the location of the routine called by cancelled speculative threads after a loop ends (label loops) or when threads are squashed after a violation (slow loops and procedures)
13	Quick Kill Vector	W	Sets the location of the routine called when threads are squashed in mid-loop (label or quick loops)
14	LTB Violation Time	R	Time recorded off of the violation time counter (register 24) when the last violating load actually was issued, on an LTB hit (0 otherwise). Valid from the time of the violation until the next speculative region is started.
15	LTB Violating Load PC	R	PC of load that caused a violation, on an LTB hit (0 otherwise). Valid from the time of the violation until the next speculative region is started.

Table 5-4: The CP2 Registers

#	Name	Use	Function
16	Violating Store PC	R	PC of store that caused a violation. Valid from the time of the violation until the next speculative region is started.
17	Violation Address	R	Address of the load/store pair causing the most recent speculative violation. Valid from the time of the violation until the next speculative region is started.
18	Receive Fork Vector	W	Sets the location of the routine called when this processor is starting a new procedure, under procedure speculation
19	State statistics control	R/W	Controls state statistics (see below) [NOTE: This register is not implemented in the LESS simulator, since these values can be set directly in C.]
20	"Bad" Statistics Flag	R/W	For writing. When read, returns the "running-discarded" state accumulator contents.
21	"Good" Statistics Flag	R/W	For writing. When read, returns the "waiting-discarded" state accumulator contents.
22	"Run" Statistics Flag	R/W	For writing. When read, returns the "running-used" state accumulator contents.
23	"Idle" Statistics Flag	R/W	For writing. When read, returns the "waiting-used" state accumulator contents.
24	Violation Count	R/W	Returns the time since the L1 cache was last advanced or backed up if read. If written, this timer is cleared. [NOTE: not in LESS]
25	"Wait" Statistics Flag	R/W	For writing. When read, returns the "overhead" state accumulator contents.
26	"Clear" Statistics Flag	R/W	For writing. When read, returns the "idle" state accumulator contents.
27–31	N/A	—	5 unused registers

5.4.3 Speculative Interrupts

For some of the commands listed in Section 5.4.2, software speculation handlers may also need to be invoked in order to properly handle the situation. When this is necessary, the coprocessor raises the interrupt line to its processor. This is possible because the external interrupt line for the processor is re-routed into the processor's CP2, instead. The CP2 screens external interrupts and adds speculative interrupts, in order to generate the interrupt line that is passed to the processor core. When any interrupt occurs, CP2 makes the “current vector” register (#6) equal to the appropriate

Table 5-5: CP2 Command Register (#0) Bit Definition

#	Name	Function
0	Enable Speculative Exceptions	Turns normal screening of speculative exceptions on and causes the "current vector" register to return the appropriate vector for any necessary speculative exception
1	Disable Speculative Exceptions	Causes the processor to ignore all speculative exception causing events, and the "current vector" register will always return the Interrupt Vector
2	Enable Speculative Memory References	Switches system so that loads and stores will now be speculative
3	Disable Speculative Memory References	Switches system so that loads and stores will be nonspeculative
4 - 5	N/A	—
6	Advance L1 Cache	Causes all speculative bits in the L1 cache to be cleared. Meanwhile, all lines with the "pre-invalidate" bit set are invalidated.
7	Backup L1 Cache	Causes all speculative bits in the L1 cache to be cleared. Meanwhile, all lines with the "modified" bit set are invalidated.
8 - 31	N/A	—

vector that was previously stored into its interrupt vector registers. This way, the primary interrupt handler can quickly determine what handler needs to be executed with a single MFC2 instruction.

Table 5-9 summarizes how the speculative commands may cause interrupts on the various processors in the system. The “Interrupts Who?” column indicates how CP2 must selectively pass interrupts into the processor core based on the current speculative state of the processor and the source of the bus command.

The “Quick Kill” vector allows optimized thread squashing routines during label-based and quick loops (as described in Section 5.5). During subroutine speculation and slow loops, however, it is set equal to the Total Kill vector by software, which processes all kills at those times.

In addition to the interrupts caused by bus commands, some speculative interrupts are caused by conditions determined within the processor that demand software attention. Table 5-10 summarizes these exceptional conditions.

Table 5-6: State Statistics Control Register (#19) Bit Definition

#	Name	Function
0	Read Head State	When set, causes registers 20-23, 25, and 26 to be read as state accumulators from the "head" (nonspeculative) speculative state
1	Read First Speculative State	When set, causes registers 20-23, 25, and 26 to be read as state accumulators from the second (first speculative) speculative state
2	Read Second Speculative State	When set, causes registers 20-23, 25, and 26 to be read as state accumulators from the third (second speculative) speculative state
3	Read Third Speculative State	When set, causes registers 20-23, 25, and 26 to be read as state accumulators from the fourth (third speculative) speculative state
4	Counter State RUN	Causes the "running" counters to be the actively incrementing ones
5	Counter State WAIT	Causes the "waiting" counters to be the actively incrementing ones
6	Accumulate State RUN	Switches the state accumulation hardware to "running" mode
7	Accumulate State IDLE	Switches the state accumulation hardware to "idle" mode
8	Sliding Violation Tracking ON	The violation tracking mechanism tracks the last 32 (could vary) speculatively loaded addresses
9	Sliding Violation Tracking OFF	The violation tracking mechanism tracks only the first 32 (could vary) speculatively loaded addresses
10	Read Lower Half of Accumulators	When set, causes registers 20-23, 25, and 26 to read the lower 32-bit halves of the 64-bit state accumulators
11	Read Upper Half of Accumulators	When set, causes registers 20-23, 25, and 26 to read the upper 32-bit halves of the 64-bit state accumulators
12–31	N/A	—

Finally, the command register bits 0 and 1 may be used to turn speculative interrupt processing on or off. When it is off, then only “normal” interrupts and exceptions that could occur in the unmodified IDT core are allowed. This feature should be used to protect any threads running nonspeculatively from receiving spurious interrupts, in case their speculation state is erroneously set to something other than “dead,” which normally deactivates all speculative exceptions.

Table 5-7: Speculative States

#	Name	Function
0	Dead	This processor is not currently running a speculative thread. It may be running something unrelated to speculation, however.
1	Free — Head	This state is not possible
2	Free — Third	This processor is waiting for a speculative thread, and is third in the queue for threads
3	Free — Second	This processor is waiting for a speculative thread, and is second in the queue
4	Free — First	This processor is waiting for a speculative thread, and will be the next to receive one
5	Running — Head	This is the most fundamental state. It represents the "head" speculative processor, that is actually non-speculative. When only one thread is running, it is this one.
6	Running — Second	This processor is running a speculative thread. It is the least speculative processor, just after the "head" processor.
7	Running — Third	This processor is running a speculative thread. It is the middle speculative processor.
8	Running — Fourth	This processor is running a speculative thread. It is running the most speculative thread.

5.4.4 Speculative Memory Accesses

While threads are running speculatively on the processors in Hydra, their memory accesses must be made in a special “speculative” mode, which causes special behavior for both reads and writes in order to allow hardware simulation of “sequential” program behavior even when speculatively executing threads are running in parallel. Speculative reads are tracked in the processor’s L1 cache, so that RAW violations between “later” threads and “earlier” threads in the system may be properly resolved. Speculative writes are buffered at the L2 cache using the special speculative buffers described in Section 5.3.3, so that the writes associated with a speculative thread may be discarded or permanently committed, depending upon whether the speculative thread is restarted or successfully completes. This section discusses the functionality that has to be in the speculative coprocessor to allow these accesses to work correctly.

If speculative memory accesses are enabled for a processor using bits 2 & 3 in the command register, then the coprocessor will set the “speculative reference” bit for most accesses going to mem-

Table 5-8: Speculative Write Bus Commands and the state changes they may cause

#	Command	Function	State Change
0	Clear Buffer	Resets L2 buffers to default positions. For use at speculation startup only.	—
1	Kill Buffer	Discards this processor's current L2 buffer.	—
2	Commit Buffer	Commits this processor's current L2 buffer.	—
3	N/A	—	—
4	N/A	—	—
5	Kill	Kill L2 buffers for this processor and all of higher order.	—
6	Kill and Halt	The head processor has finished a loop, so everyone must drop loop iterations and jump ahead to post-loop code.	—
7	Commit and Head	The head processor has finished a thread, so everyone must rotate around to the next position	Source processor goes to state 8, while other processors all have their state decreased by 1.
8	Kill Others	Same as kill, but this processor's buffer is not affected by the kill.	—
9	Fork	Start a new speculative set of subroutine completion code, grabbing the most speculative processor and starting it on the thread if possible.	Nothing if the source is in state 9. Otherwise, the processor of order 4 goes to the state 1 greater than the source processor's. Any other processors with an order higher than the source are incremented up in state by 1.
10	Start Loop	A processor is starting a loop, so all more speculative processors should join it and get loop iterations.	All free processors get their state incremented by 4, so they are now running.
11–14	N/A	—	—
15	Free	The processor has discovered that it has no work to do, so it declares that it is going idle.	Source processor goes to state 4. All processors of higher order than the source have their state decreased by 1.

ory, so that they are handled properly by the speculative hardware in the L1 and L2 caches. The mechanisms used to handle these accesses are described in Section 5.3.1 and Section 5.3.3. However, not all loads and stores made while speculative memory accesses are enabled will actually be speculative. There are two major exceptions.

Table 5-9: Speculative Write Bus Commands and the interrupts they can cause

#	Command	Interrupts Who?	Vector
0	Clear Buffer	—	—
1	Kill Buffer	—	—
2	Commit Buffer	—	—
3	N/A	—	—
4	N/A	—	—
5	Kill	All more speculative processors.	Quick Kill Vector
6	Kill and Halt	All more speculative processors.	Total Kill Vector
7	Commit and Head	—	—
8	Kill Others	All more speculative processors.	Quick Kill Vector
9	Fork	State 4 or 8 processor, whichever is running.	Receive Fork Vector
10	Start Loop	All more speculative processors.	Total Kill Vector
11–14	N/A	—	—
15	Free	—	—

Table 5-10: Internal conditions that can cause speculative interrupts.

Cause of Interrupt	Vector
RAW Violation detected by L1 cache logic	Violation Vector
L1 line with read bits set being thrown out of L1 cache by a line replacement	Hold Vector
A store attempts to overflow the L2 buffer for this processor, and the local copy of the L2 speculation buffer tags detects this condition	Hold Vector
An instruction executing in this processor's speculative thread causes a normal OS-handled exception	Exception Vector
External Interrupt Line is raised	Interrupt Vector

The first exception occurs in the interpretation of the MIPS LL (load locked) instruction. Since there is no need to use traditional MIPS LL/SC synchronization during speculation, the LL opcode is free for other use. Hydra remaps it during speculation to produce normal non-speculative loads, instead. This allows a user speculative thread to load data without enabling RAW violation detection, when this is desired. For example, this mechanism may be used to do spinlock synchronization between speculative threads in order to “protect” critical variable references that commonly cause RAW violations. This is described fully in Section 6.2.1.

Unlike LL instructions, SC instructions are illegal during speculation. If one would attempt to use them in their “normal” manner, erratic and unpredictable results would occur as the speculation system interfered with the limited “speculation” performed by LL/SC. Also, there is no compelling reason to remap SCs to any special type of store, such as a non-speculative form, during speculation. A store that bypasses the speculation logic can never be “undone,” which could result in odd behavior if a thread making the store were to be restarted. Similarly, uncached reads and writes performed speculatively will cause unpredictable behavior, since these references will avoid all of the speculative mechanisms by their very nature.

The second exception occurs when the thread running on a processor becomes the head, nonspeculative thread. At this point in time, some stores are made nonspeculatively. When each speculative store is issued, the local L2 buffer tags (Section 5.3.2) are consulted. If the store hits in a write buffer line already contained within those tags, then it continues to issue in a speculative manner. However, if the store misses in those tags, and would therefore require the allocation of another L2 write buffer entry, then the store is converted to a non-speculative one that writes directly to the L2 cache. This technique minimizes the unnecessary use of the L2 buffers that would otherwise occur after the originally speculative thread became non-speculative. This trick has three advantages over letting references pass out speculatively. First, fewer write buffer entries are required, saving memory and commit time (see Section 5.3.3.4). Second, if a processor runs out of L2 buffer space and must stall using a “hold” exception, it will automatically work correctly after it is restarted as the head processor. Finally, fewer L2 cache lines will be locked down by speculative writes (as described in Section 5.3.1).

Overall, each memory reference may be modified by 3 control lines that indicate if the reference is uncached, LL/SC, and/or speculative. While CP2 doesn't affect the uncached line, it may modify the other two lines according to the following equations:

- **LL/SC** = (LL && !SpeculativeReferences) || SC
- **Speculative** = SpeculativeReferences && !LL && !SC && !(store && HeadThread && !LocalL2TagHit)

5.4.5 Statistics Mechanisms

Simple timing of speculative threads may be done with the existing CP0 count register (#9), which counts at 1/2 the clock rate. However, this mechanism generally requires a fair amount of software support to keep track of statistics, since it is so primitive. As a result, we have added some specialized hardware statistics registers to calculate many critical speculation timing statistics while incurring minimal software overhead.

5.4.5.1 State-Time Accumulators

CP2 adds a set of state-time accumulators that add up the amount of time spent by speculative threads in overhead, useful work, discarded work, and waiting (for useful and discarded threads). Idle time (i.e. useless time between speculative regions) is also tracked. Each of the individual accumulators is actually maintained as a set of 4 separate 64-bit registers that sort the time spent in each of these states into 4 subcategories, by speculative processor order number (see Section 5.4.2). These accumulators work with only a small amount of overhead added to the speculation software handlers.

Internally, two sets of four counters are maintained — a "running" counter set and a "waiting" counter set. During speculation, one counter is always incrementing, depending upon whether the system is in "running" or "waiting" state and which speculative processor order number this processor has. The only time that a counter is not incrementing is when the speculative order state is "dead" (i.e. not speculating at all). On any cycle, the single counter selected by the processor's state and the processor's speculative order number increments by one. The counters and the single incrementer that they share are depicted in the top half of Figure 5-10.

The system is also always in one of two accumulation states — "idle" or "running." All of these states may be manipulated manually using the CP2 state statistics control register (#19), if desired. During normal operation, MTC2 operations to the 6 accumulator registers cause specific actions to be taken with the counter contents, depending upon the accumulation state, as noted in Table 5-11. The groups of adders in the lower half of Figure 5-10 are responsible for actually accumulating the counters with the accumulators selected by the particular MTC2 command. Simply by inserting the appropriate MTC2 commands into the software speculative handlers, statistics may be taken to get a reasonable picture of how the system is spending its time.

Figure 5-11 shows a couple of typical scenarios that are used by the speculation software handlers to get approximate timing information from the execution of a speculative program. The "Idle" accumulator is used to track the time that a particular processor is waiting in the "free" speculative states, waiting for a thread. When speculation is actually occurring, the "Overhead" accumulator collects the time spent in the speculation software handlers while the four "Runnning/Waiting" accumulators track the amount of time spent actually running speculative threads, or waiting to become the head processor.

5.4.5.2 Violation Tracking

Another statistic mechanism included in CP2 is a violation time counter, which tracks the time since the L1 cache was last advanced or backed up using the appropriate command register commands. This may also be cleared by writing CP2 register #24, or may be read by reading that register.

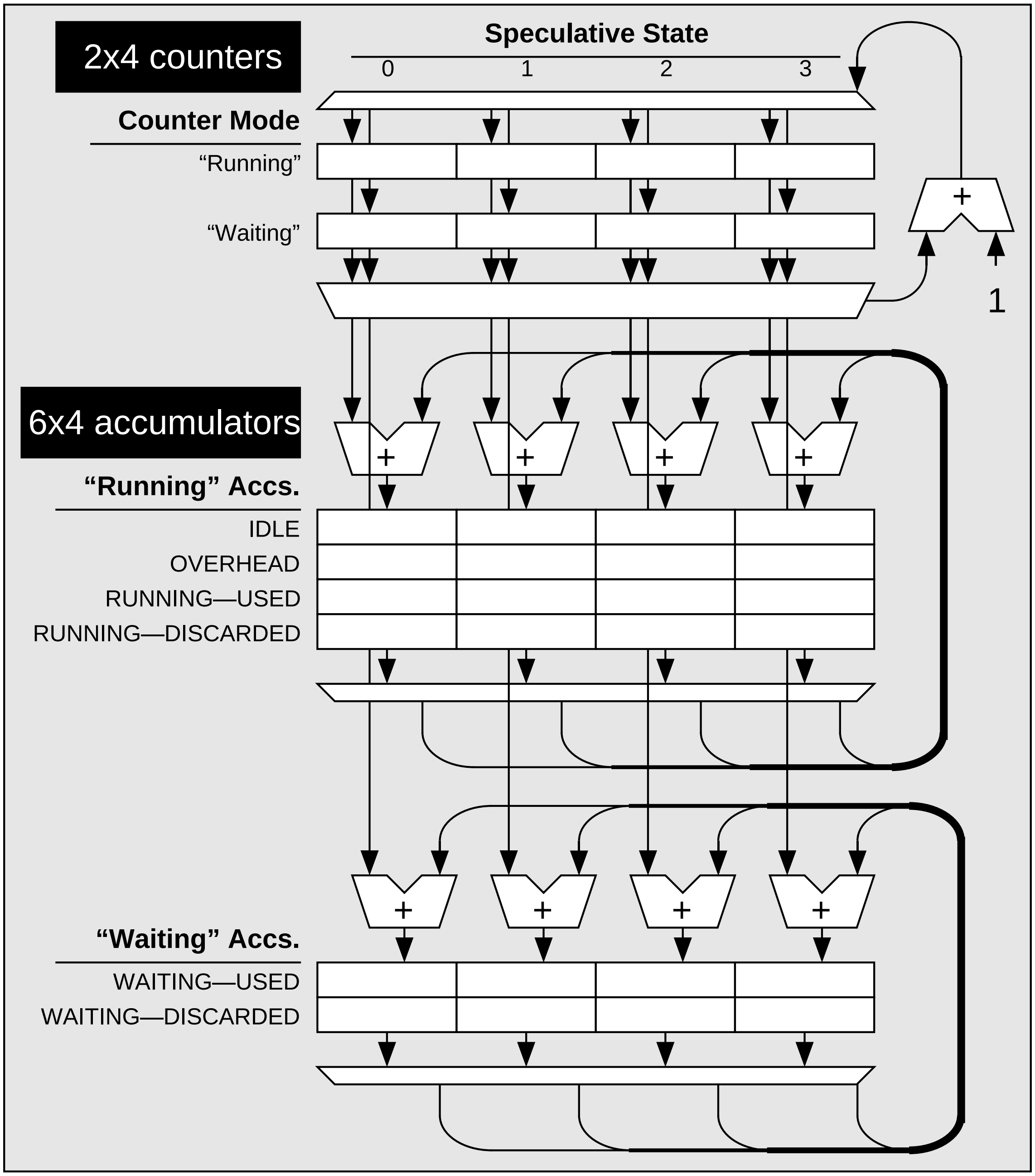


Figure 5-10: A schematic overview of the state-time accumulator system

In order to track the instructions that cause violations, the coprocessor maintains a circular Load Tracking Buffer (LTB) that tracks the 32 most recent unique speculative load addresses in the current speculative thread, if "sliding tracking" is enabled using the statistics control register (#19). If

Table 5-11: Responses to any writes to speculative state transition registers

Reg.	Transition	Action if "Idle	"Action if "Running"
20	Bad Work	Adds running counter to "idle time" accumulator. Switches to "Running" state. Counters are cleared and running counter is started.	Adds running counter to "running-discarded" accumulator and waiting counter to "waiting-discarded" accumulator. Counters are cleared and running counter is started.
21	Good Work	Adds running counter to "idle time" accumulator. Switches to "Running" state. Counters are cleared and running counter is started.	Adds running counter to "running-used" accumulator and waiting counter to "waiting-used" accumulator. Counters are cleared and running counter is started.
22	Run	Adds running counter to "idle time" accumulator. Switches to "Running" state. Counters are cleared and running counter is started.	Adds running counter to "overhead" accumulator. Counters are cleared and running counter is started.
23	Idle	Adds running counter to "idle time" accumulator. Counters are cleared and running counter is started.	Adds running counter to "overhead" accumulator. Switches to "Idle" state. Counters are cleared and running counter is started.
25	Wait	No action.	Causes the internal "waiting" counters to be the actively incrementing ones, switching off the "running" counters.
26	Clear	Clears all of the statistics counters and accumulators, resetting the state statistics system. Also, initially sets the processor to the "Idle" accumulation state.	

“sliding tracking” is not enabled, the LTB just buffers the first 32 load addresses only. This buffer is cleared, along with the violation time counter, at the beginning of each speculative thread. As the thread progresses, each speculative load with a unique load address is noted in the LTB by storing its PC, memory address, and violation time count, as is noted in Table 5-12. If the LTB overflows in “sliding tracking” mode, then the oldest entries are discarded as new ones write over them. The load address fields in the buffer are maintained as a block of CAM memory, and this CAM is checked before each load is entered into the LTB. If a load’s memory address has already been entered into the LTB by a previously executed load, then the later one is *not* recorded.

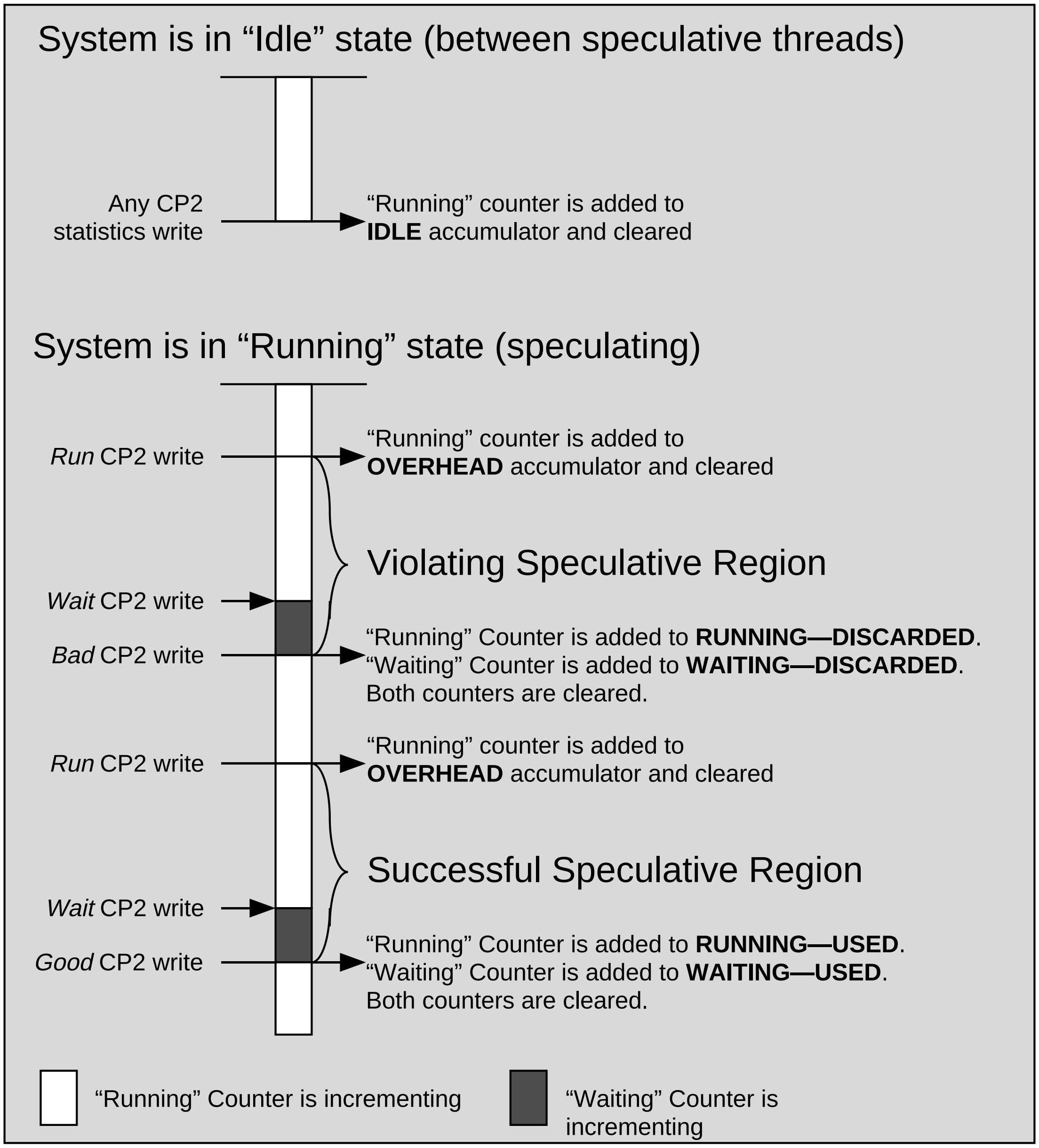

System is in “Idle” state (between speculative threads)
Any CP2 statistics write
“Running” counter is added to **IDLE** accumulator and cleared
System is in “Running” state (speculating)
Run CP2 write
“Running” counter is added to **OVERHEAD** accumulator and cleared
Violating Speculative Region
Wait CP2 write
Bad CP2 write
“Running” Counter is added to **RUNNING—DISCARDED**.
“Waiting” Counter is added to **WAITING—DISCARDED**.
Both counters are cleared.
Run CP2 write
“Running” counter is added to **OVERHEAD** accumulator and cleared
Successful Speculative Region
Wait CP2 write
Good CP2 write
“Running” Counter is added to **RUNNING—USED**.
“Waiting” Counter is added to **WAITING—USED**.
Both counters are cleared.
“Running” Counter is incrementing
“Waiting” Counter is incrementing

Figure 5-11: A state-time statistics operation example

When the L1 speculation bits detect a violation with a store passing over the write bus, then the CAM array is used in an attempt to find the load that incorrectly read speculative data too early. If found, the PC, address, and time count from the load’s LTB entry and the store PC from the write bus are made available in CP2 registers 14–17, so that a software statistics handler may use them.

Table 5-12: Load Tracking Buffer fields

Buffer Entry Section	Size	Function
Load PC	30 bits	Holds the address of a speculative load that may potentially cause a violation.
Load Address	30 bits	Holds the address of the word that was read during the load. This entry must be a CAM memory.
Violation Time Count	32 bits	A set of lines that indicate which state machine is using the write bus, so that snooping CPUs will access the proper address bus. Alternately, an entire 30-bit address bus can be placed here, to avoid muxing from the address buses.

5.5 Speculative Thread Control Software

The Multiscalar [28] and TLDS [14] speculative architectures take radically different approaches to finding speculative threads within an application. The Multiscalar architecture breaks a program as a sequence of arbitrary "tasks" to be executed, and then allocates tasks in order around its ring of processors. While the division of a program into basic block-sized "tasks" is done at compile time, all dynamic control of the threads is performed by ring management hardware at runtime. On the other hand, the TLDS architecture is based on a less tightly coupled chip multiprocessor and ends up near the opposite end of the spectrum. All thread control is handled by multiprocessor software routines that are automatically added to a program at the beginning and end of speculative "epochs" by a compiler, with only minimal hardware support.

We have used a combined hardware/software approach, similar to TLDS but with more hardware support from our CP2 coprocessor, to divide programs into threads and then to spread the resulting threads among our CPUs. While the CP2 coprocessor provides significant help in the form of special instructions and registers, the core of the control system is contained in the small assembly-language software handlers, listed in Table 5-13. These handlers attempt to divide applications using the two techniques introduced in Section 5.1.2. First, subroutine calls can cause a light-weight "fork" to occur that branches off a speculative thread. Afterwards, the original CPU immediately takes a checkpoint of the CPU state and makes the subroutine call itself, while the execution of the code following the subroutine return is handled speculatively by another CPU.

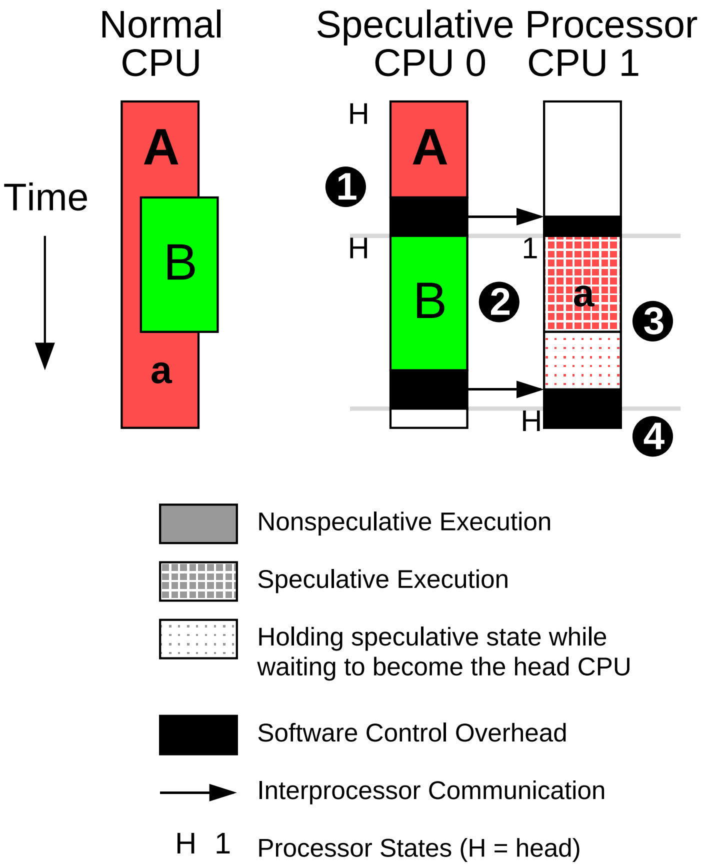
Second, when specially marked loops are found in the course of running a program, the loop iterations are spread out across all of the CPUs in the system.

5.5.1 Subroutine Threads

Subroutine speculation is initiated with a “fork” message to other processors when a subroutine call is detected in the portion of the program working on the head processor. Figure 5-12 depicts the sequence of actions in a typical fork. When a subroutine call is detected, the processor immediately notifies a free processor that it should start working on the code in the original routine following the subroutine being called. Most of the forking processor’s register state is passed along with the fork command. That processor then starts executing the post-call continuation code speculatively. Meanwhile, the original processor tries to execute the subroutine itself, non-speculatively. Functions that return a result cause a predicted return value to be used by the speculative thread. While more complex schemes are possible, we just predict the last value returned by the function, which works fine in the common case of a function that doesn’t actually return anything. This prediction is later checked against the actual result returned. If they are different, or if a memory violation is detected, the instructions from the continuation code after the subroutine call are re-executed with the correct return value.

A single subroutine fork is fairly simple, but when multiple forks start happening in quick succession the situation gets much more complicated. Subroutine speculation is actually controlled using complex assembly-language handlers that form what is essentially a miniature thread-OS system. This system tracks speculative threads using a linked list of active threads ordered from least-to-most speculative. When a thread is created, it is inserted into this active list. The head processor is always running the thread at the beginning of the active list, while more speculative processors try to execute the next three threads — the least speculative ones — from the list.

When a subroutine call is detected, several steps must occur during the actual forking operation:


Normal
CPU
Speculative Processor
CPU 0
CPU 1
Time
A
B
a
A
B
a
H
H
1
H
1
2
3
4
Nonspeculative Execution
Speculative Execution
Holding speculative state while waiting to become the head CPU
Software Control Overhead
Interprocessor Communication
H 1
Processor States (H = head)

B is a subroutine called within the A/a routine.

1. The call is intercepted during normal execution and the a thread is sent out to CPU 1, along with a newly created RPB containing its starting state and the guess for the return value of B.
2. The original caller continues by executing the B subroutine, staying the head processor as this happens.
3. Meanwhile, CPU 1 picks up the a thread, the caller's continuation code, and executes it speculatively. Upon completing this speculative thread, it must wait to become the head processor. During both the execution and the waiting time, its speculation mechanisms watch stores from B to ensure that no true dependencies between the threads are violated. The a thread is restarted immediately when such a violation is detected.
4. Upon becoming the head, CPU 1 completes and returns (or restarts and re-executes the a thread if the original return value prediction was wrong).

Figure 5-12: An example execution pattern over time of a simple subroutine fork and return, with a key for the notation used here and in Figure 5-14.

1. The processor allocates a *register passing buffer* (RPB) for the thread it is creating by allocating one from the free buffer list maintained by the speculation control support software. Since our design does not incorporate direct interconnections between the processors, a buffer in

memory is necessary to temporarily hold a processor's registers during the register passing communication from processor to processor. In addition, since these registers may need to be reloaded if a thread is restarted following any sort of speculation violation, it makes sense to allocate a buffer once that can hold a thread's starting (or restarting) state throughout the thread's lifetime.

2. The new buffer is filled with all registers that may be saved across subroutines (9 integer and 12 floating point using standard MIPS software conventions), the current global and stack pointers, the PC following the subroutine call, and a prediction of the subroutine's return value. For this paper, we used the simple *repeat last return value* prediction mechanism used in [43]. While more complex schemes are possible, this technique works well because most functions tend to either return the same thing continuously (`void` functions and functions that only return error values are good examples), or they are completely unpredictable, and therefore should not be selected for speculative execution at all. These unpredictable functions are pruned off and marked as *unpredictable* after a few mispredictions have been detected by the mechanisms described in Section 5.5.4.
3. The new buffer is then inserted into the list of active buffers, immediately after the current processor's, as depicted in Figure 5-13 (a). This allows the list of active RPBs to also act as the active thread list, since any *child* thread created will always be the next-most-speculative thread. The thread list must be maintained in memory for two reasons. First, thread may be assigned to multiple processor over the course of its lifetime, so it is necessary to keep the thread list in a central location that all may access. Second, since there are frequently more threads available than processors, it is convenient to simply leave the RPBs for these "extra," very speculative threads lying in memory at the end of the active list until a processor can be assigned to them.
4. The processor finishes by notifying a free processor that it should load the registers in the newly created RPB and continue working on the code after the subroutine call. If no free pro-

cessors are available, then most speculative running processor will drop its thread and pick up the new one.

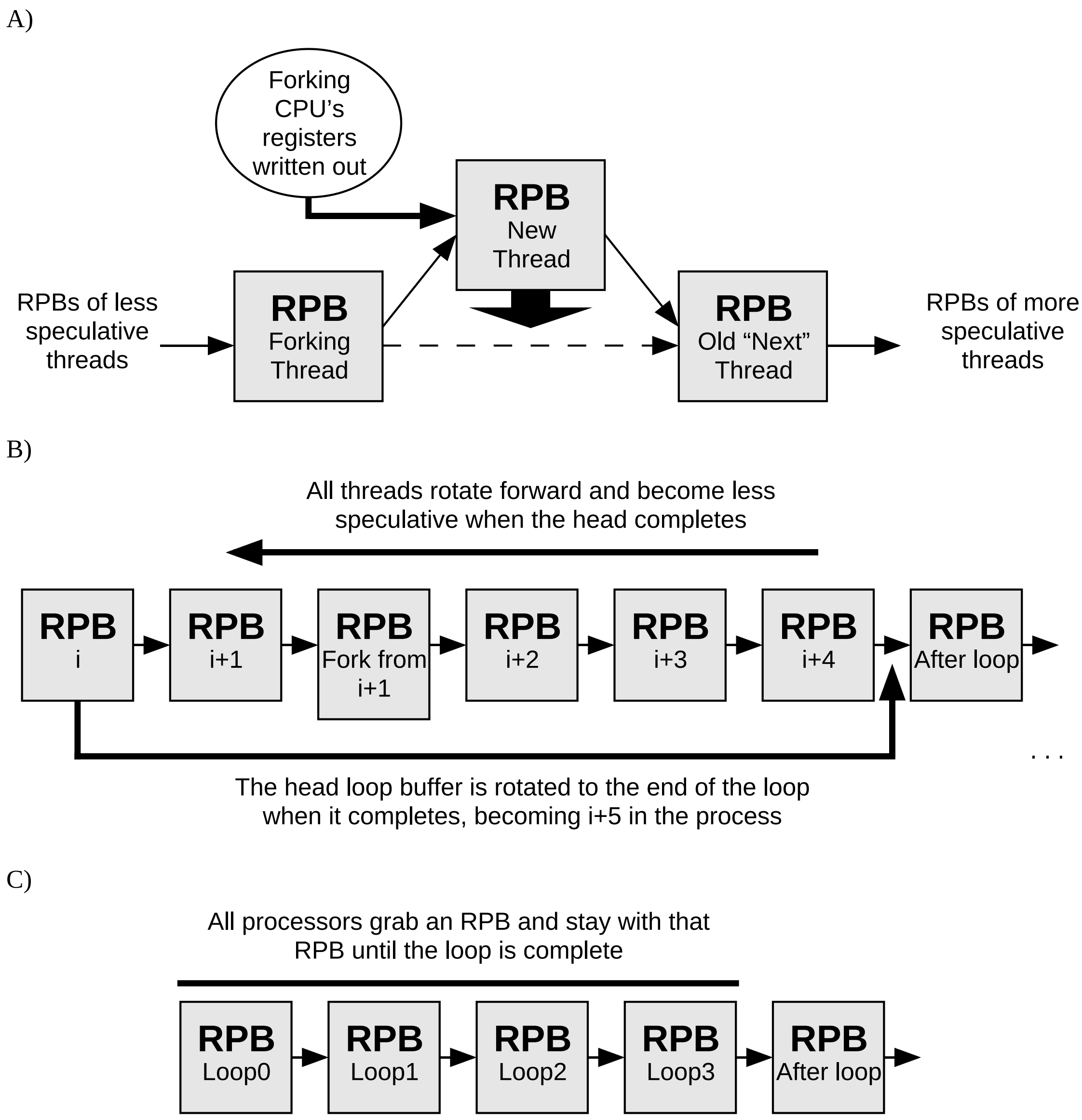

A)
Forking CPU's registers written out
RPB
New Thread
RPBs of less speculative threads
RPB
Forking Thread
RPB
Old "Next" Thread
RPBs of more speculative threads
B)
All threads rotate forward and become less speculative when the head completes
RPB
i
RPB
i+1
RPB
Fork from i+1
RPB
i+2
RPB
i+3
RPB
i+4
RPB
After loop
. . .
The head loop buffer is rotated to the end of the loop when it completes, becoming i+5 in the process
C)
All processors grab an RPB and stay with that RPB until the loop is complete
RPB
Loop0
RPB
Loop1
RPB
Loop2
RPB
Loop3
RPB
After loop

Figure 5-13: Managing the register passing buffers (RPBs). A) During a subroutine fork. B) During a "slow" loop. C) During a "quick" loop.

These steps are currently performed by an exception handler that is executed when a subroutine call is detected, so that we could use commercially available compilers to compile our benchmarks. While we have vectored exceptions for speculation that avoid the normal OS exception overhead, inlining the forking code using a speculation-aware compiler would definitely be more efficient, since only the live registers would need to be saved in the RPB. At the processor receiving the fork, another vectored exception handler gets a pointer to the new buffer from the active list, reads in the contents of the buffer into its registers, and starts executing the continuation code following the procedure call. Due to the overhead inherent in allocating a new buffer and then saving, communicating, and loading most of a processors' registers, very short subroutines cannot be handled by this mechanism without incurring excessive amounts of overhead.

When a subroutine completes after forking off its continuation code, it returns to the speculation support software, which performs several more steps to complete the forked subroutine:

1. It waits until it becomes the head processor. This is necessary because the processor must maintain its L1 speculation bits intact for this thread until after it becomes the head, since it may be restarted by dependence violations up until this point.
2. The actual return value of the subroutine is compared with the one predicted during the last fork. If a misprediction is detected, the return value is corrected in the RPB allocated during the last fork, and then all of the speculative processors are restarted so that they will execute using the new, correct return value.
3. The RPB of the current thread is returned to the free list as the next thread becomes the head.
4. The old head processor becomes the most speculative processor. At this point, it checks to see if there is a fourth RPB that is not assigned to any processor in the active list. If so, it starts running the thread associated with that RPB. Otherwise, it is freed until another fork occurs.

5.5.2 Loop Iteration Threads

Two different sets of control code are used for executing specially-marked loops (see Chapter 6) in speculative programs. When starting a large loop, in which the forking of subroutines within a loop is desirable, a circle of RPBs pointing to the loop body subroutine is inserted into the active

Table 5-13: A summary of the software handlers required to support speculation.

Group	Handler	Use	What it does	Overhead (instr.)
Subroutines	Fork	Exception, generated upon execution of a jump-to-subroutine (JALR) instruction	Allocates an RPB, saves necessary processor state into the RPB, and then sends a FORK command out to the most speculative (or free) processor.	~70
	Subroutine Completion	Routine returned to after a forked subroutine completes, instead of the normal return address	Waits to become the head, commits speculative data, checks the guessed return value from the last fork against the actual one, kills speculative processors if necessary, and then continues with a "Start Buffer" command to get a new thread.	~40 + full "Start Buffer"
	Start Buffer	Jumped to by completion commands or an exception triggered by a FORK or STARTLOOP	Selects one of the first 4 RPBs on the active list, deletes any old children generated by this thread during any restarted earlier executions, reads in its contents, and begins executing the subroutine or loop thread. If insufficient threads are available, the processor is freed.	Full: ~60 (loop) or ~75 (fork) Short: ~25 (loop) or ~40 (fork)
Loops	Start Speculative Loop	Called by spec_begin() in original program code	Sets up a ring of RPBs for "slow" loop operation, configuring their buffers correctly, then sends out a STARTLOOP bus command to other processors.	~75
	Start Speculative Loop (quick version)	Called by spec_begin_quick() in original program code	Sets up 4 RPBs for "quick" loop operation, configuring their buffers correctly, then sends out a STARTLOOP bus command to other processors.	~70
	End-of-Iteration / Terminate	Routine returned to after a speculative loop iteration completes	Waits to become the head, commits speculative data, moves its RPB to the end of the loop, making adjustments to it if it was used for a fork during the loop iteration, and claims the next available iteration with a COMPLETE command, or commits its speculative data, discards all of the RPBs within the entire loop, and kills all other loop iterations with a KILL command before picking up the buffer following the loop with a "Start Buffer" command.	~45 + short "Start Buffer"
	End-of-Iteration / Terminate (quick ver.)	Routine returned to after a speculative loop iteration completes	Same as slow version, except without RPB manipulation and adjustments during the end of every iteration.	16 (on EOI) or ~45 + short “Start Buffer” (on terminate)

Table 5-13: A summary of the software handlers required to support speculation.

Group	Handler	Use	What it does	Overhead (instr.)
Support	Violation: Initiate	Exception, initiated by a RAW hazard detection in the L1 cache speculation logic	Causes speculative data to be discarded and the iteration to be restarted, and sends out a KILL bus command to following CPUs.	~30 (20 if ina quick loop)
	Violation: Receive	Exception, initiated by KILL bus command	Causes speculative data to be discarded and the iteration to be restarted.	8 + full "Start Buffer" (11 if in quick loop)
	Hold: Buffer Full	Exception, initiated by buffer-management hardware within each CPU, due to a full L2 buffer or the possible loss of read-bit information due to an L1 replacement	Causes the CPU to stop until it becomes the head CPU, when it is allowed to continue non-speculatively, bypassing the full buffer.	15
	Hold: Exception	Exception, initiated in place of a "normal" instruction exception or system call trap encountered during speculation	Causes the CPU to stop until it becomes the head CPU, when it is allowed to continue non-speculatively with the SYSCALL or instruction exception.	25 + OS time

thread list when the loop is started, as is depicted in Figure 5-13 (b). Subsequently, when a loop iteration completes on the head thread, its RPB is recycled to the end of the loop, as the figure indicates. Aside from the RPB recycling and the fact that fewer registers must be saved and restored when starting a loop subroutine, the system works much like it does with procedure forks. Since the active RPB list works the same at all times, this model allows speculative thread forks from within a loop or even a loop within a loop to work correctly. However, a loop within a loop is impractical, even if it works correctly, because enough loop RPBs are always inserted into the active list when *any* loop is started so that all processors will always be working on the innermost loop or subroutines inside of it. Hence, RPBs from outer loop iterations will always be far enough back on the active list that they will never execute until the inner loop completes. As a result of this processor allocation scheme, if nested loops are encountered, we must choose which loop is the best choice for speculative execution. Only a system with a very large number of processors could practically consider dividing up the free processors among several different parallel or nested loops in order to run speculative iterations from more than one at a time. The second set of control code is faster, but less flexible. For loops that do not contain subroutines that need to be forked, this *quick* set of routines allocates a set of four RPBs for the loop, one per processor, and then locks each processor into an RPB, as is depicted in Figure 5-13 (c). The overhead of the control routines associated with these loops is much lower because it does not have to manipulate the active RPB list after every loop iteration to perform RPB recycling or deal with forks or nested loops inside of the loop, because these are simply executed inline. Table 5-14 shows a comparison of the dramatic overhead savings that are possible using this technique. While we have not currently implemented this feature, it would be possible for a compiler to generate code that could first use the slow but sophisticated loop control routines to dynamically measure a loop's contents, including that of any inner loops, and then select the quicker routines for loops that do not need the flexibility of the full loop handler based on its initial measurements.

Table 5-14: A comparison of typical handler overheads for "slow" and "quick" loops.

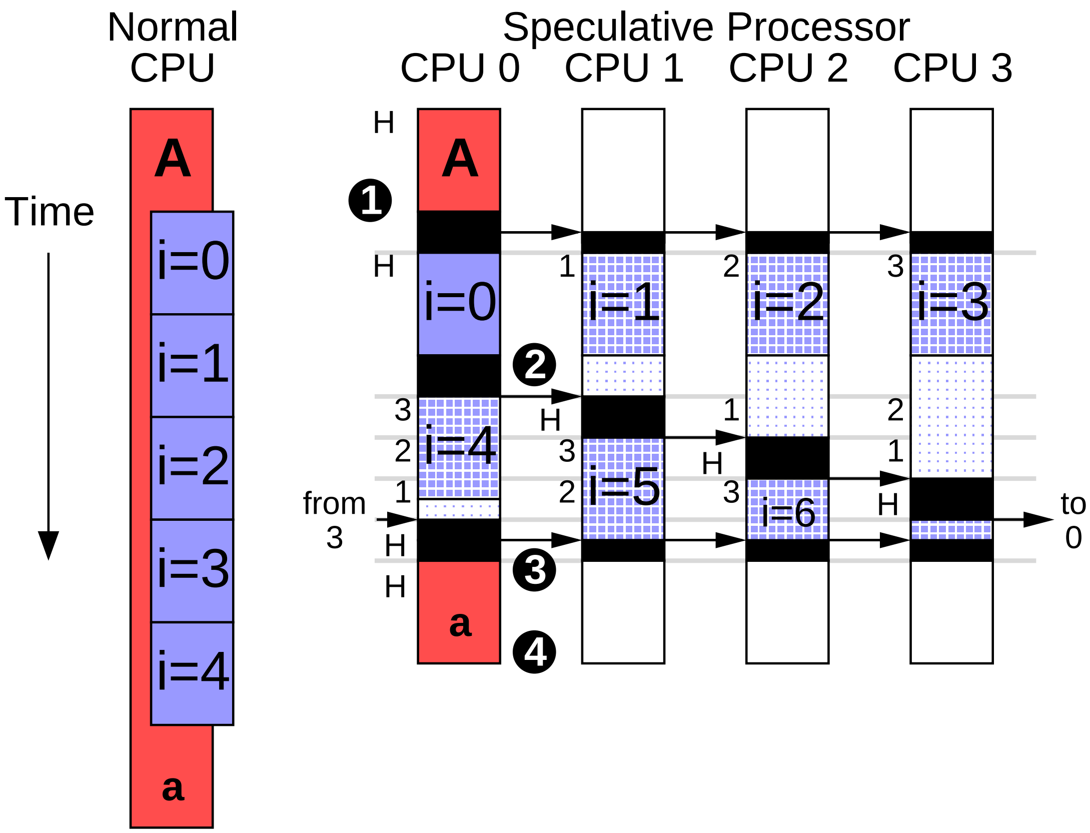
Routine	Use	"Slow" Overhead	"Quick" Overhead
Start Loop	Prepares the system to speculatively execute loop iterations, and then starts execution	~75	~70
End of each loop iteration	Completes the current loop iteration, and then attempts to run the next loop iteration (or speculative procedure thread, if present)	~80	16
Finish Loop	Completes the current loop iteration, and then shuts down processing on the loop	~80	~80
Violation: Local	Handles a RAW violation committed by this processor	~30	20
Violation: Receive from another CPU	Restarts the current speculative thread on this processor, when a less speculative processor requires this	~80	11

If the loop is a good candidate for speculation, a modified version of the loop body, transformed into a self-contained function using the techniques described in Section 6.1, is executed repeatedly. Loop iterations are executed on all available processors. They are distributed among processors so that when loop iteration *i*, running on the head processor, completes, that iteration's results are committed to memory. Afterwards, the processor starts running the next loop iteration that has not yet been allocated to a processor, usually *i+4*, becoming the most speculative processor in the process. Meanwhile, the *i+1* iteration becomes the head iteration and is allowed to repeat the cycle. This pattern continues until one of the loop iterations detects a loop terminating condition, and notifies the speculation system. When this processor becomes the head, all processors executing loop speculation are cancelled and execution returns to normal. A simple example of this execution sequence is depicted in Figure 5-14.

A possible problem with loop speculation is that it may increase the amount of memory traffic and the instruction count during the loop. The speculative version of the loop cannot register allocate variables that are shared across loop iterations, because Hydra's data speculation mechanisms cannot protect against true dependency violations in registers. We chose not to make these modifications because, unlike our L2 memory system, the register files are an integral part of each processor's pipeline, and modifications to allow communication between them would likely decrease each processor's core cycle time.

5.5.3 Loop-only Speculation

By limiting speculation to loops alone, we can completely eliminate the RPB system and thereby dramatically simplify the control routines so they require many fewer instructions. Table 5-15 shows the overhead reductions we were able to obtain by simplifying several key loop-control routines so that they would only work in an environment without procedure speculation. Table 5-16 emphasizes this point further with a side-by-side comparison showing the performance of all three types of loop speculation side-by-side. The savings obtained by simplification of the routine that must be called at the end of each loop iteration and the violation-processing routines have the most impact on overall performance.



1. A loop is started — all processors respond, and start executing loop iter-ations in the order of their current state. The initiating processor also gets an iteration as it completes the A thread.
2. When the head processor (speculative state 0) completes a loop iteration, it notifies the other processors and starts a new loop iteration while its buff-ers are committed to permanent memory. This message causes the spec-ulative states of the looping processors to shuffle. Each is decremented by 1, except the head, which now becomes the most speculative of the loop-ing processors as it starts a new loop iteration.
3. When an iteration completes after detecting the end-of-loop condition, it sends a termination message out to all of the processors. All other proces-sors will now be running iterations "beyond the end of the loop" that shouldn't actually execute, so they are simply cancelled, and are freed to execute any threads after the end of the loop.
4. The terminating processor picks up the a thread, immediately following the loop, and com-pletes it.

Figure 5-14: An example execution pattern over time for a simple speculative loop. This example uses the same notation as the subroutine example in Figure 5-12.

Making this system better yet, the programming API for loop-only speculation can be optimized, since it no longer has to be compatible with the RPB system dictated by procedure speculation. As is described in Section 6.1.3.2, the loop bodies must be encapsulated within a function call so that they can be speculatively executed. To maintain compatibility with procedure speculation, the

Table 5-15: A comparison of "label" and conventional speculative handlers.

	Routine	Use	Procedure and Loops Overhead	Loop-only Overhead
Procedures	Start Procedure	Forks off the code following a procedure call to another processor, speculatively	~70	—
	End Procedure	Completes processing for a procedure that has forked off its completion code, and starts running another speculative task on the CPU	~110	—
Loops	Start Loop	Prepares the system to speculatively execute loop iterations, and then starts execution	~70	~30
	End of each loop iteration	Completes the current loop iteration, and then attempts to run the next loop iteration (or speculative procedure thread, if present)	~80	12
	Finish Loop	Completes the current loop iteration, and then shuts down processing on the loop	~80	~22
Support	Violation: Local	Handles a RAW violation committed by this processor	~25	7
	Violation: Receive from another CPU	Restarts the current speculative thread on this processor, when a less speculative processor requires this	~80	7
	Hold: Buffer Full	Temporarily stops speculative execution if the processor runs out of primary cache control bits or secondary cache buffers	15	12
	Hold: Exception	Pauses the processor following a SYSCALL instruction, until it is the non-speculative, "head" processor	25 + OS time	17 + OS time

loop body encapsulation function for "slow" and "quick" loops must be called on *every* loop iteration, since the encapsulation function holds only a single loop body that is pretending to be a "subroutine" that can be speculatively forked off into a thread. On the other hand, the encapsulation functions for loop-only speculation hold a modified form of the original loop that executes a quarter of the original loop. This allows each processor executing speculation to only call the encapsu-

Table 5-16: Comparison of the most important handlers in all three looping schemes.

Routine	Use	"Slow" Overhead	"Quick" Overhead	"Improved" Overhead
Start Loop	Prepares the system to speculatively execute loop iterations, and then starts execution	~75	~70	~30
End of each loop iteration	Completes the current loop iteration, and then attempts to run the next loop iteration (or speculative procedure thread, if present)	~80	16	12
Finish Loop	Completes the current loop iteration, and then shuts down processing on the loop	~80	~80	~22
Violation: Local	Handles a RAW violation committed by this processor	~30	20	7
Violation: Receive from another CPU	Restarts the current speculative thread on this processor, when a less speculative processor requires this	~80	11	7

lating function once per speculative loop, instead of once per loop iteration — a significant savings.

5.5.4 Thread Size & Runtime Selection

Serious consideration must be given to the size of the threads selected, for several reasons:

- **Limited buffer size:** Since we need to buffer state from a speculative region until it commits, threads need to be short enough to avoid filling up the buffer space allocated for data speculation too often. An occasional full buffer can be handled by simply stalling the thread that is storing too much data until it becomes the head processor, when it may continue to execute while writing directly to the L2 cache. However, if this occurs too often performance will suffer because of the constant thread stalling.

- **True dependencies:** Excessively large threads will exhibit more dependencies with later threads, since they issue more loads and stores. With more true dependencies, more violations and restarts occur.

- **Restart length:** A late restart on a large thread will cause much more work to be discarded, since a checkpoint of the system state is only taken at the beginning of each thread. Having shorter threads results in more frequent checkpoints being made, with more efficient restarts as a result.

- **Overhead:** Very small threads are also inefficient, because of the speculation software handler overhead. Programs that are broken up into large numbers of small threads will waste more time on these overheads.

In order to amortize the speculation software overheads while still keeping threads short enough to avoid the long-thread problems, we have found that threads on the order of 100–1000 instructions are optimal.

Unfortunately, not all loop bodies and subroutines are in this “perfect” size range for most efficient speculation. Also, many of these possible threads have too many true dependencies across loop iterations or with their calling routines to ever effectively achieve speedups during speculative execution. With an infinite number of processing nodes, it is possible to attempt to run every loop iteration and subroutine in parallel. However, many processors would be wasted achieving negligible speedups on the “bad” routines while others sped up usefully. However, we only have a finite number of processing units. As a result, care must be taken to allocate these units among speculative threads that will actually produce helpful speedups.

Under the fastest, “label” loop variant, the loops that will be speculated are completely predetermined by the compiler or programmer, but it is possible to experiment a bit and vary the loops and subroutines chosen for speculation while executing the less efficient but more flexible varieties of speculation. In these modes, simply attempting to speculate on *every* subroutine call and loop will not typically result in selection of “good” threads. These tactics would just tend to clog up the processors with outer loops and the outermost subroutines of nested sets of routines. These usually are not the best choices. As a result, “bad” subroutines or loops must be determined and cancelled as quickly as possible to free up space for any “good” routines that can be found.

There are several heuristics that we use based on various CP2 statistics registers to find and control the cancellation of “bad” threads: the violation counters, to eliminate threads with many dependencies, thread timers, to eliminate threads that are too short or long, and stall timers, to find threads that are stalled too long waiting to become the “head” processor. Once “bad” threads are discovered, we note their poor performance using a mechanism similar to a branch prediction table. This table is consulted before the forking of each speculative thread to determine whether or not the thread is a poor target for speculative execution. If the thread has been marked as being poor, we back off and run it sequentially, instead. We currently maintain a “hardware” prediction table in our simulator, but in a real implementation it would be necessary to implement this table as a software table accessed by instructions within the handling routines. This table is currently an infinite buffer that records loops based on the address of a “check loop” call made before each loop and subroutines based on the address of the called subroutine.

5.5.5 Handling Speculative Violations

The speculation software handlers are also responsible for handling the inevitable case of a speculation that goes wrong — a violation. A graphical example of one of these events is depicted in Figure 5-15. In the example, the head processor writes out some data after the first speculative processor has read the old data value from that address too early, causing a violation. The “Violation: Initiate” handler listed on Table 5-13 is invoked on the first speculative processor after the read bits in its L1 cache notice that the data written by the head processor has unfortunately already been read. In this case, the speculative exception is internally triggered through the process described in Section 5.4.3.

After cleaning up and restarting the violated thread, the handler also has to restart all more speculative threads by sending out a KILL write bus message using the protocols defined in Section 5.4.2 and Section 5.4.3. This is required because the threads *may* have performed computation based on incorrect results already produced by the violating thread, propagating the effects of the violation further. While this is a very conservative mode of operation, tracking the passage of all data through all speculative threads is just too complex a task to realistically consider. The write bus message is passed through the coprocessors of the two most speculative processors, and the “Violation: Receive” handler is invoked. At this point, the speculative threads on these speculative processors are restarted and the system is ready to try executing the threads again, but with the correct data being read by the speculative processors.

5.5.6 Support for Precise Exceptions

If a speculative thread requires operating system services through a system call or an exception, the thread is simply stalled until it becomes the head processor. At that time, the operating system, which is not compatible with speculative execution, may be safely entered. If a thread violates or is cancelled while waiting, the operating system call or exception is simply discarded. This is a particular concern because it is quite easy for a speculative loop to start executing extra iteration or two before they are cancelled, and these iterations often will try to access nonexistent array entries after the actual end of the arrays being accessed.

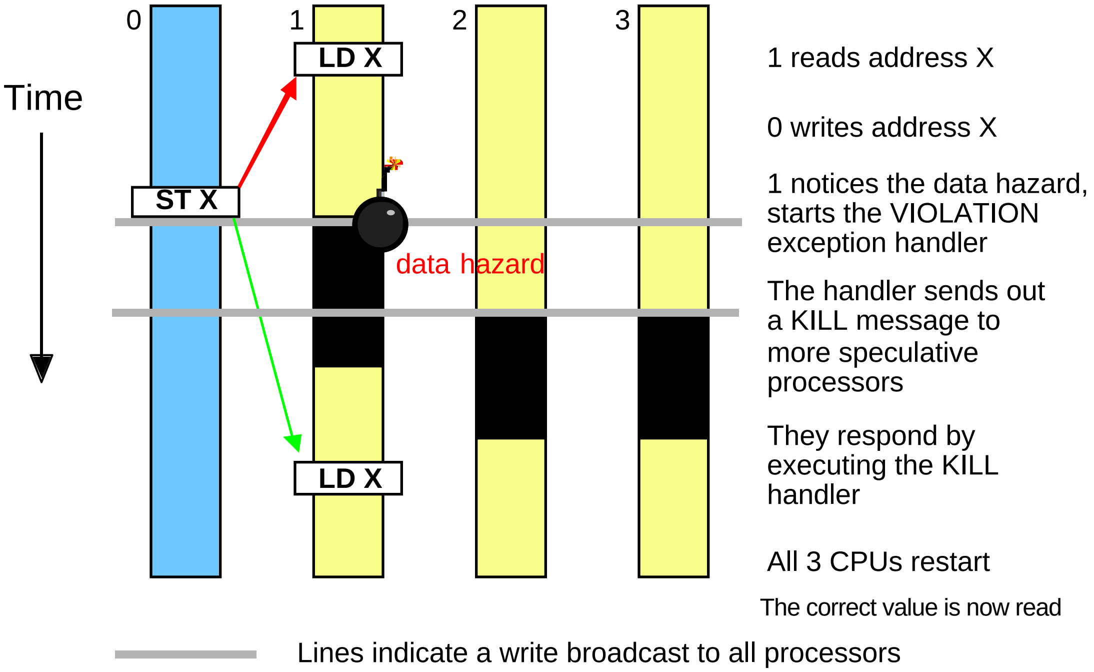

0
1
2
3
Time
LD X
ST X
data hazard
LD X
1 reads address X
0 writes address X
1 notices the data hazard, starts the VIOLATION exception handler
The handler sends out a KILL message to more speculative processors
They respond by executing the KILL handler
All 3 CPUs restart
The correct value is now read
Lines indicate a write broadcast to all processors

Figure 5-15: The sequence of events during a typical speculative violation

5.6 Design Analysis

Taken as a whole, we feel that our implementation of thread-level speculation offers a good tradeoff between capability and hardware implementation costs.

We are unable to speculate on extremely small threads (less than 50-80 instructions long) without incurring so much overhead that we cause slowdowns. An architecture with tight coupling between the various processor cores such as the Multiscalar [13] should be able to get higher performance on code that breaks up more efficiently into smaller threads, but only at the expense of some form of high-speed logic connecting the various cores together. In contrast, the Hydra speculation architecture requires very little adjustment to the design of the processor cores — just the addition of a few bits to the primary caches and the simple CP2 hardware — and no high-speed connections other than the existing bus structure. Even with this significantly reduced hardware support, we are still capable of successfully running some rather small speculative threads, but not the smallest ones. We feel that this is a reasonable limitation to accept.

Another tradeoff in the design that we made has to do with scalability. While some recent speculative systems have been designed to be scalable to large numbers of processors [38] [40], we have chosen to focus on relatively small-scale systems that only scale to 4-8 processors, maximum. In our design, the main limiting factor comes from the fact that our speculation algorithm requires writethrough primary caches and a shared, broadcast write bus between the various processors. In other words, we take advantage of the basic Hydra design for fault detection and with the structure of the grouped speculation write buffers attached to the L2 cache, as described in Section 5.3.3.2. The algorithm for synthesizing "current" cache blocks from the contents of several per-thread speculative buffers is also a problem in scalable systems, because it requires that all of the buffers for all of the threads be present at one physical location. On a large-scale system, structures like these would be difficult or impossible to implement, and there would be a strong impetus to build writeback caches to avoid sending extraneous writes everywhere over a network. Given that Hydra is a fairly small-scale MP to begin with, however, it is generally better for us to stick with a simpler protocol in order to increase performance and decrease the potential for protocol errors. This also allowed us to use most of the existing Hydra design with little or no modification. Since many of the programs we were targeting were integer ones that would not have scaled up to higher numbers of processors anyway, this design choice was a good one for us.

We found that software control, described in Section 5.5, had some definite advantages and disadvantages. The primary advantage was that it allowed the speculation coprocessor hardware (CP2, Section 5.4) to be very limited, since the hardware did not need to have any "brains." It just needs a few registers, some connections to the outside world for interrupt control, and an a coprocessor interface to the rest of the processor core. Software also allows the easy development and modification of speculation thread control protocols over time. We were able to implement all of the different speculation techniques (3 loop speeds plus procedures) on essentially the same hardware, just by changing the control routines. Unfortunately, the cost of software control is the inability of our system to speed up threads that require complex control, most notably subroutine forking. In general, losing this ability is probably a reasonable constraint to avoid the complexity of building all of these control routines into the hardware itself. However, a potential middle ground might be achieved between these two extremes by adding a few extra hardware features such as hardware support for a small number of key tasks. In particular, fast register saving / restoring and a hardware locking mechanism that connected the processors with a specialized path other than the write bus could be helpful. However, with any additional processor interconnections, one would have to worry about synchronizing changes that might be made over the new bus with changes made over the existing write bus. This tricky problem is a large part of the reason we decided to let all communication go through the write bus, where any communication is inherently ordered with respect to all speculative memory references.

Finally, the speculative cache array hardware incorporates very elegantly into the existing cache design, but it has a couple of potential drawbacks. The extra L1 bits from Section 5.3.1 need to be gang-clearable and have extra logic functions to selectively clear the “valid” bits in each line. As a custom block, this should be no problem to design. However, if one was to attempt to synthesize Hydra from an existing selection of standard library parts, no off-the-shelf memory would probably be able to support these features. The design of the L2 cache interface and the speculation write buffers in Section 5.3.3 is very good, with one glaring exception. The locking bits that can “lock down” parts of a cache set were a bug fix to handle a rare case. In the original design, it was possible for a speculative thread to complete, start to flush its buffers, and then discover that a buffered line doesn’t exist in the cache itself because it has been forced out of the L2 cache. Unfortunately, they are an ugly fix. The best solution would be to have a “pseudo-processor” at the L2 cache that is able to re-fetch the line from memory itself. However, redesigning the controllers and arbiter to handle an extra state machine controller was deemed too difficult so late in the design process, so we are stuck with a stitched-together solution. Thankfully, hacks like the locking bits were not common in the design.

6 Programming with Speculation

Loop speculation may be used to split virtually any sequential program with loops of any kind into arbitrary threads that can be parallelized across several processors. However, before the system can successfully exploit loops it must know where they occur in the code. It is actually possible to find loops as existing code executes, simply by looking for backward branches. Wherever one is encountered, the hardware may assume that a loop has been found and begin forking off speculative threads with the program counter set to the target of the backward branch in all speculative threads. This approach has actually been evaluated in [10]. However, finding loops "on the fly" like this with legacy code requires the hardware to speculatively track references to registers as well as memory. Because we judged this high-bandwidth sharing to be too expensive an addition to Hydra, it was necessary to add a compiler pass to isolate the loops in such a way that all communication between speculative threads would be forced through memory.

This chapter describes the API used to isolate and mark loops for speculative execution. The structures required by the API may be generated manually to modify loops for speculative execution, or may be automatically generated by our HydraCAT source-to-source compiler tool. Later parts of the chapter describe ways that we have enhanced the parallel execution of speculative programs using further modifications of the loop source code. Some of these are simply involve slight modifications of the existing loops, while others utilize enhancements to the API designed specifically to aid in performance tuning.

At the end of the chapter, Section 6.3 briefly addresses the software interface to subroutine speculation. Unlike the loop interface, this interface adds nothing new to C code, as subroutines are already fairly self-contained and well marked in C code and the binaries generated from them.

6.1 Simple Loop Isolation

This section describes how a programmer would modify loops for use with speculation. Two versions are provided, for use with loops + subroutines and loop-only speculation. Modifications to force registers out to memory are essentially identical between the two specifications, but the method used to mark loops has been significantly optimized in the loop-only speculation version of the system. With HydraCAT, all modifications but the global ones described in Section 6.1.1 will be done automatically to C `for` or `while` loops with a “p” typed in front of them, making them `pfor` or `pwhile` loops.

6.1.1 Global Additions

The simplest changes are just text additions or replacements necessary to modify the program for use with LESS and to tell LESS that speculation is being used. The list of modifications is fairly short:

1. The name of the `main` function must be changed to `mainX`, to allow it to be used with LESS.

2. Calls to `spec_startup();` and `spec_cleanup();` must be added to enable speculation. These should be located where they will be called before and after *all* speculative loops have executed in the program (the beginning and end of `mainX` is a good spot, for example). During loops+subroutine speculation, these indicate where subroutine speculation will start and end.

3. The following speculation header information must be added to source files that call speculative handlers:

```
extern bool sInSpeculation;
extern void *sSpecDataPointer;
extern void *sSpecFuncPointer;
```

```
void spec_startup();
void spec_cleanup();
void spec_begin(void *loopbody, void *nonlocals);
void spec_begin_label(void *spec_start, void *spec_end, void
   *spec_last_end);
void spec_eoi_label();
void spec_terminate_label();
typedef unsigned int Reg;
int spec_checklock(Reg iteration, Reg *lock, int count);
```

These adjustments are all fairly automatic and straightforward, and require no special knowledge of the code within the application.

6.1.2 Per-Loop Declarations

Next, several customized declarations must be made to encapsulate the variables used by a loop and provide a location where they can be forced into memory so that all speculative processors can access them. While any naming convention may be used, the following bits of example code use the word NAME with various single-letter suffixes to generate a family of unique names for the various structures and functions being declared. Here's the list of declarations that then need to be made:

1) Declaration of a struct containing globalized variables from the loop. Section 6.1.4 describes how to define globalized variables in more detail:

```
typedef struct NAMES
{
   << globalized variables, as defined in Section 6.1.4.1 >>
}
NAMET;
```

2) Declaration of global `NAMET *NAMEV;` at the beginning of the file where the loop is located.

3) Declaration of the encapsulating function in the form `void NAME(NAMET *nonlocals);` (loop-only speculation) or `int NAME(NAMET *nonlocals);` (for loops + subroutines speculation).

6.1.3 Marking the Speculative Loops

Rewriting the loops for speculation consists of two parts: rewriting the loop code in the middle of the loop and adding an encapsulated version of the loop code in at the end of the loop. This added code should be in the same file, so that it shares globals, declarations, and so on with the original function.

6.1.3.1 Modifications to the Loop Itself

The loop itself needs to be replaced with a fairly straightforward block of code. The additions to the code for the loop-only version of speculation are as follows:

```
if (!sInSpeculation)
{
      /* Load non-local values struct */
      << variable packing, as defined in Section 6.1.4 >>

      /* Execute the speculative loop */
      sSpecDataPointer = (void*)NAMEV;
      sInSpeculation = 1;
      sSpecFuncPointer = &NAME;
      NAME(NAMEV);
      sInSpeculation = 0;

      /* Unload the modified non-local values */
      << variable unpacking, as defined in Section 6.1.4 >>
}
else
```

```
{
     << existing loop body >>
}
```

For the loops + subroutines form of speculation, the central part of this code needs to be modified slightly:

```
     /* Execute the speculative loop */
     sInSpeculation = 1;
     spec_begin(NAME, NAMEV);
     sInSpeculation = 0;
```

With these blocks of code, the speculative version of the loop is launched if possible. Otherwise, the original uniprocessor code in the `<< existing loop body >>` clause of the `if` statement is executed normally. This is generally used to prevent re-entering of loop speculation on the inner nest of a nested loop.

6.1.3.2 The Encapsulation Functions

The "encapsulation" functions that contain the loop bodies for speculation should be placed somewhere in the same file as the original routine. The format of the functions in both the loop-only and loops + subroutines of the simulator is fairly similar, but the way the functions work is radically different, and are therefore described separately.

The loop-only encapsulation function runs a quarter of the loop being parallelized. This function is called by all four processors simultaneously so that each will execute its assigned quarter. It contains a specially modified version of the loop that calls low-overhead speculation functions on every loop iteration to notify the hardware what is happening during that iteration. When the loop completes, the function completes on all four processors and execution resumes following the loop solely on the main processor.

```
void NAME(NAMET *nonlocals)
{
     /* Purely local variables, copied from original program */
     << declare purely local variables >>

     /* Local copy of localizable variables */
     << declare local copies of localizable variables >>

     /* Localize variables */
     << perform localization >>

     spec_begin_label(&&spec_start,&&spec_terminate);
     while (1)
     {
        spec_eoi_label();
spec_start:

        /* WHILE/FOR loop termination transformation */
        << optional FOR localization >>
        if(!( << WHILE/FOR end test >> ))
           spec_terminate_label();

        /* Existing loop body */
        << rest of loop >>

        /* DO-WHILE loop termination transformation */
        if(!( << DO-WHILE end test >> ))
           spec_terminate_label();
     }
spec_terminate_last:
     << any unnecessary packing, but probably nothing >>
spec_terminate:
}
```

The loops+subroutine encapsulation function only runs a single iteration of the loop body. The low-level speculation routines automatically call it multiple times on the different processors as the loop executes. This incurs the overhead of a subroutine call during every loop iteration, but makes the loop iterations speculative-thread compatible with subroutines so that the same low-level routines can handle both interchangeably. Within the routine, however, it is very similar to the loop-only version. The primary difference is in how the low-level routines are notified about loop termination. Instead of calling termination or end-of-iteration routines directly, the return value indicates continuation. When the loop should continue, the encapsulation function returns a value of 0, while when it should terminate, it returns a value of 1.

```
void NAME(NAMET *nonlocals)
{
     /* Purely local variables, copied from original program */
     << declare purely local variables >>

     /* Local copy of localizable variables */
     << declare local copies of localizable variables >>

     /* Localize variables */
     << perform localization >>

     /* WHILE/FOR loop termination transformation */
     << optional FOR localization >>
     if(!( << WHILE/FOR end test >> )) return 1;

     /* Existing loop body */
     << rest of loop >>

     /* DO-WHILE loop termination transformation */
     if(!( << DO-WHILE end test >> )) return 1;

     return 0;
}
```

6.1.3.3 The Loop Transformations

`For`, `while`, and `do-while` loops can all be transformed into speculative versions. Each loop must be stripped off of its body before it may be placed into the encapsulating functions from the previous section. The "test" clause of the loop is preserved in the spots noted in the sample code there. `For` and `while` tests are done at the beginning of the loop body and `do-while` tests are done at the end, just as in the original loop.

The "test" clause may just be copied right out of the existing loop and put into position right where the template indicates. These clauses control an `if` statement that triggers a `spec_terminate();` call or `return 1;` which causes the loop to end. Since the clauses normally return TRUE when we want the loop to continue, the result must be inverted with a `!` before it is actually used.

Transformation and movement of the "initialization" and "increment" clauses of `for` loops are discussed below with the other variable manipulations.

In addition, `break` and `continue` statements need to be modified to work correctly in the speculative loops. `Continue` statements become `spec_eoi_label();` with loop-only speculation or `return 0;` statements with loops + subroutines. `Break` statements become `spec_terminate_label();` under loop-only speculation while they become `return 1;` statements with both loops and subroutines active.

Finally, some further adjustments must be made to handle `return` statements in the middle of speculative loops. See Section 6.1.5 for more information on this.

6.1.4 The Variables

The most complex adjustment is the moving of all shared variables into a globalized structure in memory that can be shared by all speculative threads. Essentially, all stack-allocated variables within a function used by the loop are put into a shared `struct` out on the heap or a global variable. Then, instead of using a stack-based version, all four processors use the same heap-based or global copy. This section summarizes how to process a single loop.

6.1.4.1 Classification

The first step to sorting variables is to list all of the variables used within the loop and then sort them out by how they are declared and/or used. The following five classification categories are useful for deciding the proper way to process any particular variable:

1) **Global variables:** Ignore these. They will work fine without modification, since they are already global and accessible by all processors. Any changes to globals will seen by everyone, while read-only globals will be cached by each processor and processed fairly efficiently.
2) **Loop-carried stack variables:** Stack-based variables that may be modified within the loop are the trickiest ones to handle. Every read or write must be made to or from the heap-based `struct`, so that the speculation hardware will be able to detect any violations caused by illegal changes of the variables.
3) **Localizable stack variables:** Stack-based variables that are initialized before the loop, but not modified within the loop, may be localized within each processor. Since these variables are never written, value changes do not need to be propagated through the heap stack. While these may just be treated as #2 variables, for performance purposes it may be better to have local copies of the variables initialized from the global value when the encapsulated loop function is started.
4) **Pure local variables:** Stack-based variables that are *always* initialized within each loop iteration before being used, and that are *never* read again in the code following the loop may be

localized completely within the loop encapsulation function. For example, loop iteration counters for smaller loops nested within the speculative loop are typically initialized by the smaller loop’s `for` statement, and ignored after that loop terminates. While these variables may be globalized as #2 variables, performance can be significantly enhanced by leaving them localized within each of the processors. By privatizing these variables, false-sharing conflicts can be minimized, and code can be made more efficient, since these variables may be register-allocated.

5) **FOR loop iteration counters:** If the speculative loop is a `for` loop, then it is in a unique category. The variable is treated like a hybrid between a #2 and a #4, but is handled in a somewhat different manner.

6.1.4.2 Structure declarations

Once the variables used in the loop have been classified, all #2, #3, and #5 variables used by the function must be constructed must be inserted in the `struct` defined in Section 6.1.2. Here is the basic pattern:

```
typedef struct NAMES
{
      /* Loop-carried variable */
      long int ent; /* type code_int */  <<< EXAMPLE of #2

      /* Existing, precomputed variables that can be localized */
      int hshift;                        <<< EXAMPLE of #3

      /* Loop iteration variable */
      int i;                             <<< EXAMPLE of #5
}
NAMET;
```

This is basically just a matter of writing out all of the variables into the `struct` list. Note that variables from categories #1 and #4 do not need to be in the `struct` at all.

6.1.4.3 Loading & unloading the struct in the original function

Third, the program needs to "load" and "unload" the structure before speculatively executing the loop. The correct locations for this code are noted in Section 6.1.3.1's example. The following code shows the loading and unloading of the previous `struct`:

```
/* Loading */
NAMEV.ent = ent;               <<< EXAMPLE of #2
NAMEV.hshift = hshift;         <<< EXAMPLE of #3
NAMEV.i = 0;                   <<< EXAMPLE of #5
```

Variables from categories #2 and #3 are simply copied into the `struct`, item by item, moving the variables from the stack of the original function to the `struct` for the encapsulated loop. The `for` loop variable is initialized here during the loading stage, using the initialization clause from the `for` loop itself (in this case, `for (i=0; . . .`).

```
/* Unloading */
ent = NAMEV.ent;               <<< EXAMPLE of #2
i = NAMEV.i;                   <<< EXAMPLE of #5 (maybe!)
```

Only variables from category #2 normally need to be "unloaded" back into their original stack locations. Variables from category #3 will still be correct, since they were not written during the loop. `For` loop variables are rarely used after a loop, but for safety one should write them back. As a further optimization, some or all of the unloading may be eliminated if the translator is smart enough to hunt through the rest of the original routine and see which variables are actually read and used by code, later part of the routine. Any variables that are not used again do not need to be unloaded.

6.1.4.4 Setting up at the beginning of the encapsulated loop

Similar `struct` unpacking code must also be put in the function that encapsulates the loop itself. All #3 type variables need to be localized, and the local variables (type #4) need to be declared. #5 type variables are declared, too, so that `for` loop counters may be localized. Using the locations noted in Section 6.1.3.2, the following code continues the previous example:

```
/* Purely local variables, copied from original program */
int j;                             <<< EXAMPLE #4

/* Local copy of localizable variables */
int hshift;                        <<< EXAMPLE #3
int i;                             <<< EXAMPLE #5

/* Localize variables */
hshift = nonlocals->hshift;        <<< EXAMPLE #3, continued
```

6.1.4.5 Changing the loop body

There are two variable-related modifications that may need to be made to the loop body. The first, and most straightforward, involves transforming all of the #2 type variables. They must be replaced using the following transformation:

`X` becomes `(nonlocals->X)` — where `X` is the old variable name.

The `for` loop variable optimization is slightly more complex. Just before the `for` loop escape test, the “optional `for` localization” is noted in the Section 6.1.3.2 examples. This optimization attempts to make `for` iteration counters act sort of like #3 type variables. Just before the escape test, the following code should be added:

```
X = nonlocals->X;
```

```
<< FOR increment clause, with X in the clause becoming (nonlocals-
   >X) >>;
```

Or, for the common `++` and `--` increment clauses, the following one-line transformations will work, too:

`X = (nonlocals->X)++;` or `X = (nonlocals->X)--;`

All of these methods establish the local `X` as the value `X` should have during this iteration, while immediately incrementing the global value in the `struct` to the proper count for the *next* iteration, which may then pick it up from memory when it is ready to start that iteration.

The rare `for` loops that write to their iteration counting variable within the loop itself cannot use this optimization, and as a result may be rendered unfit for speculation. Throughout the loop body, these variables should just be treated like #2-type ones, turned into `struct` references. Then, at the very end of the loop body (the `do-while` decision point in the Section 6.1.3.2 examples), the increment clause of the `for` loop must be applied — so `i++` would become `(nonlocals->i)++;` as the last line of the loop. This late write will tend to completely serialize loop operation, however, so it should be avoided if possible.

6.1.5 Return-in-the-Loop Transformation

One other case can cause serious problems for loop speculation, in either version: `return`s that occur in the middle of the loops. They are a real problem because they not only terminate the loop, but also the function block that surrounds the loop. If left alone, they will either permanently leave the system "speculating" (with loops + subroutines) or just cause the current loop iteration to complete erratically (with loop-only speculation, if a simple `break` replacement is used). This section describes the additions necessary to handle them, under both versions of speculation.

6.1.5.1 Modifications to the variable structure

The following two entries need to be included at the end of the `struct` defined for any loop with nested `return`s:

```
     int __returnValueUsed;
     <return type of original function> __returnValue;
```

For a `void` function, the `__returnValue` variable is obviously unnecessary. All references to it in the following sections may be omitted, also.

6.1.5.2 Modifications to the original function, around the loop

The program must initialize and process the new variables, which calls for the addition of a couple of lines just before and after the existing `spec_begin` calling code or loop subroutine caller:

```
     __returnValueUsed = 0;
     .
     .  load other variables
     .
     Execute the speculative loop
     .
     .  unload variables
     .
     if (__returnValueUsed) return __returnValue;
```

6.1.5.3 Modifications to the encapsulated loop:

Finally, wherever a `return`-within-the-loop occurs, it must be replaced with the following block of code:

```
{
     nonlocals.__returnValueUsed = 1;
```

```
        nonlocals.__returnValue = <expression from original return>;
        spec_terminate_label();
}
```

The entire middle line may be eliminated for functions that return `void`.

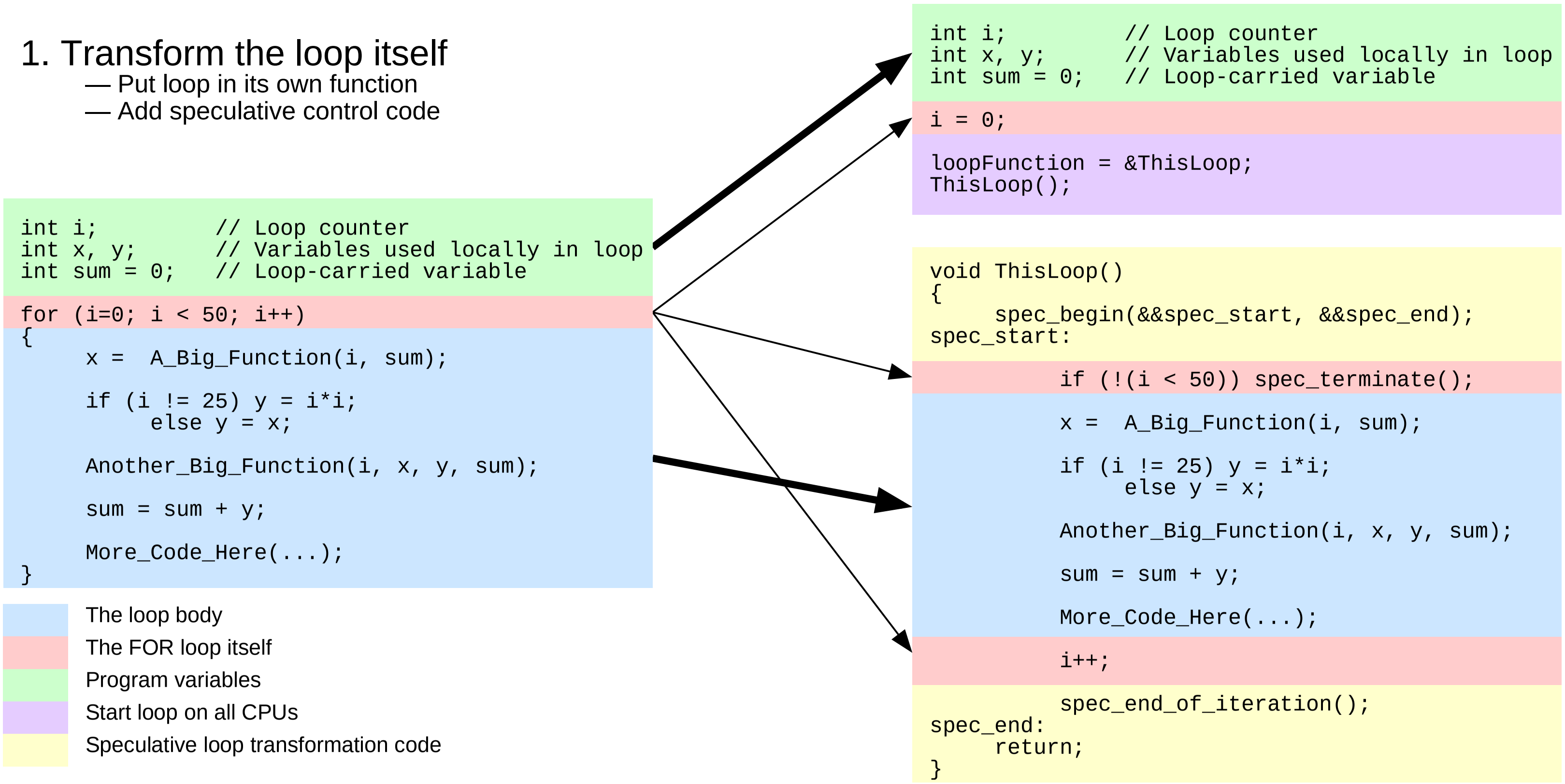
6.1.6 An Example Transformation

To illustrate how all of these optimizations look on an entire loop, the following example shows how loop-only parallelization converts a rather complex loop into its loop-only speculation speculative form. Loops + subroutines speculation results in a similar, but slightly simpler output. For space reasons, the `sInSpeculation` clause that allows the unmodified loop to execute in case of a loop nest has been omitted. The extra `while` in the encapsulated loop has also been omitted, since it is only necessary to keep the SGI compilers working properly with the modified code.

The first stage of the conversion, transforming and encapsulating the loop, is illustrated in Figure 6-1. This example demonstrates several key features of the transformation process:

1) The loop itself is replaced with the loop-only starting template that invokes the loop encapsulation function on the main processor after signaling the other processors to start it as well.
2) The loop body is encapsulated within the new `ThisLoop` encapsulation function. In the process, the necessary `spec_` marking lines are added. The names of these have been simplified a bit to allow for easier reading.
3) The three clauses of the `for` loop are distributed appropriately.

Once the basic loop transformations are complete, the loop's variables need to be globalized and their use should be optimized. This second stage of the transformation process is illustrated in Figure 6-2 and described below.



1. Transform the loop itself

— Put loop in its own function

— Add speculative control code

```
int i;          // Loop counter
int x, y;       // Variables used locally in loop
int sum = 0;    // Loop-carried variable

for (i=0; i < 50; i++)
{
     x =  A_Big_Function(i, sum);

     if (i != 25) y = i*i;
          else y = x;

     Another_Big_Function(i, x, y, sum);

     sum = sum + y;

     More_Code_Here(...);
}
```

🟦 The loop body

🟥 The FOR loop itself

🟩 Program variables

🟪 Start loop on all CPUs

🟨 Speculative loop transformation code

```
int i;          // Loop counter
int x, y;       // Variables used locally in loop
int sum = 0;    // Loop-carried variable

i = 0;

loopFunction = &ThisLoop;
ThisLoop();
```

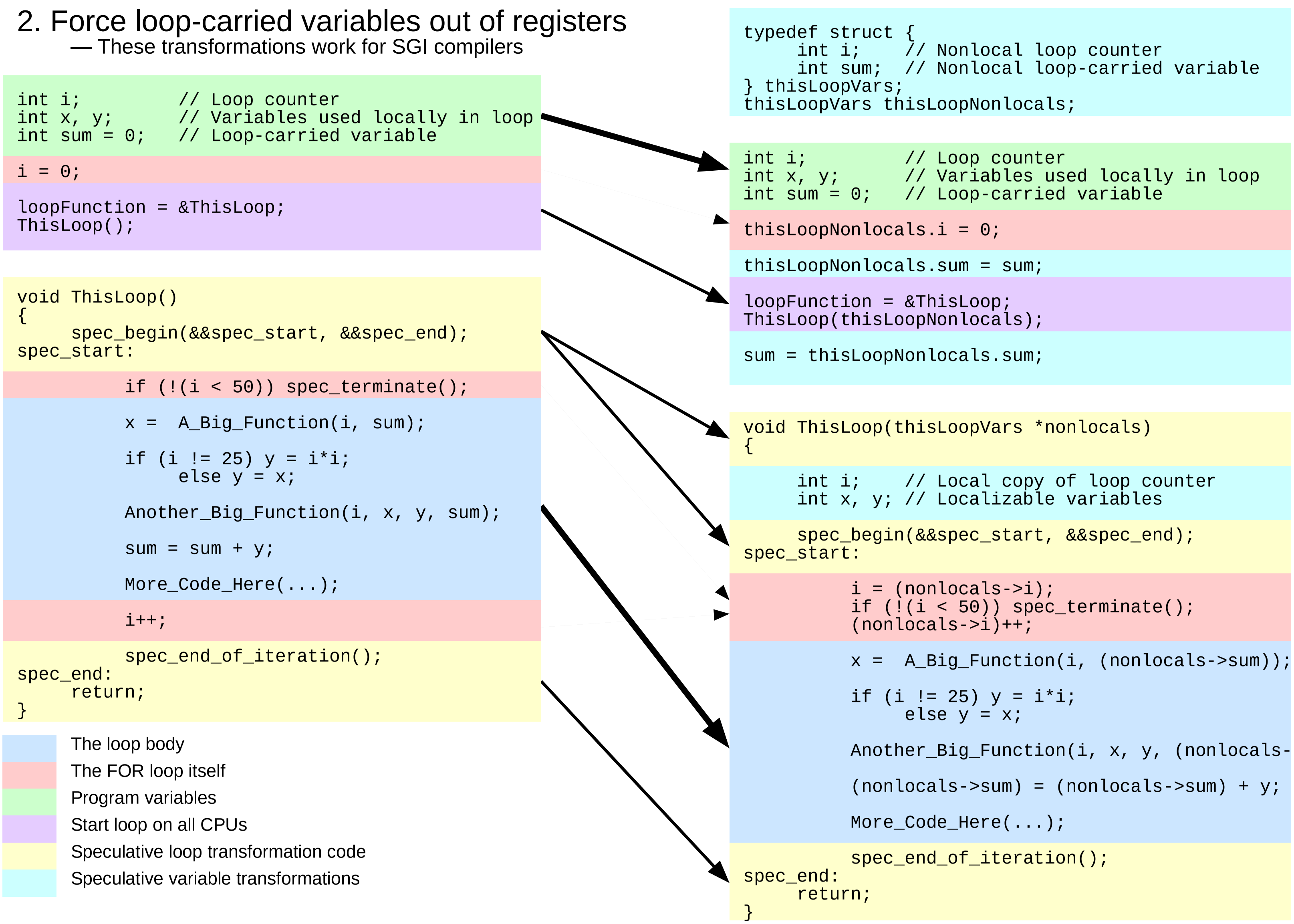
```
void ThisLoop()
{
     spec_begin(&&spec_start, &&spec_end);
spec_start:
          if (!(i < 50)) spec_terminate();

          x =  A_Big_Function(i, sum);

          if (i != 25) y = i*i;
               else y = x;

          Another_Big_Function(i, x, y, sum);

          sum = sum + y;

          More_Code_Here(...);

          i++;

          spec_end_of_iteration();
spec_end:
     return;
}
```

Figure 6-1: An example of speculation loop transformations.

4) A `struct` containing all of the variables that need globalization is generated and placed near the beginning of the C source file containing the loop.
5) Variables are packed and unpacked before starting the speculative loop and after returning from it. A pointer to the `struct` is also added to the encapsulated loop function call so that it may be accessed within the loop.
6) Local copies of the `for` loop counter and all localizable variables are declared within the encapsulation function.
7) The `for` loop counter increment is moved to the *beginning* of the encapsulated loop body to avoid creating an unnecessary dependence between the end of one speculative iteration and the beginning of the next one. This will allow the loop to be much more parallel.
8) All references to globalized variables are modified to use the `struct` containing them.

While this example was fairly simple, the techniques demonstrated scale up in a straightforward fashion to much larger loops which may modify many different variables of different types.

When run, the transformed loop will execute in parallel using speculation. After all four processors start executing the encapsulating function, the `spec_begin` call starts the actual speculation control handlers and sets up the initial sequencing of threads, with the four processors in Hydra each being allocated one of the first four iterations of the loop. The loop body executes normally on each processor except for additional loads and stores to loop-carried variables that were forced out of registers by the addition of `nonlocals` dereferencing. At the end of each loop iteration, the `spec_endofiteration` call triggers a handler that waits to make sure it is running on the head processor. When the thread is safely nonspeculative, the handler commits the contents of its L2 cache buffer with a write bus command (see Section 5.3.3.4), triggers the clearing of its primary cache speculation bits with a CP2 command register write (see Section 5.4.1), invalidating any “pre-invalidated” lines in the process, and then jumps back into the user code to start another loop iteration as the most speculative processor. Meanwhile, violations may be detected by the



2. Force loop-carried variables out of registers
— These transformations work for SGI compilers

```
int i;          // Loop counter
int x, y;       // Variables used locally in loop
int sum = 0;    // Loop-carried variable
i = 0;
loopFunction = &ThisLoop;
ThisLoop();

void ThisLoop()
{
     spec_begin(&&spec_start, &&spec_end);
spec_start:
          if (!(i < 50)) spec_terminate();
          x =  A_Big_Function(i, sum);
          if (i != 25) y = i*i;
               else y = x;
          Another_Big_Function(i, x, y, sum);
          sum = sum + y;
          More_Code_Here(...);
          i++;
          spec_end_of_iteration();
spec_end:
     return;
}
```

```
typedef struct {
     int i;    // Nonlocal loop counter
     int sum;  // Nonlocal loop-carried variable
} thisLoopVars;
thisLoopVars thisLoopNonlocals;

int i;          // Loop counter
int x, y;       // Variables used locally in loop
int sum = 0;    // Loop-carried variable
thisLoopNonlocals.i = 0;
thisLoopNonlocals.sum = sum;
loopFunction = &ThisLoop;
ThisLoop(thisLoopNonlocals);
sum = thisLoopNonlocals.sum;

void ThisLoop(thisLoopVars *nonlocals)
{
     int i;    // Local copy of loop counter
     int x, y; // Localizable variables
     spec_begin(&&spec_start, &&spec_end);
spec_start:
          i = (nonlocals->i);
          if (!(i < 50)) spec_terminate();
          (nonlocals->i)++;
          x =  A_Big_Function(i, (nonlocals->sum));
          if (i != 25) y = i*i;
               else y = x;
          Another_Big_Function(i, x, y, (nonlocals-
          (nonlocals->sum) = (nonlocals->sum) + y;
          More_Code_Here(...);
          spec_end_of_iteration();
spec_end:
     return;
}
```

🟦 The loop body
🟥 The FOR loop itself
🟩 Program variables
🟪 Start loop on all CPUs
🟨 Speculative loop transformation code
▢ Speculative variable transformations

Figure 6-2: An example of speculation variable transformations.

read bits in speculative processors and cause interrupts that vector the processor to the `spec_violation` handler. This handler flushes the contents of the secondary cache buffer, clears primary cache speculation bits and invalidates any “modified” lines, sends a message to “later” processors telling them to restart too (using a write bus command), and finishes by restarting the loop body. Finally, when the loop completes, the `spec_terminate` call is triggered by the loop body instead of `spec_endofiteration`. This call acts just like `spec_endofiteration`, except that when it is complete it jumps to the fall-through code after the loop and sends a message to the other processors (again, using a write bus command) telling them to shut down using the `spec_killandhalt` handler.

6.2 Optimizing the Basic Loops

Our simulator produces output that explicitly identifies loads and stores in the application that cause violations. In a real system, similar feedback could be obtained first by adding memory to each processor that records the address of each executed speculative load instruction, next by broadcasting store instruction program counters along with data on the write bus, and finally by interpreting the results from these hardware structures using instrumentation code built into the speculation software routines. This code would only be needed at profiling time, so normally the overhead imposed by such code could be eliminated. We developed a tool to translate the load and store addresses produced by our simulator to source code locations, for easier analysis. Armed with this information, we attempted to reduce the number of violations caused by the most critical dependencies using several different techniques.

While we used the feedback information from the violation statistics to make manual modifications to our benchmarks, many of these modifications could be performed using a feedback-directed compiler targeted to a speculatively parallel architecture. In fact, one of the key features of a speculatively parallel architecture is that it allows programmers — or compilers — to execute their programs in parallel quickly, without a full static analysis of data dependencies that may exist between threads in the program, while still maintaining proper program execution. As a compiler is normally required to do both a fully static analysis of dependencies and to produce correctly executing object code, this can greatly aid the design of a compilation system. Also, once code has been compiled, the feedback hardware can automatically determine the amount of parallelism present in the program and find potential trouble spots where true data dependencies frequently occur simply by executing the code and noting when dependencies exist. Armed with this information, it is possible for programmers or compilers to concentrate their efforts on just the most critical portions of programs being parallelized.

6.2.1 Explicit Synchronization

The first technique we used to minimize effects from these critical data dependencies was to add explicit synchronization, causing any dependent threads to stall instead of causing a dependency violation. This was achieved simply by adding a way to issue a non-speculative load instruction even while the processor is executing speculatively. As depicted in Figure 6-3, this special load may be used to test lock variables that protect the critical regions of code in which pairs of loads and stores exist that cause frequent dependency violations. Before entering a critical region, synchronizing code spins on the lock variable until the lock is released by another processor. Because the special load is non-speculative, a violation does not occur when the lock is released by a store from another processor. Once the lock is freed, the speculative processor may perform the load at the beginning of the critical region. Finally, when a processor has performed the store at the end of the region, it updates the lock so that the next processor may enter the critical region. This process eliminates all restarts caused by dependent load-store pairs in the critical region, at the expense of forcing the speculative processors to serialize through the critical regions, eliminating any possibility of finding parallelism there. As a result, this technique should only be used for small critical regions which are the source of very frequent violations. The lock handling code also adds a small software overhead to the program.

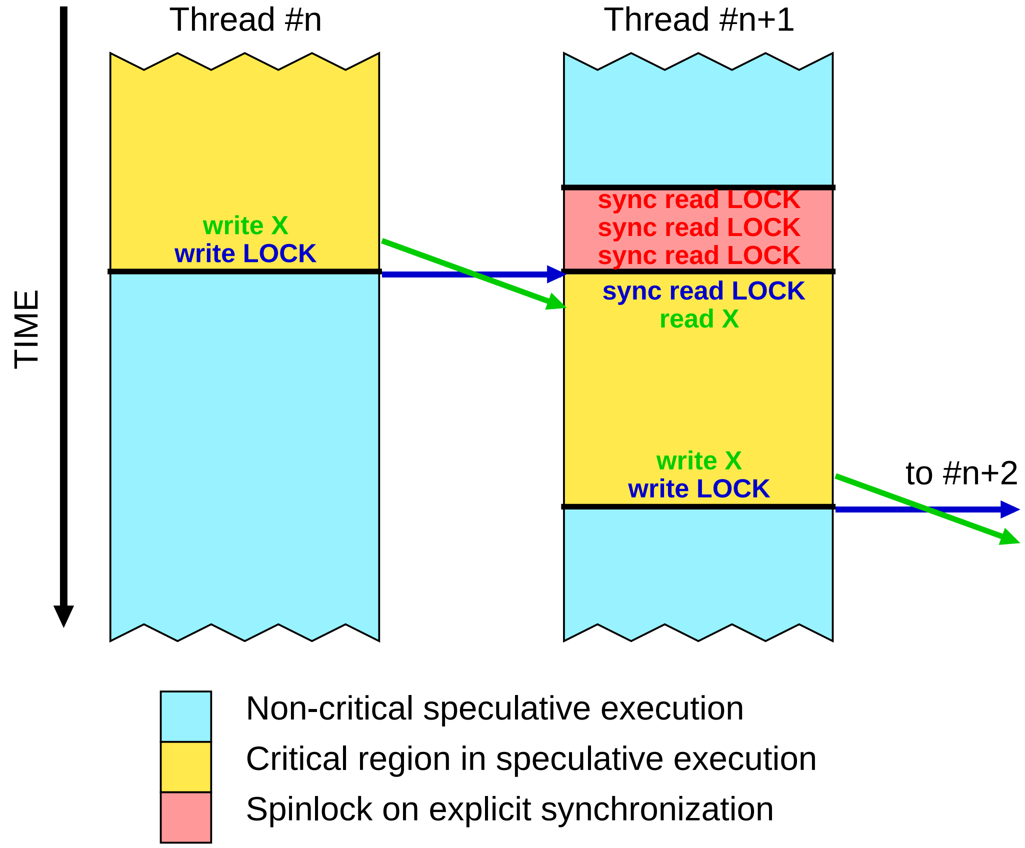


Figure 6-3: The timing of explicit synchronization

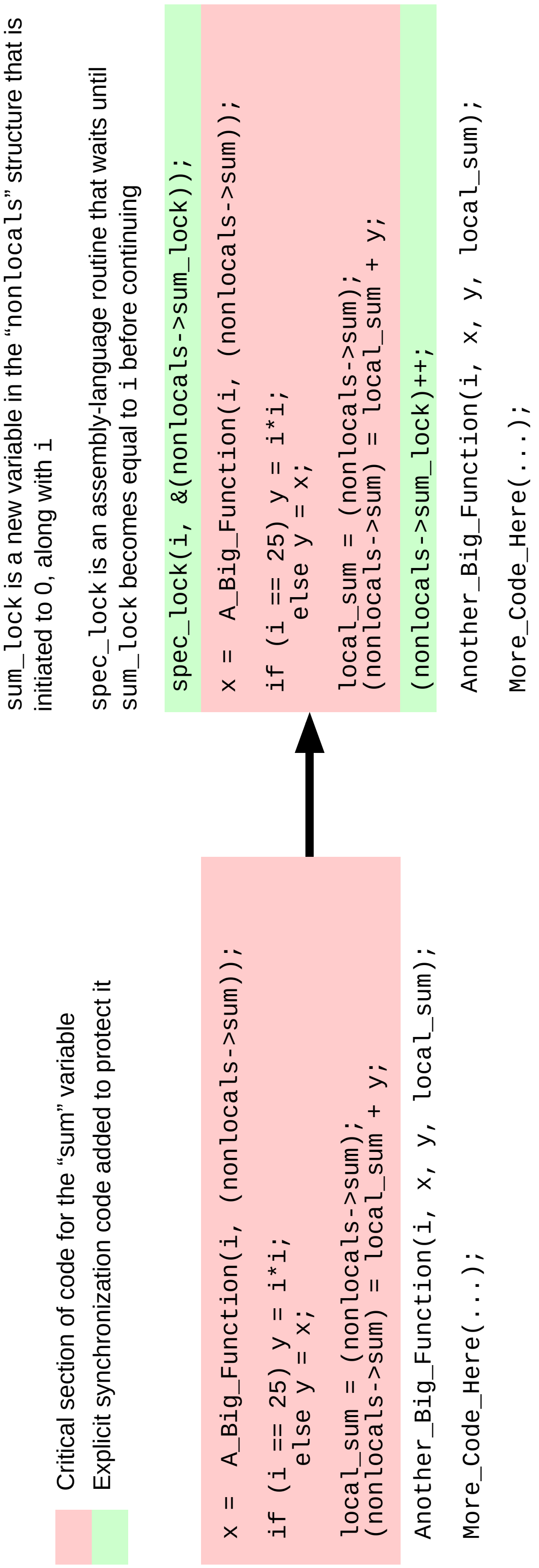
The following code example shows how this might be inserted into an actual program:

```
// Start of speculative region
threadNumber = (globals->threadNumber)++;

. . .

// Spin-lock to protect variable “X”
while (LL(globals->lockThread) < threadNumber);

// First read of variable “X”
y = x + 1;
```

```
. . .

// Last write of variable “X”
x = a + b + c;

// Release the lock for the next processor
(globals->lockThread)++;

. . .
```

The current thread number is loaded early in a speculative thread using a normal load that is fully synchronized using the speculative memory system. The global thread number is initially 0 and is incremented by each processor as it acquires a local copy of the thread number, as is shown. In the middle of the loop, the region between the first read and last write of the `x` variable forms a critical region. This region is outlined by adding two bits of code. The spinlock loop at the top uses a non-speculative LL to test a lock variable, which specifies the number of the thread that may safely use the variable. At the bottom of the region, a simple increment of the global variable releases the lock for the next processor. Figure 6-4 shows the critical region of an example program, with the `while` loop from the previous example pulled into a small assembly-language routine.

Similar synchronization mechanisms have been proposed before. In [14], special loads and stores were used to pass data between processors directly and perform explicit synchronization at the same time. This method avoided the overhead of extra synchronization code, but required more complex hardware synchronization mechanisms to handle the special loads and stores. An all-hardware data dependence prediction and synchronization technique was explored in [13] for use with the Multiscalar architecture. Special hardware structures tracked dependencies and then automatically used synchronizing hardware to prevent restarts due to dependent load-store pairs. This hardware-based technique allows a limited degree of automatic synchronization even without programmer intervention, and it would be adaptable for use on a Hydra-like architecture. An alterna-



🟥 Critical section of code for the “sum” variable

🟩 Explicit synchronization code added to protect it

```
x =  A_Big_Function(i, (nonlocals->sum));

if (i == 25) y = i*i;
     else y = x;

local_sum = (nonlocals->sum);
(nonlocals->sum) = local_sum + y;
Another_Big_Function(i, x, y, local_sum);

More_Code_Here(...);
```

sum_lock is a new variable in the “nonlocals” structure that is initiated to 0, along with i

spec_lock is an assembly-language routine that waits until sum_lock becomes equal to i before continuing

```
spec_lock(i, &(nonlocals->sum_lock));

x =  A_Big_Function(i, (nonlocals->sum));

if (i == 25) y = i*i;
     else y = x;

local_sum = (nonlocals->sum);
(nonlocals->sum) = local_sum + y;
(nonlocals->sum_lock)++;

Another_Big_Function(i, x, y, local_sum);

More_Code_Here(...);
```

Figure 6-4: Example showing the insertion of synchronization code.

tive design for a superscalar architecture that tracks sets of stores that commonly supply data to a following, dependent load was proposed in [11]. Finally, we presented some preliminary results on the use of synchronization with the compress benchmark in [12].

6.2.2 Code Motion

While explicit synchronization prevents critical dependencies from causing violations, it also forces the speculative processors to serialize their execution. For small critical regions, this is perfectly acceptable, but for large ones it can easily eliminate all of the parallelism that speculative threads are attempting to exploit. To avoid this situation the second technique we have used to improve speculation performance is to manually move source code lines with dependent loads and stores in order to shrink the critical path between frequently violating load-store pairs. This makes it possible to reduce the number of violations and often increases the inherent parallelism in the program by lengthening the sections of code that could be overlapped on different processors without causing violations.

This works in two ways. At the tops of critical regions, loads can sometimes be delayed by rearranging code blocks in order to move code without critical dependencies higher up in the loop body. However, this is usually only possible in large loops built up from several non-dependent sections of code that can be interchanged freely. More significant changes came from moving stores. We were frequently able to rearrange code to make stores to shared variables occur earlier. Induction variables are an obvious target for early stores. Since the store that updates the induction variable is not dependent upon any computation within the loop, these can safely be moved to the top of the loop body. When this is done, a “local” copy of the old value is made at the same time, with a small amount of additional code, and this local copy is used throughout the remainder of the loop body while the “global” variable is used by other processors starting their own loop iterations. The optimization performed on the loop induction variable in Figure 6-2 showed an example of this type of optimization. Other variables, that *do* depend upon results calculated in a loop iteration, will not be improved as dramatically by scheduling their critical stores early, but performance can often still be improved significantly over unmodified code using these techniques. Figure 6-5 shows an example of this type of optimization as it might be performed in the middle of a large loop body.

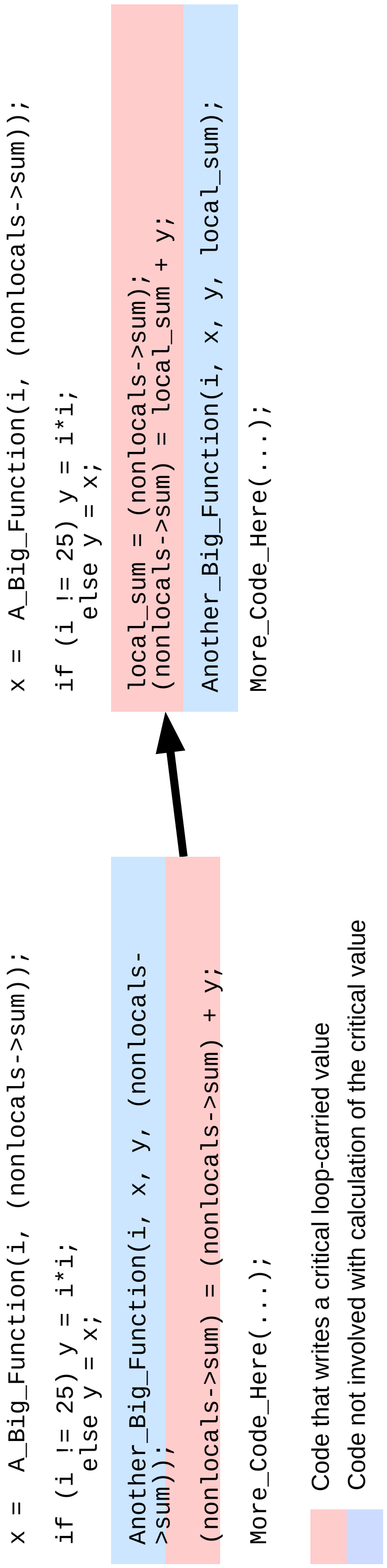
We have done all code motion by hand based on violation statistics provided by our simulator. However, while synchronization may easily slow down a program if implemented in an improper fashion, code motion will rarely degrade performance. Hence, a feedback-directed compiler could perform this job almost as well as a human programmer by aggressively moving references that frequently cause violations within speculative code, up to the limits imposed by existing data and control dependencies, in order to reduce the critical path between dependent load-store pairs.

6.2.3 Early Computed Value Prediction

Code motion on a speculative processor is somewhat different from that on a conventional multiprocessor, since the hardware is constantly checking to see if dependencies exist among the non-synchronized variables. As a result, code motion can just be considered a way of decreasing the probability of restarts, while not eliminating them. This view can be taken further by treating variables that often — but not always — act a lot like induction variables, with a very predictable update being the common case, just like induction variables at the top of the loop. Should they later act differently, the variable may be re-written, probably causing restarts on the more speculative processors. However, as long as the value computed and written early is used most of the time, the amount of available parallelism in the program may be vastly increased. Figure 6-6 shows an example of this optimization at work.

6.2.4 Loop Body Slicing and Chunking

More radical forms of code motion and thread creation are possible by breaking up a loop body up into smaller chunks that execute in parallel or its converse, combining multiple speculative threads

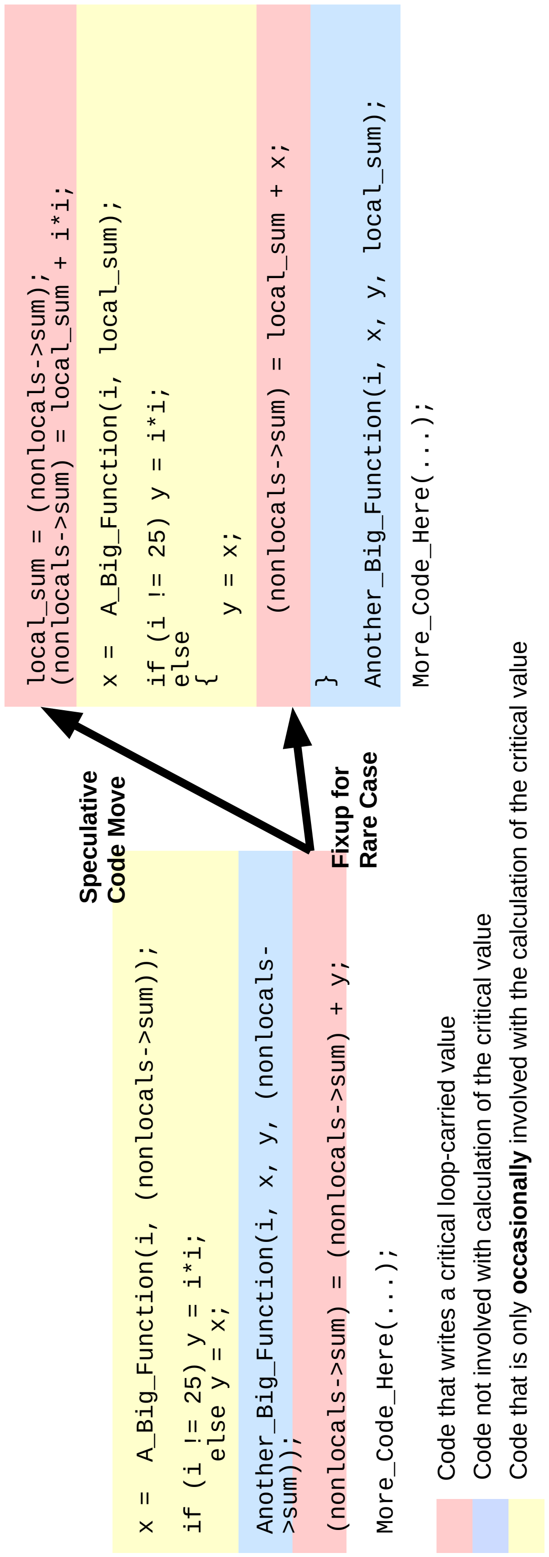


```
x =  A_Big_Function(i, (nonlocals->sum));

if (i != 25) y = i*i;
    else y = x;

Another_Big_Function(i, x, y, (nonlocals-
>sum));

(nonlocals->sum) = (nonlocals->sum) + y;

More_Code_Here(...);
```

```
x =  A_Big_Function(i, (nonlocals->sum));

if (i != 25) y = i*i;
    else y = x;

local_sum = (nonlocals->sum);
(nonlocals->sum) = local_sum + y;

Another_Big_Function(i, x, y, local_sum);

More_Code_Here(...);
```

🟥 Code that writes a critical loop-carried value
🟦 Code not involved with calculation of the critical value

Figure 6-5: Example showing code motion within a loop body.



```
x =  A_Big_Function(i, (nonlocals->sum));

if (i != 25) y = i*i;
    else y = x;
Another_Big_Function(i, x, y, (nonlocals->sum));
(nonlocals->sum) = (nonlocals->sum) + y;

More_Code_Here(...);
```

```
local_sum = (nonlocals->sum);
(nonlocals->sum) = local_sum + i*i;

x =  A_Big_Function(i, local_sum);

if (i != 25) y = i*i;
else
{
    y = x;
    (nonlocals->sum) = local_sum + x;
}
Another_Big_Function(i, x, y, local_sum);

More_Code_Here(...);
```

Speculative Code Move
Fixup for Rare Case
🟥 Code that writes a critical loop-carried value
🟦 Code not involved with calculation of the critical value
🟨 Code that is only **occasionally** involved with the calculation of the critical value

Figure 6-6: Example showing value prediction within a loop body.

together into a single thread. With the former technique, loop slicing, a single loop body is spread across several speculative iterations running on different processors, instead of running only as a single iteration on a single processor. In the latter case, loop chunking, multiple loop iterations may be chunked together into a single, large loop body. Loop bodies that were executed on several processors are combined and run on one. This generally only results in better performance if there are no loop-carried dependencies besides induction variables, which limits the versatility of loop chunking. However, if a parallel loop can be found, chunking can allow one to create speculative threads of nearly optimal size for the speculative system. Figure 6-7 shows conceptually how slicing and chunking work, while Figure 6-8 shows a code examples from sample loops using both techniques.

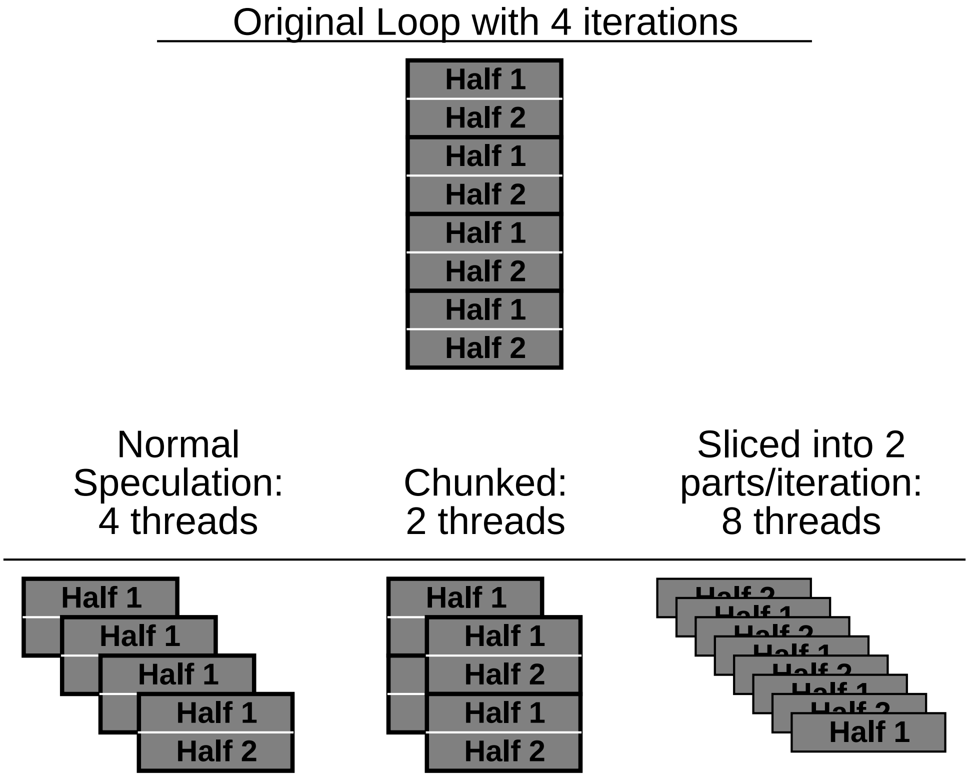


Figure 6-7: Graphical view of loop body chunking and slicing.

A) Original loop:

```
for (i=0; i < N; i++)
{
     . . . small amount of work . . .
}
```

Final speculative loop body, with chunking:

```
     . . .

start_of_loop_label:

     /* localization and increment of for variable */
     i = (globals->i);
     unroll_i = (globals->i) += UNROLL_FACTOR;

     /* “unrolling” for loop */
     for (; i < unroll_i; i++)
     {
          /* original for loop termination clause */
          if (!(i < N)) spec_terminate();

          /* original loop body */
          . . . small amount of work . . .
     }

     /* normal end of loop iteration */
     spec_end_of_iteration();

     . . .
```

B) Original loop:

```
while (1)
{
     . . . first big code block . . .
     . . . second big code block . . .
     importantGlobal = CalculateMe();
     . . . third big code block . . .
}
```

Final speculative loop body, with slicing:

```
     . . .

start_of_loop_label:

     /* localization and increment of slice variable */
     slice = (globals->slice)++;
     if (slice == 2) globals->slice = 0;

     switch(slice)
     {
          case 0:
               . . . first big code block . . .
               break;
          case 1:
               . . . second big code block . . .
               break;
          case 2:
               /* importantGlobal is “precomputed” now */
               importantGlobal = CalculateMe();
               . . . third big code block . . .
               break;
     }

     /* normal end of loop iteration */
     spec_end_of_iteration();

     . . .
```

Figure 6-8: Example showing loop body chunking and slicing.

While loop chunking only needs to be performed if the body of the loop is so small that the execution time is dominated by speculation overheads, the motivation for loop slicing are more complex. The primary reason is to break down a single, large loop body, made up of several fairly independent sections, into smaller parts if they are more optimally sized for speculation. In a very large loop body a single memory dependence violation near the end of the loop can result in a large amount of work being discarded. Also, the large loop body may overflow the buffers holding the speculative state. Buffer overflow prevents a speculative processor from making forward progress until it becomes the head, non-speculative processor, so this should normally be avoided whenever possible. Loop slicing is also a way to perform code motion to prevent violations. If there is code in the loop body that calculates values that will be used in the next loop iteration and this code is not *usually* dependent upon values calculated earlier in the same loop iteration, then this code may be sliced off of the end of the loop body and assigned to its own speculative thread. In this way, the values calculated in the sliced-off region are essentially “precomputed” for later iterations, since they are produced in parallel with the beginning of the loop iteration. The advantage of slicing over normal code motion is that no data dependency analysis is required to ensure it is legal to perform the code motion, since the violation detection mechanism will still enforce all true dependencies that may exist.

Loop chunking may be implemented in a compiler in a similar manner to the way loop unrolling is implemented today, since both are variations on the idea of combining loop iterations together. In fact, the two operations may be merged together, so that a loop is unrolled as it is chunked into a speculative thread. As a result, adding this capability to current compilers should not be difficult. Effective slicing, however, will require more knowledge about the execution of a program than loop unrolling, although the ability to analyze the control structure of the application combined with violation statistics should be sufficient. Slicing should not be performed indiscriminately because it may allow speculative overheads to become significant and/or it may result in significant load imbalance among the processors if some slices are much larger than others. The former problem will obviously slow the system down, while the latter problem will degrade performance in the current Hydra design, since a speculative processor that completes its assigned thread must stall until it becomes the head processor, wasting valuable execution resources in the process. This problem can only be overcome by making the speculation system allow multiple speculative regions within each CPU, so that a processor that completes a speculative region may immediately go on to another while it waits to commit the results from the first. However, implementing this capability has significant hardware overheads since bits and/or buffers in both the L1 and L2 caches will need to be replicated for each thread that may be waiting on each processor.

6.3 Subroutine Speculation

A subroutine speculation API is not strictly necessary, and therefore not discussed in this thesis. On most processors, subroutine calls must follow a predetermined protocol of register saving and restoring that eliminates the possibility of most register-register dependencies. As a result, the low-level subroutine speculation handlers can quite easily handle all register-register dependencies that may occur between threads simply by passing registers from one processor to another during thread forking and then watching for changes in a few key registers at thread completion. On the MIPS architecture, checking the result-value returning register(s) is sufficient.

As a result, the only API enhancements that might be included would be performance-enhancing hints about which subroutines to target for speculation and suggestions for return values that they might produce. While we did not pursue optimized subroutine execution enough to develop a specialized source-code API, we added a feature to LESS to preselect one or more routines as subroutine speculation targets. With this modified version of LESS, we were able to speed up the execution of `eqntott`, a program dominated by a recursive series of `quicksort()` calls. More recently, we have achieved similar or better speedups by converting similar trees of recursive calls into iterative loops and using high-speed loop processing on the resulting loop [reference to Manohar's current work or delete this].

6.4 Analysis of the Speculative Programming Model

The design of the software interface for speculation was a great success. With it, we were able to parallelize programs much more quickly and easily than with any conventional parallelization tool. For example, `ijpeg` was parallelized in only three days by a single person, without a single error. Many other applications were also parallelized in a fraction of the time that a conventional parallelization effort would have taken. With aid from `hydracat`, these times could be cut even further. Most optimizations that we developed were fairly simple to implement, although determining the best places to use the various optimizations often took some analysis, even with our automatic tools to find “hot spots” where loads and stores frequently cause faults.

In the future, the basic API for speculation should stay essentially the same. However, the tools supporting programming for speculation will need to develop. `Hydracat` or a successor to it needs to be more robust and able to support features such as automatic insertion of common optimizations and automatic profiling-based feedback. With a profiling compiler, it should be possible to determine which loops are the best targets for speculation automatically, since loops may be converted to speculative loops automatically. Also, a compiler should be able to take feedback information on the location of “hot spots” in the code and automatically insert simple optimizations like synchronization to automatically handle them. With these additions, a speculative compilation system could be truly powerful.

7 The Performance of Speculation

This chapter gives an overview of our evaluation of the speculative Hydra system.

7.1 Benchmarks and Simulation Methodology

To evaluate our speculative system and identify potential areas for improvement, we used eleven representative benchmarks that cover a significant portion of the general-purpose application space. These benchmarks are listed in Table 7-1. The speculative versions of the benchmarks were created using Hydracat. The binaries were generated using GCC 2.7.2 with optimization level `-O2` on an SGI workstation running IRIX 5.3. Currently, Hydracat only works on C programs. This prevents us from experimenting with any of the SPEC95 floating-point benchmarks, which are all written in FORTRAN. However, from the point of view of extracting parallelism, the C SPEC92 floating point benchmarks are more challenging than the SPEC95 floating point benchmarks, so we would expect the performance on SPEC95 benchmarks to be similar or better.

All the performance results we present are obtained by executing the benchmarks on the LESS simulator described in Chapter 2. For the most part, this simulation properly modeled the speculative Hydra memory system described in Chapter 3 and Chapter 5. However, the sizes of some of the memories were enlarged to more properly simulate a future high-end processor. These differences are summarized in Table 7-2. The execution time of the speculative binaries includes the time spent in the run-time system. Even though we used the capability of LESS to fast-forward through large sections of the benchmarks, code representing at least 95% of the original sequential execution time has been simulated to ensure that the simulations are representative.

7.2 Baseline Performance

The performance of the base speculative Hydra CMP running the more flexible but slower loops + subroutines speculation is shown in Figure 7-1 (speedups obtained) and Figure 7-2 (how proces-

Table 7-1: A summary of the application benchmarks used to evaluate speculative threads. Ones that were also used with the baseline system are noted in italics.

	Application	Source	Input Data Set
General Integer	*compress*	SPEC95 [48]	train
	eqntott	SPEC92 [53]	reference
	grep	UNIX utility [51]	190 line file
	m88ksim	SPEC95 [48]	test
	vortex	SPEC95 [48]	test, simplified
	wc	UNIX utility	10,000 character file
Multimedia Integer	ijpeg (compression)	SPEC95 [48]	train
	mpeg-2 (decoding)	MediaBench [46]	test.m2v
Floating Point	alvin	SPEC92 [48]	reference
	cholesky	Numeric Recipes [52]	100 x 100
	ear	SPEC92 [53]	reference
	simplex	Numeric Recipes [52]	40 variables

Table 7-2: The Hydra memory hierarchy characteristics used for these evaluations. These are identical to those used previously for the non-speculative simulations.

	L1 Cache	L2 Cache	Main Memory
Configuration	Separate I & D SRAM cache pairs for each CPU	Shared, on-chip SRAM cache	Off-chip DRAM
Capacity	16KB each	2 MB	128 MB
Bus Width	32-bit connection to CPU	256-bit read bus + 32-bit write buses	64-bit bus at half of the CPU speed
Access Time	1 CPU cycle	5 CPU cycles	at least 50 cycles
Associativity	4-way	4-way	N/A
Line Size	32 bytes	64 bytes	4 KB pages
Write Policy	Writethrough, no allocate on writes	Writeback, allocate on writes	"Writeback" (virtual memory)
Inclusion	N/A	Inclusion enforced by L2 on L1 caches	Includes all cached data

sors spent time during speculation). The speedups represent the execution time of one of the processors in Hydra running unmodified C code divided by the execution time of the speculative

Hydra CMP running speculatively parallelized C code, to show the effect of using the speculative threads to allow the other three processors in Hydra to help speed up program execution. We see that the performance is highly dependent on the application and varies from 0.6 to 3.4. To understand the performance profile in more detail, we explain how each benchmark was parallelized and what limits the speculative parallel performance on the benchmark.

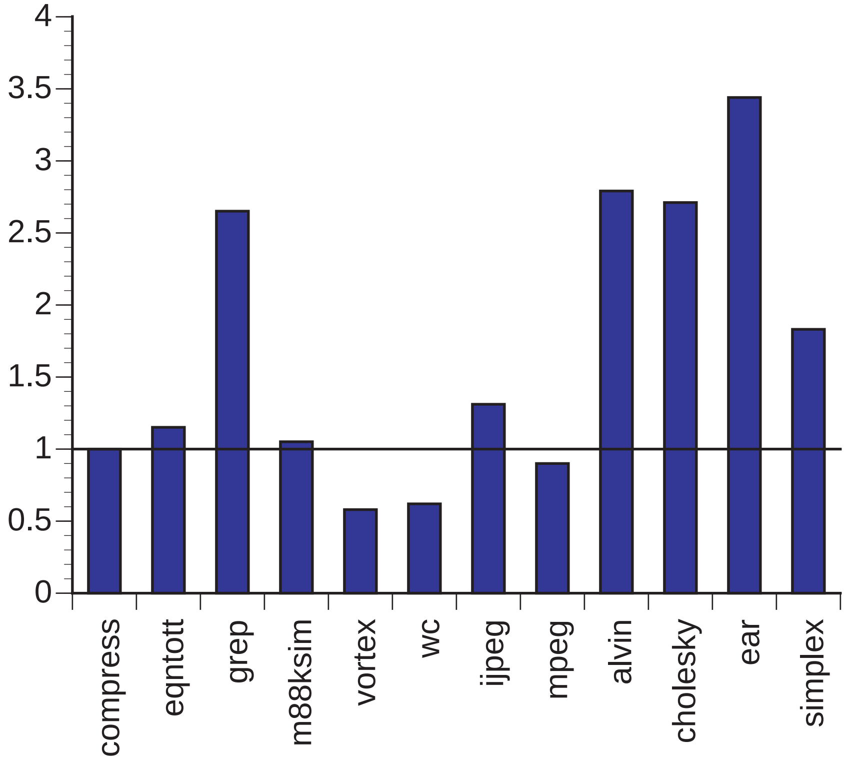


Figure 7-1: Baseline speculation system performance, in speedup of 4-processor system over a single processor executing the original uniprocessor benchmark.

compress (compression): This benchmark compresses an input file using entropy encoding. The core of `compress` is fairly small loop — about 100 instructions or 140 cycles per iteration — that has complex control flow but is large enough so that the speculation and communication overheads, while drastically limiting the amount of parallelism usable in the code, do not completely overwhelm its execution time. As observed in [63], and contrary to the analysis in [14] there is a loop-carried dependency in variable `ent` that limits the parallelism in this loop.

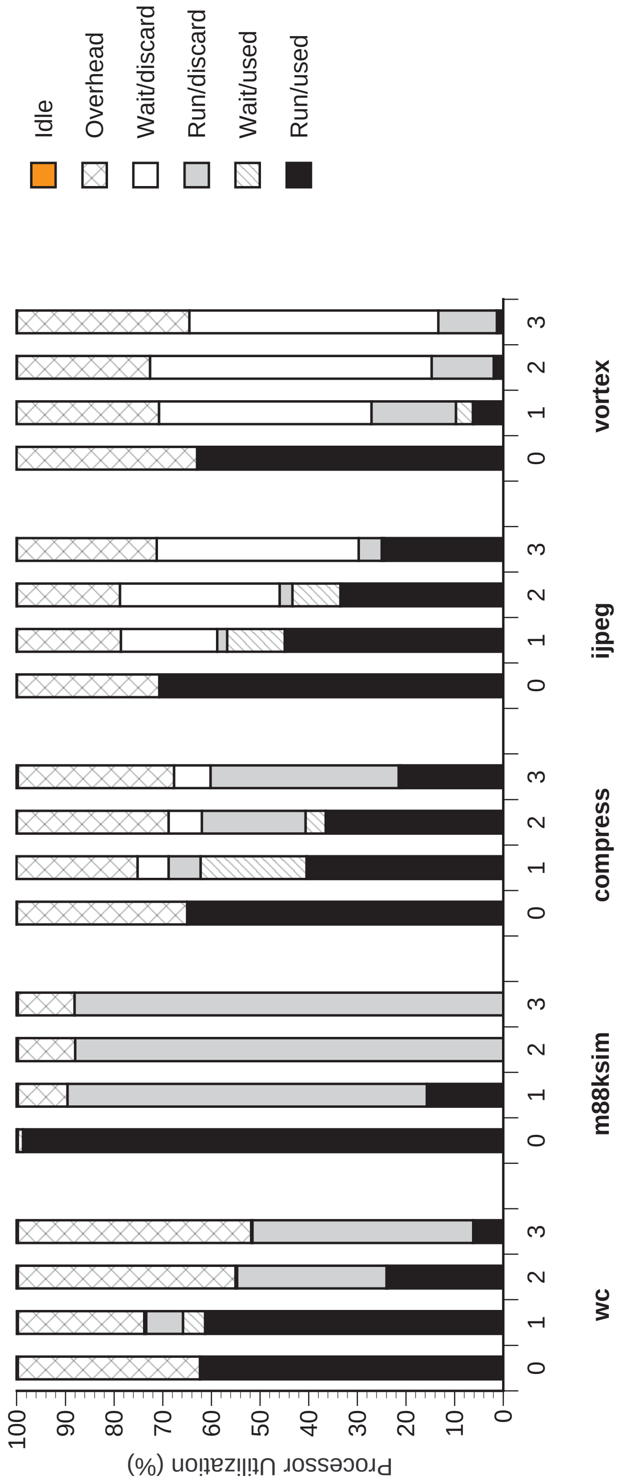


Figure 7-2: Speculative processor usage breakdown for five of the benchmarks. This graph shows the distribution of work and wasted time on speculative processors, from the least speculative (0) to the most speculative (3).

When we left this critical loop-carried variable alone, performance was essentially equal to the uniprocessor version, because enough parallelism was exploited through the overlap of the I/O routines from separate loop iterations to balance out the speculation overheads.

eqntott: This benchmark performs logic minimization on a set of input equations. Unlike the other benchmarks, we did not parallelize the loops in this application, because the only good candidate loop (in the `cmppt` function) is too small to be effectively parallelized on Hydra without significant programmer help. As a result, we used procedure-call based speculation. Despite a reasonable amount of parallelism in the large, recursive `quicksort` procedure, the speculation software overheads and squashing of non-parallel procedures limit speedup.

grep: This benchmark is the well-known regular expression tool that matches a regular expression to each line of an input file and prints those lines that contain matching strings. The main loop of the program iterates over all the lines of the input file. The speculative thread version of this loop performs very well because the only dependencies that occur are during file input and when a match is found. However, most lines do not match the expression and so once they have been read from the input file they are processed completely in parallel.

m88ksim: This program performs simulates a Motorola 88000 RISC CPU on a cycle-by-cycle basis. The loop in `m88ksim` is very large for a speculative loop, running for about 5000 cycles and executing an average of 4500 dynamic instructions during each iteration. With such a long loop, the overhead associated with speculation control and inter-processor communication had a minimal impact on the overall execution time. Instead, some of `m88ksim'`s global variables are read and written at various locations in the loop, some inside subroutines, that severely curtail the amount of parallelism that may be exploited. The first speculative processor can use about 15% of its time usefully by overlapping the beginning of each loop iteration with the end of the previous one, but most of this time is simply spent overcoming the communication inefficiencies, limiting speedup to 3.5%. Meanwhile, the second and third processors contribute nothing, as they must work on iterations two or more ahead that cannot overlap with the head iteration at all due to true dependencies. Previous work has shown that an actual parallelizing compiler, designed from the start to divide code into threads and optimize the inter-thread communication, might expose more parallelism and thereby allow more speedup by moving loads of shared data as late as possible in threads and stores of shared data as early as possible [43][14]. However, in the baseline case we were just using a standard compiler to generate code that could not make these distinctions.

vortex: This is a SPEC95 benchmark that simulates a small database server designed in a pseudo-OOP variant of C and running on a pseudo-OS. `vortex` indicates that subroutine parallelism is very difficult to utilize due to control software overhead and a lack of parallelism between the code in subroutines and the continuation code following them. Our simple *last value* return value prediction mechanism was able to obtain a 96.6% successful prediction accuracy when speculating on the pseudo-OOP `vortex` code, thanks to the large number of functions that return nothing or rarely-raised error flags. However, the severe misprediction penalty for the remaining 3.4% of the predictions — complete flushing of the system's speculative state — combined with frequent memory dependence violations originating from the side effects of the functions made parallelism virtually impossible to find. Increasing the amount of parallelism exploitable would require a very sophisticated compiler that performed interprocedural optimizations to increase the distance between loads and stores that *might* be communicated between processors running different subroutines in parallel.

Another significant problem demonstrated by `vortex` is load imbalance. Unlike `eqntott`, whose subroutine behavior mostly consists of very large and balanced calls to its `quicksort` routine, `vortex` calls a wide variety of different routines, all of different sizes. Most of the speculative processors get stuck with relatively small routines and then spend their time waiting to become the head, since long subroutines tend to move up quickly and then tie up most of the time of the head processor. To improve performance, a compiler would be forced to break up longer subroutines into smaller parts to help solve load balancing problems.

It should also be noted that the overhead associated with software control of speculation is exceptionally high because `vortex` is parallelized *only* using speculative procedure continuation, which must use the full subroutine control protocol instead of the low-overhead looping protocols. As a result, the overhead is comparable to that associated with fairly small loops like `wc` or `compress` even though the subroutines are generally much larger than the loops in those benchmarks, since small subroutines are simply pruned off and avoided by our speculative thread selection mechanisms.

wc: This is the UNIX word count utility, which counts the number of characters, words, and lines in an input file. Our results demonstrate that the software control overheads associated with our implementation of speculation can severely limit speculation performance. The core of `wc` is a single loop that takes an average of only 27 cycles per iteration with fully-optimized uniprocessor code, other than the occasional iterations when the call to `getchar()` within the loop must request more characters from the OS. Even using the "quick" version of the loops + subroutines runtime system, the speculation control software requires approximately 40% of the time on all of the processors just to handle the frequent iteration completions (on the head and #3 processor) and dependency violations (on processors 1-3), since the 10-15 instruction overhead of these operations is about half of the entire loop time! Even with this overhead, the parallelism that speculation is able to expose in the loop still allows about 40% of the system's processor time to work on the actual code. If all of this time could be exploited productively, a speedup of 1.6 could be obtained. However, an entire processor's worth of performance is lost to two factors related with interprocessor communication. First, `wc` has two critical loop carried-dependencies that cannot be avoided — the buffer pointer in the `getchar()` library call, and the local *in a word* variable that is used to count words. While the uniprocessor hits in its data cache when accessing these variables, a speculative processor must devote ten or more cycles to handling the data cache misses associated with this communication. Additionally, this communication forces the compiler to insert loads and stores to move the values to and from the `nonlocals` memory during every iteration to facilitate communication, preventing the register allocation of these commonly-used variables that may be used in the uniprocessor code. The combined effect of these two communication-related inefficiencies consumed a processor's worth of execution time on this small loop.

ijpeg (compression): This is an integer, multimedia benchmark that compresses an input RGB color image to a standard JPEG file, using a lossy algorithm consisting of color conversion, downsampling, and DCT steps. The multiple loops in these steps possess a large amount of inherent parallelism, but were originally coded so that the parallelism is often obscured by existing program semantics, such as numerous pointers, poor code layout, and odd loop sizes. The speculatively parallel version of `ijpeg` exposes much of the available parallelism, but performance is limited by the coding of the loops, parallel loop coverage, and speculation software overheads. This benchmark clearly indicates that our loop speculation mechanisms are able to exploit parallelism in code when that parallelism exists, even without extensive compiler optimizations. At the same time, it also emphasizes the need for aggressive optimizations, as these results should be even better.

mpeg (decoding): This is a benchmark from the Mediabench suite [46] that decodes a short MPEG-2 video sequence to an RGB frame buffer. Parallelization occurs at the macro block level, where the variable-length decoding (VLD) is performed. The VLD step is completely serial, but speculation is able to overlap the processing performed during the other, more parallel stages of decoding (IDCTs and motion estimation) with the serial VLD step of later macroblocks. This parallelization technique is described in greater detail for a hand-parallelized version of mpeg in [44]. Despite a significant amount of potential parallelism, MPEG-2 decoding slows down because of a loop-carried dependency that unnecessarily serializes execution.

alvin: This is a neural network training application for autonomous land vehicle navigation. It is composed of four key procedures which are dominated by doubly-nested loops. Although these loops are parallel, the parallelism is obscured by the way in which the application is coded with pointer variables. However, these loops are easily speculatively parallelized and result in good parallel performance.

cholesky: Cholesky decomposition and substitution is a well-known matrix kernel. The multiply nested loops in the decomposition procedure have significant amounts of parallelism, but they also contain loop-carried dependencies that occur infrequently. In conventional parallelization, these dependencies would have to be detected and synchronized, but with speculation we can obliviously parallelize them. The loops in the substitution routine are also parallel, but here each loop is written in so that a loop-carried dependency serializes the speculative loop. Fortunately the decomposition procedure dominates the sequential execution time and speedup is still quite good.

ear: This benchmark simulates the propagation of sound in the human inner ear. The conventional wisdom is that the structure of this program is a sequential outer loop and a sequence of parallel inner loops [47]. The parallel inner loops are extremely fine grained and therefore are not good candidates for speculative threads on the Hydra CMP. However, the outer loop can be speculatively parallelized and achieves very good speedup. The dependent outer loop pipelines across the processors quite well because each iteration is only dependent on the previous iteration in a way that allows much of the computation to be overlapped. Memory renaming ensures that, even though each iteration of the outer loop uses the same data structures, each processor dynamically gets a private copy.

simplex: This kernel solves problems in linear programming. The three procedures in this kernel have a number of small loops. The dependences between the iterations of these loops are not at all obvious. The speculative parallelization of these loops achieves good speedup, but less than the other numerical benchmarks. This is mainly due to the speculation software overheads that dominate the short running loops.

The memory buffering system described in Section 5.3 worked very well in our simulations. We found that the memory system added very little latency beyond the basic L2 cache hit time required after every communication invalidation. Also, some test simulations that we performed showed that the full per-word sets of write bits in the L1 cache proved to be essential. All of our simulations done without them resulted in the useful work done by the speculative processors dropping to nearly zero in virtually all cases due to false violations on WAW hazards.

7.3 Write Buffer Requirements

A key architectural metric is the size of the speculative write state in the buffers described in Section 5.3.3. This metric is important because it indicates the feasibility of implementing a Hydra CMP architecture with reasonably sized speculative write buffers that achieves good performance. Figure 7-3 shows the numbers of 64 byte L2 buffers filled during each successful speculative thread on several of our integer benchmarks. Similarly, Table 7-4 lists the number of 32 byte cache line buffers required to hold the maximum size write state for the benchmarks after the loop-only speculation handlers are used. Both sets of results clearly indicate that a buffer of 24-32 lines (1.5 KB – 2 KB) per processor is sufficient to handle even fairly large loop iterations or subroutines in most cases. Even the huge `m88ksim` loop iterations would have fit in a 21-line buffer.

The only two loop-oriented benchmarks that might have trouble with these limits were the alvin and ear benchmarks. These two applications would require 8KB write buffers for maximum performance. However, it would be possible to reduce the memory requirements of the alvin benchmark to an arbitrary degree simply by using loop iteration chunking on the inner loop of alvin, as described in Section 6.2.4, instead of our original scheme of speculating on the outer loop.

It should also be noted that a small number of the subroutines speculated on in vortex and ijpeg required more buffer space, but these routines were infrequent enough that simply halting them and then completing the iteration non-speculatively after becoming the head processor would probably have little impact on performance. For that matter, most of these routines would be running on the head processor by the time they had managed to fill up a 2 KB buffer anyway. As a result, the write-through head-processor buffers described in Section 5.4.4, which were not used during these simulations, might eliminate the need to halt at all.

These results indicate that the basic Hydra memory system lends itself well to speculative operation. Even allowing for full double-buffering with pairs of processors, the system requires only about 19KB of extra on-chip memory — or 13KB without full double buffering. This is about the same size as an extra pair of L1 caches. Even allowing for a fair amount of control logic overhead, it seems reasonable to believe that all of the hardware we propose would not be larger than a single simple processor, each of which is only a small fraction of the total die area.

7.4 Improved Performance Results

After switching to loop-only speculation (from Section 5.5.3) and applying the various software optimizations described in Section 6.2 and Section 6.3 to the benchmarks, we collected a new set of performance results for all benchmarks except vortex. As a conceptual experiment, we also tested the performance of the fully optimized applications on a hypothetical version of Hydra with optimal, and virtually impossible to build, restart behavior after violations. Instead of following

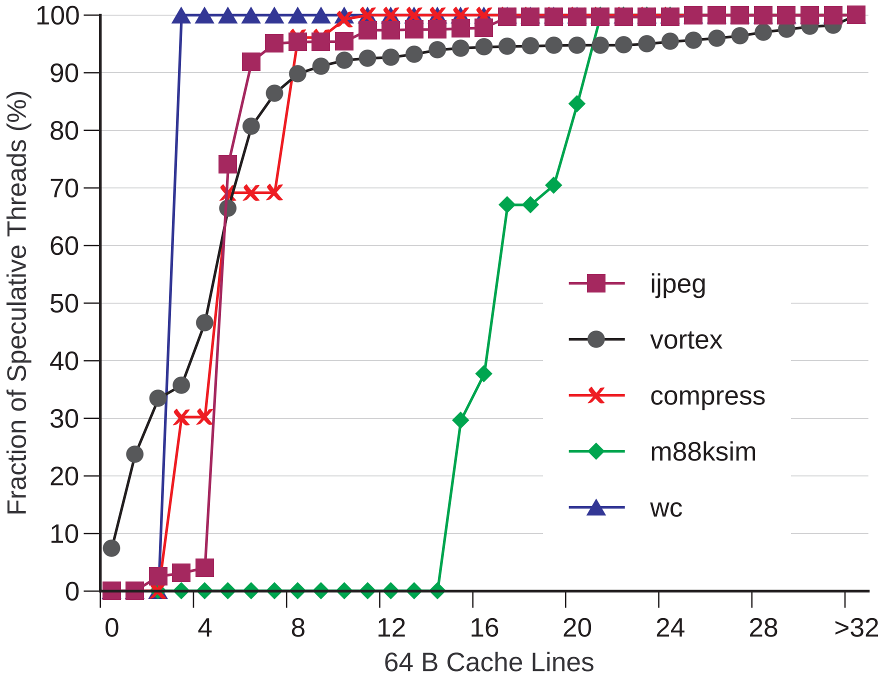


Figure 7-3: Speculative store buffer size requirements for some benchmarks.

the normal Hydra violation protocol described in Section 5.5.5, the test version of Hydra with “Hardware Checkpoints” could back up just to the load that actually caused the violation. This would allow it to keep all of the work that it had managed to complete before the offending load, potentially saving a lot of time when the violating loads occur late in speculative threads. Of course, more speculative processors behind the violating processor still had to do full restarts after receiving KILL messages, so the overall savings potential was limited.

The results of these simulations are shown in Table 7-3 and Figure 7-4. From these results it is clear that the lower overheads of loop-only speculation provide significant performance improvements for most benchmarks. The most dramatic improvement is seen with the `wc` benchmark, whose performance more than doubles. This is consistent with `wc’`s small loop body, that was completely dominated by the speculation software overheads in the base run-time system.

Table 7-4 shows some key performance characteristics that we obtained from the loop-only versions of the benchmarks (except for `eqntott`, since loops were not used there) and used as a guide for directing our software optimizations. Except for the `ijpeg` and `simplex` benchmarks, more than 90 percent of each benchmark can be speculatively parallelized. However, high coverage does not guarantee good performance when there are a significant number of violations. The number of restarts per committed thread (restart rate) gives a good indication of the inherent parallelism in an application. As expected, the benchmarks fall into two main classes: integer benchmarks with restart rates that are much greater than one and numerical benchmarks with restart rates less than one. The combination of the coverage, the restart rate, the amount of work that is lost due to the restarts, and the speculation software overheads determines the fraction of the time the processors are doing useful work (CPU utilization), which ultimately determines performance.

Table 7-3: Performance of the system with various enhancements.

	Application	**Procedures and Loops**	**Loops Only**	**After Software Optimization**	**With Hardware Checkpoints**
General Integer	compress	1.00	1.42	1.57	1.71
	eqntott	1.15	—	1.75	1.8
	grep	2.65	2.86	—	2.85
	m88ksim	1.05	1.13	1.45	1.56
	wc	0.62	1.57	—	1.64
Multimedia Integer	ijpeg (compression)	1.31	1.56	—	1.6
	mpeg (decoding)	0.90	0.96	2.14	2.14
Floating Point	alvin	2.79	3.02	—	—
	cholesky	2.71	2.85	3.16	3.25
	ear	3.44	3.87	—	3.88
	simplex	1.83	2.67	—	2.72

Table 7-4: Loop-only speculative performance characteristics summary. The speculative write state is presented in units of 32-byte cache lines.

	Application	Coverage (%)	Restarts per thread	CPU utilization (%)	Maximum Speculative write state
General Integer	compress	100	6.4	28	24
	eqntott	93	12.1	32	40
	grep	90	1.1	75	11
	m88ksim	94	15.5	29	28
	wc	100	4.6	69	8
Multimedia Integer	ijpeg (compression)	60	2.35	42	32
	mpeg (decoding)	92	1275	25	56
Floating Point	alvin	96	0.31	81	158
	cholesky	91	0.88	74	4
	ear	96	0.32	97	82
	simplex	86	0.14	60	14

We were able to optimize the software for five (`compress`, `eqntott`, `m88ksim`, `mpeg`, and `cholesky`) of the eleven benchmarks using the optimizations described in Section 6.2 and Section 6.3.

compress: Synchronization was added to `compress` to prevent the key loop-carried dependency from causing violations. This reduced the number of restarts by a factor of 3 (to 2.1 per iteration), allowing a greater speedup.

eqntott: The optimization of `eqntott` was actually performed using the more selective speculation software described in Section 6.3. Instead of speculating on all procedure calls, we modified the system so that *only* calls to the recursive `quicksort` procedure generate new speculative threads. This eliminates the performance losses from procedure speculation of other procedures which have little or no parallelism.

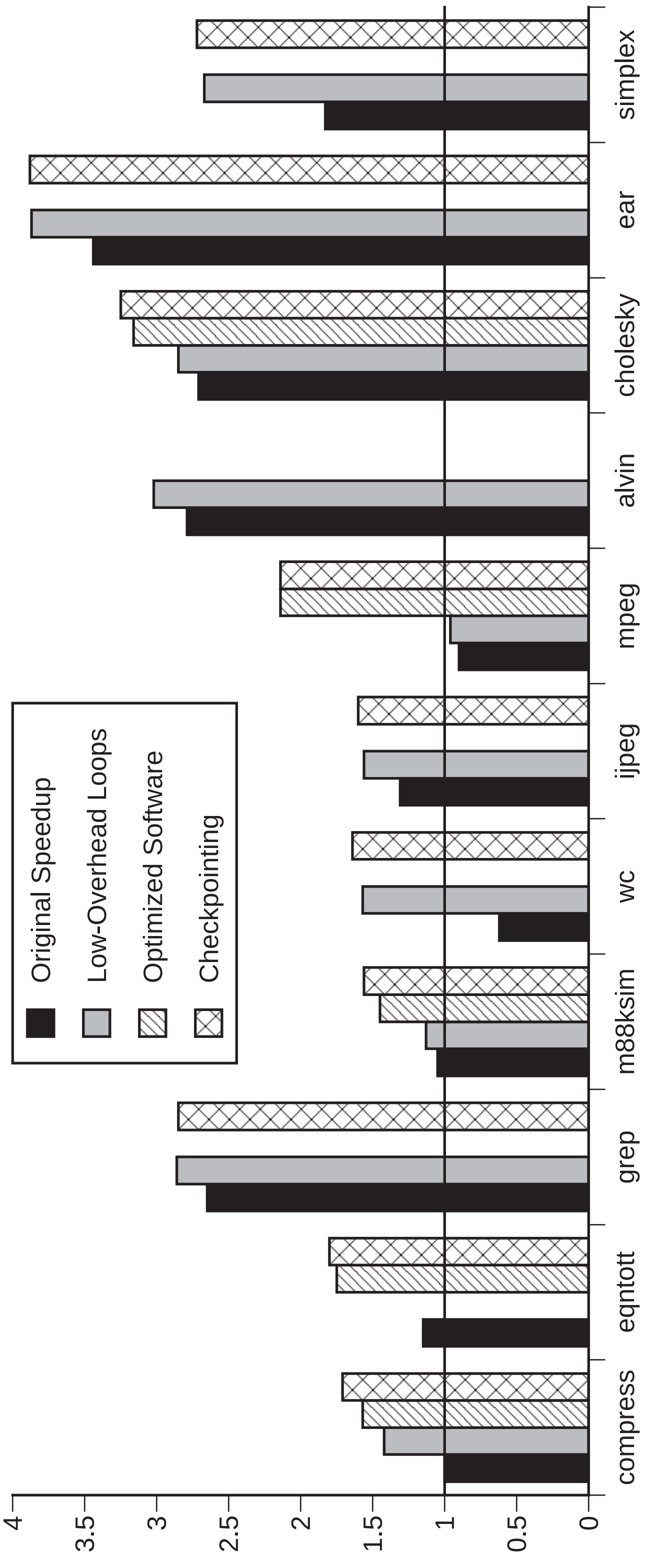


Figure 7-4: A performance comparison of the baseline and enhanced systems.

m88ksim: The large loop body in `m88ksim` was sliced to expose parallelism within the loop body and to allow code motion for a data value that is read early in each iteration, but is normally computed late in the previous iteration. This code motion decreased the number of restarts by a factor of 5 (to 3.1 per iteration), as the amount of available parallelism was dramatically increased.

mpeg: The performance of the MPEG-2 benchmark is improved substantially by moving an induction-like variable update from the end of the loop body to before the time-consuming DCT step. This code motion makes it possible for DCTs of multiple macroblocks to overlap. The same performance improvement was also achieved with loop slicing. As with `m88ksim`, the additional parallelism exposed by these modifications reduced the number of restarts by a factor of 5 (to 280 per iteration). Our speculatively parallel version of this code was actually faster than the macroblock-based hand-parallelized version in [44], because the speculative system does not have to synchronize between processors in a conservative fashion.

cholesky: In the Cholesky substitution program stage, the direction of the inner loop is reversed so that the loop carried dependency occurs on the last loop iteration, instead of the first. This allows the inner loop iterations from consecutive outer loop iterations to overlap.

These modifications resulted in significant performance improvements in these benchmarks, as is shown in Table 7-3. In particular, we were able to achieve a speedup of 2.14 on `mpeg`, whereas the baseline system slowed down.

The performance improvements from the more optimal “hardware checkpointing” scheme executing with the optimized software are quite modest. Not enough work was saved by the smaller backups to make a noticeable impression on the speedup figures. Furthermore, we found that mixing explicit software synchronization and hardware checkpointing is not useful, based on our observations with the `compress` benchmark. In effect, the optimized violation behavior in our "hardware checkpointing" scheme is essentially an inherent form of synchronization, since the violating processor is just pulled back to the "synchronization point" formed by the violating load. As explained in Section 6.2.1, software synchronization is essentially a way to prevent violations that result in a large amount of needlessly lost work — the same goal of hardware checkpointing. As a result, software synchronization just added some additional overhead to `compress`, slowing down the system while supplying little or no benefit.

Table 7-5 summarizes the gains from these improvements for each of the major application classes using the harmonic means.

Table 7-5: Harmonic mean summaries of the performance results.

Category	Procedures and Loops	Loops Only	After Software Optimization	With Hardware Checkpoints
General Integer	1.04	1.45	1.73	1.81
Multimedia Integer	1.07	1.19	1.80	1.80
Floating Point	2.56	3.04	3.12	3.15
Total	1.33	1.71	2.10	2.14

7.5 Conclusions

Speculation proved to be a very effective technique for extracting available parallelism out of our benchmarks with a minimum of programmer intervention. However, it is not a magic wand. If there is limited parallelism in the way a program is coded, then no significant speedup is possible. However, at least speculation cannot cause code to run incorrectly, unlike a lot of manual parallelization techniques. Also, it can be turned off, so it cannot hurt performance.

We had originally hoped to use speculation primarily on integer benchmarks that are difficult to parallelize, in an effort to parallelize them with little or no programmer intervention. In reality, we were usually able to get competitive performance, but often only after the application of one or more manual optimization techniques to improve the usable parallelism of our loops. Explicit synchronization, code motion, and adjustment of the loop structure itself were found to be useful to a greater or lesser degree with a variety of benchmarks. If these optimizations could be automated more in the future, then the use of speculation to help bridge between current uniprocessors and the CMPs of the future may become reality.

We also attempted to use speculation with a selection of applications that were known to have significant amounts of parallelism, generally dense matrix-oriented floating point and multimedia benchmarks. Our selection focused on ones that do not lend themselves to conventional matrix-oriented parallelization because they are coded in such a way that they are difficult to statically analyze. While we occasionally had to apply an optimization to extract the parallelism, in general we were much more successful at achieving very good speedups with little intervention. The speedups obtained were usually not *quite* as good as the conventionally parallelized versions of these programs, due to some of the inherent overheads of the speculation process, but they were still significantly better than efforts to find parallelism in these tasks using techniques such as ILP extraction in a superscalar.

Interestingly enough, when we compare our results with what has been published by other groups, we find that we are in agreement more often than we differ. While individual architectures may vary somewhat, the general trend among speculative architectures is to achieve speedups of 1.1 – 2 on most integer applications, almost regardless of the number of processors thrown at the problem, and more linear-with-processor-count results on the applications with more fundamental parallelism in their algorithms. [13] [14] [10?] This indicates to us that while we may not have thrown as much hardware [13] or compiler technology [14] at the problem than other groups working in the field, we are all doing a fairly effective job of finding the extractable parallelism with our technique, which attempts to balance the use of hardware and software in order to speculate. Another interesting point of comparison is the results we obtained with a highly optimal speculation technique that effectively checkpointed the system at every load and only backed up the work that was absolutely necessary. Even though this system allowed us to achieve nearly optimal memory system behavior under speculation, speedups increased only slightly. These observations lead us to conclude that available parallelism in the code was the limit, and not limitations caused by artifacts in our design. These results tend to validate the fact that our design for a speculation system is fairly reasonable.

Finally, we checked the L2 buffering requirements associated with speculation, as described in Section 7.3. By and large, reasonable thread sizes produced write states of about 1-2 kilobytes. Since the occasional large iteration can be handled by stalling a processor until it becomes the "head" processor, there is little reason to have more than about 2K per processor of buffer space (i.e. we can size buffers for the average, not the extreme). In a system where the typical L1 cache is already 16K, a handful of 2K buffers will not contribute significantly to the overall area of the processor core design.

8 Overall Conclusions and Future Directions

As more transistors are integrated onto larger dies, single-chip multiprocessors integrated with large amounts of cache memory are becoming a feasible alternative to the large, monolithic uniprocessors that dominate today’s microprocessor marketplace. The earlier work of the Hydra group [5][6][7][8] has clearly shown that CMPs are an excellent alternative to other single-chip microprocessor architectures, such as conventional superscalars, as integration densities of chips continue to increase.

The work in this thesis describes and evaluates the design of the Hydra architecture itself [50], instead of CMPs in general, although many of the opinions that we have been able to draw from Hydra can be generalized to other CMPs. Hydra offers a promising way to build a small-scale MP-on-a-chip with a fairly simple design while maintaining excellent performance on a wide variety of applications. Simple buses are used to connect several optimized, single-ported caches together. With the write bus based architecture, no sophisticated coherence protocols are necessary to keep the on-chip caches coherent. Arbitration of key resources and addresses is accomplished with two custom mechanisms that have been highly tuned, although their design could have probably been simplified somewhat while still maintaining good performance. With this balance between cost, complexity, and performance, we believe Hydra offers a promising model for future MP-on-a-chip designs.

However, the baseline design is limited by the fact that there is still only a limited amount of parallel code available that allows an explicitly multithreaded architecture like a CMP to function at maximum effectiveness. Hydra has one major advantage over conventional multiprocessors that we can exploit to make parallelization of code easier. It has very low interprocessor communication latencies of approximately 10 cycles or so, even though there are no special and expensive interprocessor communication links. Thanks to this simple and effective L2 cache communication mechanism, we were able to consider a much simpler parallelization technique that programmers of conventional multiprocessors could never dream of using with the long interprocessor communication delays they must contend with: the nearly automatic parallelization afforded by thread-level speculation.

8.1 Thread-Level Speculation

We have demonstrated that by judicious use of hardware and software mechanisms it is possible to add data speculation capability to a CMP [12][49]. Our results indicate that a data speculation system similar to ours can extract “hidden” parallelism from loops (and occasionally from procedures) in uniprocessor code. It does this by allowing compilers to *obliviously* parallelize loops that cannot be fully analyzed for dependencies at compile time due to problems such as C pointer disambiguation. If there is parallelism already inherent in the way that the program has been coded, as in `ijpeg` or many of the floating-point benchmarks, then the application can speed up significantly. Even if there is not, the application is still guaranteed to work, even though speculation may provide little or no benefit. While there is a potential for speculation to slow down the system, it can be deactivated completely in Hydra, unlike some competing designs [28].

Our first implementation of thread-level speculation attempted to extract parallelism from both procedures and loops, sometimes simultaneously. This architecture is effective at exploiting speculative parallelism when large amounts of parallelism are already present in the application, as our results in Section 7.2 for the floating point benchmarks demonstrate. However, it has difficulty extracting parallelism from conventional integer applications. A key problem with Hydra while it is using our original set of protocol handlers, capable of speculating on loops and procedures, is the high overhead incurred by the speculation software to properly handle procedures. This problem is especially acute with integer applications because the number of overhead-incurring restarts during the speculative parallelization of these applications is higher due to a larger number of dependencies between loop iterations and among procedures. Lastly, procedures typically degrade performance because too much time is spent starting speculative threads for procedures that turn out to have little parallelism.

We have also shown that a few simple optimization techniques can dramatically improve the performance of speculative threads extracted from loops. With all optimizations, we achieved a 60 percent overall performance improvement on eleven benchmarks and a 75 percent performance improvement on several general integer benchmarks. As an initial optimization technique, we eliminated procedures and focused on loop-only speculation, to make our loop iteration threads run as quickly as possible, with much lower speculation software overheads. This effort was very successful and resulted in significant speedups on most of our applications. Violation statistic gathering mechanisms allowed us to identify data-dependent pairs of loads and stores in the benchmarks that frequently resulted in violations. On half of our applications, we were able to use this information to guide several software optimization techniques that modified parts of the original source code. These modifications also improved performance significantly. Finally, we simulated a "hardware checkpointing" scheme that provides nearly optimal performance by eliminating most of the delay caused by frequently violating loads. The performance gains that we were able to achieve with this technique were fairly small, considering the hardware investment that would be necessary to actually implement it, and indicate that our more realistic implementations of Hydra are already able to handle violations fairly quickly and efficiently.

While our experiments to improve loop-based threads worked well, our mechanisms to extract subroutine parallelism were hard to improve. We were hindered by the previously noted high software overhead of the complex control code, the load imbalance caused by running a mix of subroutines of varying sizes, and frequent memory dependencies caused by the side effects of subroutines. `Eqntott` was the only benchmark that achieved significant speedup using procedure speculation. Much of the execution time in `eqntott` is dominated by a recursive `quicksort()` procedure call instead of any sort of iterative loop of reasonable size. By manually limiting procedure speculation to this key `quicksort()` routine, instead of speculating on subroutines blindly, we were able to get a significant improvement. This strongly indicates that procedure speculation is still a viable alternative, but only for environments where it can be used very judiciously. Another example is the Java virtual machine, as we showed in [41]. Also, our results clearly indicate that software control of subroutine speculation only makes sense if the control protocols are extremely simple. Instead, more complex thread-generation and control algorithms clearly demand more sophisticated hardware support, such as that included in the Multiscalar architecture [28]. Except on this particular family of applications, however, our simpler hardware support often did just as well as more sophisticated architectures like Multiscalar.

Overall, our results demonstrate two main conclusions. First, speculation is a viable technique for parallelizing code in a relatively automatic fashion. Our performance results indicate that it is a very sound technique for finding a large amount of the “hidden” parallelism that may exist in sequential programs — when that parallelism actually exists, of course. On many programs, its speedups are comparable to those obtainable by simply building a large superscalar uniprocessor using the same die area [6], while on ones with significant amounts of loop-level parallelism the speedups can be much better. Because of this, a CMP with speculative support is potentially a viable alternative as a replacement for conventional processors. Also, it is always possible to simply deactivate speculation on architectures such as Hydra and run them as conventional multiprocessors when a multitude of non-speculative threads are naturally available. Therefore, the CMP should be able to achieve higher performance on inherently parallel applications, such as floating point numerical code and multimedia applications, than a comparable cost superscalar architecture. Second, there is a promising migration path from our current, simple sequential-to-speculatively parallel conversion tools, such as Hydracat, to more sophisticated compilation tools that can tune and optimize code for a speculatively parallel architecture. Today, these optimizations still require programmer intervention, but it should be relatively easy to automate them significantly, possibly using feedback mechanisms.

8.2 Future Directions

Several directions for follow-on work have been identified as this thesis was written. Some of these involve improving some of the techniques that we already have used in a limited fashion in order to make the use of technologies like speculation much more powerful and increasingly automatic. Others make use of some of the technology in Hydra and extend upon it in new and sometimes unusual ways.

The most obvious area for further development is in the design of the compilers that target Hydra. While we have made tools for automatically parallelizing loops that have been identified by the programmer as good speculation targets, it would be advantageous if the compiler could automatically find loops — and possibly subroutines — by itself. With a profiling compiler, it would seem reasonable to make a system that could "speculatively" speculate on every important loop in a program, obtain statistics on the speedups obtained by various loops, and then deactivate speculation on all but the most promising loops. Unlike most parallelization techniques, speculation lends itself to this type of automatic analysis because it does not require that data be pre-distributed among processor nodes, a task that is notoriously difficult to automate.

A logical extension on the automatic parallelization idea would be to develop new and more powerful parallelization enhancements, and have those applied automatically as well. Many potential enhancements are possible, such as optimizations to speed up routines that are dependent upon recursion or other techniques that the current implementation of speculation does not handle well. Like speculation itself, a large number of these optimizations might potentially be added automatically by the compiler wherever possible and then selectively whittled down via some sort of profiling and performance analysis process.

On the hardware side, logical extensions might look at scaling the Hydra architecture to take advantage of larger numbers of base CPU cores. For a small design (8-16 processors), this would probably only involve re-optimizing the memory system design to deal with the larger number of processors. The write bus would probably need replication — the splitting up of the bus into multiple lanes that could all work simultaneously, allowing each reference to effectively only take a fraction of a clock cycle. These would have to feed into a multi-banked L2 cache to avoid conflict. Some of the centralized control mechanisms like the central arbiter might also need to be distributed more. However, much of the design could remain very similar or essentially identical. For a larger design, however, things would have to change considerably. Many of the techniques applied to earlier distributed shared memory machines such as DASH [56] or FLASH [57] might be applied in these instances. Scalable speculation such as that proposed by the TLDS [38] or Illinois [40] groups might also be used with such a system, too.

Further extensions on the Hydra design could be even more exotic. One proposal being investigated by the Hydra group is to adapt the speculative memory hardware structures in Hydra to build a more general purpose transactional memory system. This can then be used for a variety of purposes such as backing up and restarting code when exceptional conditions or errors are discovered during the execution of the code at runtime. For example, it may be possible to execute threads redundantly on more than one processor and compare the results as they sit in the L2 write buffers, before committal, to ensure that no dynamic runtime errors occurred on either of the processors.

Other technologies may be developed based on Hydra, as well. These brief summaries only offer views of a few of the near-term projects that are taking advantage of possibilities that Hydra has opened up for investigation.

9 References

This section is partially full of entire references, but mostly full of incomplete references that just give me a quick mnemonic to the paper, thesis, or book actually being referenced. Needless to say, all will be fully filled-out before the final version goes to press. << NOTE: Any suggestions to additional references or different, better citations than these would obviously be appreciated. >>

[1] ref to WD Webster thesis & possibly tech report on Tango

[2] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "The SimOS Approach," *IEEE Parallel and Distributed Technology*, vol. 4, no. 3, 1995.

[3] MINT reference

[4] SimpleScalar reference

[5] B. Nayfeh, L. Hammond, and K. Olukotun, "Evaluation of Design Alternatives for a Multiprocessor Microprocessor," *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 67–77, Philadelphia, PA, 1996.

[6] K. Olukotun, K. Chang, L. Hammond, B. Nayfeh, and K. Wilson, "The case for a single chip multiprocessor," *Proceedings of the 7th Int. Conf. for Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 2–11, Cambridge, MA, 1996.

[7] L. Hammond, B. Nayfeh, and K. Olukotun, "A Single Chip Multiprocessor," *IEEE Computer*, September 1997, pp. 79-85.

[8] Basem's thesis

[9] SGI/IRIX reference

[10] Illinois paper describing inter-register communication bus with backward-branch detection, about 1998, in my drawer.

[11] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," *Proceedings of 25th Annual International Symposium of Computer Architecture*, pp. 142–153, Barcelona, Spain, June 1998.

[12] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," *Proceedings of Eighth International Conference on Architectural Support for Pro-*

gramming Languages and Operating Systems (ASPLOS VIII), pp. 58–69, San Jose CA, October 1998.

[13] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, “Dynamic speculation and synchronization of data dependences,” *Proceedings of 24th Annual International Symposium of Computer Architecture*, pp. 181–193, Denver, CO, June 1997.

[14] J. G. Steffan and T. Mowry, “The potential for using thread-level data speculation to facilitate automatic parallelization,” *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, <<PAGES>>, Las Vegas, NV, February 1998.

[15] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition*, San Francisco, CA: Morgan Kaufmann, 1996.

[16] Y. Patt, “First Let’s Get the Uniprocessor Right,” *Microprocessor Report*, August 5, 1996, pp. 23–24.

[17] VLIW-EPIC reference

[18] D. Tullsen, S. Eggers, and H. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” *Proceedings of the 22nd Annual International Symposium of Computer Architecture*, pp. 392–403, <<LOCATION>>, <<MONTH>> 1995.

[19] IRAM reference

[20] Wiring ref. From 1B Computer issue

[21] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy, *The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler*, Stanford University Computer Systems Laboratory Technical Report No. CSL-TR-94-620, May 1994.

[22] DEC 21164 reference (µP Report or otherwise)

[23] DEC 21264 / 21364 reference (µP Report, ISSCC)

[24] IBM Power4 reference (probably µP Report, maybe multiple)

[25] Sun MAJC reference (Hot Chips or otherwise)

[26] Reference (HC 3??) describing limitations of parallelizing compilers with C

[27] T. Knight, “An architecture for mostly functional languages,” *Proceedings of the ACM Lisp and Functional Programming Conference*, pp. 500–519, August 1986.

[28] M. Franklin and G. S. Sohi, “The expandable split window paradigm for exploiting fine-

grain parallelism," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 58–67, Gold Coast, Australia, May 1992.

[29] TI CMP-DSP from 1994 in µP Report

[30] Piranha reference at ISCA or otherwise

[31] Reference to one (or more, if desired) of the microcontroller-DSP combinations that have appeared in the past few years. Motorola, ARM, etc. could all be sources.

[32] T. Koyama, E. Iwata, H. Yoshikawa, H. Hanaki, K. Hasegawa, M. Aoki, M. Yasue, T. Schrobenhauser, M. Aikawa, I. Kumata, and H. Koyanagi, "A 250 MHz Single-Chip Multiprocessor for A/V Signal Processing," *2001 International Solid-State Circuits Conference Digest of Technical Papers*, pp, 146–147, San Francisco, CA, February 2001.

[33] IDT core reference manual

[34] Reference to one (or more, if desired) of the on-chip L2 cache processors released in recent years (most notably various Intel P6es, AMD K6s and Athlons, and latest IBM PPC G3)

[35] M. Franklin and G. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, May 1996.

[36] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi, "Speculative Versioning Cache," *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, <<PAGES>>, Las Vegas, NV, February 1998,.

[37] More recent TLDS reference, perhaps a thesis or tech report, describing low-level design

[38] ISCA 2000 TLDS paper

[39] Fairly recent Illinois reference, perhaps a thesis or tech report

[40] Illinois ISCA 2000 paper

[41] M. Chen and K. Olukotun, "Exploiting method-level parallelism in single-threaded Java programs," Proceedings of Parallel Architectures and Compilation Techniques (PACT 98), pp. 176–184, Paris, France, October 1998.

[42] Jouppi victim cache paper that we're always referencing

[43] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun, *Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors,* Stanford University Computer Systems Laboratory Technical Report CSL-TR-98-765, July 1998.

[44] E. Iwata and K. Olukotun, *Exploiting coarse-grain parallelism in the MPEG-2 Algorithm,* Stanford University Computer Systems Laboratory, Technical Report CSL-TR-98-771,

September 1998.

[45] Transactional memory paper that I read eons ago (in my drawer)

[46] C. Lee, M. Potkonjak, and W. Mangione-Smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communications systems,” *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.

[47] S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang, and W. Lee, “Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor,” *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 306–317, Barcelona, Spain, June 1998.

[48] SPEC95 reference

[49] K. Olukotun, L. Hammond, and M. Willey, “Improving the Performance of Speculatively Parallel Applications on the Hydra CMP,” Proceedings of the 1999 ACM International Conference on Supercomputing, <<PAGES>>, <<LOCATION>>, <<MONTH>> 1999.

[50] L. Hammond and K. Olukotun, Considerations in the Design of Hydra: a Multiprocessor-on-a-Chip Microarchitecture, Stanford University Computer Systems Laboratory Technical Report CSL-TR-98-749, February 1998.

[51] B. Kernighan and R. Pike, *The Practice of Programming*, Reading, Massachusetts: Addison-Wesley, 1999.

[52] W. H. Press, S. A. Teulosky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge, England: Cambridge University Press, 1992.

[53] SPEC92 reference

[54] TPC-B reference

[55] D. E. Culler, J. P. Singh, with A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, San Francisco, CA: Morgan Kaufmann Publishers, 1999.

[56] DASH reference, possibly Lenoski thesis

[57] FLASH reference, possibly Kuskin thesis or ISCA 94 paper

[58] P. Sindhu, J.-M. Frailong, and M. Cekleov, *Formal Specification of Memory Models*, Technical Report (PARC) CSL-91-11, Xerox Corp. Palo Alto Research Center, Palo Alto, CA, 1991.

[59] SUN Microsystems, *The SPARC Architecture Manual*, #800-199-12, Version 8 (January),

Mountain View, CA: SUN Microsystems, 1991.

[60] M. Papamarcos and J. Patel, “A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories,” *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 348–354, June 1984.

[61] IBM Simple Pipeline, running at a GHz (probably refer to ISSCC or µP article)

[62] J. Oplinger, D. Heine, M. Lam, and K. Olukotun, *In Search of Speculative Thread-Level Parallelism*, << CHECK FINAL LOCATION >>.

[63] G. Sohi, S. Breach, and T. Vijaykumar, “Multiscalar Processors,” *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425, Ligure, Italy, June 1995.

[64] more . . . ?