THE STANFORD HYDRA CMP

CHIP MULTIPROCESSORS OFFER AN ECONOMICAL, SCALABLE ARCHITECTURE FOR FUTURE MICROPROCESSORS. THREAD-LEVEL SPECULATION SUPPORT ALLOWS THEM TO SPEED UP PAST SOFTWARE.

Lance Hammond
Benedict A. Hubbert
Michael Siu
Manohar K. Prabhu
Michael Chen
Kunle Olukotun
Stanford University

The Hydra chip multiprocessor (CMP) integrates four MIPS-based processors and their primary caches on a single chip together with a shared secondary cache. A standard CMP offers implementation and performance advantages compared to wide-issue superscalar designs. However, it must be programmed with a more complicated parallel programming model to obtain maximum performance. To simplify parallel programming, the Hydra CMP supports thread-level speculation and memory renaming, a paradigm that allows performance similar to a uniprocessor of comparable die area on integer programs. This article motivates the design of a CMP, describes the architecture of the Hydra design with a focus on its speculative thread support, and describes our prototype implementation.

Why build a CMP?

As Moore's law allows increasing numbers of smaller and faster transistors to be integrated on a single chip, new processors are being designed to use these transistors effectively to improve performance. Today, most microprocessor designers use the increased transistor budgets to build larger and more complex uniprocessors. However, several problems are beginning to make this approach to microprocessor design difficult to continue. To address these problems, we have proposed that future processor design methodology shift from simply making progressively larger uniprocessors to implementing more than one processor on each chip.[1] The following discusses the key reasons why single-chip microprocessors are a good idea.

Parallelism

Designers primarily use additional transistors on chips to extract more parallelism from programs to perform more work per clock cycle. While some transistors are used to build wider or more specialized data path logic (to switch from 32 to 64 bits or add special multimedia instructions, for example), most are used to build superscalar processors. These processors can extract greater amounts of instruction-level parallelism, or ILP, by finding nondependent 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 the processor executes because instructions from the same sequence are typically highly interdependent. As a result, processors that use this technique are seeing diminishing returns as they attempt to execute more instructions per clock cycle, even as the logic required to process multiple instructions per clock cycle increases quadratically. A CMP avoids this limitation by primarily using a completely different type of parallelism: thread-level parallelism. We obtain TLP by running completely separate sequences of instructions on each of the separate processors simultaneously. Of course, a CMP may also exploit small amounts of ILP within each of its individual processors, since ILP and TLP are orthogonal to each other.

Wire delay

As CMOS gates become faster and chips become physically larger, the delay caused by interconnects between gates is becoming more

0272-1732/00/$10.00 © 2000 IEEE

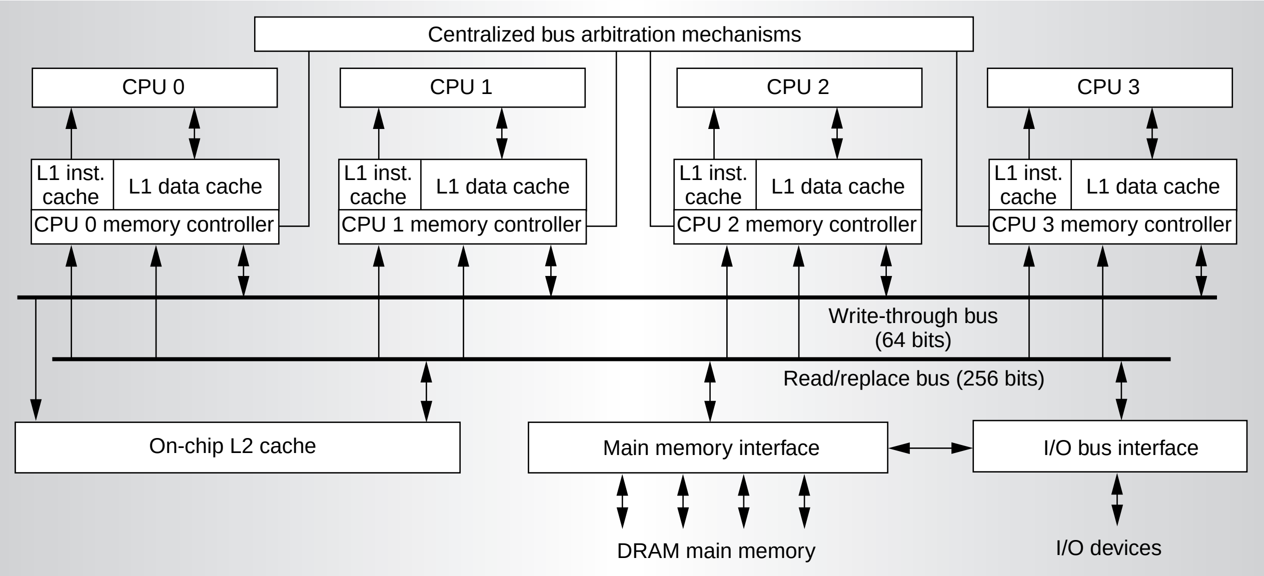

Figure 1. An overview of the Hydra CMP.

significant. Due to rapid process technology improvement, within the next few years wires will only be able to transmit signals over a small portion of large processor chips during each clock cycle.[2] However, a CMP can be designed so that each of its small processors takes up 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 the processors need to be long.

Design time

Processors are already difficult to design. Larger numbers of transistors, increasingly complex methods of extracting ILP, and wire delay considerations will only make this worse. A CMP can help reduce design time, however, because it allows a single, proven processor design to be replicated multiple times over a die. Each processor core on a CMP can be much smaller than a competitive uniprocessor, minimizing the core design time. Also, a core design can be used over more chip generations simply by scaling the number of cores present on a chip. Only the processor interconnection logic is not entirely replicated on a CMP.

Why aren't CMPs used now?

Since a CMP addresses all of these potential problems in a straightforward, scalable manner, why aren't CMPs already common? 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 been only partially successful at automatically handling these tasks for programmers.[3] As a result, acceptance of multiprocessors has been slowed because only a limited number of programmers have mastered these techniques.

Base Hydra design

To understand the implementation and performance advantages of a CMP design, we are developing the Hydra CMP. Hydra is a CMP built using four MIPS-based cores as its individual processors (see Figure 1). Each core has its own pair of primary instruction and data caches, while all processors share a single, large on-chip secondary cache. The processors support normal loads and stores plus the MIPS load locked (LL) and store conditional (SC) instructions for implementing synchronization primitives.

Connecting the processors and the secondary cache together are the read and write buses, along with a small number of address and control buses. In the chip implementation, almost all buses are virtual buses. While they logically act like buses, the physical wires are divided into multiple segments using repeaters and pipeline buffers, where necessary, to avoid slowing down the core clock frequencies.

The read bus acts as a general-purpose system bus for moving data between the processors, secondary cache, and external interface to off-chip memory. It is wide enough to handle an entire cache line in one clock cycle. This is an advantage possible with an on-chip bus that all but the most expensive multichip systems cannot match due to the large number of pins that would be required on all chip packages.

The narrower write bus is devoted to writing all writes made by the four cores directly to the secondary cache. This allows the permanent machine state to be maintained in the secondary cache. The bus is pipelined to allow single-cycle occupancy by each write, preventing it from becoming a system bottleneck. The write bus also permits Hydra to use a simple, invalidation-only coherence protocol to maintain coherent primary caches. Writes broadcast over the bus invalidate copies of the same line in primary caches of the other processors. No data is ever permanently lost due to these invalidations because the permanent machine state is always maintained in the secondary cache.

The write bus also enforces memory consistency in Hydra. Since all writes must pass over the bus to become visible to the other processors, the order in which they pass is globally acknowledged to be the order in which they update shared memory.

We were primarily concerned with minimizing two measurements of the design: the complexity of high-speed logic and the latency of interprocessor communication. Since decreasing one tends to increase the other, a CMP design must strive to find a reasonable balance. Any architecture that allows interprocessor communication between registers or the primary caches of different processors will add complex logic and long wires to paths that are critical to the cycle time of the individual processor cores. Of course, this complexity results in excellent interprocessor communication latencies—usually just one to three cycles. Past results have shown that sharing this closely is helpful, but not if it extends the access time to the registers and/or primary caches. Consequently, we chose not to connect our processors this tightly. On the other hand, these results also indicated that we would not want to incur the delay of an off-chip reference, which can often take 100 or more cycles in modern processors during each interprocessor communication.

Because it is now possible to integrate reasonable-size secondary caches on processor dies and since these caches are typically not tightly connected to the core logic, we chose to use that as the point of communication. In the Hydra architecture, this results in interprocessor communication latencies of 10 to 20 cycles, which are fast enough to minimize the performance impact from communication delays. After considering the bandwidth required by four single-issue MIPS processors sharing a secondary cache, we concluded that a simple bus architecture would be sufficient to handle the bandwidth requirements for a four. This is acceptable for a four- to eight-processor Hydra implementation. However, designs with more cores or faster individual processors may need to use either more buses, crossbar interconnections, or a hierarchy of connections.

Parallel software performance

We have performed extensive simulation to evaluate the potential performance of the Hydra design. Using a model with the memory hierarchy summarized in Table 1 (next page), we compared the performance of a single Hydra processor to the performance of all four processors working together. We used the 10 benchmarks summarized in Table 2 to generate the results presented in Figure 2 (p. 75).

The results indicate that for multiprogrammed workloads and highly parallel benchmarks such as large matrix-based or multimedia applications, we can obtain nearly linear speedup by using multiple Hydra processors

Table 1. The Hydra system configuration used for our simulations.

Characteristic	L1 cache	L2 cache	Main memory
Configuration	Separate I and D SRAM cache pairs for each CPU	Shared, on-chip SRAM cache	Off-chip DRAM
Capacity	16 Kbytes each	2 Mbytes	128 Mbytes
Bus width	32-bit connection to CPU	256-bit read bus + 32-bit write bus	64-bit bus at half 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-Kbyte pages
Write policy	Write through, no allocate on write	Write back, allocate on writes	Write back (virtual memory)
Inclusion	N/A	Inclusion enforced by L2 on L1 caches	Includes all cached data

Table 2. A summary of the conventionally parallelized applications we used to make performance measurements with Hydra.

Purpose	Application	Source	Description	How parallelized
General uniprocessor applications	compress	SPEC95	Entropy-encoding file compression	Not possible using conventional means
	eqntott	SPEC92	Logic minimization	On inner bit vector comparison loop
	m88ksim	SPEC95	CPU simulation of Motorola 88000	Simulated CPU is pipelined across processors
	apsi	SPEC95	Weather and air pollution modeling	Automatically by the SUIF compiler
Matrix and multimedia applications	MPEG2	Mediabench suite	Decompression of an MPEG-2 bitstream	"Slices" in the input bitstream distributed among processors
	applu	SPEC95	Solver for partial differential equations	Automatically by the SUIF compiler
	swim	SPEC95	Grid-based finite difference modeling	Automatically by the SUIF compiler
	tomcatv	SPEC95	Mesh generation	Automatically by the SUIF compiler
Multiprogrammed workloads	OLTP	TPC-B	Database transaction processing	Different transactions execute in parallel
	pmake	Unix command	Parallel compilation of several source files	Compilations of different files execute in parallel

working together. These speedups typically will be 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.[1] In addition, multiprogrammed workloads are inherently parallel, while today's compilers can automatically parallelize most dense matrix Fortran applications.[3]

However, there is still a large category of less parallel applications, primarily integer ones that are not easily parallelized (*eqntott*, *m88ksim*, and *apsi*). The speedups we obtained with Hydra on these applications would be difficult or impossible to achieve on a conventional multiprocessor, with the long interprocessor communication latencies required by a multichip design. Even on Hydra, the speed improvement obtained after weeks of hand-parallelization is just comparable to that obtainable with a similar-size ILP processor with no programmer effort. More troubling, *compress* represents a large group of applications that cannot be parallelized at all using conventional techniques.

Thread-level speculation: A helpful extension

Applications such as database and Web servers

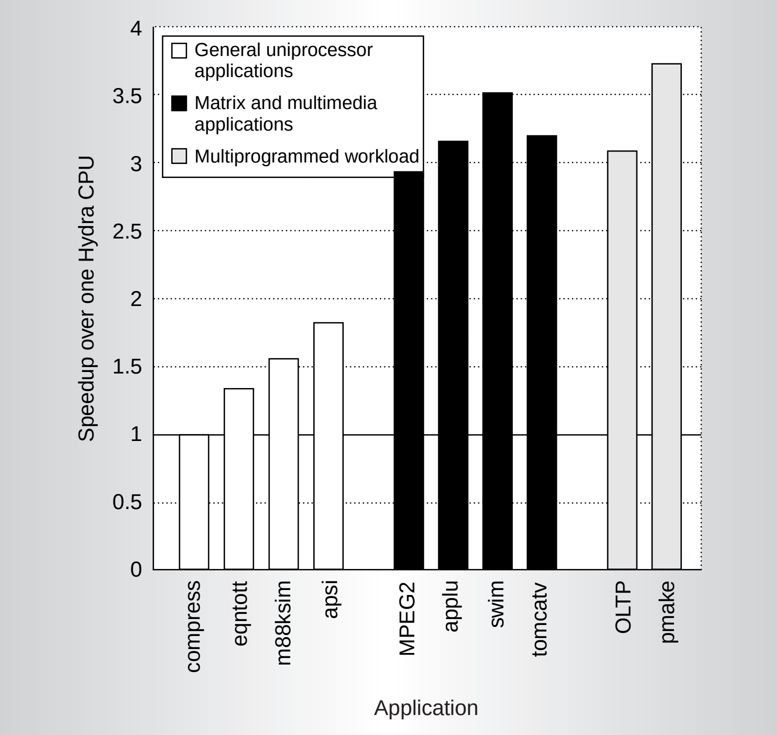

Figure 2. Speedup of conventionally parallelized applications running on Hydra compared with the original uniprocessor code running on one of Hydra's four processors.

perform well on conventional multiprocessors, and therefore these applications will provide the initial motivation to adopt CMP architectures, at least in the server domain. However, general uniprocessor applications must also work well on CMP architectures before they can ever replace uniprocessors in most computers. Hence, there needs to be a simple, effective way to parallelize even these applications. Hardware support for thread-level speculation is a promising technology that we chose to add to the basic Hydra design, because it eliminates the need for programmers to explicitly divide their original program into independent threads.

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. To ensure that each program executes the same way that it did originally, hardware must track all interthread dependencies. When a "later" thread in the sequence causes a true dependence violation by reading data too early, the hardware must ensure that the misspeculated thread—or at least the portion of it following the bad read—re-executes with the proper data. This is a considerably different mechanism from the one used to enforce dependencies on conventional multiprocessors. There, synchronization is inserted so that threads reading data from a different thread will *stall* until the correct value has been written. This process is complex because it is necessary to determine *all possible* true dependencies in a program before synchronization points may be inserted.

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 while the program is executing. This greatly simplifies the parallelization of programs because it eliminates the need for human programmers or compilers to statically place synchronization points into programs by hand or at compilation. All places where synchronization would 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. The speculative hardware will ensure that the parallel code always performs the same computation as the original sequential 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 many potential dependencies that may result in a true dependency, but where dependencies seldom if ever actually occur during the execution of the program. A speculative system may attempt to run the threads in parallel anyway, and only back up the later thread if 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. Routines that modify data objects through pointers in C programs are a

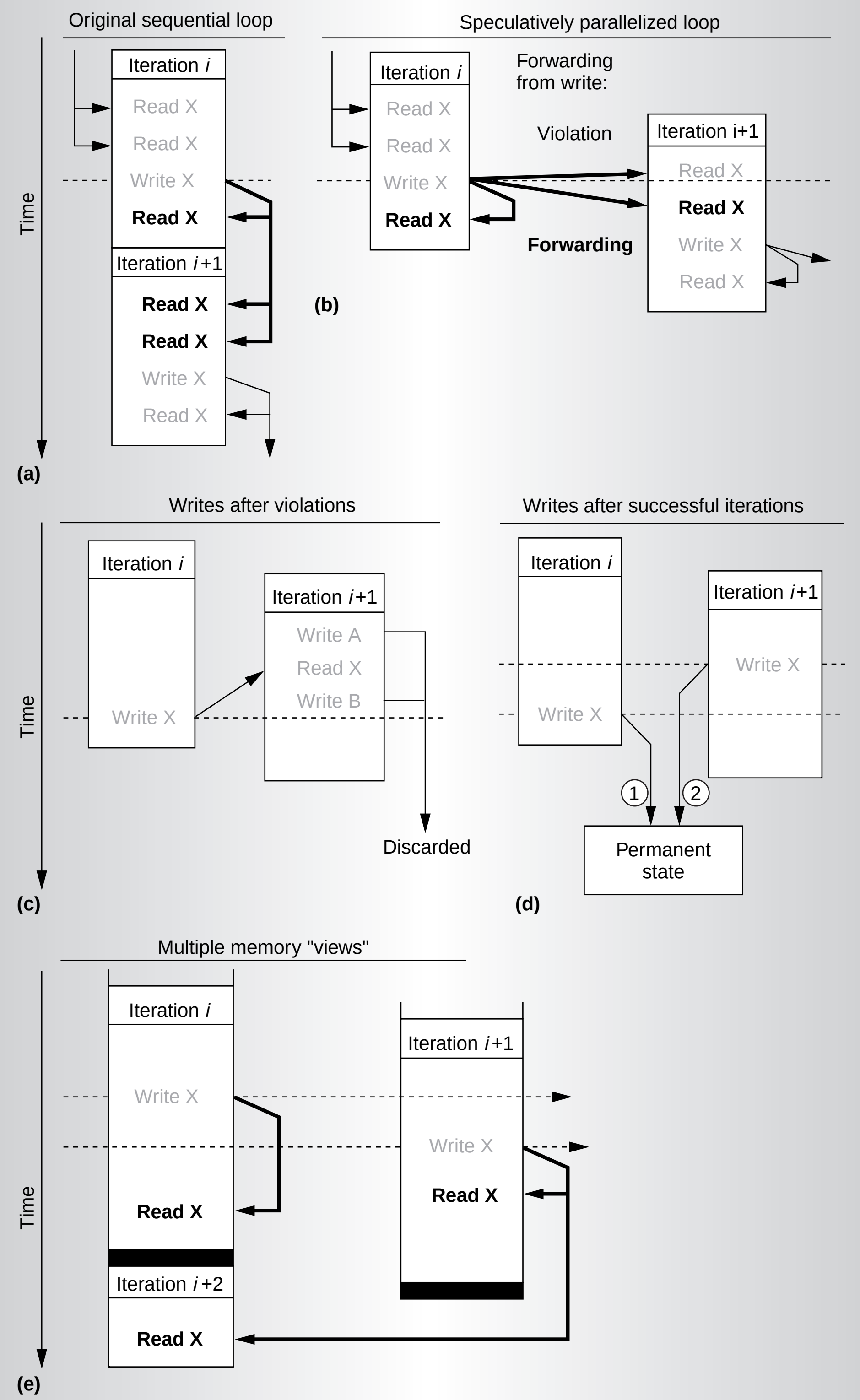

Figure 3. Five basic requirements for special coherency hardware: a sequential program that can be broken into two threads (a); forwarding and violations caused by interaction of reads and writes (b); speculative memory state eliminated following violations (c); reordering of writes following thread commits (d); and memory renaming among threads (e).

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 is written, and therefore wasted as a compiler conservatively parallelizes a program.

Note 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, a programmer or feedback-driven compiler can still add synchronization into a speculatively parallelized program if that helps the program execute faster. In our experiments, we found a few cases where synchronization protecting one or two key dependencies in a speculatively parallelized program produced speedup by dramatically reducing the number of violations that occurred. Too much synchronization, however, tended to make the speculative parallelization too conservative and was a detriment to performance.

To support speculation, we need special coherency hardware to monitor data shared by the threads. This hardware must fulfill five basic requirements, illustrated in Figure 3.

The figure shows some typical data access patterns in two threads, *i* and *i* + 1. Figure 3a shows how data flows through these accesses when the threads are run sequentially on a normal uniprocessor. Figures 3b-d show how the hardware must handle the requirements.

1. *Forward data between parallel threads.* While good thread selection can minimize the data shared among threads, typically a significant amount of sharing is required, simply because the threads are normally generated from a program in which minimizing data sharing was not a design goal. 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. Figure 3b depicts this.
2. *Detect when reads occur too early (RAW hazards).* The speculative hardware must provide a mechanism for tracking reads and writes to the shared data memory. If a data value is read by a later thread and subsequently written by an earlier thread, the hardware must notice that the 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. See Figure 3b.
3. *Safely discard speculative state after violations.* As depicted in Figure 3c, 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, while no permanent machine state may be lost in the process.
4. *Retire speculative writes in the correct order (WAW hazards).* 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 3d illustrates.
5. *Provide memory renaming (WAR hazards).* Figure 3e depicts an earlier thread reading an address after a later processor has already written it. 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. This process is complicated by the fact that each processor will eventually be running newly generated threads (*i* + 2 in the figure) that *will* need to "see" the changes.

In some proposed speculative hardware, the logic enforcing these requirements monitors both the processor registers and the memory hierarchy.[4] However, in Hydra we chose to have hardware only enforce speculative coherence on the memory system, while software handles register-level coherence.

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 hardware and software that finds good places in a program to create new, speculative threads. The system 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. Initial program counter positions (and, for Hydra, register states) must be generated when threads are started. As a result, we investigated two ways to divide a program into threads: loops and subroutine calls. With loops, several iterations of a loop body can be started speculatively on multiple processors. As long as there are only a few straightforward loop-carried dependencies, the execution of loop bodies on different processors can be overlapped to achieve speedup. Using subroutines, a new thread can start to run the code following a subroutine call's return, while the original thread actually executes the subroutine itself (or vice-versa). As long as the return value from the subroutine is predictable (typically, when there is no return value) and any side effects of the subroutine are not used

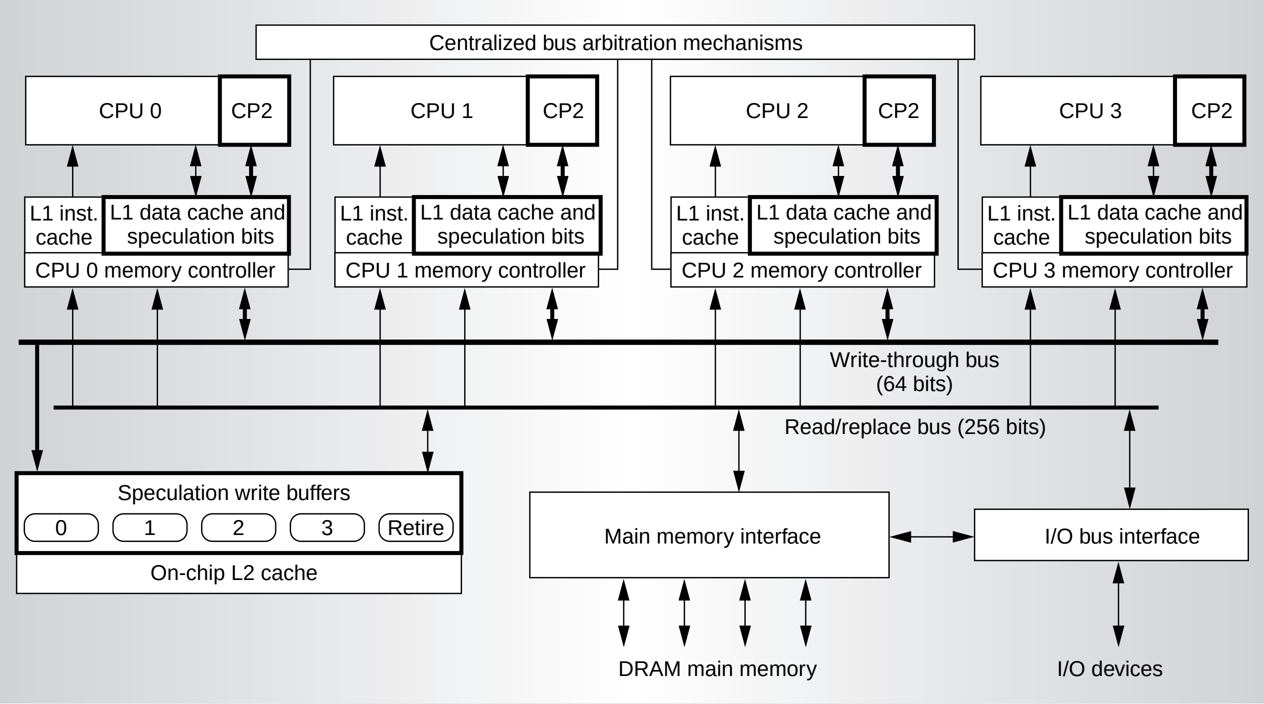

Figure 4. An overview of Hydra with speculative support.

immediately, the two threads can run in parallel. In general, achieving speedup with this technique is more challenging because thread sequencing and load balancing among the processors is more complicated with subroutines than loops.

Once threads have been created, the speculation runtime system must select the four least speculative threads available and allocate them to the four processors in Hydra. Note 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 can handle events that cannot normally be handled speculatively (such as operating system calls and exceptions). Since all threads eventually become the head thread, simply stalling a thread until it becomes the head will allow the thread to process these events during speculation.

Implementing speculation in Hydra

Speculation is an effective method for breaking an existing uniprocessor program into multiple threads. However, the threads created automatically by speculation often require fast interprocessor communication of large amounts of data. After all, minimizing communication between arbitrarily created threads is not a design consideration in most uniprocessor code. A CMP like Hydra is necessary to provide low-enough interprocessor communication latencies and high-enough interprocessor bandwidth to allow the design of a practical speculative thread mechanism.

Among CMP designs, Hydra is a particularly good target for speculation because it has write-through primary caches that allow all processor cores to snoop on all writes performed. This is very helpful in the design of violation-detection logic. Figure 4 updates Figure 1, noting the necessary additions. The additional hardware is enabled or bypassed selectively by each memory reference, depending upon whether a speculative thread generates the reference.

Most of the additional hardware is contained in two major blocks. The first is a set of additional tag bits added to each primary cache line to track whether any data in the line has been speculatively read or written. The second is a set of write buffers that hold speculative writes until they can be safely committed into the secondary cache, which is guaranteed to hold only nonspeculative data.

One buffer is allocated to each speculative thread currently running on a Hydra processor, so the writes from different threads are always kept separate. Only when speculative threads complete successfully are the contents of these buffers actually written into the secondary cache and made permanent. As shown in Figure 4, one or more extra buffers may be included to allow buffers to be drained into the secondary cache in parallel with speculative execution on all of the CPUs. We have previously published[5] more details about the additional primary cache bits and secondary cache buffers.

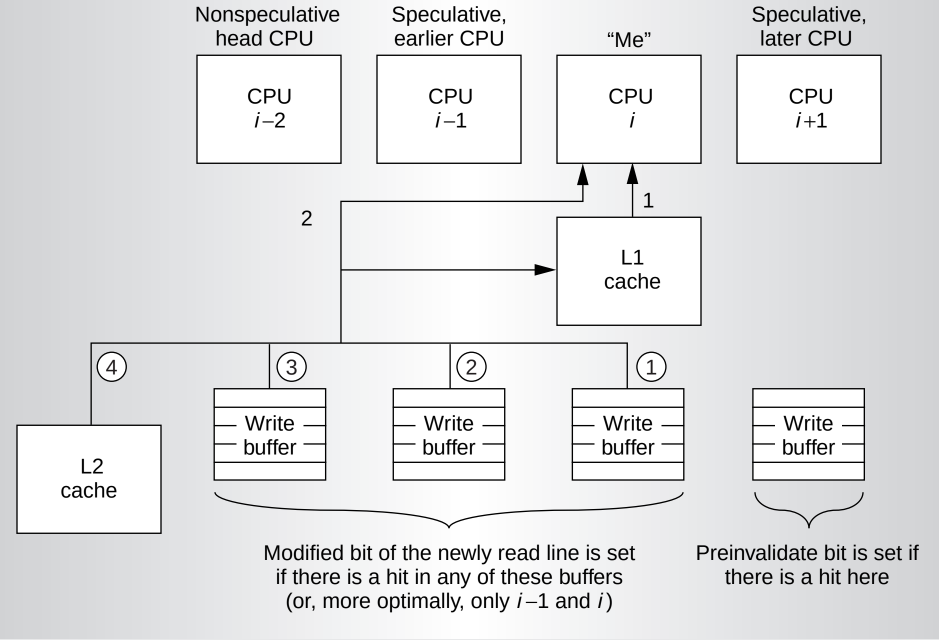

Figure 5. How secondary cache speculative buffers are read. 1) A CPU reads from its L1 cache. The L1 read bit of any hit lines are set. 2) The L2 and write buffers are checked in parallel in the event of an L1 miss. Priority encoders on each byte (indicated by priorities 1-4 here) pull in the newest bytes written to a line. The appropriate word is delivered to the CPU and L1 with the L1 modified and preinvalidate bits set appropriately.

To control the thread sequencing in our system, we also added a small amount of hardware to each core using the MIPS coprocessor interface. These simple "speculation coprocessors" consist of several control registers, a set of duplicate secondary cache buffer tags, a state machine to track the current thread sequencing among the processors, and interrupt logic that can start software handlers when necessary to control thread sequencing. These software handlers are responsible for thread control and sequencing. Prior publications[5,6] provide complete details of how these handlers work for sequence speculative threads in the Hydra hardware.

Together with the architecture of Hydra's existing write bus, the additional hardware allows the memory system to handle the five memory system requirements outlined previously in the following ways:

1. *Forward data between parallel threads.* When a speculative thread writes data over the write bus, all more-speculative threads that may need the data have their current copy of that cache line invalidated. This is similar to the way the system works during nonspeculative operation. If any of the threads subsequently need the new speculative data forwarded to them, they will miss in their primary cache and access the secondary cache. At this point, as is outlined in Figure 5, the speculative data contained in the write buffers of the current or older threads replaces data returned from the secondary cache on a byte-by-byte basis just before the composite line is returned to the processor and primary cache. Overall, this is a relatively simple extension to the coherence mechanism used in the baseline Hydra design.
2. *Detect when reads occur too early.* Primary cache bits are set to mark any reads that may cause violations. Subsequently, if a write to that address from an earlier thread invalidates the address, a violation is detected, and the thread is restarted.
3. *Safely discard speculative state after violations.* Since all permanent machine state in Hydra is always maintained within the secondary cache, anything in the primary caches may be invalidated at any time without risking a loss of permanent state. As a result, any lines in the primary cache containing speculative data (marked with

Other CMP and TLS efforts

Several other research groups have investigated the design of chip multiprocessors and the implementation of thread-level speculation mechanisms. While there are too many to mention all here, a few of the most important follow:

- *Wisconsin Multiscalar.*[4,7] This CMP design proposed the first reasonable hardware to implement TLS. Unlike Hydra, Multiscalar implements a ringlike network between all of the processors to allow direct register-to-register communication. Along with hardware-based thread sequencing, this type of communication allows much smaller threads to be exploited at the expense of more complex processor cores. The designers proposed two different speculative memory systems to support the Multiscalar core. The first was a unified primary cache, or address resolution buffer (ARB). Unfortunately, the ARB has most of the complexity of Hydra's secondary cache buffers at the primary cache level, making it difficult to implement. Later, they proposed the speculative versioning cache. The SVC uses write-back primary caches to buffer speculative writes in the primary caches, using a sophisticated coherence scheme.
- *Carnegie-Mellon Stampede.*[8] This CMP-with-TLS proposal is very similar to Hydra, including the use of software speculation handlers. However, the hardware is simpler than Hydra's. The design uses write-back primary caches to buffer writes—similar to those in the SVC—and sophisticated compiler technology to explicitly mark all memory references that require forwarding to another speculative thread. Their simplified SVC must drain its speculative contents as each thread completes, unfortunately resulting in heavy bursts of bus activity.
- *MIT M-machine.*[9] This one-chip CMP design has three processors that share a primary cache and can communicate register-to-register through a crossbar. Each processor can also switch dynamically among several threads. As a result, the hardware connecting processors together is quite complex and slow. However, programs executed on the M-machine can be parallelized using very fine-grain mechanisms that are impossible on an architecture that shares outside of the processor cores, like Hydra. Performance results show that on typical applications extremely fine-grained parallelization is often not as effective as parallelism at the levels that Hydra can exploit. The overhead incurred by frequent synchronizations reduces the effectiveness.

Recently, Sun and IBM announced plans to make CMPs. Sun's plans offer limited TLS support.

- *IBM Power4.*[10] This is the first commercially proposed CMP targeted at servers and other systems that already make use of conventional multiprocessors. It resembles Hydra but does not have TLS support (an unnecessary feature for most types of servers) and has two large processors per chip.
- *Sun MAJC.*[11] This CMP with a shared primary cache is designed to support Java execution. It also supports a subroutine-based TLS scheme. MAJC has interrupt hardware similar to that in our speculative coprocessors, but it implements speculative memory using only software handlers invoked during the execution of each sharable load or store. This is possible with Java, the target language, because the rare loads and stores to global objects that can be shared among subroutines are clearly defined in Java bytecode binaries. However, even with the shared primary cache allowing quick interprocessor communication of changes to the speculation-control memory structures and the low percentage of shared memory accesses, it may be difficult for this scheme to scale to more than two processors due to the overhead of the software handlers.

a special modified bit) may simply be invalidated all at once to clear any speculative state from a primary cache. In parallel with this operation, the secondary cache buffer for the thread may be emptied to discard any speculative data written by the thread without damaging data written by other threads or the permanent state of the machine in the secondary cache.

4. *Retire speculative writes in the correct order.* Separate secondary cache buffers are maintained for each thread. As long as these are drained into the secondary cache in the original program sequence of the threads, they will reorder speculative memory references correctly. The thread-sequencing system in Hydra also sequences the buffer draining, so the buffers can meet this requirement.
5. *Provide memory renaming.* Each processor can only read data written by itself or earlier threads when reading its own primary cache or the secondary cache buffers. Writes from later threads don't cause immediate invalidations in the primary cache, since these writes should not be visible to earlier threads. This allows each primary cache to have its own local copy of a particular line. However, these "ignored" invalidations are recorded using an additional pre-invalidate primary cache bit associated with each line. This is because they must be processed before a different speculative or nonspeculative thread executes on this processor. If a thread has to load a cache line from the secondary cache, the line it recovers only contains data that it should actually be able to "see," from its own and earlier buffers, as Figure 5 indicates. Finally, if future threads have written to a particular line in the primary cache, the pre-invalidate bit for that line is set. When the current thread completes, these bits allow the processor to quickly simulate the effect of all stored invalidations caused by all writes from later processors all at once, before a new thread begins execution on this processor.

Based on the amount of memory and logic required, we estimate that the cost of adding speculation hardware is comparable to adding an additional pair of primary caches to the system. This enlarges the Hydra die only by a few percent.

Table 3. A summary of the speculatively parallelized applications used to make performance measurements with Hydra. Applications in italics were also hand-parallelized and run on the base Hydra design.

Application	Source	Description	How parallelized
compress	SPEC95	Entropy-encoding compression of a file	Speculation on loop for processing each input character
eqntott	SPEC92	Logic minimization	Subroutine speculation on core quick sort routine
grep	Unix command	Finds matches to a regular expression in a file	Speculation on loop for processing each input line
m88ksim	SPEC95	CPU simulation of Motorola 88000	Speculation on loop for processing each instruction
wc	Unix command	Counts the number of characters, words, and lines in a file	Speculation on loop for processing each input character
ijpeg	SPEC95	Compression of an RGB image to a JPEG file	Speculation on several different loops used to process the image
MPEG2	Mediabench suite	Decompression of an MPEG-2 bitstream	Speculation on loop for processing slices
alvin	SPEC92	Neural network training	Speculation on 4 key loops
cholesky	Numeric recipes	Cholesky decomposition and substitution	Speculation on main decomposition and substitution loops
ear	SPEC92	Inner ear modeling	Speculation on outer loop of model
simplex	Numeric recipes	Linear algebra kernels	Speculation on several small loops

Speculation performance

We extended our model of the Hydra system with speculation support to verify our thread-level speculation mechanisms. On all applications except *eqntott*—which was parallelized using subroutine speculation—we used our source-to-source loop-translating system to convert loops in the original programs to their speculative forms. Even with our simple, early programming environment, we could parallelize programs just by picking out which loops and/or subroutines we wanted speculatively parallelized. We then let the tools do the rest of the work for us.

Both because this design environment is C-based and Fortran programs can often be automatically parallelized using compilers such as SUIF, we limited our set of applications to a wide variety of integer and floating-point C programs. Table 3 lists them. Many of these programs are difficult or impossible to parallelize using conventional means due to the presence of frequent true dependencies. However, all of the more highly parallel applications listed in Table 3 can be parallelized by hand. Still, automatically parallelizing compilers are stymied by the presence of many C pointers in the original source code that they cannot statically disambiguate at compile time.

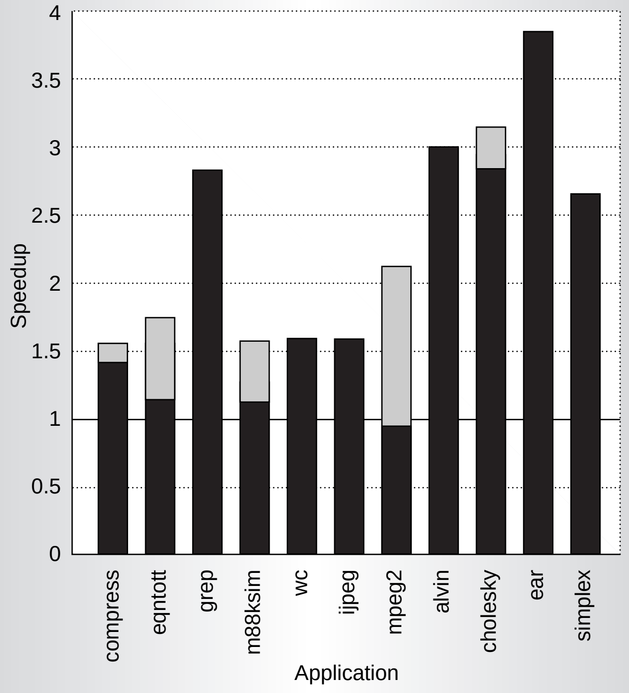

Figure 6. Speedup of speculatively parallelized applications running on Hydra compared with the original uniprocessor code running on one of Hydra's four processors. The gray areas show the improved performance following tuning with feedback-based code.

Figure 6 summaries our results. After our initial speculative runs with unmodified loops from the original programs, we used feedback from our first simulations to optimize our benchmarks by hand. This avoided the most critical violations that caused large amounts of work to be discarded during restarts. These optimizations were usually minor—usually just moving a line of code or two or adding one synchronization point. [6] However, they had a dramatic impact on benchmarks such as MPEG2.

Overall, these results are at least comparable to and sometimes better than a single large uniprocessor of similar area running these applications, based on our past work simulating CMPs and uniprocessors.[1]

Of course, a CMP can also perform faster by running fully parallelized programs without speculation, when those programs are available. A uniprocessor cannot. It is even possible to mix and match using multiprogramming. For example, two processors could be working together on a speculative application, while others work on a pair of completely different jobs. While we have not fully implemented it in our prototype, we could relatively easily enhance the speculative support routines so that multiple speculative tasks could run simultaneously. Two processors would run one speculative program, and two would run a completely different speculative program. In this manner, it is possible for a CMP to nearly always outperform a large uniprocessor of comparable area.

Speedups are only a part of the story, however. Speculation also makes parallelization much easier, because a parallelized program that is guaranteed to work exactly like the uniprocessor version can be generated automatically. As a result, programmers only need to worry about choosing which program sections should be speculatively parallelized and tweaked for performance optimization. Even when optimization is required, we found that speculative parallelization typically took a single programmer a day or two per application. In contrast, hand parallelization of these C benchmarks typically took one programmer anywhere from a week to a month, since it was necessary to worry about correctness *and* performance throughout the process. As a result, even though adding speculative hardware to Hydra will make the chip somewhat harder to design and verify, the reduced cost of generating parallel code will offer significant advantages.

Prototype implementation

To validate our simulations, develop more speculative software, and verify that the Hydra architecture is as simple to design as we believe it to be, we are working with IDT to manufacture a prototype Hydra. It will use IDT's embedded MIPS-based RC32364 core and SRAM macrocells. We have a Verilog model of the chip and are moving it into a physical design using synthesis. With 8-Kbyte primary instruction and data caches and approximately 128 Kbytes of on-chip secondary cache, the die (depicted in Figure 7) will be about 90 mm^2 in IDT's 0.25-micron process. We based these area and layout estimates on the current RC32364 layout and area estimates of new components obtained using our Verilog models of the different sections of the Hydra memory system.

The memory system we are designing to connect the IDT components together consists of the following:

- our speculative coprocessor,
- interconnection buses,
- controllers for all memory resources,
- speculative buffers and bits,
- a simple off-chip main memory controller, and
- an I/O and debugging interface that we can drive using a host workstation.

We are designing most of this using a fairly straightforward standard cell methodology. The clock rate target for the cores is about 250 MHz, and we plan on inserting pipeline stages into our memory system logic as necessary to avoid slowing the cores. The most critical part of the circuit design will be in the central arbitration mechanism for the memory controllers. This circuit is difficult to pipeline, must accept many requests for arbitration every cycle, and must respond to each request with a grant signal. The large numbers of high fan-in and fan-out gates that must operate during every cycle make it a challenging circuit design problem.

A chip multiprocessor such as Hydra will be a high-performance, economical alternative to large single-chip uniprocessors. A CMP of comparable die area can achieve performance similar to a uniprocessor on integer

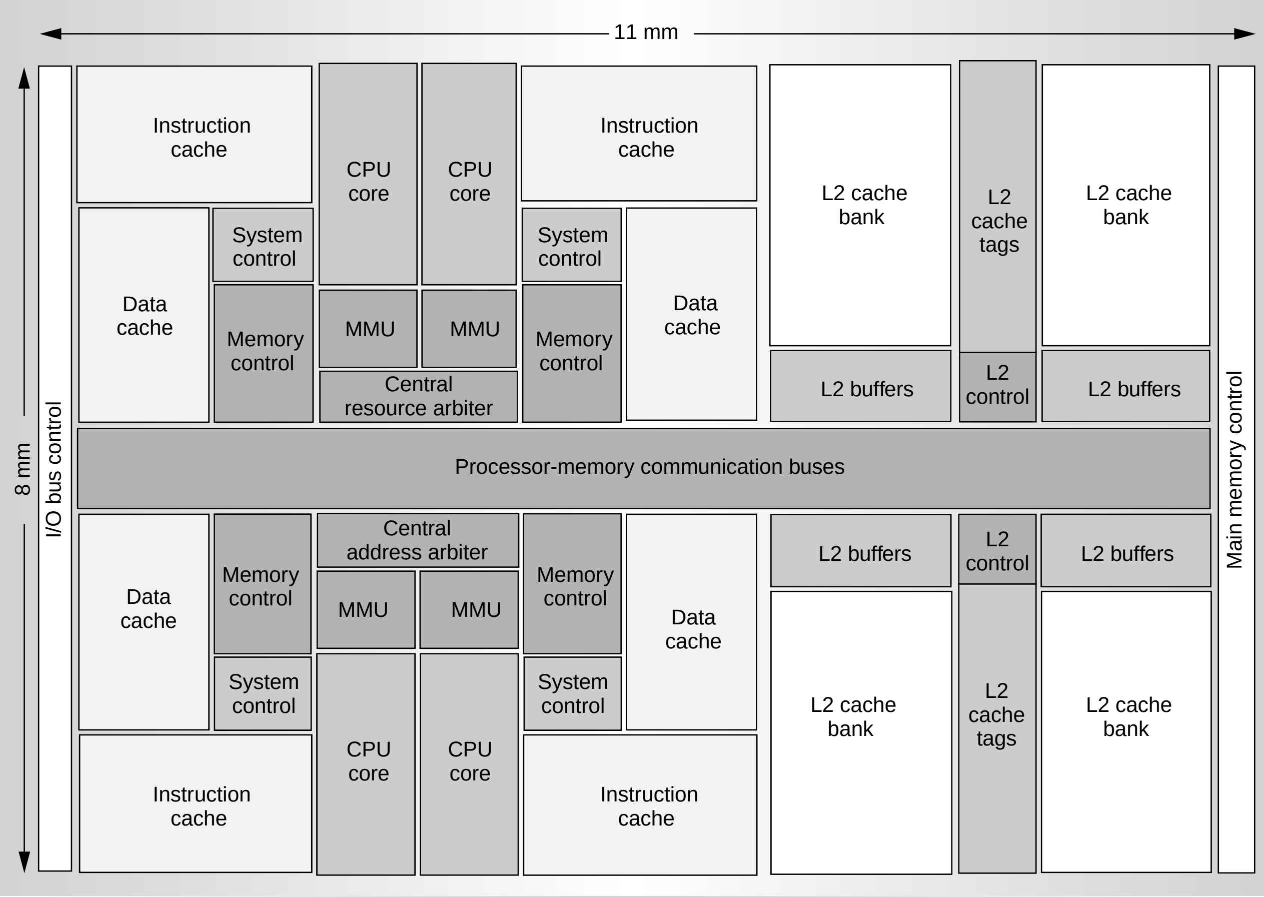

Figure 7. The floor plan of the Hydra implementation.

programs using thread-level speculation mechanisms. In addition, with multiprogrammed workloads or highly parallel applications a CMP can significantly outperform a uniprocessor of comparable cost by operating as a multiprocessor. Furthermore, the hardware required to support thread-level speculation is not particularly area-intensive. Inclusion of this feature is not expensive, even though it can significantly increase the number of programs that can be easily parallelized to fully use the CMP. MICRO

Acknowledgments

Many people gave us invaluable help during the Hydra design. Basem Nayfeh guided the early development of Hydra's memory system and thread-level speculation hardware. Monica Lam, Jeff Oplinger, and David Heine from the SUIF group provided help with compiler support and analysis of early speculation algorithms. Nick Kucharewski, Raymond M. Chu, Tuan Luong, and Maciek Kozyrczak at IDT worked closely with us during the process of building the Hydra prototype.

US Defense Advanced Research Projects Agency contracts DABT 63-95-C-0089 and MDA 904-98-C-A933 provided research funding.

References

1. K. Olukotun et al., "The Case for a Single Chip Multiprocessor," *Proc. Seventh Int'l Conf. for Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII),* ACM Press, New York**,** Oct. 1996, pp. 2-11.
2. D. Matzke, "Will Physical Scalability Sabotage Performance Gains?," *Computer,* Sep. 1997, pp. 37-39.
3. M.W. Hall et al., "Maximizing Multiprocessor Performance With the SUIF Compiler," *Computer,* Dec. 1996, pp. 84-88.
4. G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Annual Int'l*

Symp. Computer Architecture, ACM Press, Jun. 1995, pp. 414-425.

5. L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII),* ACM Press, Oct. 1998, pp. 58-69.
6. K. Olukotun, L. Hammond, and M. Willey, "Improving the Performance of Speculatively Parallel Applications on the Hydra CMP," Proc. 1999 *Int'l Conf. Supercomputing,* ACM Press, June 1999, pp. 21-30.
7. S. Gopal et al., "Speculative Versioning Cache," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture (HPCA-4),* IEEE Computer Society Press, Los Alamitos, Calif., Feb. 1998, pp. 195-205.
8. J.G. Steffan and T. Mowry, "The Potential for Using Thread-level Data Speculation to Facilitate Automatic Parallelization," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture (HPCA-4),* IEEE CS Press, Feb. 1998, pp. 2-13.
9. S. Keckler et al., "Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor," *Proc. 25th Ann. Int'l Symp. Computer Architecture,* ACM Press, Jun.-Jul. 1998, pp. 306-317.
10. J. Kahle, "The IBM Power4 Processor," Microprocessor Forum presentation reported in: K. Diefendorff, "Power4 Focuses on Memory Bandwidth," *Microprocessor Report,* Oct. 6, 1999, pp. 11-17.
11. M. Tremblay, "MAJC(tm): An Architecture for the New Millennium," *Proc. Hot Chips 11,* Aug. 1999, pp. 275-288. Also reported in B. Case, "Sun Makes MAJC With Mirrors," *Microprocessor Report,* Oct. 25, 1999, pp. 18-21.

Lance Hammond's biography is not available.

Benedict A. Hubbert is a PhD candidate in electrical engineering at Stanford University. His research interests include parallel computer architecture and software. Hubbert received a BS in electrical engineering from the University of Michigan and a MS in electrical engineering from Stanford University.

Michael Siu is a senior design engineer at Advanced Micro Devices, where he participated in the design of the K6, Athlon (K7), and most recently the Sledgehammer (K8) processors. His responsibilities include architecture, logic design, and physical implementation. Siu received a BS degree in electrical engineering from the University of Florida and an MS in electrical engineering from Stanford University.

Manohar K. Prabhu is a doctoral candidate in the Department of Electrical Engineering at Stanford University. He is currently investigating methods of parallel program development for chip multiprocessors. Prabhu has an SB degree in engineering sciences from Harvard University and an MS in electrical engineering from Stanford University. His research has been funded in part by the National Science Foundation and by the Alliance for Innovative Manufacturing at Stanford.

Michael Chen is a PhD candidate in electrical engineering at Stanford University. His research interests are in high-performance and parallel computer architectures and optimizations for improving the performance of modern programming language features such as garbage collection and dynamic compilation. Chen received a BS and an MS in electrical engineering and an MS in industrial engineering from Stanford University.

Kunle Olukotun is an associate professor of electrical engineering at Stanford University, where he leads the Hydra project (http://www-hydra.stanford.edu). He is interested in the design, analysis, and verification of computer systems using ideas from computer and computer-aided design. Olukotun received a BS from Calvin College and an MS and PhD in computer engineering from the University of Michigan. He is a member of the IEEE and the ACM.

Direct questions concerning this article to Kunle Oluktoun, Stanford University, Room 302, Gates Computer Science 3A, Stanford, CA 94305-9030; kunle@ogun.stanford.edu.