CONSIDERATIONS IN THE DESIGN OF HYDRA: A MULTIPROCESSOR-ON-A-CHIP MICROARCHITECTURE

Lance Hammond
Kunle Olukotun

Technical Report No.: CSL-TR-98-749

February 1998

This work was supported by DARPA contract DABT63-95-C-0089.

Considerations in the Design of Hydra: A Multiprocessor-on-a-Chip Microarchitecture

Lance Hammond and Kunle Olukotun

CSL-TR-98-749

February 1998

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Gates Computer Science Building, #408
Stanford, CA 94305-9040
pubs@shasta.stanford.edu

Abstract

As more transistors are integrated onto larger dies, single-chip multiprocessors integrated with large amounts of cache memory will soon become a feasible alternative to the large, monolithic uniprocessors that dominate today's microprocessor marketplace. Hydra offers a promising way to build a small-scale MP-on-a-chip using a fairly simple design that still maintains excellent performance on a wide variety of applications. This report examines key parts of the Hydra design — the memory hierarchy, the on-chip buses, and the control and arbitration mechanisms — and explains the rationale for some of the decisions made in the course of finalizing the design of this memory system, with particular emphasis given to applications that stress the memory system with numerous memory accesses. With the balance between complexity and performance that we obtain, we feel Hydra offers a promising model for future MP-on-a-chip designs.

Keywords & Phrases: microprocessors, multiprocessors, MP-on-a-chip (CMP), cache memories, cache coherence, SimOS, SUIF

Considerations in the Design of Hydra: A Multiprocessor-on-a-Chip Microarchitecture

Lance Hammond and Kunle Olukotun
Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070
email: lance@leland.stanford.edu, kunle@ogun.stanford.edu, http://www-hydra.stanford.edu

1. Introduction

The Hydra microarchitecture is a research vehicle currently being designed at Stanford in an effort to evaluate the concept of a multiprocessor on a chip as an alternative for future microprocessor development, when large numbers of transistors and RAM may be integrated on a single chip. We have previously demonstrated the value of this approach, in a generalized processor [1]. This technical report provides a more detailed view of the system we are currently designing.

Hydra is composed of four 2-way superscalar MIPS CPUs on a single chip, each similar to a small R10000 processor [2] with individual L1 instruction and data caches attached. A single, unified L2 cache supports the individual L1 caches and provides a path for rapid and simple communication between the processors. These two levels of the memory hierarchy and the bus interconnections between them are the focus of the Hydra design effort described here. However, the design would be incomplete without a state-of-the-art off-chip memory system. On one side of the chip, a 128-bit L3 cache bus attaches to an array of high-speed cache SRAMs, while on the other side Rambus channels directly connect main memory to Hydra and a more conventional bus connects Hydra to I/O devices. Fig. 1 shows a diagram of the architecture, while fig. 2 depicts a possible layout for the completed design.

This paper gives a brief overview of the microarchitecture and attempts to describe some of the trade-offs that have been evaluated in the course of revising the design. Section 2 gives a brief overview of the simulation environment we are using to evaluate Hydra, and presents a few of our most interesting results obtained through simulation. Section 3 presents a descriptive overview of Hydra's memory hierarchy, along with a qualitative view of how the different levels interact. The communication buses used to transmit data between the different parts of memory are described in section 4. Control mechanisms, including the resource and address arbiters, are described briefly in section 5. Finally, section 6 concludes.

2. Simulation Methodology and Results

Hydra is currently being evaluated using a sophisticated, cycle-accurate memory simulator written in C++ that is interfaced with the SimOS machine simulator [7]. SimOS allows us to simulate four fully functional MIPS-II ISA CPUs and a suite of I/O devices with enough realism to boot and execute the IRIX 5.3 operating system under our tested applications. As a result, system calls and I/O in our benchmarks were simulated with exceptional realism. Hydra simulates the memory system using a group of interconnected state machine controllers to evaluate the memory's response to all memory references, both instruction and data, supplied by SimOS. These controllers communicate through shared models of the central arbitration mechanisms and the caches in order to accurately model the time required to complete all accesses.

This paper focuses on describing the Hydra hardware qualitatively, but some key numbers from two applications that we evaluated are used throughout the text to illustrate the rationale for key design features of Hydra. The numbers of interest are plotted in figs. 3–6. Representative samples from the core loops of the swim and tomcatv SPEC95FP benchmarks, parallelized automatically using the SUIF compiler system [8], were executed on the simulator to get these results. While we have examined several other applications from the SPEC suite, these two have exhibited the most interesting memory system behavior 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. In the future, we may examine other applications with large data sets, such as databases.

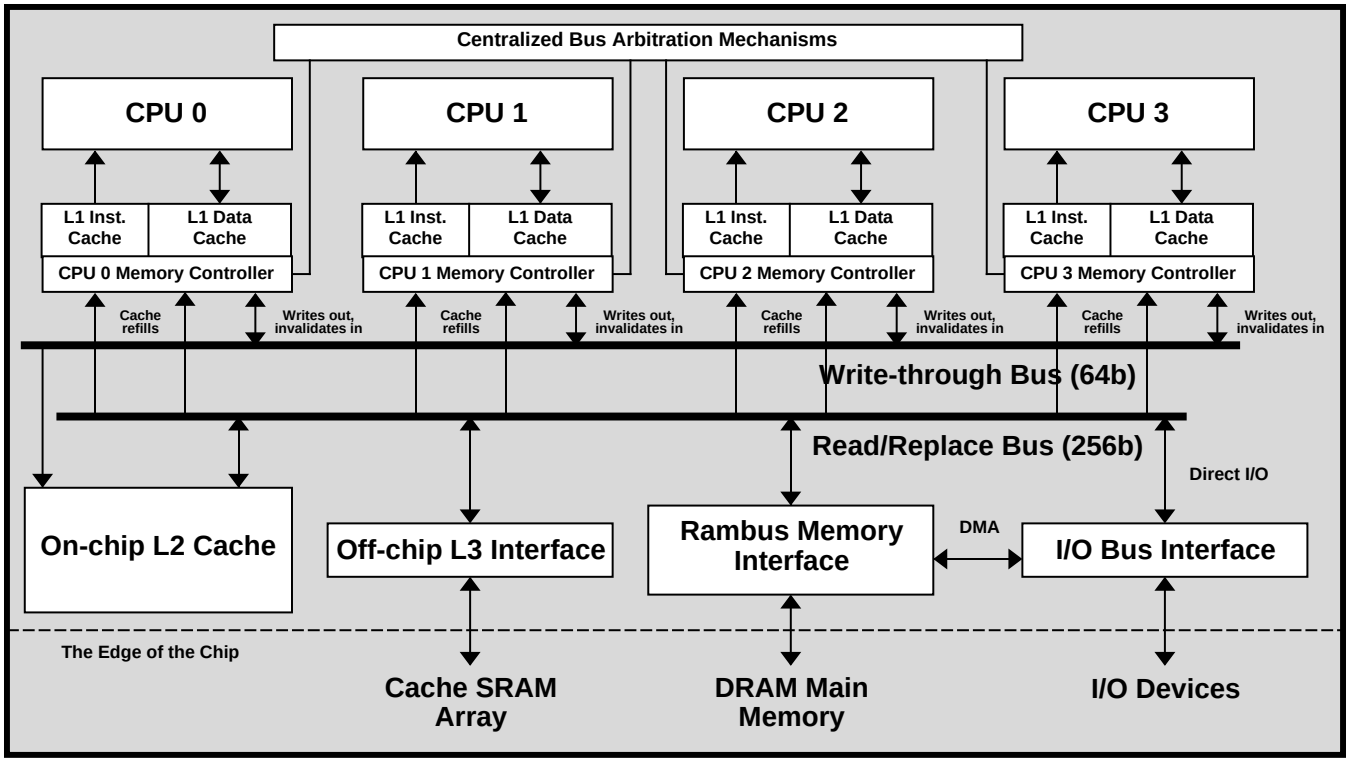

Figure 1: A schematic overview of Hydra

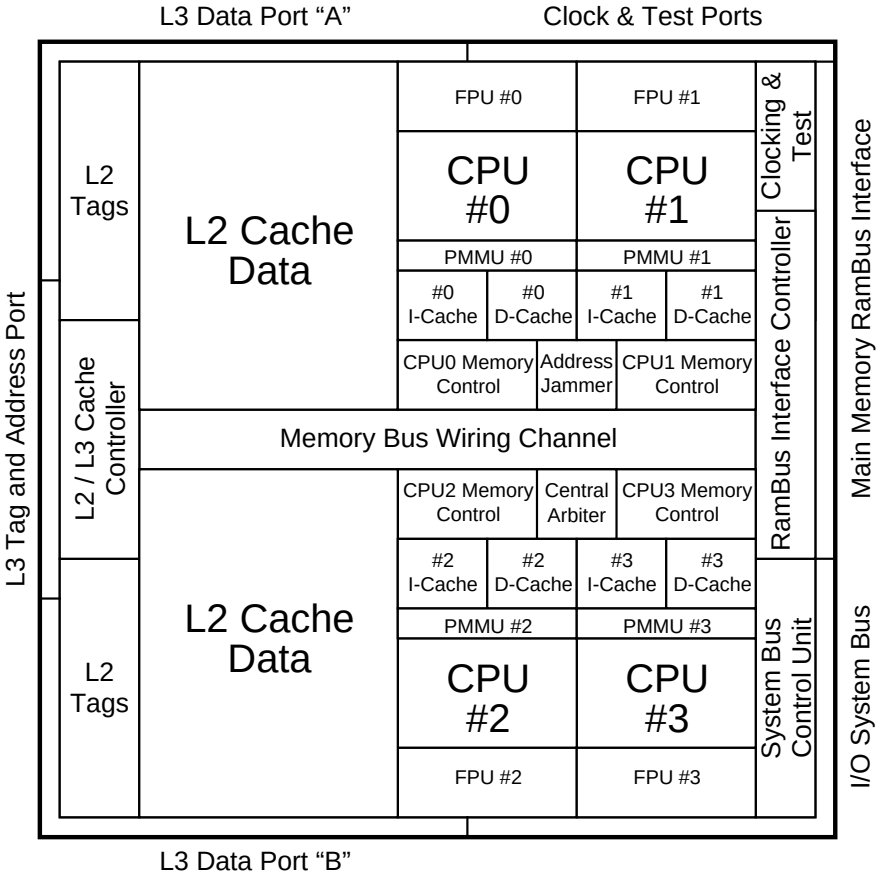

Figure 2: A possible layout of Hydra

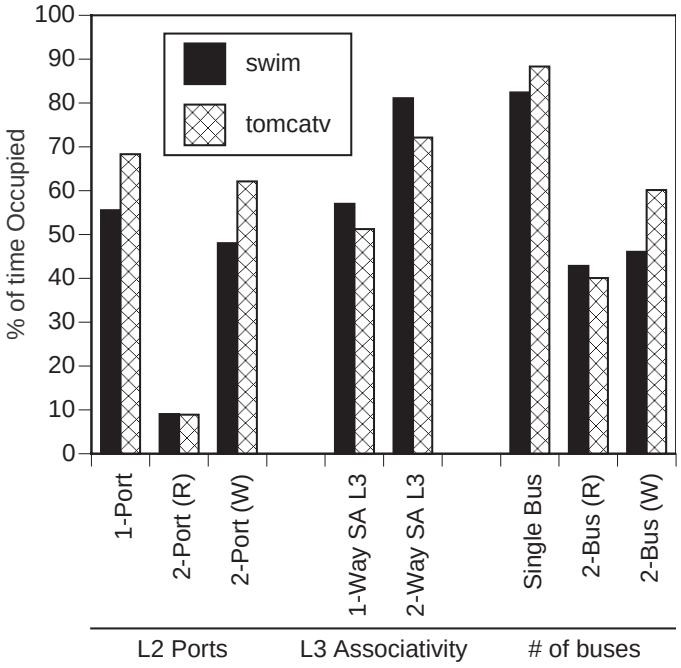

Figure 3: Resource occupancies in Hydra for several simulated situations. (see section 3 for the L2/L3 cache discussion and section 4 for discussion of the buses)

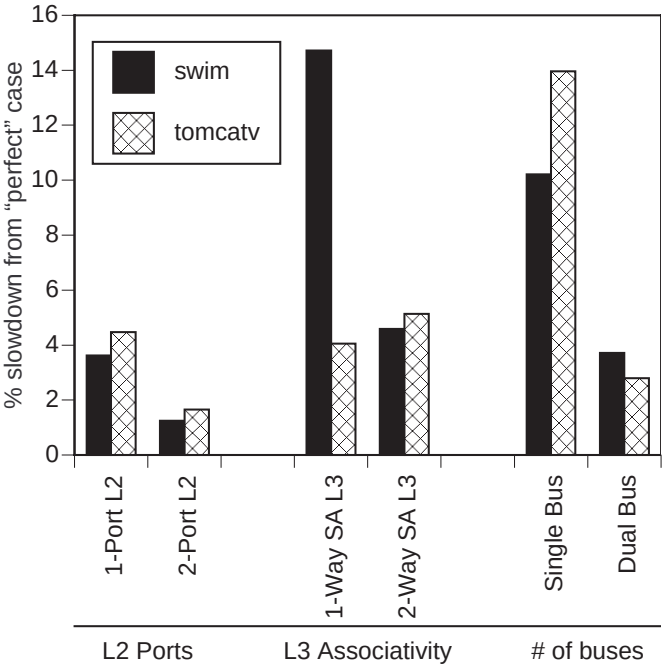

Figure 4: The % increase in execution time seen as a result of several architectural decisions, over "perfect" scenarios. (see section 3 for the L2/L3 cache discussion and section 4 for discussion of the buses)

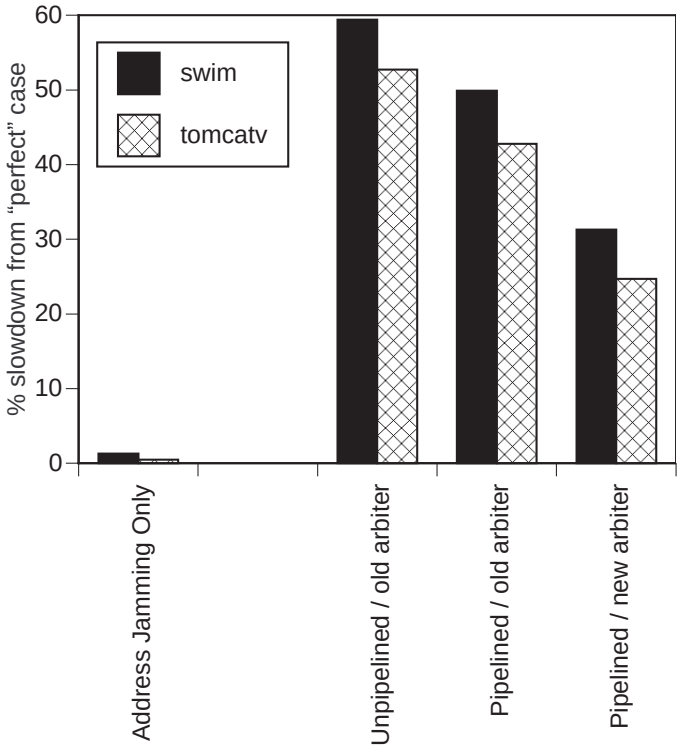

Figure 5: The % increase in execution time, over "perfect" scenarios with no arbitration at all between fully pipelined state machines, seen as a result of variations in the arbitration mechanisms and pipelining of the read and write control state machines. (see section 5 for discussion)

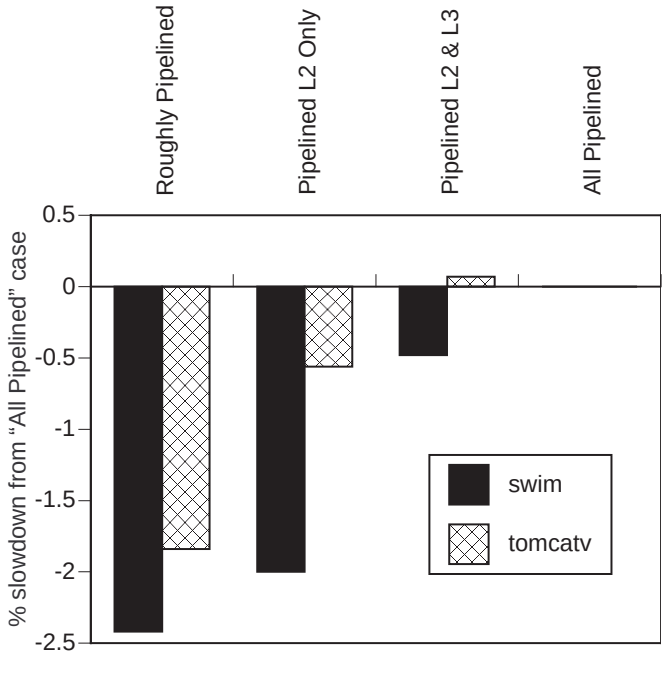

Figure 6: The % increase in execution time seen as a result of variations in the degree of pipelining used in the read and write control state machines. (see section 5 for discussion)

	L1 Cache	L2 Cache	L3 Cache	Main Memory
Configuration	Separate I & D SRAM cache pairs for each CPU	Shared, on-chip SRAM cache	Shared, off-chip SRAM cache	Off-chip DRAM
Capacity	16KB each	512KB	8 MB	128 MB
Bus Width	64-bit connection to CPU	256-bit read bus + 64-bit write bus*	128-bit synchronous SRAM (running at half the CPU speed)	32-bit Rambus (running at the full CPU speed)
Access Time	1 CPU cycle	5 CPU cycles	10 cycles to first word	at least 50 cycles
Associativity	4-way	4-way	2-way*	N/A
Line Size	32 bytes	32 bytes	64 bytes	4 KB pages
Write Policy	Writethrough, no allocate on write	Writeback, allocate on writes	Writeback	"writeback" (virtual memory paging)
Inclusion	N/A	Inclusion not enforced by L2 on L1 caches	Includes all data in L1 and/or L2 caches	Includes all cached data

Table 1: A summary of the cache hierarchy. The two entries marked with asterisks, L2 cache bus width and L3 cache associativity, were varied during the experiments documented in figures 2 and 3. A third experiment, comparing a single L2 port against a pair of separate read and write ports, is also documented in the graphs but is not reflected in this table, since all other levels of memory are strictly single-ported.

3. The Memory Hierarchy

The Hydra memory system uses a four-level arrangement, summarized in Table 1. Each level is explored further below.

3.1. The L1 Caches

The individual L1 instruction and data caches associated with each processor are designed to supply data to each processor in a single cycle while supporting the very high memory bandwidth needed to sustain processor performance. Since 30–40% of instructions in a typical MIPS-II ISA instruction stream are loads and stores, dynamically [3], a reasonably efficient 2-way superscalar implementation will have to deal with about two data memory accesses from each processor every three cycles, in addition to a constant stream of instruction fetches. Multiplied by four processors, the first level data caches considered together must be able to support a throughput of approximately three accesses per cycle, on average.

A single, shared cache would have to be extensively multiported and/or multibanked in order to sustain this bandwidth. Additionally, such a shared cache would have to be larger than the four separate data caches we propose, since it would have to hold the active working sets of all four processors. Third, a shared cache would require a crossbar and/or fairly sophisticated arbitration logic between the L1 cache and the processors. Such hardware would tend to have an impact on the processor's clock speed and/or increase the number of cycles required for each L1 cache access, which would negatively impact the load-use penalty seen by executing code and cause more pipeline stalls. The combination of these three effects makes the design of a shared-L1 cache a problematic solution, as shown in [4].

In contrast, Hydra's independent 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. The only concession that must be made to allow multiprocessing is that each data cache must snoop the write bus and invalidate any lines to which other processors write, in order to maintain cache coherence. This can be easily performed using a duplicate set of L1 tags dedicated to handling snoops on the write bus, and then only interrupting the rest of the L1 cache when an invalidation is actually necessary.

3.2. The L2 Cache

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 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 [3], 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 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.

The communication mechanism through the L2 allows the L1 caches to be simplified — they only have to support the external invalidations mentioned in the previous section, instead of full MESI protocols and L1 cache-to-L1 cache data transfers, since they cannot hold private data. With this simple protocol, inter-processor communication speed is still quick — just the round trip time to write to the L2 and then read back into a different processor's L1 cache — about 10 cycles, minimum. Another advantage that this design offers on a low level is that the L1 and L2 caches may be individually tailored to their distinct purposes — the L1 caches to handling high-speed, high-bandwidth accesses in a single clock cycle, and the L2 to high capacity, filling up all die area not required by the processors with as much cache memory as possible.

Fig. 3 shows the occupancies of the cache ports when the L2 cache is single-ported or dual-ported, 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, fig. 4 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.

3.3. The L3 Cache

The off-chip L3 cache offers a cache with another order of magnitude larger capacity than the L2, but which can only be accessed through a relatively narrow, 128-bit port that operates at half of the processor speed. As a result, it provides good hit rates with a reasonably short access time of about 15 cycles to the first word, minimum (including L1 and L2 miss time), but with bandwidth restrictions that are severe compared to the on-chip caches.

The design of this level of the memory hierarchy was fairly straightforward, but we did examine the consequences of having the L3 cache be both direct-mapped and 2-way set associative. Since we do not have on-chip cache tags, the 2-way version of the cache speculatively starts fetching from both cache lines contained in a set until the correct way is known. The former consumes less bandwidth, since erroneous data from the "wrong" way of the set will not be speculatively fetched, as shown in fig. 3, while the latter offers higher hit rates. The 2-way L3 exhibits extremely high occupancies on the L3 interface, but its increased hit rate and flexibility in the face of cache conflicts make it better choice, in general. Fig. 4 shows that the two applications slowed down about 4% in both cases due to contention at the interface with the 2-way associative cache. However, swim incurred a significant penalty from a cache conflict that could not be averted in the direct mapped cache but was easily avoided in the set-associative one. Since the latency of main memory is an order of magnitude longer than the L3 latency, a small number of these conflicts can cause a significant slowdown.

3.4. Main Memory

Even with a large L3 cache, some applications have large enough working sets to miss in all of the caches frequently, severely taxing the main memory subsystem. In order to keep up with the processors on swim and tomcatv, we have found that the memory system would have to be able to deliver a 32-bit word to the chip on every processor cycle, on average — half the bandwidth of the L3 cache. With processor speeds of tomorrow, this will take multiple Rambus-style memory interfaces directly attached to the processor to supply the necessary bandwidth while keeping the number of pins connecting Hydra to the DRAM reasonable.

3.5. Other Alternatives

While the current Hydra design addresses a fairly high-end and expensive CPU design, with many high-speed SRAM and DRAM chips directly attached to the Hydra chip, alternative designs are possible in systems with different constraints.

A small variation could be made by pulling the L3 cache tags on-chip, like those in the PowerPC 750 [9]. In the current design, the L2 cache acts primarily as a buffer for writes and interprocessor communication, due to the L2's low local hit rate on read accesses. As a result, most data not found in the L1 has to be recovered from the L3 cache. Trading some L2 cache for L3 tags, in order to improve the performance of the L3 cache, might therefore be a reasonable trade-off. The L2 hit rate will decrease, but that should not affect overall performance much. This would help improve L3 performance in two ways. First, the L3 cache could be made highly associative. While it is not practical to check more than one or two off-chip tags at once, many on-chip tags can be checked in parallel. The increased associativity would tend to reduce L3 miss rates and thereby increase performance. Second, the L3 tags could be checked in parallel with each L2 access. Accesses that miss in the L3 entirely could be routed directly to main memory, not wasting any of the L3 cache port's limited bandwidth on useless accesses. Also, even a highly associative L3 would only exhibit bandwidth requirements similar to the tested direct-mapped version. Instead of wasting bandwidth speculatively reading multiple lines from a set of associative cache lines, the on-chip tags would simply select the correct line before the off-chip access was started. The reduced contention at the L3 cache port would both improve performance, as our tomcatv results in fig. 4 show, and save power by reducing the number of accesses to off-chip memory.

Another interesting variation is a design with no off-chip SRAM L3 cache at all. This arrangement is beneficial for several reasons. First, high-speed cache SRAM chips are expensive, so eliminating them would reduce the system cost dramatically. Second, approximately half of the I/Os on Hydra are devoted to the L3 cache interface. Without this cache, a much cheaper package and simpler motherboard could be used to hold each Hydra part. Third, design of the Hydra and the system would be simpler and therefore cheaper without the L3, since the design of the L3 interface, from both electrical and control perspectives, is a nontrivial task. Finally, as time passes it is becoming possible to integrate larger amounts of cache memory on the processor die. Thus, the on-chip SRAM L2 may eventually become large enough to have a good local hit rate on read accesses, and therefore perform the job of the L3 cache in the current design. Another possibility is that the SRAM L2 cache could be replaced with on-chip DRAM, a larger but slower on-chip memory technology, in order to fully replace the L3 cache with an on-chip alternative of similar size and speed. Larger on-chip SRAM caches and on-chip DRAM are explored in [6].

4. Communication Buses

The read/replace (or "read") and write-through (or "write") buses, depicted in fig. 1, are the principal paths of communication across the Hydra chip. Both buses are controlled in a pipelined manner, with bus arbitration occurring at least a cycle before bus use, to keep throughput at one access per cycle each bus. Such throughput is necessary to avoid bus saturation in the current implementation of Hydra on applications such as tomcatv and swim.

4.1. The Read Bus

The read/replace bus is the largest and most important data highway across the Hydra chip. It is 256 bits wide, matching the L1 and L2 cache line size, in order to allow entire lines to be transmitted across the chip at once. It is used to move cache lines around among the on-chip caches and the off-chip interfaces, since all connect to it. While only one interface may broadcast data at a time, many may listen to data broadcast across the read bus — for example, data brought in from memory can be accepted by the L1 cache, L2 cache, and L3 cache interface on the same cycle, from a single broadcast.

Since it is not a crossbar, the read bus must be arbitrated for before each data broadcast, as described in section 5. However, while it performs many tasks, we have found that the bus

is typically occupied less than 50% of the time, as shown in fig. 3, even with our most memory-intensive applications. Contention for this particular resource is thus not a problem. As fig. 4 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.

4.2. Write Bus

While the read bus is a general-purpose data bus, the write bus has one specific job — it takes write-through traffic from the CPUs and sends them into the L2 cache. Since it only carries data from one store at a time, it only needs to be 64 bits wide, at most — the width of the widest MIPS-II store instruction. This dedicated bus is necessary because our L1 caches are writethrough and all stores from the CPUs must be broadcast due to our simple coherence scheme. Since the L1 caches do not filter this bandwidth, bus traffic can be quite high — sometimes exceeding 60% full, as is shown in fig. 3, even though each access only reserves the bus for a single cycle — and the bus can be saturated 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 they execute bursts of store instructions. Merging this bus with the read bus would typically result in a completely saturated bus and significantly reduced performance, as figs. 3 and 4 indicate.

As well as giving writes a clear path to the L2 cache, the write bus also serves as a crucial mechanism for synchronizing the processors. Writes are serialized as they pass through the write bus, so multiple writes to the same address will update the permanent state of the machine in a consistent order. Since all state changes are broadcast in this one location, L1 cache coherence may be maintained by having the L1 caches snoop on just this bus, and then invalidate their own contents when they contain a line to which new data is being written. Since the write bus carries written data along with addresses that need to be invalidated, it is fairly easy to use an update cache protocol instead, should that be desired. We chose not to use an update protocol since this generally results in many useless interventions into other processors' L1 caches.

One critical issue that should be noted is that the write bus is not scalable to larger numbers of processors, or to a four-way MP made up of more complex, and therefore faster, processors. This problem is easily addressed in several ways, however. First, it would be possible to have multiple write buses on a single Hydra chip. As long as they did not carry data being written to the same address at once, the multiple write buses would appear just like a single, faster write bus to the CPUs. This scheme would require that extra ports be supplied to the L1 tags, to allow multiple snoops and invalidations per cycle, and to the L2 cache, to allow multiple data words to be written every cycle. This latter modification would probably significantly enlarge the L2 cache, and thus could make this solution impractical. A second scheme would be to have a single, line-wide write bus. Since many bursts of writes will fall within a cache line or two, merging writes that fall on single cache lines together in the CPU write buffers would allow greater store throughput on these critical sections of code. Third, the write bus could be discarded altogether and a more conventional MESI coherence protocol between the L1 caches could be adopted. This would eliminate the write bus altogether, and would make the system look like a conventional shared-bus multiprocessor centered around the read bus. As such systems have already been examined [5], and have significant control overhead, we chose to look at a more simple and novel architecture. Finally, in the long-term future, multiple Hydra-like architectures could be connected hierarchically on a single chip, to achieve scalability while still including high-performance clusters of processors. Such an architecture is beyond the scope of this technical report.

5. Control Mechanisms

The buses and caches in Hydra are controlled by a group of memory control pipelines, each devoted to a particular CPU or interface. To prevent the pipelines from attempting to use resources or addresses simultaneously, access to these two limiting factors is controlled through the resource and address arbiters.

5.1. Memory Control Pipelines

Eight of the ten different "memory control pipelines" are contained in the CPUs. Each of the CPU memory controllers in fig. 1 contains two independent pipelines, one for handling read accesses — both from instruction fetches and loads — and one for handling writes from stores. These pipelines are used for any access that requires the use of any level of memory below the L1 caches. While a processor will make approximately two read accesses per cycle on typical code [3], 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 cycles [3], 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.

The read and write pipelines in each CPU are very similar. Each consists of three main stages: the L2 cache access (which is about 5 cycles long), the L3 cache access (about 15 cycles), and the initiation of a main memory access (about 5–10 cycles). Each access goes through each stage in succession until the required cache line is located in one of the levels of the memory hierarchy. Resource arbitrations are made at the start of each section of the pipeline in order to ensure that the pipeline has reserved needed resources before initiating an access to a cache or using a bus. The only real difference between the two pipelines is that the write pipeline writes its data into the L2 from the write bus at the end of every access, while the read pipeline forces the desired cache line out onto the read bus.

In the process of tuning Hydra, these pipelines were adjusted extensively to improve performance. Initially, the pipelines were not actually pipelines — 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 — controllers that could start processing a new access every cycle. As shown in fig. 5, performance improved, but not as much as we had hoped. In particular, we found that switching to such an extensively pipelined design undermined our arbitration priority scheme. For best performance, a good heuristic is that the oldest accesses should get resources before newer ones. However, our old arbiter was often letting young accesses, going to the L2 cache, pass up older accesses trying to use the L3 cache or main memory — a problem often called "priority inversion." As a result, we improved the resource arbiter (see the next subsection), and managed to dramatically improve results. Since the "perfect" baseline represented in fig. 5 could be obtained only with full crossbars instead of buses and with infinitely-ported caches, we feel that our results are quite good, given our realistic resource limits. In a later experiment, we reduced the degree of pipelining by eliminating the cycle-by-cycle pipelining in some or all of the three main pipeline access stages. This allowed only a single access to be handled in each of these main stages at a time. The surprising results are shown in fig. 6. The more coarse pipelines were actually slightly better than the full cycle-by-cycle ones, even though they are much simpler and easier to build. The reason is simple — along with the improved arbiter, the coarser pipeline reduced the amount of priority inversion that occurred. Handling more than one access per main pipe stage, in contrast, turned out to be unnecessary — the coarser stages were sufficient.

The remaining two pipelines are in the main memory and I/O interfaces. These controllers process data returning from memory or I/O requests, sending it to the appropriate caches on the Hydra chip as it arrives. They are much simpler cousins to the CPU controllers, since they basically just respond to all data arriving from off-chip in a programmed manner. Like the other pipelines, they were gradually pipelined from simple state machines. However, the limited bandwidth of the main memory and I/O buses also imposed a limit on the degree of pipelining that was necessary. The main memory controller only needed to be divided into two main pipeline sections — one for sending data to the L1/L2 and one for recording cache lines into the L3 cache. Currently, the I/O controller is still completely unpipelined, since that is enough to handle our simulated I/O traffic.

5.2. Resource Arbiter

The resource arbiter is responsible for allocating the chip's key resources — the buses, caches, and some buffers — for the next several cycles ahead. Many arbitration mechanisms handle requests on a cycle-by-cycle or resource-by-resource basis, but the Hydra arbiter processes entire "resource requests" at once — requests for a group of resources required by a pipeline over the course of the next 5-20 cycles. Each CPU

pipeline sends the arbiter a resource request before each of the three main pipe stages, and then stalls until the arbiter tells the pipeline it may continue. The memory and I/O pipelines work similarly, but are designed to make several small resource requests as words of data are brought into Hydra from off-chip. Since pipelines send out their resource requests at least a cycle before any of the critical resources are actually needed for processing, the resource arbiter causes no unnecessary delays when resources are free. Some of the resource requests, especially those involving the L3 cache, may be for quite complicated combinations of buses and caches on different cycles. The arbiter imposes a fixed priority on the requests, depending upon the pipeline and the request type. All memory and I/O pipeline requests always get first priority, followed by main memory requests, L3 cache requests, and finally L2 cache requests. Among the CPU pipeline request types, the read pipelines generally have priority over the write pipelines. However, each CPU may select to invert the priority of its read and write pipelines intentionally on a cycle-by-cycle basis, giving its writes higher priority, in case its write buffer is nearly full. We found this feature to be quite helpful with memory-intensive applications that must frequently recover cache lines from the L3 cache or main memory in order to handle their writethrough traffic, since even a few L2 cache misses on writethroughs can allow a CPU fill up its write buffer. Priorities among the different CPUs are changed round-robin, so that no CPU may hog the arbiter. We tried more complex inter-CPU priority schemes, also, but they did not affect performance significantly.

As shown in fig. 5 and discussed in the previous subsection, the arbiter has gone through two main stages of development in order to prevent accidental priority inversions. The first version of the arbiter checked all pending resource requests from all pipelines on every cycle. If a request failed, it was tried again on the next cycle. If a request succeeded, its resource reservations for the next several cycles were recorded, and the requesting pipeline was allowed to continue. This scheme had the unfortunate side effect of giving the simple L2 cache accesses higher priority than the complex L3 accesses. Since these simpler requests demanded fewer resources, it was easier for them to find all of their resources free on any given cycle. They would then reserve resources for the next several cycles ahead, and frequently block pending L3 cache accesses further in the process. This resulted in some references taking thousands of cycles to complete, simply because they were never able to acquire resources. The second version of the arbiter solved this problem by handling each resource request only once, on the cycle it arrives at the arbiter. On that cycle, the arbiter scans ahead until it finds the earliest time that the request can be handled. After finding this time, the arbiter reserves the resources immediately and notifies the pipeline how many cycles it must wait. While the simple L2 cache accesses are still usually delayed less than the more complex L3 accesses by this arbitration mechanism, the arbiter can guarantee that the L3 accesses are able to get the minimum delays that their resource requirements allow.

5.3. Address Arbiter

If multiple accesses to the same address are processed simultaneously by different pipeline controllers in Hydra, problems may occur. The first problem is that multiple copies of a single cache line may be generated within an associative set of lines. If a pipeline controller misses in a cache, it will immediately attempt to recover the line from the next level of the cache hierarchy. Unfortunately, if another controller is already processing a miss to the same line, both will return with their own copies. Along with wasting precious cache space on redundant data, duplicate lines may cause functional errors if writes are made to both copies, since eventually both copies of the cache line will be written back to lower levels of memory — so the writes to the copy written back first will be erroneously overwritten. A similar problem can occur if one controller is writing back a dirty line to the next level of memory. While the line is being written back, it is no longer listed in the tag directory of the cache that is discarding it. Hence, another controller may miss in the cache and subsequently fetch the old, stale version from the lower level of memory while the more recent version is still in the process of being written back. The cache will refill with the stale data, and the written-back modifications to the line will be permanently lost. Making these problems worse, each access may involve up to three different cache lines — the line requested, the L2 cache writeback, and the L3 cache writeback.

Hydra's address arbiter solves these problems by simply forbidding multiple accesses to the same address in the L2/L3/main memory system until all previous accesses to that address have completed. In order to protect all three cache lines that an access may use with only a single arbitration, the Hydra address arbiter actually checks only the address bits that are used for indexing into *both* the L2 and L3 — 8 bits in the

current implementation. This works because a line and any writebacks it may generate will always be in the same associative set, so by protecting its sets in both the L2 and L3 we automatically protect all three lines that might be active at once. The address arbiter "checks in" each access when it initially tries to access the L2 cache tags — always the first step of any access. Most accesses continue on without problems, but if an address collision is detected the L2 cache access is aborted and the access is queued up behind all previous accesses to that particular index. To keep the mechanism simple, no prioritizing is performed among the queued accesses. When all previous accesses to that index have completed, the access is freed and it retries the aborted L2 cache access.

While this mechanism might appear to be rather heavy-handed and excessive, in reality it has a negligible impact on performance, as fig. 5 demonstrates. 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 reasonably small. Second, even 8-bit indices offer 256 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, 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.

6. Conclusions

As more transistors are integrated onto larger dies, single-chip multiprocessors integrated with large amounts of cache memory will soon become a feasible alternative to the large, monolithic uniprocessors that dominate today's microprocessor marketplace. 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. Off-chip memory is controlled through two glueless ports to attached SRAM and DRAM chips. Arbitration of key resources and addresses is accomplished with two custom mechanisms that have been highly tuned. With this balance between complexity and performance, we believe Hydra offers a promising model for future MP-on-a-chip designs.

References

[1]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.

[2]K. Yeager, "The MIPS R10000 superscalar microprocessor," IEEE Micro, vol. 16, No. 2, pp. 28–40, April 1996.

[3]D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd Ed., Morgan Kaufmann Publishers, Inc., pp. 105–6, San Francisco 1996.

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

[5]M. Takahashi, H. Takano, E. Kaneko, and S. Suzuki, "A Shared Bus Control Mechanism and a Cache Coherence Protocol for a High-performance On-chip Multiprocessor," *Proceedings of the 2nd Conference on High Performance Computer Architecture (HPCA-2)*, pp. 314–322, San Jose, CA 1996.

[6]T. Yamauchi, L. Hammond, and K. Olukotun, "A Single-Chip Multiprocessor Integrated with DRAM," Stanford University Technical Report No. CSL-TR-97-731, August 1997.

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

[8]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 Technical Report No. CSL-TR-94-620, May 1994.

[9]L. Gwennap, "Arthur Revitalizes PowerPC Line," *Microprocessor Report*, pp. 10–13, February 17, 1997.