Rationale, Design and Performance of the Hydra Multiprocessor

Kunle Olukotun, Jules Bergmann, Kun-Yung Chang and Basem Nayfeh

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

In Hydra four high performance processors communicate via a shared secondary cache. The shared cache is implemented using multichip module (MCM) packaging technology. The Hydra multiprocessor is designed to efficiently support automatically parallelized programs that have high degrees of fine grained sharing. This paper motivates the Hydra multiprocessor design by reviewing current trends in architecture and development in parallelizing compiler technology and implementation technology. The design of the Hydra multiprocessor is described and explained. Initial estimates of the interprocessor communication latencies show them to be much better than current bus-based multiprocessors. These lower latencies result in higher performance on applications with fine grained parallelism.

1 Introduction

The performance of microprocessors has been improving at a phenomenal rate for the last ten years. This performance growth has been driven by innovation in compilers, improvements in architecture and tremendous improvements in implementation technology. Currently high performance microprocessors use multiple instruction issue superscalar design techniques; however, extensions to these superscalar design techniques cannot sustain the current growth in microprocessor performance. As the number of instructions issued every cycle increases there will be diminishing returns in performance and significant increases in design and implementation complexity. An alternative to superscalar design is to develop multiprocessor microarchitectures that have multiple threads of control. These architectures promise to be easier to build than wide issue superscalar machines and will offer equal or better performance.

To understand the complexities of building a multiprocessor microarchitectures and to demonstrate that these architectures can achieve high performance we are embarking on the Hydra multiprocessor project. The key elements of the Hydra multiprocessor are multiple microprocessors sharing a secondary cache and the use of multichip module packaging technology. The goal of the Hydra multiprocessor project is to demonstrate that the cost/performance of general purpose computing can be improved by combining parallelizing compilers, high performance single chip microprocessors and advanced packaging technology.

The purpose of this paper is to present the rationale for the Hydra multiprocessor and to describe its operation. The remainder of the paper is organized as follows. The next section presents the rational for the Hydra multiprocessor. This is followed by a section that describes the design and operation of the Hydra multiprocessor. Following this is a section that presents preliminary performance results that indicate that the Hydra architecture will have an advantage over existing multiprocessor designs. The last section presents the conclusions.

2 Rational for the Hydra Multiprocessor

The design of Hydra multiprocessor is motivated by trends in microprocessor architecture, and developments in parallel compilers and packaging technology. We describe these trends and developments in this section and indicate how the Hydra architecture will exploit these developments to improve the cost/performance of microprocessors.

2.1 Trends in single chip microprocessor design

Improvements in silicon integrated circuit technology continue to provide faster transistor switching speeds and higher integration density. Currently, transistor speed is improving at 40% per year and integration density is increasing at 60% per year [7]. The performance of microprocessors built with this technology is increasing at the unprecedented rate of 60% per year. The increase in microprocessor speed relative to main memory speed and the desire to maintain microprocessor performance growth has resulted in two trends in microprocessor design. These are large on-chip caches and complex microarchitectures designed to exploit instruction level parallelism (ILP).

As the performance of processors improve they require higher bandwidth and lower latency memory hierarchies. Increasing the size and complexity of the on-chip cache is one way to match improvements in processor performance with improvements in memory hierarchy latency and bandwidth. Currently, the fastest microprocessor, the DEC Alpha 21164, has a two-level cache with a total of over a 100 KB of SRAM. However, there are alternatives to architectures with large on-chip caches. An example of an alternative are the HP PA-RISC microprocessor architectures. These architectures have achieved high performance with very little on-chip cache by using the fastest commodity SRAMs in an off-chip cache and a highly optimized design of the electrical interface between the processor and the cache SRAMs. The result is an off-chip primary cache access time that is close to that of an on-chip cache and a cache size that is much larger than it is possible to have on a single chip.

The other trend in microprocessor design is the use of increasingly complex architectures to exploit ILP. There are two approaches that are used to exploit ILP in programs: superscalar architectures and VLIW architectures. Both approaches attempt to issue multiple instructions to independent function units every CPU clock cycle. Superscalar architectures use hardware to dynamically find data-independent instructions in an instruction window and to issue these instructions to independent functional units. On the other hand, VLIW architectures rely on the compiler to find the ILP and to statically schedule the execution of independent instructions. Superscalar architectures are appealing because they can improve the performance of existing application binaries. However, the ease with which applications can be accelerated is not without costs. Superscalar architectures are complex to design and difficult to implement. Looking for parallelism in a large window of instructions requires a significant amount of hardware and usually does not improve performance as much as one might expect. Due to the complexity of superscalar design it is difficult to get the architecture correct and at the same time optimize the pipelines and circuits to achieve high clock frequencies. For this reason, superscalar designs with complex architectures have tended to have lower clock frequencies than simpler designs.

VLIW CPUs rely on the compiler to find *packets* of independent instructions. VLIW processors do not require the hardware for dynamic scheduling and so can be simpler to design and to implement. However, they require significant compiler support such as trace scheduling to find the ILP in application programs. Because programs must be recompiled before they will run correctly on a new VLIW architecture, VLIW processors will not execute binary programs compiled for an existing instruction set architecture (ISA). This is a serious disadvantage in the computer marketplace but one that may be overcome by the use of software for emulating existing ISAs. VLIW architectures will be preferred over superscalar architectures when the issue width is large enough so that dynamic issue hardware in the superscalar architecture is too complex and costly to build. However, even in the VLIW processor, such a wide machine will have a centralized register file which must have many ports to supply operands to the independent functional units. The access time of such a register file and the complexity of the bypass buses connecting the functional will ultimately limit the clock frequency of the of a very wide VLIW implementation. A final disadvantage of VLIW architectures is that VLIWs force all instructions in a packet to execute together. If one instruction in the packet stalls the whole packet must stall. This limits the VLIW architecture's ability to deal with unpredictable events such as data accesses that miss the cache.

To understand how architects have used greater silicon resources to increase the performance of microprocessors we have collected area and performance data for microprocessors that will have "working silicon" by the end of 1994 [14]. Their performance is measured on the SPEC92 benchmark suite [13]. Figure 1 shows the change in the utility of silicon area as the area of the microprocessor increases. Silicon utility is defined as the sum of the SPEC integer and SPEC floating point numbers of a microprocessor divided by the normalized chip area of the microprocessor. To make the area comparisons reasonably fair, the area of each microprocessor is normalized to that of a 0.5 micron process.

Three die area regions can be identified on Figure 1: are processors with areas less than 150 mm^2, which we will call small, processors with areas ranging from 150 mm^2 to 250 mm^2, which we will call medium, and processors areas with over 250 mm^2, which we will call large. If we look at the processor that achieves the highest utility of silicon area for each size range, we see that medium size microprocessors achieve higher utility than small size microprocessors, but large size microprocessors do not improve utility, and may decrease it.

If we investigate the reasons for these differences in utility, we see that the small and medium size processors focus on maximizing clock frequency, while the large size processors, excepting the DEC 21164, focus on maximizing instructions per clock (IPC). For example, the MIPS R4600, which uses a classic RISC five stage pipeline that is optimized for speed, is able to achieve high performance with little silicon area. The DEC 21064A achieves a significant increase in performance over a design such as the R4600 by using larger transistors and deeper pipelining to achieve a higher clock frequency. The DEC chip also uses a two-way superscalar design, but careful consideration of the design ensures that this design feature does not impact the clock frequency. The large size chips use aggressive superscalar and dynamic scheduling techniques and large caches to achieve much higher performance than the medium size chips. This performance comes at the cost of significant increases in die area. The large size chip that achieves the best utility is the DEC 21164, it does so by emphasizing high clock rate over maximizing IPC. The other large processors emphasizes low CPI and have utilities that are even worse than the small size microprocessors.

Two conclusions are evident from Figure 1. First, the best utility of silicon area is not achieved by the largest chips that have large-on chip caches and advanced superscalar techniques and second, trading off clock frequency for architectural complexity significantly reduces silicon utility.

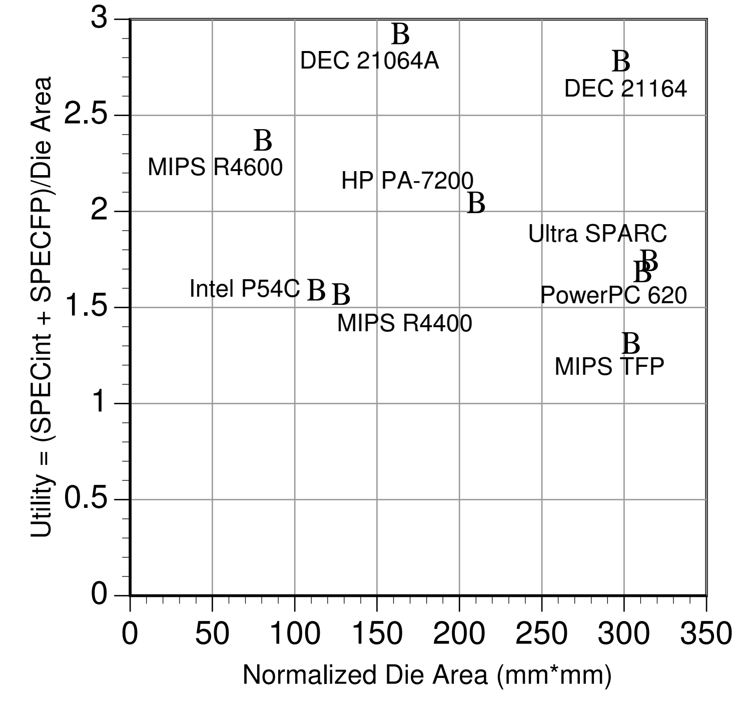

Figure 1. Silicon utility versus die area for recent microprocessors.

2.2 Parallel architectures and parallelizing compilers

Advanced superscalar and VLIW architectures improve computer performance by increasing the issue width from a single instruction stream. Another way to improve computer performance is to increase the number of issue streams by using a multiprocessor. Multiprocessor architectures are currently used to accelerate large-scale scientific applications that have been parallelized by hand, to increase the throughput of jobs in a multiprogramming environment and for transaction processing. Most of multiprocessors designed for these purposes have been small (4–32 processors) bus-based multiprocessors that use snoopy-based protocols to maintain cache coherence [5]. These processors typically use large per-processor caches to keep the bus bandwidth requirements low enough so that using a snoopy coherence protocol is feasible. The typical remote access times for these machines are in the range of a microsecond [8]. This translates to between 100 and 200 CPU cycles with current microprocessor clock frequencies.

Although multiprocessors have been able to accelerate parallel applications and improve the throughput of multiprogramming environments, multiprocessors have had limited success at improving the performance of sequential programs. This is partly due to the limitations of compiler technology and partly due to the 100 to 200 CPU cycles that it takes to communicate in a bus-based multiprocessor.

Recently, advances in compiler technology have extended the range of applications that can be successfully parallelized [1]. These advances include algorithms for interprocedural analysis of data dependencies, array privatization and C pointer analysis. Interprocedural analysis allows the compiler to find parallelism over wide regions of the program and array privatization makes it possible to parallelize loops that use arrays as temporary work areas in the body of the loop. Array privatization make these loops parallel by giving each parallel loop an independent copy of the array. A significant amount of data dependence analysis is required for a compiler to perform array privatization. C programs uses pointers; pointers can refer to the same object. Such aliases prevent parallelization and without further information the compiler must assume all pointers are aliases of each other. Using C pointer analysis, the compiler is able to identify the pointer aliases that actually occur in the program. This greatly increases the potential for parallelization

As an example of what advanced compiler technology is capable of, Table 1 shows the amount of loop-level parallelism extracted from four SPEC floating point benchmarks using the state of the art Stanford University intermediate format (SUIF) compiler system [1]. The percentage of parallel coverage on each benchmark is the fraction of the sequential run time that is spent in the parallelized loops. The FORTRAN benchmarks require interprocedural analysis to get the high parallel coverage reported in Table 1. Without this analysis the parallel coverage on mdljdp2 drops to 13% and on ora the coverage drops to zero.The C benchmarks, alvinn and ear, require pointer disambiguation in addition to interprocedural analysis in order to find parallelism.

Benchmark	**Language**	**Description**	**Source Lines**	**Parallel Coverage (%)**
alvinn	C	Neural network training	272	99
ear	C	Human ear simulation	5237	88
mdljdp2	FORTRAN	Molecular dynamics model	3883	89
ora	FORTRAN	Ray tracing	533	100

Table 1 Four SPEC FP benchmarks

Even though compiler parallelization and data locality optimization algorithms are improving there are many programs that, although they have sufficient parallelism, do not speed up on existing parallel computers because the communication and synchronization latencies are too high. The Hydra architecture will reduce the communication and synchronization latencies of a small scale multiprocessor by an order of magnitude compared to present parallel computers by changing the architecture and implementation of the memory hierarchy. This improvement will make it possible to improve the parallel performance of a much larger number of sequential programs. We expect the Hydra multiprocessor will also work well on applications that have unstructured parallelism and large amounts of fine grained sharing. This will create the incentive to develop compiler algorithms that can extract this type of parallelism from programs that do not have loop level parellelism.

2.3 Multichip module packaging technology

In an MCM package, multiple die are placed in close proximity on a single substrate. This has a number of advantages over packaging the chips in single chip packages and placing them on a printed circuit board (PCB). First, the chips can be placed much closer together and the wires between the chips can be made very short. Second, the electrical characteristics of the MCM substrate are better than those on-chip. MCM wires have lower parasitic capacitance, inductance and resistance compared to on-chip wires [2]. This means given optimized I/O pads that are optimized for the MCM environment it can take less time to propagate a signal using an MCM substrate wire than to propagate the signal using a on-chip wire. Even though the electrical characteristics of MCM interconnect are not quite as good as those of the interconnect on a PCB, the short distances on the MCM result in much lower overall interchip delay. Finally, by eliminating the individual chip packages, MCM packaging eliminates the high parasitic inductances of the package, this further reduces inter-chip delay. With the use of flip-chip bonding of the chip to the substrate the inductance of the bonding wire can be eliminated completely [12].

MCM packaging affects system design in a number of ways. MCM packaging allows large amounts of silicon to be placed close together both physically and electrically. This reduces the performance penalty of chip crossings. MCM technology also provides the capability of having pads that cover the area of the chip which greatly increases the number of signal connections that a chip can have. In a large system, MCMs allow many high bandwidth low latency interconnections without requiring the system to be implemented as a single chip. Ultimately, systems built from multiple chips and MCM packaging could provide lower cost and higher performance than those implemented as a single large chip. However, to get the best advantage of MCM technology, it is not sufficient to take existing board systems and place them on an MCM. While this may decrease the size of the system, it will not significantly increase the performance of the system. When systems are designed specifically to take advantage of MCM technology and I/O drivers are optimized for the MCM electrical environment, both performance and size improvements will result.

The Hydra multiprocessor is an architecture that relies heavily on MCM packaging technology. It would not be feasible to implement this architecture on a single chip or with multiple single chip packages on a PCB. A single chip would not provide enough silicon area. A design with multiple chips on a PCB would have much lower performance because it would have fewer pins and a lower system clock frequency.

2.4 Improving performance and cost

The main purpose for developing of the Hydra multiprocessor is to present an alternative to wide issue superscalar architectures that will achieve comparable performance at a lower cost. The Hydra multiprocessor has multiple microprocessors that are tightly coupled using MCM packaging technology and relies on parallelizing compiler technology to generate code for this architecture.

To understand the potential performance and cost advantages of tightly coupled microprocessors, we will look at the performance and cost of recent microprocessors. Figure 2 shows the relative performance and relative cost for the microprocessors shown in Figure 1. Cost is estimated as being proportional to the square of the normalized die area, which is consistent with die area and cost estimates of recent microprocessors [14]. In these cost estimates we ignore the cost of the large SRAM secondary caches that are required for the microprocessors to reach the performance levels shown. It is assumed that all the processors will use large secondary caches and main memories and so the cost of the off chip memory system will be cancelled out.

The line labeled single chip microprocessors shows the increase in cost and performance and roughly corresponds to a doubling in cost for each unit increase in performance. By coupling together high utility processors using MCM packaging, the Hydra multiprocessor will achieve a greater increases in performance for smaller increases in cost. The ideal increase in performance and cost is shown by the lines labeled MCM multiprocessors. These lines originate from the best performing small and medium sized processors. Even though the small size processors show the best improvement in cost/performance, we would argue that the medium sized processors are preferred because the individual processors have higher performance. This means fewer medium size processors will be required to reach a certain performance level and the task of compiling sequential programs will be easier. Furthermore, fewer chips will require less MCM area which will ultimately translate into a cost savings. Finally, we do not expect this style of multiprocessor architecture to scale beyond eight processors.

In our cost estimates for MCM multiprocessor we have not included the cost of the MCM. Instead of factoring in the cost of the MCM, it is assumed that a single chip processor needs to be packaged and that the packaging of large single chip die with high numbers of pins is a significant portion of the total cost of the packaged chip. In volume production the cost of placing a chip on an MCM-D substrate is expected to be cheaper than the cost of individually packaged chips [3][4].

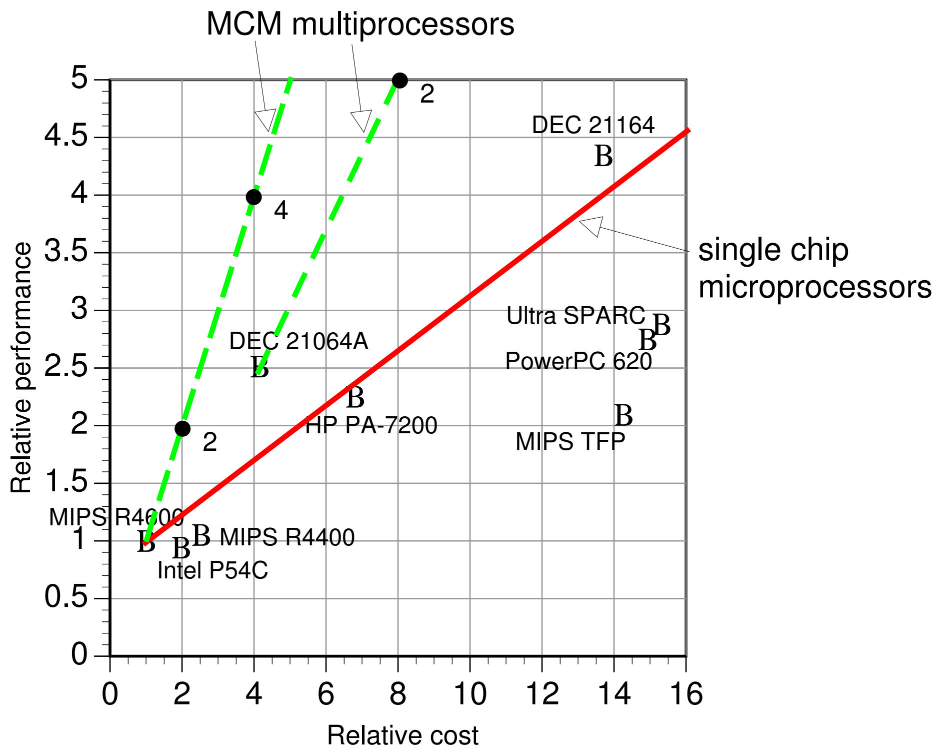

Figure 2. The relative cost and performance of recent microprocessors.

3 The Hydra Multiprocessor Design

3.1 Architecture

Hydra is composed of four high-performance off-the-shelf microprocessors with on-chip cache, a shared second-level cache, and a custom cache controller. Besides providing low-latency and high-bandwidth cache access, the cache controller supports special synchronization primitives that enable low latency coordination of the processors. The processors, the cache controller, and the secondary cache are integrated on an MCM.

The Hydra architecture has a number of advantages from both a computer organization and an implementation technology point of view. First, because Hydra does not rely on extracting ILP from a single thread of control, the microprocessors can be simpler and less complicated than a wide superscalar or a VLIW processor and thus can be made smaller and perhaps faster. Second, the processors communicate through a high bandwidth low latency nonblocking secondary cache, which makes it feasible to exploit fine grained parallelism. Third, the advanced packaging technology allows many short wires which provide low latency connections between the processors and the cache and enough cache bandwidth to make sharing a cache among multiple processors feasible. Fourth, the large cache is constructed from commodity SRAMs which are both cheaper and faster than a large on-chip cache.

The main disadvantage of Hydra architecture is that generating code for it will require sophisticated compilers that can extract multiple independent threads of execution from sequential programs. Given that the practical limit of the amount of ILP that can be found statically or dynamically from a single thread of control is in the range of four to eight instructions per cycle [17], the use of multiple parallel threads of execution will be required to provide further gains in processor performance. The Hydra multiprocessor is designed to exploit multiple threads. While compiler algorithms for parallelizing floating point loop intensive applications are starting to appear (see Section 2.2), it is difficult to parallelize integer programs because of their inherent lack of parallelism. However, Hydra will be able to execute multiple processes in parallel which will provide higher throughput for multiple integer applications running under an operating system.

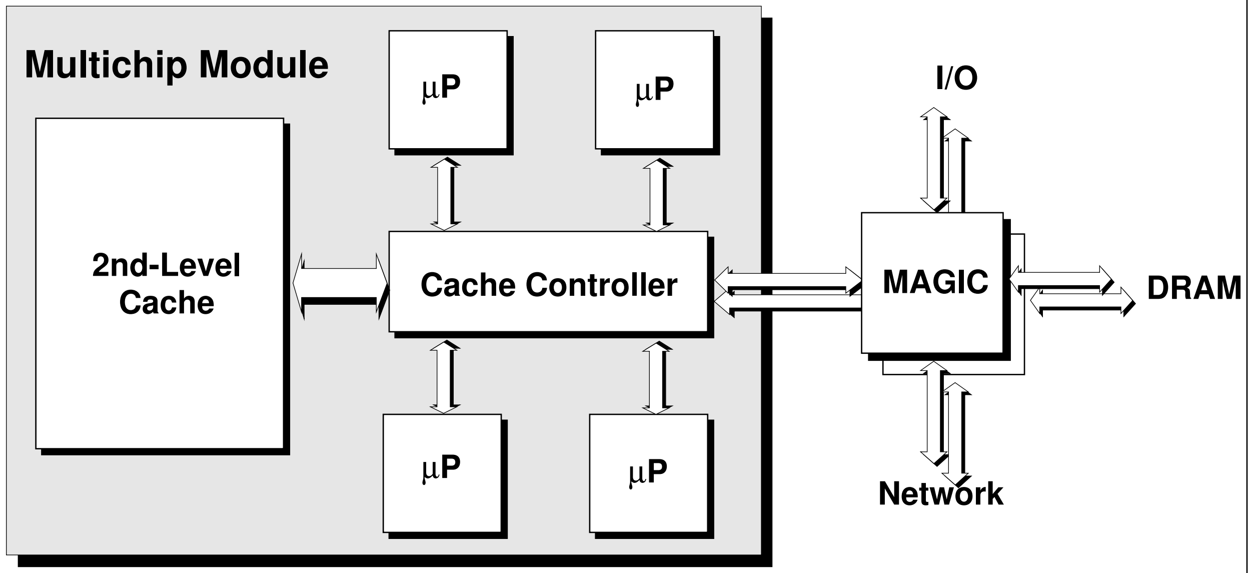

Figure 3. The Hydra architecture.

The initial implementation of the Hydra multiprocessor architecture will use four high-performance DEC Alpha 21064A microprocessors which are interfaced to a secondary cache. As Figure 3 shows, the cache controller is interfaced to the *memory and general interconnect chip* (MAGIC) being developed under the Stanford FLASH multiprocessor project [10]. MAGIC will provide access to DRAM, high performance network and I/O devices. Processors will operate at a frequency of 300 MHz and the communication and synchronization latencies between the processors under 100 ns. These latencies are an order of magnitude shorter than those of recent multiprocessors will make it possible to improve the performance of parallel programs with fine grained sharing. The use of a high performance microprocessor is critical part of the Hydra project because it will demonstrate that tight coupling is possible even under demanding bandwidth and latency requirements. It will also allow the Hydra architecture to have a high overall performance which rivals or surpasses the next generation of microprocessors, demonstrating that the architecture is a realistic microprocessor design alternative.

The major design effort in the Hydra multiprocessor will be definition of the shared cache. The design of a low latency high-speed shared secondary cache presents a number of architectural challenges. These are achieving low cache miss rates providing high bandwidth, achieving low latency communication and synchronization and keeping the primary caches coherent.

The secondary cache miss rate will be kept low by using a large direct mapped cache. The shared cache allows single applications to communicate and share the data through the cache efficiently [13]. For multiprogramming workloads, applications running on different processors will interfere with each other in the cache; however, the multiprocessor operating system that manages the applications will benefit from the shared cache. Because the cache is direct mapped there is the potential for pathological conflict misses. The probability of these misses is increased because four processors will quadruple the number of references seen by the cache. To prevent these misses from becoming a performance bottleneck, a large highly set-associative tertiary cache will be implemented inside the cache controller. This cache will capture the secondary cache conflict misses and eliminate the main memory accesses that would otherwise result form these misses.

A shared cache would not be very effective if it did not provide sufficient bandwidth to fully utilize the capabilities of the processors. To provide this bandwidth we will use wide buses, independent memory banks, pipelining, and nonblocking caches. Very wide secondary cache data buses of 128 bits or greater are becoming standard in high-performance microprocessors. The cache controller with associated data buffers will connect each processor cache bus to any one of the multiple memory modules. To prevent a miss caused by one processor from stalling the other processors the cache must be nonblocking [11]. The design of a nonblocking cache that can support multiple outstanding misses adds a large amount of complexity to the design of the cache, but is necessary to achieve the performance goals. Using these techniques, initial estimates place the cache bandwidth in excess of 10 GB/s. This bandwidth is sufficient to ensure that the cache will not be a performance bottleneck.

Given the number of interconnections and the buffering that is required to provide high-bandwidth, keeping the latency low at the same time is especially difficult. We will achieve low latency by implementing the cache controller in an advanced CMOS process, and the use of thin film MCM (MCM-D) packaging. The high interconnection density provided by MCM-D technology are ideally suited to a shared cache architecture. The Hydra shared cache design will use the interconnection density available on an MCM to achieve far greater bandwidths and only slightly longer latencies than an architecture in which an MCM is used to interconnect a single microprocessor to a secondary cache. Although single processor architectures benefit form MCM packaging, they do not fully exploit the interconnect density and electrical performance advantages of MCM-D. By placing multiple microprocessors on a single substrate using a shared cache architecture like Hydra, MCMs will provide much higher computational density and higher overall system performance.

Low latency synchronization is required to achieve high performance on fine-grained parallel applications. We will implement low latency versions of standard synchronization primitives *load locked* and *store conditional* for implementing course-grain synchronization locks, semaphores and barriers. We will also implement fine-grain synchronization primitives that allow a processor to *wait* for a *signal* from another processor to enforce read-after-write data dependencies. These synchronization primitives will be implemented using a multiported synchronization register file within the cache controller. The primitives will be accessed by reference to predefined memory locations within the address space of each processor. The primitives will provide low overhead coordination of doaccross loops and applications that have unstructured parallelism. These primitives can also be used to implementing low latency barriers and fuzzy barriers [6].

In Hydra the individual processor caches are kept coherent using an invalidation protocol. Each secondary cache line has a full bit vector directory which keeps track of processor primary cache lines that have a copy of the line. When the line is written or replaced the primary cache lines are invalidated. The memory consistency model used in Hydra corresponds to partial store ordering (PSO) [10].This model keeps the order of a read followed by a read and a read followed by a write but allows a write followed by a read and a write followed by a write to complete out of order. PSO allows Hydra to hide write latency because the invalidations that must be performed for a write do not stall subsequent memory operations.

To ensure correct operation there is a total ordering of writes from all the processors at the secondary cache. If two processors that have a copy of a line in their primary caches write to this line at the same time, both of the processors will receive invalidations. In this situation, the write from the first processor that reaches the secondary cache will cause the other processor’s cache line to be invalidated. The bit that indicates that first processor has a copy of the line is kept on. When the write form the second processor reaches the secondary cache it will invalidate the first processor and turn on the bit that indicates that the second processor has a copy of the line even though this copy has already been invalidated.

3.2 Logical design.

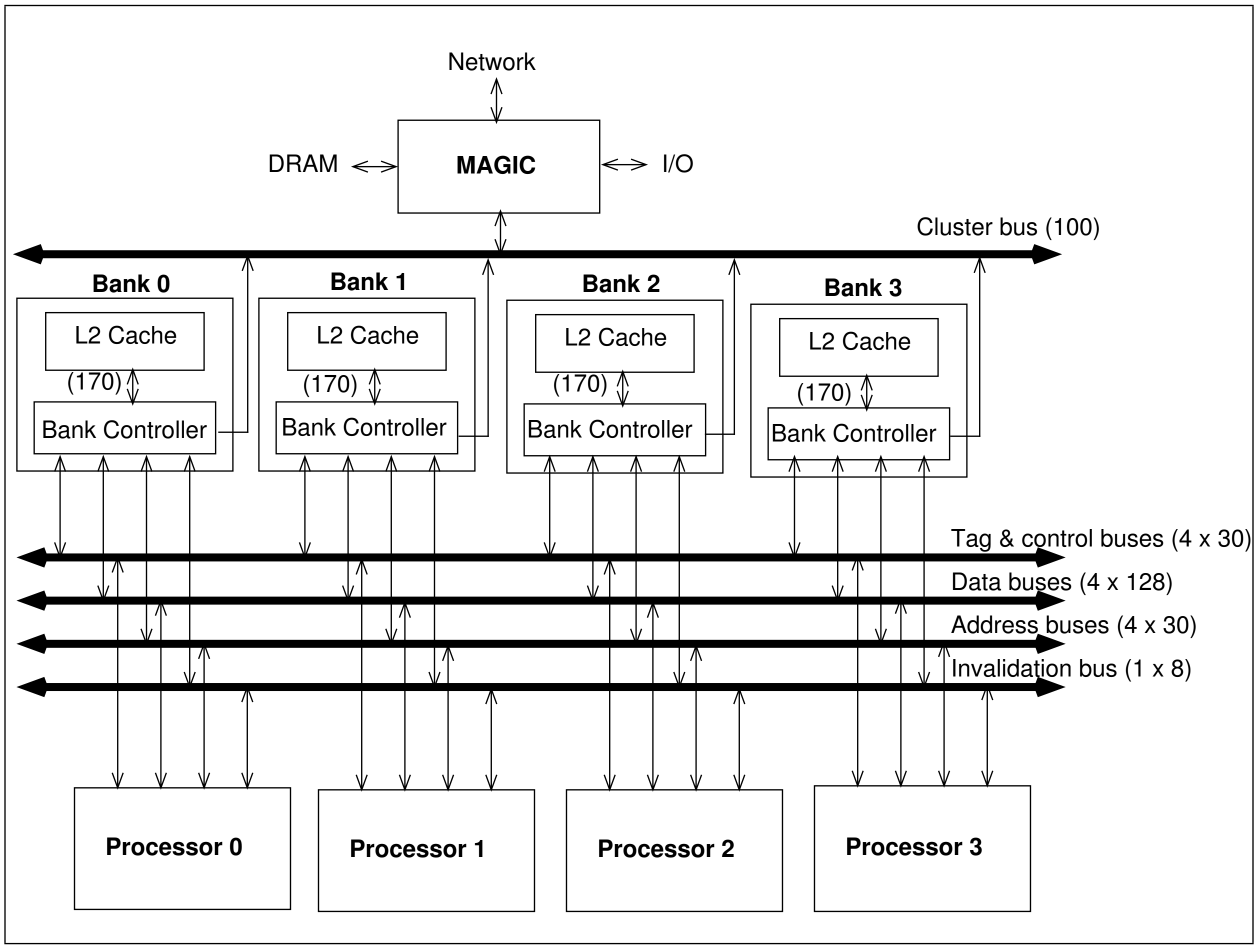

Figure 4. Logical organization of the Hydra Multiprocessor.

The philosophy we have followed in the initial logical design of Hydra multiprocessor is to keep the complexity as low as possible to get correct behavior without creating any obvious performance bottlenecks. A direct consequence of keeping the complexity of the cache controller low is that we introduce as little latency as possible in the access path between the processors and the cache. This minimizes the drawback of sharing the cache. As we will see, our design is able to approach the lowest secondary cache access time possible with the DEC 21064A processor.

Figure 4 shows the logical organization of the Hydra multiprocessor. There are four processors and four cache banks. Each processor has a direct mapped instruction cache and a direct-mapped write-through data cache. Every processor has it own set of address, data, tag, and invalidation signals which are used for memory requests, synchronization requests and to maintain cache coherency. Each cache bank has a separate set of connections for the address, data and tag buses of each processor, but the invalidation bus is shared among the four processors.The numbers in parentheses in the Figure 4 indicate the number of signal wires in each of the buses

As Figure 4 shows that the cache controller is partitioned so that each bank is controlled by its own controller. These controllers communicate with each other using the cluster bus and dedicated control lines. There are a number of advantages to this organization. The operation of the banks is relatively independent, making the controllers separate and thus simpler. Separate controllers also require fewer pins. The cluster bus, which is controlled by the MAGIC, makes the design of a cache controller composed of individual bank controller design easier. The cluster bus is a split-transaction bus that provides support for multiple outstanding memory transactions. To the cluster bus each bank of the secondary cache operates as a separate source of memory transactions. These memory transactions are coordinated by the cluster bus. This eliminates the need to provide extra logic with in the bank cache controllers to coordinate the memory transactions among the banks.

The main components of a cache bank controller are arbitration logic, invalidation logic, synchronization logic, MSHRs (miss status holding registers), tertiary cache and cluster bus interface. The arbitration logic is used to determine which processor accesses the cache each clock cycle. The invalidation logic sends invalidations using the common invalidation bus to all processors which have a copy of a line that has been written or replaced. The information that indicates which processors are sharing a line is kept in a directory within each tag. This information is not exact since the on-chip caches are write thorough, the only way for a processor to leave the sharing list of a line is by being invalidated. Thus, even though the line may have been replaced in one of the processors caches, it will still be invalidated when the corresponding secondary cache line is replaced or written. These invalidations are called useless. We expect that the useless invalidations that are performed will have a small impact on performance because the miss rate and hence the replacement rate of the second level cache will be much smaller than the primary cache and repeated writes to a line will only create useless invalidations on the first write. Because the invalidation bus is shared the cache controllers must arbitrate for the use of this bus, but lines in the caches of all processors can be invalidated at the same time. The synchronization logic implements the synchronization file primitives that were discussed in the first part of this section. The MSHRs are used to hold information on outstanding misses that have been sent to MAGIC, ensuring that subsequent misses to the same line are merged and do not generate new requests [9]. The tertiary cache is a large highly associative cache that is uses to hold lines that have been replaced from the secondary cache. A cache line is only written back to main memory when it is replaced in the tertiary cache. The cluster bus interface generates the correct protocols for the cluster bus. This involves generating requests to and receiving responses from MAGIC.

3.3 MCM floorplan

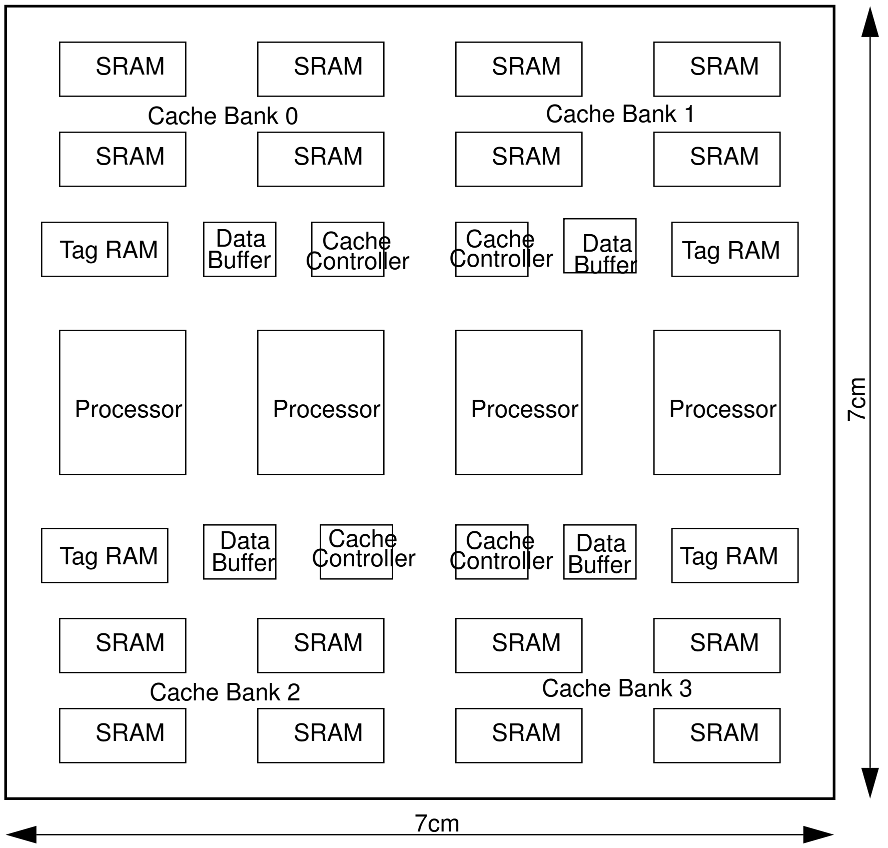

Figure 5. The Hydra multiprocessor MCM floorplan.

A preliminary floorplan for the MCM is shown in Figure 5. The size of the entire module is 7cm by 7cm. At present, only the effect of electrical and area constraints have been considered in the MCM layout. The addition of thermal considerations might change the final layout. On the module there are four processors and four cache banks. Each cache bank is composed of four 32KBx32 synchronous SRAM chips, one tag SRAM chip, and bank controller chips. The tag SRAM uses the same SRAM as the data memory. The SRAM chip is expected to have a 10ns access time, a 5ns cycle time and a die size of 5mm by 10mm. The processor has a die size of 12 mm by 14 mm. The output and bidirectional pads of the processors are designed to drive a capacitive load of up to 40pF without performance degradation. Our analysis shows that due to the low parasitic capacitance of the MCM interconnect this does not constrain the MCM floorplan. Using MCM parameters from MicroModule Systems analysis shows that It will only take 1.4 ns to drive the longest net on the substrate.

The number of signal pins on the cache controller chip is estimated to be 930. This high number of signal pins will only be possible with area array pads such as those available with IBM's C4 process [12]. To reduce the pin requirements we could break the controller into a controller chip and a data buffer chip. The controller chip would provide the arbitration, synchronization and cluster interface while the data buffers would multiplex the data and control lines of each processor to the address and data signals of the SRAM. This partitioning is shown in the floorplan.

3.4 Operation of Hydra

To illustrate the operation of the cache controller we have discussed in the previous sections we show an example of a communication transaction.The following is an example of microprocessor communication through the cache with synchronization (see Figure 6). In this example, the cache controller clock (MClk) has a period of 5 ns and the processor clock (PClk) has a period of 3.3 ns. Processor 2 writes a line into the second level cache and issues a *signal* synchronization. Processor 1blocks on a *wait* synchronization and then reads that line from the cache.

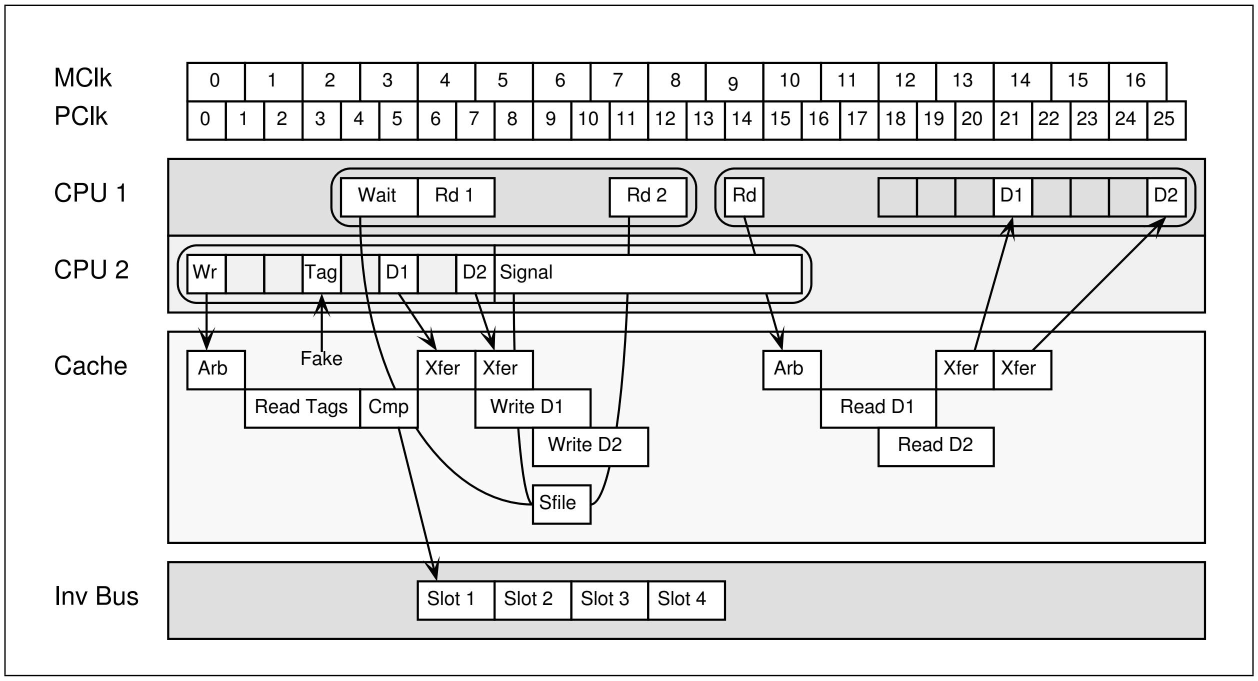

Figure 6. An interprocessor communication example.

A processor write to the secondary cache consists of a tag read/compare cycle, followed by two data write cycles to write the line (assuming a cache hit). On a cache miss, the processor aborts the operation and instead starts a new external memory write. Because of the latency through the interconnection network, it makes little sense to have the processor do the tag compare. Instead, we return a fake tag to the processor and assume responsibility for handling a miss in the cache controller.

When the write request is issued, arbitration is performed to get the path from the processor to the cache ('Arb', 1 memory cycle). The tag memory is then read ('Read Tags', 2 memory cycles) and a comparison is done. If the write is a 'hit', invalidations are sent out to the processors potentially caching the line. Because of the possibility of contention for the invalidation bus (the other processors could be writing), in the worst case the cache controller must wait 3 cycles to perform the invalidations. Meanwhile data is transferred through the network and written into the cache ('Xfer', 'Write D1 & D2').

Processor 1 then performs a *signal* operation. This is encoded as a write to special range of memory locations. Upon seeing the address of the write, the cache controllers can recognize the type of operation and perform the desired action. The rest of the write operation is treated as a no-op. Conceivably, a microprocessor designed with support for low cost synchronization built in would be able to issue such operations more efficiently and concisely. The operation sets a bit in the synchronization file.

Meanwhile, processor 2 performs a *wait* operation. This is encoded as a read from a set of reserved addresses that blocks the processor until the corresponding *signal* is received by the synchronization file. To minimize the time between the receipt of the *signal* and unblocking the processor, we only stall the second read cycle ('Rd 2')

After the *wait* completes, processor 2 issues a normal read operation to the cache (by this time the invalidations from the write will have completed). Arbitration and network latency to the cache take 1 memory cycle (Arb), followed by 2 memory cycles to read the first half of the line ('Read D1'). This goes back to the processor through the network ('Xfer') and is followed immediately by the second half. The processor expects the data back from the cache in a minimum for 4 cycles from the time the address is issued. Because of network latency and memory pipelining, the initial delay is several cycles longer, while the subsequent delay is several cycles less. We handle this by stalling processor until the first data comes back ('D1'), and then buffer the second part ('D2').

In summary, data can be transferred between processors in a minimum of 22 cycles (this is the time from when the initial write address issued by the sender until the critical data is given to sendee —we do not include internal cycles). It is interesting to note that a write followed immediately by a read (representing zero latency synchronization) can only be satisfied in 16 cycles. The latency of a store/signal pair is 15 cycles, and the latency of a wait/load pair is 16 cycles (as seen by the issuing processors).

Given a processor clock cycle time of 3.3 ns, 22 cycles represents 73 ns. This is the minimum time that it would take to communicate some data to another processor in the Hydra architecture. This represents over an order of magnitude lower latency compared to current bus-based multiprocessors.

Shared cache architectures provide a lower interprocessor communication latency at the cost of a higher cache hit time. Table 2 lists the uniprocessor latencies and Hydra latencies for common secondary cache access types in terms of 3.3 ns CPU cycles. The uniprocessor latencies are the lowest that are achievable with the 21064 processor. As expected, the read latency of Hydra is worse than the uniprocessor latency because of the arbitration and chip crossing delay required to share the cache; however, the write latency is better than a uniprocessor because the cache controller in Hydra behaves as a write buffer.

Memory Operation	Uniprocessor latency	Hydra latency
read	8	11
write	12	8

Table 2 Memory latencies.

Memory Operation	Uniprocessor latency	Hydra latency
synchronization	8	8

Table 2 Memory latencies.

4 Preliminary performance of Hydra

To investigate the benefits of shared cache multiprocessor we compare three four-processor machine models. They are a perfect cache model, a bus based model and a Hydra based model.

4.1 Architectural assumptions

The first model is a multiprocessor with a nearly perfect memory system. In this model all memory accesses take one CPU clock cycle. This model represents the upper bound of parallel performance i.e. the performance of this model is determined solely by the parallel coverage of the program (see section 2.2). The second model is a bus based multiprocessor which is representative of most small scale multiprocessor architectures. In the bus model, bus transactions to read data from main memory or from another processors cache have a result latency of 100 cycles with an issue latency of 25 CPU cycles. These bus latencies are consistent with a 300 Mhz processor and a 75 MHz bus which is modeled after the Power Path-2 bus used in the SGI Challenge multiprocessor [6]. The third model is a shared cache secondary cache multiprocessor which is very similar to Hydra.

In all three models, each processor is connected to a 16 Kbyte write-through primary data cache. In the perfect and bus machine models, every primary data cache interfaces to a separate 512 Kbyte write-back secondary cache. In the shared model, all the primary caches interface to a single 2 Mbyte write-back secondary cache. Both the primary and secondary caches are two-way set associative and use 128 byte lines. The latencies of the memory access to the caches for the different models are shown in Table 10. Notice that the cost of sharing the secondary cache is four clock cycles of latency. In this simulation study it is assumed that the processor uses partial store ordering to completely hide the latency of store instructions [7].

Model	primary cache hit	secondary cache hit
Perfect	1 cycle	1 cycle
Bus	1 cycle	10 cycles
Shared	1 cycle	14 cycles

Table 3 Model Cache Memory Latencies

4.2 Simulation Methodology

For this study we use an event driven simulator to simulate the memory system behavior of the different machine models in detail. The simulations are performed by using properly interleaved memory references from a parallel memory reference generator are used to drive the machine models [20]. Because we are primarily interested in the memory system behavior of these models the all instructions except loads are assumed to have a CPI of 1.0. The CPI of loads is accurately determined by the architecture of the different models. Contention in all the models is accurately modeled by the simulator. This includes contention on the shared bus for bus requests that arrive less than 25 cycles apart and the contention at the secondary cache ports for the shared cache model. Due to partial store ordering, the simulation model does not stall the processor for writes; however, the occupancy of writes at the secondary cache ports is modeled in the shared machine and the occupancy caused by writes on the bus is modeled in the bus machine.

4.3 Simulation Results

The performance of the three models on the four SPEC floating point benchmarks (alvinn, ear, mdljdp2,and ora) introduced in Section 2.2 is shown in Figure 5. The bars show the normalized execution time normalized to the execution time of the perfect machine model. Each bar is divided into five sections. The bottom section represents the time that all four processors are busy. This section above this labeled sequential is the time that one of the processors is executing the sequential portions of the application. The section above this represents the time that the processor is stalled waiting for reads. The section labeled *contention* represents contention time at the shared bus in the bus model or contention time at the secondary cache in the shared model. The topmost section labeled *Synchronization* accounts for the time the processor is stalled due to locks and barriers. The height of the sections in each bar is the average over the four processors. The numbers above each bar give the speedup of the machine model over a single processor. For the perfect machine model the single processor execution time is measured with a perfect memory system and for the shared and bus machine models the single processor execution times are measured with the bus model memory system.

The data shows that the Hydra model improves the performance of ear and alvinn substantially. Ear shows the most dramatic performance improvement. The speedup of the Hydra model is 2.9 versus a speedup of 1.6 on the bus machine model. This increase in performance is due to the high parallel coverage of the SUIF compiler coupled with the small granularity and high levels communication in the parallel loops. In contrast, the performance of ora and mdljdp2 is not improved much by the Hydra architecture. Ora shows perfect speedup on all three architectures because the SUIF compiler achieves complete parallel coverage and there is almost no interprocessor communication. Mdljdp2 has roughly the same parallel coverage as ear but it does not reach the same parallel performance because of load imbalance. Mdljdp2 does not communicate very much so the Hydra model shows very little performance improvement compared to the bus model.

5 Conclusions

Improvements in parallelizing compilers, diminishing returns from exploiting ILP in uniprocessors and thin-film MCM packaging technology motivate the exploration of tightly coupled multiple microprocessors. These architectures promise to provide higher performance and lower cost than conventional microprocessor architectures. Multiprocessors execute multiple threads of control which means that they can exploit the parallelism in programs without compromising the clock frequency of the processor as superscalar architectures do. The Hydra multiprocessor is an example of this style of a multiprocessor architecure.The Hydra multiprocessor is designed to reduce the

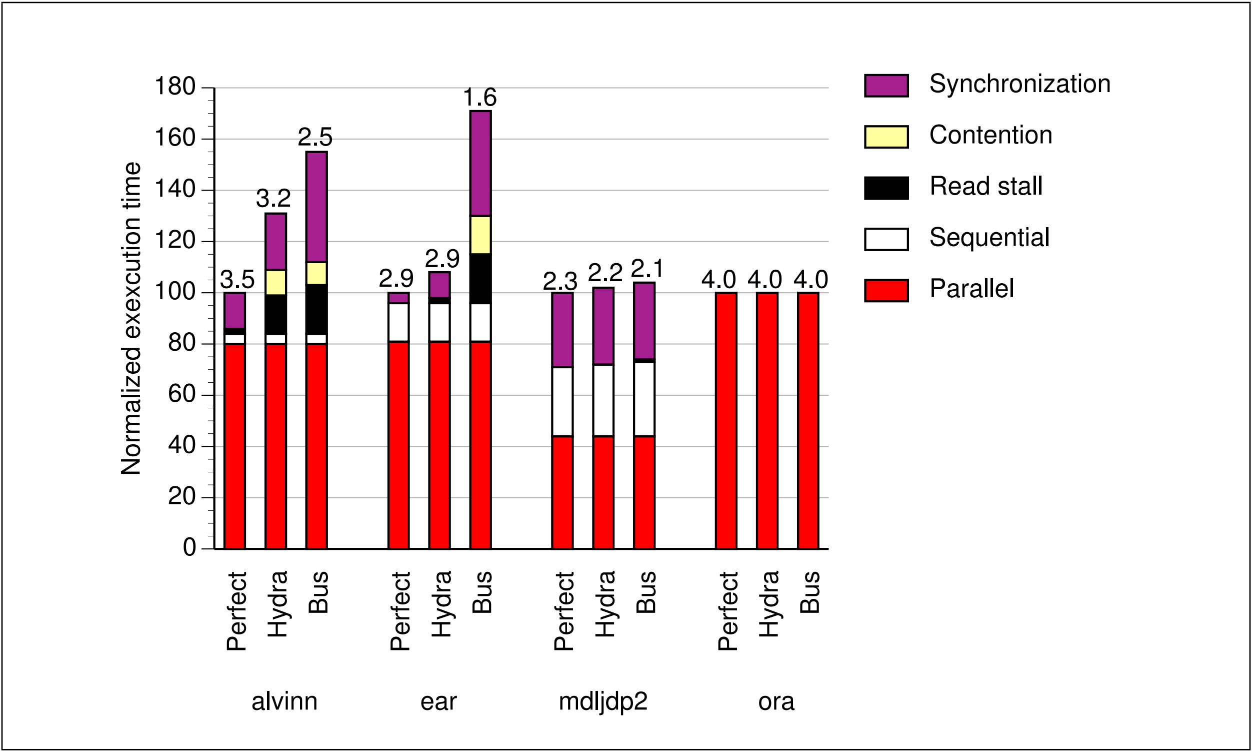

Figure 7. Performance of the Three Machine Models

communication and synchronization latency by an order of magnitude when compared with current shared memory multiprocessors. A key component in achieving this low latency is a second level cache that is shared among the processors. MCM technology makes it feasible to provide the large number of connections required to interconnect high performance processors in a shared cache organization. Initial latency estimates indicate that the interprocessor communication times of under 100 ns are possible. This reduction in communication latency can significantly improve the performance of applications that with fine grained parallelism.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C.-W. Tseng, “An overview of the SUIF compiler for scalable parallel machines,” in Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, 1995.

[2] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[3] T. Costlow, “MCM makers cut costs to gain momentum,” in Electronic Engineering Times, pp. 18, November 28, 1994.

[4] Z. Cvetanovic and D. Bhandarkar, “Characterization of Alpha AXP performance using TP and SPEC workloads”, in 21st Annual Int. Symp. Computer Architecture, Chicago, pp. 60–69, 1994.

[5] P. Franzon, “Solving Problems in Interconnect,” Invited talk at Stanford University, August, 1994.

[6] M. Galles, “The Challenge Interconnect: Design of a 1.2 GB/s coherent multiprocessor bus,” in *Hot Interconnects*, Stanford, CA, pp. 1.1.1-1.1.7, 1993

[7] J. Goodman, “Cache Memories and Multiprocessors-- Tutorial Notes,” in Third Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), Boston, MA, 1989.

[8] R. Gupta, “The fuzzy barrier: a mechanism for high speed barrier synchronization,” in Third Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), Boston, MA, pp. 1989.

[9] J. Hennessy and N. Jouppi, “Computer technology and architecture: an evolving interaction,” IEEE Computer Magazine, vol. 24, no. (9), pp. 18–29, 1991.

[10] J. L. Hennessy and D. A. Patterson, Computer Architecture A Quantitative Approach, 2nd ed. Beta, Morgan Kaufman Publishers, Inc., San Mateo, California, 1993.

[11] D. Kroft, “Lockup-free instruction fetch/prefetch cache organization,” in Proc. 8th Annual Int. Symp. Computer Architecture, pp. 81-87, 1981.

[12] J. Kuskin, et. al, “The Stanford FLASH Multiprocessor,” in 21st Annual Int. Symp. Computer Architecture, Chicago, pp. 302–313, 1994.

[13] B. A. Nayfeh and K. Olukotun, “Exploring the Design Space for a Shared-Cache Multiprocessor,” in 21st Annual Int. Symp. Computer Architecture, Chicago, pp. 166–175 1994.

[14] D. P. Seraphim, R. C. Lasky, and C.-Y. Li, Principles of Electronic Packaging, McGraw-Hill, Inc., New York, New York, 1989.

[15] SPEC, “SPEC Benchmark Suite Release 2.0,” System Performance Evaluation Cooperative, 1992.

[16] Various issues of *Microprocessor Report,* 1993-1994.

[17] D. W. Wall, “Limits of Instruction-Level Parallelism,” WRL Research Report 93/6, Digital Western Research Laboratory, November 1993.