Evaluation of Design Alternatives for a Multiprocessor Microprocessor

Basem A. Nayfeh, Lance Hammond and Kunle Olukotun
Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070
{bnayfeh, lance, kunle}@ogun.stanford.edu

Abstract

In the future, advanced integrated circuit processing and packaging technology will allow for several design options for multiprocessor microprocessors. In this paper we consider three architectures: shared-primary cache, shared-secondary cache, and shared-memory. We evaluate these three architectures using a complete system simulation environment which models the CPU, memory hierarchy and I/O devices in sufficient detail to boot and run a commercial operating system. Within our simulation environment, we measure performance using representative hand and compiler generated parallel applications, and a multiprogramming workload. Our results show that when applications exhibit fine-grained sharing, both shared-primary and shared-secondary architectures perform similarly when the full costs of sharing the primary cache are included.

1 Introduction

With the use of advanced integrated circuit (IC) processing and packaging technology several options for the design of high-performance microprocessors are available. A design option that is becoming increasingly attractive is a multiprocessor architecture. Multiprocessors offer high performance on single applications by exploiting loop-level parallelism and provide high throughput and low interactive response time on multiprogramming workloads [2][15]. With the multiprocessor design option, a small number of processors are interconnected on a single die or on a multichip module (MCM) substrate. The abundance of wires available on-chip or on-MCM make it possible to construct interprocessor communication mechanisms which have much lower latency and higher bandwidth than a single bus-based multiprocessor architecture. Given the multiprocessor communication implementation options available for improving interprocessor communication performance, it is important to understand which mechanism provides the best overall performance on important application classes. The objective of this paper is to characterize the benefits and costs of realistic implementations of two proposed cache-sharing mechanisms that exploit the increased wire density: shared level-1 (L1) cache and shared level-2 (L2) cache. To provide a point of reference, the performance of these architectures is compared to that of a conventional single bus-based shared-memory multiprocessor. All three architectures are simulated using a complete system simulation environment which models the CPU, memory hierarchy and I/O devices in sufficient detail to boot and run the Silicon Graphics IRIX 5.3 operating system. Within our simulation environment, we evaluate the performance of the three architectures using representative hand and compiler generated parallel applications, and a multiprogramming workload. Both kernel and user level references are included in our results.

We present two sets of results. One set with a simple CPU model that does not include latency hiding or the true latencies of the shared-L1 architecture, and a second set with a very detailed and completely accurate CPU model. The results from the simple CPU model are used to classify the parallel applications into three broad classes: applications with a high degree of interprocessor communication, applications with a moderate degree of interprocessor communication and applications with little or no interprocessor communication. For applications in the first class we find that the shared-L1 architecture usually outperforms the other two architectures substantially. For applications in the second class the shared-L1 architecture performs less than 10% better than the other architectures. Finally, for applications in the third class, contrary to conventional wisdom, the performance of the shared-L1 is still slightly better than the other architectures. The second set of results include the effects of dynamic scheduling, speculative execution and non-blocking memory references. These results show that when the additional latencies associated with sharing the L1 cache are included in the simulation model, the performance advantage of the shared-L1 architecture can diminish substantially.

The rest of this paper is organized as follows. Section 2 introduces the three multiprocessor architectures and the architectural assumptions used throughout the paper. Section 3 describes the simulation environment and benchmark applications that are used to study these architectures. Simulation results of the performance of the three multiprocessor architectures are presented in Section 4. In Section 5 we discuss related work and we conclude the paper in Section 6.

In proceedings of 23rd Annual International Symposium on Computer Architecture (ISCA) May 22-24, Philadelphia, PA

2 Three Multiprocessor Architectures

The distinguishing characteristic of shared-memory multiprocessor architectures is the level of the memory hierarchy at which the CPUs are interconnected. In general, a multiprocessor architecture whose interconnect is closer to the CPUs in the memory hierarchy will be able to exploit fine-grained parallelism more efficiently than a multiprocessor architecture whose interconnect is further away from the CPUs in the memory hierarchy. Conversely, the performance of the closely interconnected multiprocessor will tend to be worse than the loosely interconnected multiprocessor when the CPUs are executing independent applications. With this in mind, the challenge in the design of a small-scale multiprocessor microprocessor is to achieve good performance on fine-grained parallel applications without sacrificing the performance of independent parallel jobs. To develop insight about the most appropriate level for connecting the CPUs in a multiprocessor microprocessor we will compare the performance of three multiprocessor architectures: shared-L1 cache, shared-L2 cache, and a conventional single-bus shared main memory. We will see that these architectures are natural ways to connect multiple processors using different levels of the electronic packaging hierarchy. Before we discuss the features that distinguish the three multiprocessor architectures, we will discuss the characteristics of the CPU, which is used with all three memory architectures.

2.1 CPU

This study uses a 2-way issue processor that includes the support for dynamic scheduling, speculative execution, and non-blocking caches that one would expect to find in a modern microprocessor design. The processor executes instructions using a collection of fully pipelined functional units whose latencies are shown in Table 1. The load latency of the CPU is specific to the multiprocessor architecture. To eliminate structural hazards there are two copies of every functional unit except for the memory data port.

Integer	Latency	Floating Point	Latency
ALU	1	SP Add/Sub	2
Multiply	2	SP Multiply	2
Divide	12	SP Divide	12
Branch	2	DP Add/Sub	2
Load	1 or 3	DP Multiply	2
Store	1	DP Divide	18

Table 1 CPU functional unit latencies.

Other characteristics of the processor are 16 Kbyte two-way set associative instruction and data caches, a 32 entry centralized window instruction issue scheme and a 32 entry reorder buffer to maintain precise interrupts and recover from mispredicted branches. Branches are predicted with a 1024 entry branch target buffer. The non-blocking L1 data cache supports up to four outstanding misses.

The CPU is modeled using the MXS simulator [4] which is capable of modeling modern microarchitectures in detail. In this simulator the MIPS-2 instruction set is executed using a decoupled pipeline consisting of fetch, execute and graduate stages. In the fetch stage up to two instructions are fetched from the cache and placed into the instruction window. Every cycle up to two instructions from the window whose data dependencies have been satisfied move to the execute stage. After execution, instructions are removed from the instruction window and wait in the reorder buffer until they can graduate, *i.e.*, update the permanent machine state in program order.

2.2 Shared-L1 Cache Multiprocessor

By the end of the century it will be possible to place multiple processors on a single die. A natural way to interconnect these processors will be at the first level cache as illustrated in Figure 1. The figure shows four CPUs that share a common, 4-way banked write-back L1 cache through a crossbar switching mechanism. This architecture is similar to the M-machine [8]. The primary advantage of this architecture compared to other multiprocessor architectures is that it provides the lowest latency interprocessor communication possible using a shared-memory address space. Low latency interprocessor communication makes it possible to achieve high performance on parallel applications with fine-grained parallelism. Parallel application performance is also improved by processors that prefetch shared data into the cache for each other, eliminating cache misses for processors that use the data later. Other advantages of a shared-L1 cache are that it eliminates the complex cache-coherence logic usually associated with cache-coherent multiprocessors and implicitly provides a sequentially consistent memory without sacrificing performance. This makes the hardware implementation simpler and programming easier.

There are some disadvantages to the shared-L1 cache architecture. The access time of L1 cache is increased by the time required to pass through the crossbar between the processors and cache. We assume that the added overhead of the crossbar switching mechanisms and cache bank arbitration logic would make the total latency of the L1 cache three cycles, even though the cache banks would be pipelined to allow single-cycle accesses. However, all of the memory references performed by the processors will enter the shared-memory, so there is some probability of extra delays due to bank conflicts between memory references from different processors. A third disadvantage is the converse of the shared-data advantage: processors working with different data can conflict in the shared cache, causing the miss rate to increase.

Given the clock rates and complexity of the CPU-cache interface of future microprocessors a single die implementation of the shared-L1 cache is essential in order to maintain a low L1 cache latency. If chip boundaries were crossed, either the L1 latency would be increased to five or more cycles or the clock rate of the processors would be severely degraded. Either of these would have a significant impact on processor performance. The major drawback to the single die implementation today would be the large area and high cost of the die. However, the increasing density of integrated circuit technology will soon make it possible to put four processors on a chip with a reasonable die area. We estimate the die area required for four processors of the complexity of the DEC Alpha 21064A [6] (a dual issue statically scheduled superscalar processor with 32 KB

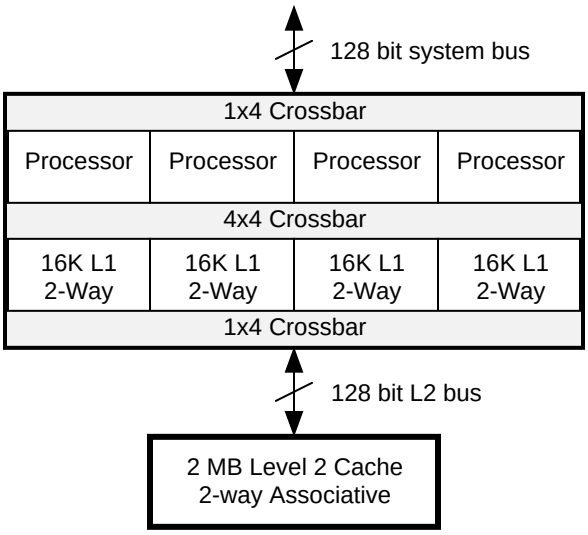

Figure 1. Shared primary cache multiprocessor.

of on-chip cache) and the crossbar interconnect to be 320 mm^2 in 0.35 micron technology. This is the area of the largest microprocessor chips produced today. In a 0.25 micron CMOS technology, that will be available by the end of 1997, the area is reduced to 160 mm^2, which is a medium-sized chip.

The L2 cache and main memories are uniprocessor-like in this system since they are not involved in interprocessor communication. This makes them relatively simple. They are designed with low latencies and heavy pipelining. The degree of pipelining is primarily limited by the 128 bit L2 bus and the 32-byte cache line size that we assume. The transfer time of two cycles sets the lower bounds on L2 cache occupancy. For the purposes of this paper we assume memory latencies and bandwidths that could be attained in a 200 MHz microprocessor with commodity SRAM L2 cache memory and multibanked DRAM main memory: an L2 with 10-cycle latency and 2-cycle occupancy (no overhead), and a main memory with a 50-cycle latency and a 6-cycle occupancy [7]. No cache-coherence mechanisms between the four processors on the chip are required at these levels of the memory hierarchy, since they are below the level of sharing. Only logic to keep the L2 cache coherent with other, completely separate processors on the system bus is required.

2.3 Shared-L2 Cache Multiprocessor

The second multiprocessor architecture we consider shares data through the L2 cache instead of the L1 cache. A possible implementation of this scheme is illustrated in Figure 2. Here four processors and the shared-L2 cache interface are separate dies which are interconnected using MCM packaging [16]. The four processors and their associated write-through L1 caches are completely independent. This eliminates the extra access time of the shared-L1 cache, returning the latency of the L1 cache to 1 cycle. However, the shared-L2 cache interface increases the L2 cache latency from 10 cycles to 14 cycles. These extra cycles are due to crossbar overhead and the delay for additional chip boundary crossings [17].

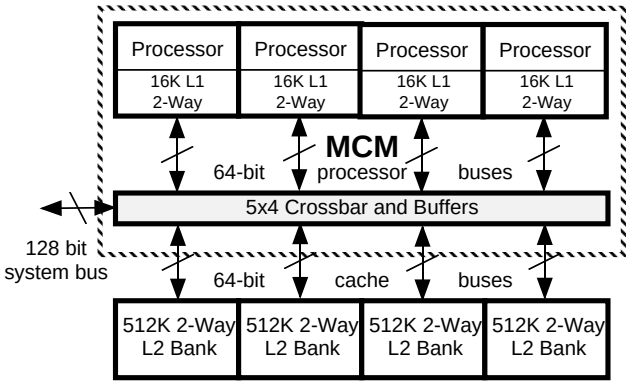

Figure 2. Shared secondary cache multiprocessor.

The write-back L2 cache has four independent banks to increase its bandwidth and enable it to support four independent access streams. To reduce the pin count of the crossbar chip, which must support interfaces to the four processors as well as the four cache banks, the L2 cache datapath is 64 bits instead of 128 bits used in the shared-L1 cache architecture. This does have the side effect of increasing the occupancy of the L2 cache from two to four cycles for a 32-byte cache line transfer. Since we assume L2 cache is designed to supply the critical-word-first, this does not have a significant performance impact. While the additional latency of the crossbar will reduce L2 cache performance compared to the shared-L1 case, only memory accesses that miss in the L1 cache will have to contend with the reduced-performance L2 cache. For the purposes of sharing, the 14 cycle communication latency will allow relatively fine-grained communication on multiprocessor programs but this latency is still much greater than the three cycle sharing latency of the shared-L1 cache architecture.

The shared-L2 architecture implemented with separate chips results in a large number of interchip wires in the system. However, the performance critical path between a processor and its L1 cache remains on chip. The less-frequently used path between the L1 and L2 caches is more tolerant of a few cycles of additional overhead from crossing die boundaries since it is already 10 cycles long. Thus, a system in which smaller dies are packaged on an MCM may have a performance that is close to a shared-L2 cache implemented on a single die while potentially being less expensive to build. Figure 2 shows that the four processor dies and the crossbar die are packaged on an MCM, while the four separate 64 bit datapath interfaces to the cache banks would go off of the MCM to separate SRAMs. Even with the narrower L2 cache datapaths the crossbar chip will still require several hundred signal pins for the interfaces to the processors and cache banks. This high pin count is only feasible today using chips with area pads that are packaged using MCM technology [17].

The main memory for this architecture is identical to the main memory from the shared-L1 case, since the system below the L2 cache is essentially a uniprocessor memory hierarchy. For the purposes of this paper we assume 50 cycles of latency and 6 cycles of occupancy per access. With this configuration, some hardware must also be installed to keep the L1 caches coherent, at least for shared

regions of memory. The simplest way to do this is to assume that the L1 cache uses a write-through policy for shared data and that there is a directory entry associated with each L2 cache line. When there is a change to a cache line caused by write or a replacement all processors caching the line must receive invalidates or updates [17]. This implementation of cache-coherency saves a considerable amount of snooping control logic on the processors. If this control logic could be eliminated the processors could be made simpler than current microprocessors which support snoopy cache coherence.

2.4 Shared-Memory Multiprocessor

The final architecture we consider is a traditional bus-based multiprocessor. The processors and their individual L1 caches run at full, single-cycle cache speeds. This is much like the shared-L2 system. In addition, each processor has its own separate bank of L2 cache that it can access at the full speed of the SRAMs, much like the shared-L1 system (latency = 10 cycles, occupancy = 2 cycles).

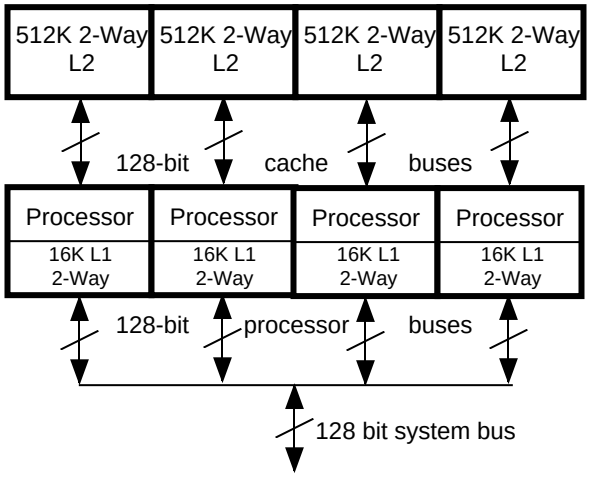

Figure 3. shared-memory multiprocessor.

However, in order to communicate each processor must access main memory through the shared system bus, with its high latencies (still, latency = 50 cycles, occupancy = 6 cycles). This will tend to limit the degree of communication that is possible — each exchange will take 50 or more cycles. Even with systems designed to support cache-to-cache sharing of shared data, the typical times seen will still have a latency of approximately 50 cycles since all three of the other processors on the bus must check their cache tags for a match, agree which processor should source the data, and then recover the necessary data from the correct cache. Since this will usually require accesses to the off-chip L2 caches controlled by the other processors while these caches are busy with local cache traffic, and because we must wait for the slowest processor's response in order to ensure coherency, typical times will often be comparable to memory access times in bus-based systems [7][9].

This architecture represents the capabilities and limitations of current printed circuit board based systems. It is worth noting that the processors must support full snoopy cache coherence of both their L1 and L2 caches. This level of support is included in the latest designs from most leading manufacturers of microprocessors.

System	Access Type	Latency Cycles	Occupancy Cycles
Shared-L1	Level 1 Cache	3	1
	Level 2 Cache	10	2
	Main	50	6
Shared-L2	Level 1 Cache	1	1
	Level 2 Cache	14	4
	Main	50	6
Shared-Mem.	Level 1 Cache	1	1
	Level 2 Cache	10	2
	Main	50	6
	Cache-to-Cache	>50	>6

Table 2 A summary of the ideal memory latencies of three multiprocessor architectures in CPU clock cycles (1 cycle = 5 ns).

Table 2 shows the contention-free access latencies for the three multiprocessor architectures. A common theme is the increased access time to the level of the memory hierarchy at which the processors communicate. A direct result of this is that the further away from the processor communication takes place, the less impact it will have on uniprocessor performance.

3 Methodology

Accurately evaluating the performance of the three multiprocessor architectures requires a way of simulating the environment in which we would expect these architectures to be used in real systems. In this section we describe the simulation environment and the applications used in this study.

3.1 Simulation Environment

To generate the parallel memory references we use the SimOS simulation environment [20]. SimOS models the CPUs, memory hierarchy and I/O devices of uniprocessor and multiprocessor systems in sufficient detail to boot and run a commercial operating system. SimOS uses the MIPS-2 instruction set and runs the Silicon Graphics IRIX 5.3 operating system which has been tuned for multiprocessor performance. Because SimOS actually simulates the operating system it can generate all the memory references made by the operating system and the applications. This feature is particularly important for the study of multiprogramming workloads where the time spent executing kernel code makes up a significant fraction of the non-idle execution time.

A unique feature of SimOS that makes studies such as this feasible is that SimOS supports multiple CPU simulators that use a common instruction set architecture. This allows trade-offs to be made

between the simulation speed and accuracy. The fastest CPU simulator, called Embra, uses binary-to-binary translation techniques and is used for booting the operating system and positioning the workload so we can focus on interesting regions of the execution time. The medium performance CPU simulator, called Mipsy, is two orders of magnitude slower than Embra. Mipsy is an instruction set simulator that models all instructions with a one cycle result latency and a one cycle repeat rate. Mipsy interprets all user and privileged instructions and feeds memory references to the memory system simulator. The slowest, most detailed CPU simulator is MXS, which supports dynamic scheduling, speculative execution and non-blocking memory references. MXS is over four orders of magnitude slower than Embra.

The cache and memory system component of our simulator is completely event-driven and interfaces to the SimOS processor model which drives it. Processor memory references cause threads to be generated which keep track of the state of each memory reference and the resource usage in the memory system. A call-back mechanism is used to inform the processor of the status of all outstanding references, and to inform the processor when a reference completes. These mechanisms allow for very detailed cache and memory system models, which include cycle accurate measures of contention and resource usage throughout the system.

3.2 Applications

We would expect a multiprocessor microprocessor architecture to be used in both high-performance workstations and servers. Therefore, we have chosen workloads that realistically represent the behavior of these computing environments. The parallel applications we use fall into three classes: hand parallelized scientific and engineering applications, compiler parallelized scientific and engineering applications and a multiprogramming workload.

To simulate each application we first boot the operating system using the fastest CPU simulator and then checkpoint the system immediately before the application begins execution. The checkpoint saves the internal state of CPU and main memory and provides a common starting point for simulating the three architectures. Checkpoints also help to reduce the total simulation time by eliminating the OS boot time.

3.2.1 Hand-Parallelized Applications

Most parallel applications are ones which have been developed for conventional multiprocessors. The majority of these applications come from scientific and engineering computing environments and are usually floating point intensive. In selecting applications we have attempted to include applications with both fine- and coarse-grained data sharing behavior.

Eqntott is an integer program from the SPEC92 benchmark suite [27] that translates logic equations into truth tables. To parallelize this benchmark, we modified a single routine — the bit vector comparison that is responsible for about 90% of the computation in the benchmark. Most of the program runs on one *master* processor, but when the comparison routine is reached the bit vector is divided up among the four processors so that each processor can check a quarter of the vector in parallel. The amount of work per vector is small so that the parallelism in this benchmark is fine-grained.

MP3D [14] is a 3-dimensional particle simulator application and is one of the original SPLASH benchmarks described in [22]. MP3D places heavy demands on the memory system because it was written with vector rather than parallel processors in mind. The communication volume is large, and the communication patterns are very unstructured and read-write in nature. As such, it is not considered to be a well-tuned parallel application, but could serve as an example of how applications initially written for vector machines perform as they are ported to shared-memory multiprocessors. In our experiments we simulated MP3D with 35,000 particles and 20 time steps.

Ocean is a well written and highly optimized parallel application that is part of the SPLASH2 benchmark suite [26]. Ocean simulates the influence of eddy and boundary currents on the large-scale flow in the ocean using a multigrid solver method. The ocean is divided into a $n \times n$ grid and each processor is assigned a square sub-grid. Each processor communicates with its neighbors at the boundaries of the subgrid. Each processor's working set is basically the size of the processor's partition of a grid, and is mostly disjoint from the working sets of the other processors. For the results in this paper we use an input data set that has 130×130 grid points.

Volpack is a graphics application that implements a parallel volume rendering algorithm using a very efficient technique called shear-warp factorization [12]. The parallel algorithm uses a image based task decomposition in which each processor computes a portion of the final image in parallel. There are three steps to the parallel algorithm. In the first step a lookup table is computed in parallel for shading the voxels (volume elements), in the second step each processor computes a portion of the intermediate image by selecting tasks from a task queue. Each task entails computing voxels of contiguous scan lines that intersect the portion of the assigned portion of the intermediate image. In the last step, the intermediate image is warped in parallel. To minimize load imbalance, the algorithm uses dynamic task stealing among the processors. The application uses a 128^3 voxel medical data set with a task size of two scanlines. The small task size is selected to maximize processor data sharing and minimize synchronization time.

3.2.2 Compiler Parallelized Applications

Recent advances in parallel 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. Aliases occur since C programs use pointers and 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

In our experiments we use two SPEC92 floating point benchmarks that have been parallelized by the Stanford University Intermediate Format (SUIF) compiler system [2]: ear and the FFT kernel from nasa7. Ear is a C program that models the inner ear, while the FFT kernel is written in FORTRAN. The FFT kernel can be parallelized using vector-like compiler analysis, but ear requires pointer disambiguation because it is a C application and interprocedural analysis in order to detect the parallelism. Using these techniques, over 90% of the execution time of these programs can be parallelized by the SUIF compiler.

The parallel behavior of the two automatically parallelized programs varies widely. In FFT the compiler is able to find outer loops that are parallelized across procedure boundaries so that the granularity of parallelism is fairly large. However, ear consists of very short running loops that perform a small amount of work per loop iteration; consequently, it is has an extremely small grain size.

3.2.3 Multiprogramming and OS Workload

One of the main uses of multiprocessor architectures will be for increasing the throughput of a multiprogramming workload that consists of both user activity and OS activity. To model this type of compute environment we use a program development workload that consists of the compile phase of the Modified Andrew Benchmark [18]. The Modified Andrew benchmark uses the gcc compiler to compile 17 files. Our benchmark eliminates the two phases of the benchmark that install the object files in a library and delete the object files. We use a parallel make utility that allows up to four compilation processes to run at a time. Two of these parallel makes are launched at a time.

4 Results

In this section, we present the results for the three architectures running the applications described in Section 3. First we present the results for all seven of the applications using the Mipsy CPU model. These results are simple to understand because the CPU model stalls for all memory operations that take longer than a cycle. Thus, all the time spent in the memory system contributes directly to the total execution time. The Mipsy results for the shared-L1 architecture are optimistic because we do not include bank contention and we assume a 1-cycle hit time. The motivation for this is to avoid penalizing the shared-L1 architecture on a CPU simulator that has no support for the latency hiding mechanisms of non-blocking caches or dynamic instruction scheduling. To investigate the effect of these latency hiding mechanisms, the three cycle hit latency, and bank contention on the performance of the shared-L1 architecture, we use the MXS simulator. We present results from the MXS simulator for three applications with different sharing and working-set characteristics.

We present the Mipsy results for each application normalized to the speed of the baseline shared-memory processor. Each graph breaks down the execution time in to the different sources of delay modeled in the CPU and memory-system models. Since over half of the time in each benchmark is devoted to the CPU, the graphs are truncated, and only show the top 50% of the execution time. This portion of the graphs allows us to focus on the percentage of time spent waiting for delays caused by various portions of the memory system; the component of interest in this investigation. The time spent waiting for a spin lock or for barrier synchronization is included in the CPU time. The speed of the load-linked and store-conditional memory operations used to implement these synchronization primitives affects the amount of time the processors spend synchronizing. Consequently, the level of data sharing in the memory hierarchy which affects the speed of these memory operations and the load balance in the application changes the amount of CPU time shown in the graphs.

To provide insight into application behavior we present miss rate data on the working set and sharing characteristics of the applications. We break down the miss rates for the L1 data cache and unified L2 cache for all three architectures. Each cache miss rate is broken into two components: replacement miss rate (L1R, L2R) and invalidation miss rate (L1I, L2I). Replacement misses are composed of cold, capacity and conflict misses. These misses are affected by the organization of the cache and the level of the memory hierarchy at which the processors share data. Invalidation misses are due to communication and are most affected by the level of sharing in the architecture, although the cache line size will affect the number of false sharing misses. All the cache miss rates we present are local miss rates: they are measured as misses per reference to the cache.

4.1 Hand Parallelized Applications

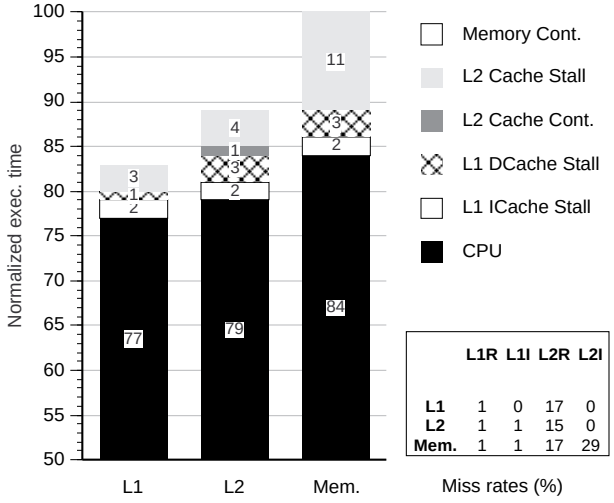

Figure 4. Eqntott performance.

Figure 4 shows that on the Eqntott benchmark, the shared-L1 cache architecture enjoys a significant performance advantage compared to the other architectures. The low L1R miss rate in the shared-L1 architecture and the high L1I miss rate in the shared-L2 and shared-memory architectures indicate that Eqntott is characterized by a small working set and a high communication to computation ratio. The L1I miss rate of 1% may not seem very high, but we shall see that it is high compared with the other benchmarks. The high communication to computation ratio is due to the fact that Eqntott is parallelized inside the inner, vector comparison loop. Every time this loop is executed, the four processors synchronize at a barrier

and the master processor transmits copies of the last three quarters of the vectors being compared to the slave processors so they can perform their portion of the comparison. With the shared-L1 architecture, these copy operations are free—all processors are reading from the same L1 caches, and can read the information directly without any overhead. With the shared-L2 and shared-memory architectures, however, the vectors must be copied from the L1 cache of the master processor to the L1 caches of each slave. This operation requires several cache misses in each of the slaves, slowing down the machines proportionally to the communication time required for each of these cache misses. With a larger data set the advantage enjoyed by the shared-L1 architecture would be less pronounced because the L1 cache replacement misses would make the communication miss time a smaller percentage of the total execution time.

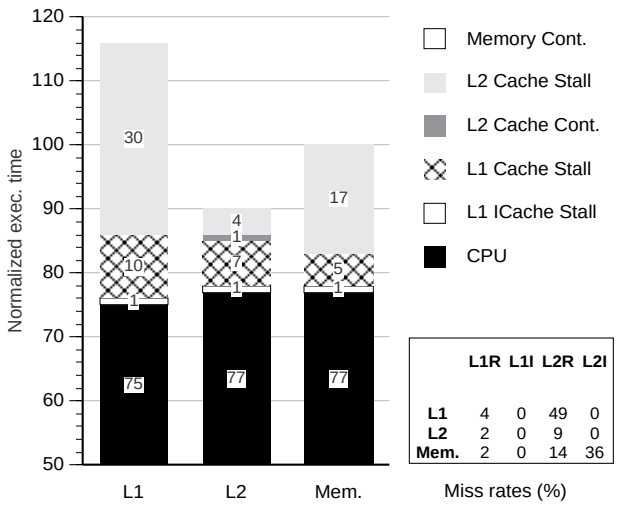

Figure 5. MP3D performance.

The performance of MP3D on the three architectures, which is shown in Figure 5, provides an interesting result. MP3D is known as a parallel application with large amounts of communication traffic; however, the L1 miss rates of all three architectures is dominated by replacements misses (L1R). This indicates that even though there is a lot of communication between the processors cache lines are replaced before they can be invalidated. The L1R miss rate for shared-L1 architecture is over twice the miss rate of the other two architectures which indicates that the references from different processors are conflicting in the L1 cache. The L2R miss rate of shared-L2 cache is relatively low which means that all the important working sets of MP3D fit into the 2 MB L2 cache. We see that the L2 miss rate of the shared-memory architecture is dominated by invalidation misses due to the heavy communication requirements of the application. The communication latency advantage that the shared-L2 architecture has over the shared-memory architecture results in a significant performance improvement. This performance advantage does not extend to the shared-L1 cache because the L1 cache is not large enough to contain all of the important working sets of MP3D. In fact, the high L1R miss rate causes a substantial increase in the L2R miss rate and causes the shared-L1 architectures to perform much worse than the shared-memory architecture. To verify that the high L2 miss rate is due to conflict misses we increased the set associativity of the L2 cache. When the L2 cache is 4-way set associative, the miss rate drops to 10% which is similar to the miss rates of the other two architectures. MP3D demonstrates that there are applications with reference patterns that can cause the shared-L1 architecture to perform poorly even when the application has a significant amount of shared read-write data.

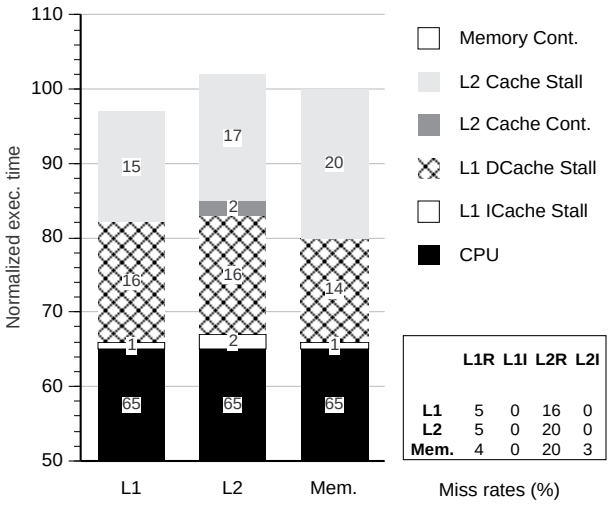

Figure 6. Ocean performance.

The performance of Ocean is shown in Figure 6. Ocean causes large numbers of L1R misses on all three architectures. The large bandwidth required to support these misses puts the shared-L2 architecture at a disadvantage because its narrower buses provide less bandwidth and the shared-L2 cache has a higher hit time. The contention at the L2 cache caused by the write-through policy of the L1 caches also degrades the performance of the shared-L2 architecture. The shared-L1 and shared-memory architectures avoid most of these performance losses through the use of wider buses and write-back L1 caches. Only a relatively small proportion of the cache misses are required for communication purposes — there is only a small amount of communication required at the edges of the four 65x65 subgrids assigned to each processor compared with the amount of work within each subgrid — the shared-cache architectures do not offer much advantage over the shared-memory architecture. The net result of all these effects is that the shared-L1 architecture performs slightly better than the shared-memory architecture and the shared-L2 architecture performs slightly worse.

Volpack performance is shown in Figure 7. Volpack's behavior is characterized by a low L1R miss rate of 1% and a negligible L1I miss rate. Under these conditions the memory system performance of the shared-L1 architecture and the shared-L2 architecture is similar. These architectures both outperform the shared-memory architecture slightly because there is a non-negligible L2I miss rate due to communication in the shared-memory architecture. There is a significant amount of synchronization in Volpack and the reduction in synchronization time provided by the shared cache architectures shows up as a reduction in CPU time in the graph.

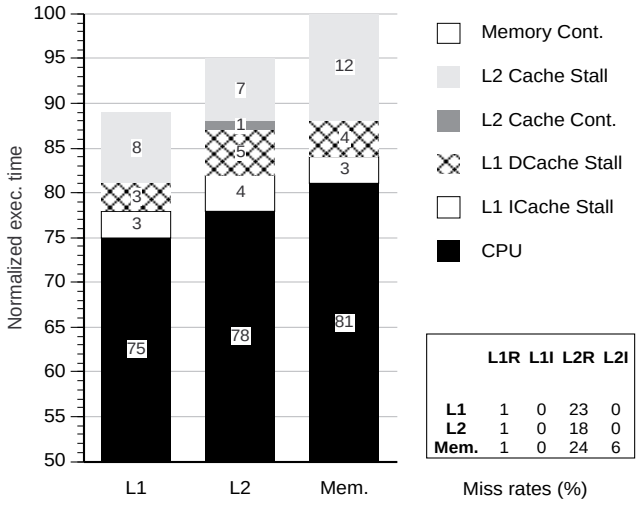

Figure 7. Volpack performance.

4.2 Compiler Parallelized Applications

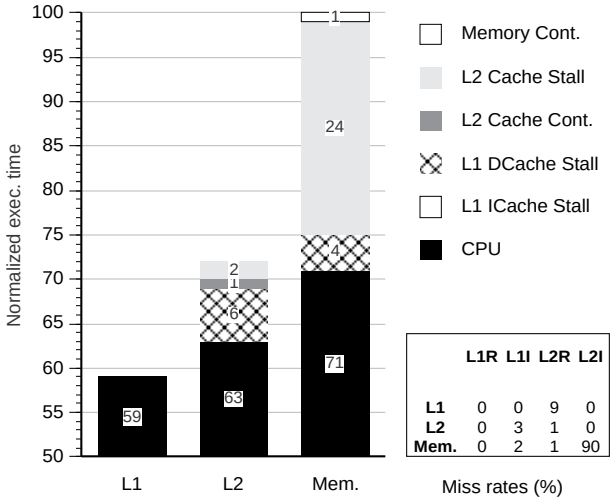

Figure 8. Ear performance.

The results from the automatically parallelized Ear benchmark, shown in Figure 8, are similar to the hand-parallelized Eqntott benchmark, since both are very small, fine-grained parallel benchmarks that have a high ratio of communication to computation. Ear is even more fine-grained than Eqntott. Ear has a negligible L1 miss rate on the shared-L1 cache architecture which indicates that all important working sets fit completely inside the L1 cache; but it has the highest L1I miss rate for the shared-L2 and shared-memory architectures of any of the applications we study which indicates a high rate of interprocessor communication. Consequently, the performance of the shared-L1 architecture on Ear is impressive: there are almost no memory system stalls. The performance of the shared-L2 architecture is not quite as good, but is considerably better than the shared-memory architecture.

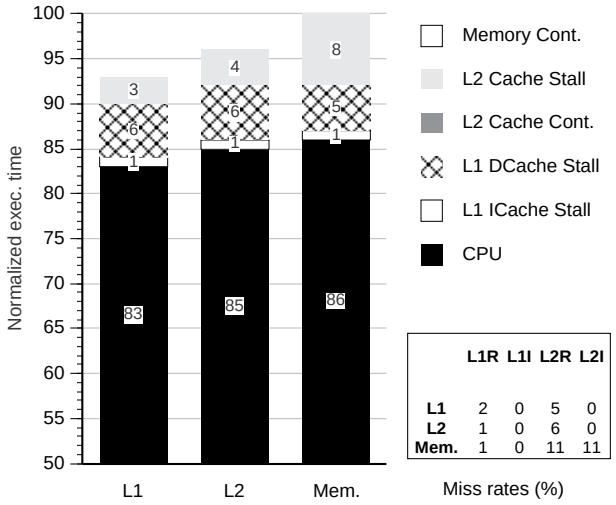

Figure 9. FFT performance.

Figure 9 shows that the results from FFT are similar to those of Volpack. Again, the low L1R and L1I miss rates result in very similar memory system performance from the shared-L1 and shared-L2 architectures. Both of these architectures do slightly better than the shared-memory architecture due to increased L2R and L2I misses in the shared-memory architecture. FFT shows that for applications with relatively large grain sizes and little shared data, the three architectures all offer fairly similar performance.

4.3 Multiprogramming and OS Workload

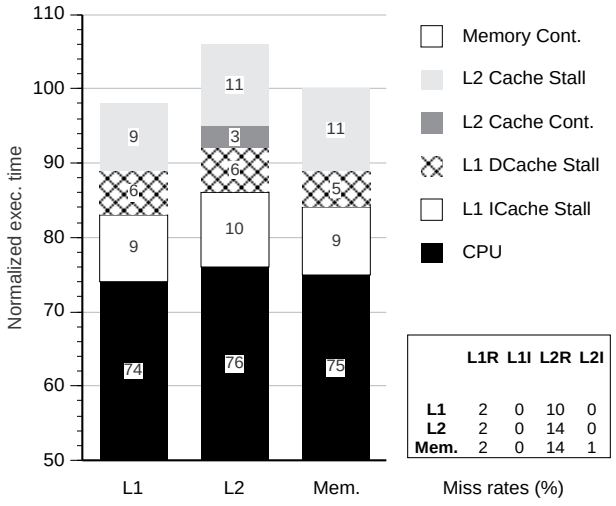

Figure 10. Multiprogramming and OS workload performance.

The multiprogramming and OS workload differs from the applications we have considered so far in that no user-level data is shared

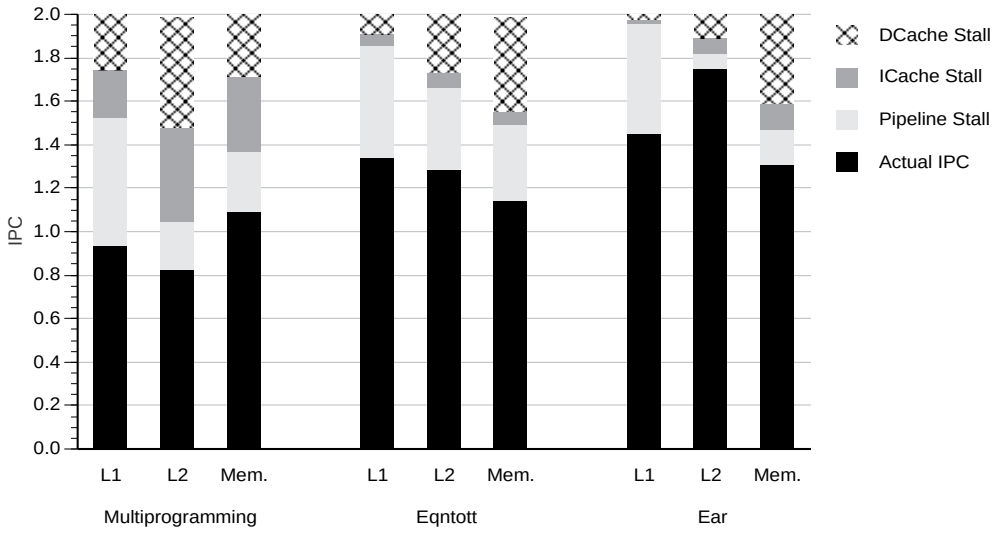

Figure 11. Performance of the three architectures with dynamic superscalar processors.

since the programs are run as multiple independent processes with separate address spaces. Figure 10 shows the performance of multiprogramming OS and workload on the three architectures. Another key difference between the multiprogramming workload and the other applications in this study is the large size of the instruction working set. So far we have concentrated on data references because these dominate in the previous applications. In the multiprogramming workload, the code path lengths in gcc compiler are much longer than the other applications and there is a significant amount of kernel activity and so the instruction working set is very large. The instruction cache stall time, which is an indicator of whether the instruction working set fits into the separate 16KB processor instruction caches, is not important in the other applications (except Volpack), but it takes up 9–10% of the execution time on the multiprogramming workload.

Surprisingly, the shared-L1 architecture does not suffer a higher L1R cache miss rate than the other architectures even though this workload is composed of multiple independent processes that should conflict with each other in the L1 cache. The reason for this is that the OS workload is comprised of relatively small processes with small data working sets that fit comfortably into the 64 KB shared cache [21]. Furthermore, 16% of the non-idle execution time is spent in the kernel. As the kernel executes on different processors the shared-L1 cache provides overlap of the kernel data structures and lower communication and synchronization latency.

The overall memory system performance of the shared-L1 architecture and shared-memory architecture is similar. The shared-L1 architecture has slightly more L1 data cache stall time and slightly less L2 cache stall time than the shared-memory architecture. The shared-L2 architecture performs 6% worse than the shared-memory architecture due to the added cost of L1 cache misses and the contention at the L2 cache ports caused by write data from the write-through L1 data cache. The port contention is significant because the OS workload has a much larger percentage of stores than the other applications.

4.4 Dynamic Superscalar CPU Results

Using our most detailed dynamic superscalar CPU model, MXS, we now include the effects of dynamic scheduling, speculative execution and non-blocking memory references in determining overall performance. In addition, we also include the three cycle L1 hit latency and L1 bank contention for the shared-L1 architecture.

We select three applications to evaluate under MXS based on their working set and sharing characteristics. The multiprogramming workload exhibits relatively large instruction and data working sets with no sharing since the programs run in different address spaces, although a minimal amount of sharing occurs within the kernel. Eqntott has moderate instruction and data working set sizes and a medium to high amount of sharing. Finally, Ear has small data working sets, but exhibits a large amount of fine-grain sharing.

Figure 11 shows the relative performance, measured in instructions per cycle (IPC), of the three applications. As described in Section 2.1, we use a 2-way issue processor model with an ideal IPC of 2. In addition to the actual IPC achieved, we show the loss in IPC due to data and instruction cache stalls, and pipeline stalls. For the shared-L1 architecture, the effects of the additional shared-L1 cache latency and L1 bank contention are counted as pipeline stalls. Using IPC to compare performance between different multiprocessor architectures can be problematic because the architectures execute different numbers of instructions depending on synchronization time; however we find that the IPC results track the actual application performance.

We focus first on the results of the multiprogramming and OS workload. Comparing the detailed CPU results to the simple CPU results (see Figure 10) we see that the shared-memory architecture

now outperforms the shared-L1 architecture by 17% and the performance gap between the shared-memory and the shared-L2 architecture has widened to 33%. The reason for this is that there is little or no interprocessor communication in the multiprogramming workload, so the cost of sharing the cache leads to performance losses. In the case of the shared-L1 architecture, the cost of sharing the cache is the three cycle L1 hit time which increases the losses due to pipeline stalls. For the shared-L2 architecture, the performance losses are due to limited L2 bandwidth and bank contention.

Turning our attention to Eqntott, we see that the shared-L1 architecture performs 18% better than the shared-memory architecture and the shared-L2 architecture performs 12% better. Thus, the performance of the three architectures stays in the same order as the simulation results with the simple CPU model (see Figure 4); however, the performance gap between the shared-L1 architecture and the shared-memory architecture has narrowed; again, this is due to the inclusion of the three cycle hit time and bank contention.

Finally, Figure 11 shows that Ear's instruction cache and data cache stall times decrease substantially in going from the shared-memory to shared-L1 cache architecture. This is consistent with the simple CPU model results (see Figure 8). However, a large increase in pipeline stalls occurs due to the additional shared-L1 cache hit-time and bank contention which is not completely hidden by the processor's dynamic instruction scheduling. In contrast, the shared-L2 cache architecture is also able to reduce the instruction cache and data cache stall times considerably, without the associated costs of the shared-L1 cache architecture. Thus, the shared-L2 architecture achieves the best performance overall.

5 Related Work

Shared caches have been proposed and investigated in the context of multiprocessors and multithreaded processors. Shared primary caches have been proposed by Dally, *et. al.*, [8] for the M-machine and by Sohi, *et al.*, [23] for Multiscalar processors; however, the performance costs of the shared cache are not really addressed in these proposals. Nayfeh, *et. al.*, [15] evaluated the performance of shared primary cache for a single chip multiprocessors. Their results showed that for a variety of applications at cache sizes larger than 32 Kbytes adding another processor to a uniprocessor chip improves the performance more than doubling the cache size. Tullsen, *et. al*., [24] described a shared primary cache for a processor with simultaneous multithreading. They compared private primary caches with separate primary caches for up to eight threads from separate processes and concluded that the cache configuration with the best overall performance is a separate instruction cache per thread with a combined data cache; however, when fewer than eight threads are running some of the private caches are not used at all. Though previous work has considered shared cache behavior, to our knowledge this is the first time a study of shared cache multiprocessors has been done using a detailed and realistic machine architecture, a complete operating system and varying application domains.

6 Conclusions

In this paper we compare the performance of three realistic architectures that could be used in a multiprocessor microprocessor. These architectures demonstrate the interconnection of processors at different levels of the memory hierarchy, with widely varying communication characteristics as a result. We have evaluated these architectures with parallel applications that have been parallelized by hand and by a compiler, and with a multiprogramming workload. All programs are simulated running on a commercial operating system using SimOS and our memory system simulator.

Our results, obtained with the simple CPU model, Mipsy, show that the parallel applications fall into three broad classes: applications such as Ear, MP3D and Eqntott that have a high degree of interprocessor communication, applications such as Volpack and FFT that have a moderate degree of interprocessor communication and applications such as Ocean and the multiprogramming workload with little or no interprocessor communication. For applications in the first class we find that the shared-L1 architecture usually out performs the shared-memory architecture substantially (20–70%). The exception is MP3D which performs 16% worse than the shared-memory architecture due to conflict misses in the L2 cache caused by conflict misses in the shared-L1 cache. For applications in the second class the shared-L1 architecture performs 10% better than the shared-memory architecture. Finally, for applications in the third class, contrary to conventional wisdom, the performance of the shared-L1 is still slightly better than the shared-memory architecture. There are two reasons for this. First, the shared 64KB 2-way associative L1 data cache is large enough to accommodate the important working sets of independent threads running on different processors so that the conflict miss rate is low. We noticed that the miss rate of the shared-L1 data cache is very similar to the miss rates of the individual data caches in the other architectures. Second, when there is interprocessor communication, it is handled very efficiently in the shared-L1 architecture. The shared-L2 architecture tracks the performance gains of the shared-L1 architecture, but to a lesser degree because the interprocessor communication latencies are a factor of ten larger than the shared-L1 architecture. Again, the exception is MP3D for which the shared-L2 performs 11% better than the shared-memory. For applications in the third class the shared-L2 architecture performs slightly worse than the shared-memory architecture due to contention at the L2 cache ports.

Using our most detailed CPU model, MXS, we included the effects of dynamic scheduling, speculative execution and non-blocking memory references, as well as, the three cycle L1 cache hit latency and L1 bank contention for the shared-L1 architecture. Our results for three applications with different sharing characteristics show that the relative performance of the shared-L1 architecture can diminish substantially, even with dynamic scheduling. In contrast, both the shared-L2 and shared-memory architectures retain much of the relative performance predicted by the simple CPU results. These MXS results indicate that for programs which exhibit fine-grained sharing, the shared-L2 cache architecture provides the same performance benefits as the shared-L1 architecture.

Acknowledgments

We would like to thank Steve Herrod, Edouard Bugnion and Mendel Rosenblum for their help with SimOS, Jennifer Anderson for her help with the compiler parallelized benchmarks, Phil Lacroute for letting us use the Volpack benchmark, Jim Bennet for developing MXS, the reviewers for their insightful comments and Trevor Mudge for serving as our shepherd. This work was supported by ARPA contract DABT63-95-C-0089.

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," *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Compiler*, San Francisco, 1995.

[2] S. Amarasinghe et.al., "Hot compilers for future hot chips," presented at Hot Chips VII, Stanford, CA, 1995.

[3] J. Archibald and J. Baer "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems*, Vol 4, no 4, pp. 273–298.

[4] J. Bennett and M. Flynn, "Performance factors for superscalar processors," Technical report CSL-TR-95-661, Computer Systems Laboratory, Stanford University, February 1995.

[5] Z. Cvetanovic and D. Bhandarkar, "Characterization of Alpha AXP performance using TP and SPEC workloads", *Proc. 21st Annual Int. Symp. Computer Architecture*, Chicago, pp. 60–69, 1994.

[6] *DECchip 21064A Hardware Reference Manual*, Digital Equipment Corporation, Maynard, Massachusetts, 1994.

[7] D. Fenwick, D. Foley, W. Gist, S. VanDoren, and D. Wissell, "The AlphaServer 8000 Series: high-end server platform development," *Digital Technical Journal*, vol. 7, pp. 43–65, 1995.

[8] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The M-Machine multicomputer," *Proc. 28th Annual IEEE/ACM International Symp. on Microarchitecture*, December 1995.

[9] 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

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

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach, 2nd ed*, Morgan Kaufman Publishers, Inc., San Mateo, California, 1996.

[12] P. Lacroute, "Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization," 1*995 Parallel Rendering Symposium*,1995.

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

[14] J. McDonald and D. Baganoff, "Vectorization of a particle simulation method for hypersonic rarefied flow," *AIAA Thermodynamics, Plasma dynamics and Lasers Conference*, June 1988.

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

[16] B. A. Nayfeh, K. Olukotun and J. P. Singh, "The Impact of Shared-Cache Clustering in Small-Scale Shared-Memory Multiprocessors",*Proceedings of the Second Annual Symposium on High-Performance Computer Architecture*, San Jose, CA. February 1996.

[17] K. Olukotun, J. Bergmann, and K. Chang, "Rationale and Design of the Hydra Multiprocessor," Computer Systems Laboratory Technical Report CSL-TR-94-645, Stanford University, 1994.

[18] J. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?," *Summer 1990 USENIX Conference*, pp. 247–256, June 1990.

[19] R10000 Users Manual, version 1.0, Silicon Graphics International. 1995.

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

[21] M. Rosenblum, E. Bugnion, S. Herrod, E. Witchel, and A. Gupta, "The impact of architectural trends on operating system performance," *Proc. 15th ACM symposium on Operating Systems Principles*, Colorado, 1995.

[22] J.P. Singh, W.-D. Weber and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory", *Computer Architecture News*, 20(1):5-44, March 1992.

[23] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar Processors," *22nd Annual Int. Symp. Computer Architecture*, Santa Margherita, Italy, June 1995.

[24] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," *22nd Annual Int. Symp. Computer Architecture*, Santa Margherita, Italy, 1995.

[25] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," *ACM SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems, Philadelphia*, 1996.

[26] S. C. Woo, M. Ohara, E. Torrie, J.P. Singh and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations", *22nd Annual Int. Symp. Computer Architecture*, Santa Margherita, Italy, June 1995.

[27] SPEC, "SPEC Benchmark Suite Release 2.0," System Performance Evaluation Cooperative, 1992.