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.

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

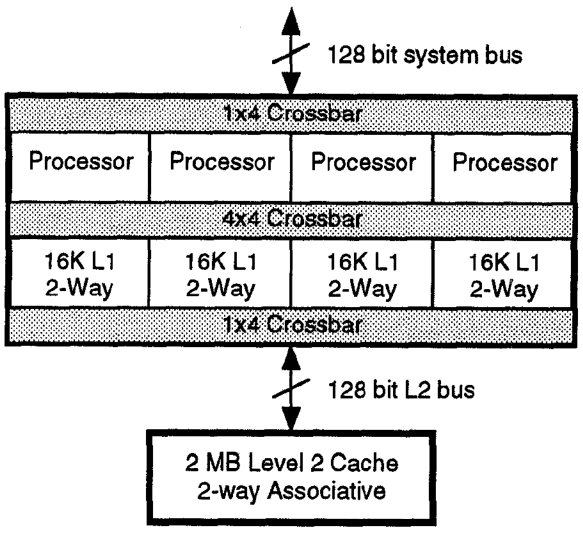

Figure 1. Shared primary cache multiprocessor.

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].

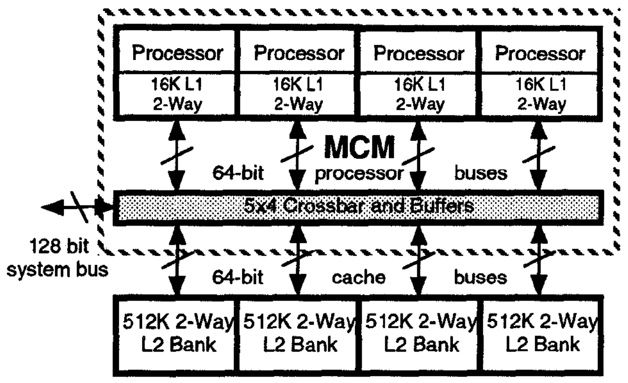

Figure 2. Shared secondary cache multiprocessor.

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).

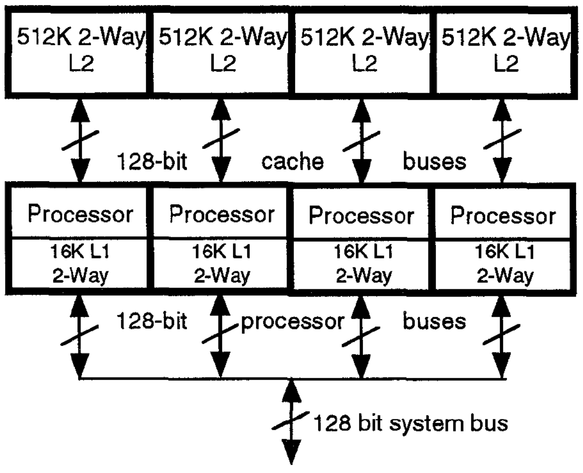

Figure 3. shared-memory multiprocessor.

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

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.