Conjoined-core Chip Multiprocessing

Rakesh Kumar[†], Norman P. Jouppi[‡], Dean M. Tullsen[†]

[†]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

[‡]HP Labs
1501 Page Mill Road
Palo Alto, CA 94304

Abstract

Chip Multiprocessors (CMP) and Simultaneous Multithreading (SMT) are two approaches that have been proposed to increase processor efficiency. We believe these two approaches are two extremes of a viable spectrum. Between these two extremes, there exists a range of possible architectures, sharing varying degrees of hardware between processors or threads.

This paper proposes conjoined-core chip multiprocessing – topologically feasible resource sharing between adjacent cores of a chip multiprocessor to reduce die area with minimal impact on performance and hence improving the overall computational efficiency. It investigates the possible sharing of floating-point units, crossbar ports, instruction caches, and data caches and details the area savings that each kind of sharing entails. It also shows that the negative impact on performance due to sharing is significantly less than the benefits of reduced area. Several novel techniques for intelligent sharing of the hardware resources to minimize performance degradation are presented.

1 Introduction

As the number of transistors available on a die has been increasing according to Moore's Law, the efficiency of their utilization in mainstream processor architectures has been decreasing. At some point, given increased transistor counts, increases in processor performance will saturate. This expectation has stimulated much research in multiprocessor architectures, including single-chip multiprocessors [7] (CMP). Another way to improve the efficiency of large out-of-order processors is to run more than one thread on each processor with multithreading, for example simultaneous multithreading [15] (SMT).

We believe these two approaches to increased efficiency (CMP and SMT) are actually two extremes of a viable spectrum. Most CMP approaches use relatively simple processors [3], which have higher inherent efficiency. SMT processors are usually larger and more complex, resulting in lower single-threaded efficiency, but share almost all processor resources between threads to increase efficiency. Between these two extremes, it is possible to imagine a range of processors sharing varying degrees of hardware between threads. At the end of the range closest to CMPs, pairs of modestly more complex processors could be designed to share a few common components. At the end of the range closest to SMTs, processors could be designed that possess private copies of particular critical resources.

So while the endpoints (CMP and SMT) of this design continuum have been studied extensively, we assert that it is important to study the middle ranges of this spectrum as well, based on two premises. First, that it is unlikely that either extreme represents the optimal design point over the entire range. Second, that the performance costs of targeted sharing of particular resources can be minimized through the application of intelligent, complexity-effective sharing mechanisms. Thus, the selection of the optimal design point cannot be identified without understanding these optimizations.

There are several benefits to sharing hardware between more than one processor or thread. Time-sharing a lightly-utilized resource saves area, increases efficiency, and reduces leakage. Dynamically sharing a large resource can also yield better performance than having distributed small private resources, statically partitioned [15, 6].

Topology is a significant factor in determining what resources are feasible to share and what the area, complexity, and performance costs of sharing are. For example, in the case of sharing entire floating-point units (FPUs), since processor floorplans often have the FPU on one side and the integer datapath on the other side, by mirroring adjacent processors FPU sharing could present minimal disruption to the floorplan. For the design of a resource-sharing core, the floorplan must be co-designed with the architecture, otherwise the architecture may specify sharings that are not physically possible or have high communication costs. In general, resources to be shared should be large enough that the additional wiring needed to share them does not outweigh the area benefits obtained by sharing.

With these factors in mind we have investigated the possible sharing of FPUs, crossbar ports, first-level instruction caches, and first-level data caches between adjacent pairs of processors. Resources could potentially be shared among more than two processors, but this creates more topological problems. Because we primarily investigate sharing between pairs of processors, we call our approach *conjoined-core chip multiprocessors*.

There are many ways that the shared resources can be allocated to the processors in a conjoined configuration. We consider both simple mechanisms, such as fixed allocation based on odd and even cycles, as well as more intelligent sharing arrangements we have developed as part of this work. All of these sharing mechanisms must respect the constraints imposed by long-distance on-chip communication. In fact, we assume in all of our sharing mechanisms that core-to-core delays are too long to enable cycle-by-cycle arbitration of any shared resource.

It is also possible that the best organization of a shared resource is different than the best organization of a private resource. For example, the right banking strategy may be different for shared memory structures than for private memory structures. Therefore, we also examine tradeoffs in the design of the shared resources as part of this study.

The chief advantage of our proposal is a significant reduction in per-core real estate with minimal impact on per-core performance, providing a higher computational capability per unit area. This can either be used to decrease the area of the whole die, increasing the yield, or to support more cores given a fixed die size. Ancillary benefits include a reduction in leakage power due to a smaller number of transistors for a given computational capability.

2 Related Work

Prior work has evaluated design space issues for allocating resources to thread execution engines, both at a higher level and at a lower level than is the target of this paper. At a higher level, CMP, SMT, and CMPs composed of SMT cores have been compared. At a lower level, previous work has investigated both multithreaded and single-threaded clustered architectures that break out portions of a single core and make them more or less accessible to certain instructions or threads within the core.

Krishnan and Torrellas study the tradeoffs of building multithreaded processors as either a group of single-threaded CMP cores, a monolithic SMT core, or a hybrid design of multiple SMT cores in [10]. Burns and Gaudiot [4] study this as well. Both the studies conclude that the hybrid design, a chip multiprocessor where the individual cores are SMT, represents a good performance-complexity design point. They do not share resources between cores, however.

There has been some work on exploring clustering and hardware partitioning for multithreaded processors. Collins and Tullsen [5] evaluate various clustered multithreaded architectures to enhance both IPC as well as cycle time. They show that the synergistic combination of clustering and simultaneous multithreading minimizes the performance impact of the clustered architecture, and even permits more aggressive clustering of the processor than is possible with a single-threaded processor.

Dolbeau and Seznec [6] propose the CASH architecture as an intermediate design point between CMP and SMT architectures for improving performance. This work is probably the closest prior work to ours. CASH shares caches, branch predictors, and divide units between dynamically-scheduled cores. CASH pools resources from two to four cores to create larger dynamically shared structures with the goal of higher per-core performance. However, the CASH work did not evaluate the area and latency implications of wire routing required by sharing. In our work we consider sharing entire FPUs and crossbar ports as well as caches, and attempt to more accurately account for the latency and area of wiring required by sharing. We also consider more sophisticated sharing scheduling techniques which are consistent with the limitations of global chip communication.

3 Baseline Architecture

Conjoined-core chip multiprocessing deviates from a conventional chip multiprocessor design by sharing selected hardware structures between adjacent cores to improve processor efficiency. The choice of the structures to be shared depends not only on the area occupied by the structures but also whether it is topologically feasible without significant disruption to the floorplan or wiring overheads. In this section, we discuss the baseline chip multiprocessor architecture and derive a reasonable floorplan for the processor, estimating area for the various on-chip structures.

3.1 Baseline processor model

For our evaluations, we assume a processor similar to Piranha [3], with eight cores sharing a 4MB, 8-banked, 4-way set-associative, 128B L2 cache. The cores are modeled after Alpha 21164 (EV5). EV5 is a 4-issue in-order processor. The various parameters of the processor are given in Table 1. The processor was assumed to be implemented in 0.07micron technology and clocked at 3.0 GHz.

For the baseline processor, each core has 64KB, 2-way associative L1 caches (I/D). The ICache is single-ported while the DCache is dual-ported (2 R/W ports). The L1 cache sizes are similar to those of Piranha cores. A maximum of 4 instructions can be fetched in a given cycle from the ICache. Linesize for both the L1 caches is 64 bytes. Each core has a private FPU. Floating point divide and square root are non-pipelined. All other floating point operations are fully pipelined. The latency for all operations is modeled after EV5 latencies.

Cores are connected to the L2 cache using a point-to-point fully-connected blocking matrix crossbar such that each core can issue a request to any of the L2 cache banks every cycle. However, one bank can entertain a request from only one of

2K-gshare branch predictor
Issues 4 integer instrs per cycle, including up to 2 Load/Store
Issues 2 FP instructions per cycle
4 MSHRs
64 Byte linesize for L1 caches, 128 Byte linesize for L2 cache
64k 2-way 3 cycle L1 Instruction cache (1 access/cycle)
64k 2-way 3 cycle L1 Data cache (2 access/cycle)
4MB 4-way set-associative, 8-bank 10 cycle L2 cache (3 cycle/access)
4 cycle L1-L2 data transfer time plus 3 cycle transfer latency
450 cycle memory access time
64 entry DTLB, fully associative, 256 entry L2 DTLB
48 entry ITLB, fully associative
8KB pages

Table 1. *Simulated Baseline Processor.*

the cores any given cycle. Crossbar link latency is assumed to be 3 cycles, and the data transfer time is 4 cycles.

Each bank of the L2 cache has a memory controller and an associated RDRAM channel. The memory bus is assumed to be clocked at 750MHz, with data being transferred on both edges of the clock for an effective frequency of 1.5GHz and an effective bandwidth of 3GB/s per bank (considering that each RDRAM memory channel supports 30 pins and 2 data bytes). Note that for any reasonable assumption about power and ground pins, the total number of pins that this memory organization would require would be well within the ITRS [1] limits for the cost/performance market. Memory latency is set to 150ns.

3.2 Die floorplan and area model

The baseline architecture and its floorplan is shown in Figure 1. We use CACTI to estimate the size and dimensions of the L2 cache. Each 512KB bank is $8.08mm^2$. CACTI gives the aspect ratio to be 2.73. So, each bank is $1.7mm \times 4.7mm$. Total L2 cache area is $64.64mm^2$. The area of the EV5-like core (excluding L1 caches) was calculated using similar assumptions and methodology as was used in [11], which also featured multiple Alphas cores on a die and technology scaling. Each core excluding caches is $2.12mm^2$. CACTI gives the area of of the L1 64KB, 2-way ICache to be $1.15mm^2$ and 64KB, 2-way DCache to be $2.59mm^2$. Hence, including the area occupied by private L1 caches, core area is $5.86mm^2$. If we assume an aspect-ratio of 1, it is $2.41mm \times 2.41mm$. The total area for the eight cores is $46.9mm^2$.

The crossbar area calculations involve measuring the area occupied by the interconnect wires. Each link from a core to a cache bank consists of roughly 300 lines. Of those, 256 lines correspond to a set of 128 unidirectional wires from the L2 to the cores and another 128-bit data bus from the cores to the L2 cache. We assume 20 lines correspond to the 20-bit unidirectional addressing signals while the rest correspond to control signals. Since each of the cores needs to be able to talk to each of the banks, there is a switched repeater

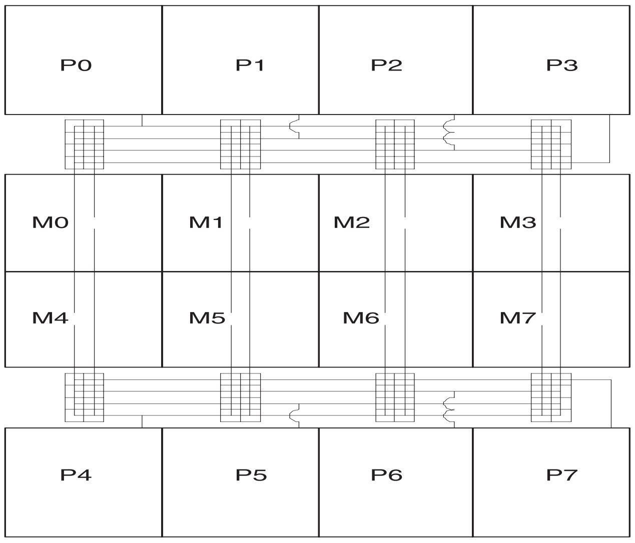


Figure 1. *Baseline die floorplan, with L2 cache banks in the middle of the cluster, and processor cores (including L1 caches) distributed around the outside.*

corresponding to each core-to-bank interface. Therefore, the number of horizontal tracks required per link would be approximately 300. The total number of input ports is equal to the number of cores. So, the number of horizontal tracks required will be $8 \times 300 = 2400$. This would determine the height of the crossbar. For the layout of the baseline processor (shown in Figure 1), the crossbar lies between the cores and the L2 banks. Also, there are two clusters of interconnects. The clusters are assumed to be connected by vertical wires in the crossbar routing channels and by vertical wires in upper metal layers that run over the top of the L2 cache banks (see Figure 1).

We assume that all the (horizontal) connecting lines are implemented in the M3/M5 layer. ITRS [1] and the "Future of Wires" paper by Horowitz, et al. [8] predict that wire pitch for a semi-global layer is 8-10λ. Assuming 10λ for $0.07micron$, the pitch is $350nm$. Then the width of the crossbar is $2400 \times 350nm = 0.84mm$. Hence, the area occupied by the crossbar for the baseline processor is $16.22mm^2$. This methodology of crossbar area estimation is similar to that used in [9].

Therefore, the total area of the processor is $127.76mm^2$ out of which $46.9mm^2$ is occupied by the cores, $16.22mm^2$ by the crossbar and $64.64mm^2$ by the L2 cache.

4 Conjoined-core Architecture

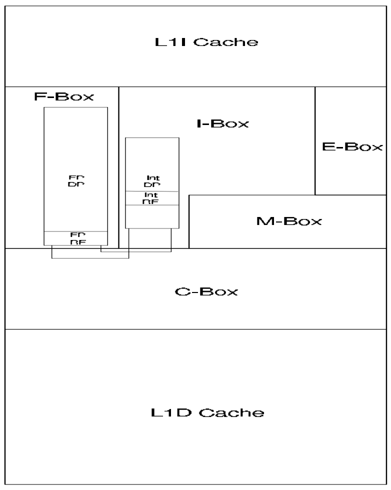
For the conjoined-core chip multiprocessor, we consider four optimizations – instruction cache sharing, data cache sharing, FPU sharing, and crossbar sharing. For each kind of sharing, two adjacent cores share the hardware structure. In this section, we investigate the mechanism for each kind of sharing and discuss the area benefits that they accrue. We talk about the performance impact of sharing in Section 6. The usage of the shared resource can be based on a policy decided either statically, such that it can be accessed only during fixed cycles by a certain core, or the accesses can be determined based on certain dynamic conditions visible to both cores (given adequate propagation time). The initial mechanisms discussed in this section all assume the simplest and most naive static scheduling, where one of the cores gets access to the shared resource during odd cycles while the other core gets access during even cycles. More intelligent sharing techniques/policies are discussed in Section 7. All of our sharing policies, however, maintain the assumption that communication distances between cores are too great to allow any kind of dynamic cycle-level arbitration for shared resources.

Note that in modern high-performance pipelines (beginning with the DEC Alpha 21064), variable operation latency past the issue point as a result of FIFO-type structures is not possible. This is because in modern pipelines, each pipestage is only a small number of FO4 delays, and global communication plus control logic overhead for implementing stalling on a cycle-by-cycle basis would drastically increase the cycle time. Such stalling is required by variable delays because instructions must be issued assuming results of previous operations are available when expected. Instead any delay (such as that required by a DCache miss instead of an expected hit) results in a flush and replay of the missing reference plus the following instructions. Although flush and replay overhead is acceptable for rare long latency events such as cache misses, it is unacceptable for routine operation of the pipeline. By assuming very simple fixed scheduling in the baseline sharing case we guarantee that the later pipe stages do not need to be stalled, and the cycle time of the pipeline is not adversely affected. Later we examine more complex techniques for scheduling sharing that remain compatible with high-speed pipeline design.

Due to wiring overheads, it only makes sense to share relatively large structures that already have routing overhead. FPUs, crossbars, and caches all have this property. In contrast, ALUs in a datapath normally fit under the datapath operand and result busses. Thus, placing something small like an individual ALU remotely would actually result in a very significant increase in bus wiring and chip area instead of a savings, as well as increased latency and power dissipation.

4.1 ICache sharing

We implement ICache sharing between two cores by providing a shared fetch path from the ICache to both the pipelines. Figure 2 shows a floorplan of two adjacent cores sharing a 64KB, 2-way associative ICache. Because the layout of memories is a function of the number of rows and columns, we have increased the number of columns but re-



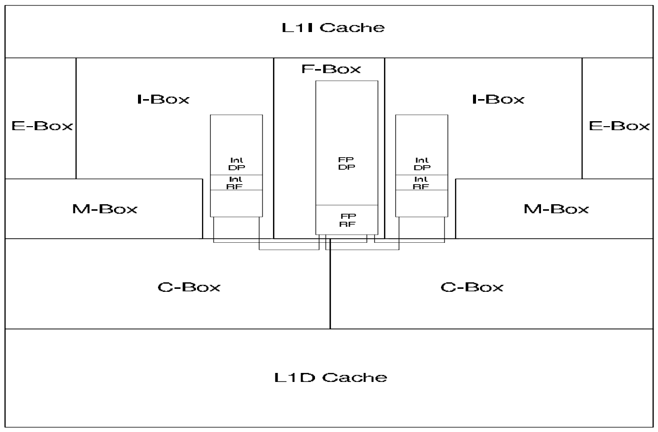


Figure 2. *(a)Floorplan of the original core (b)Layout of a conjoined-core pair, both showing FPU routing. Routing and register files are schematic and not drawn to scale.*

duced the number of rows in the shared memory. This gives a wider aspect ratio that can span two cores.

As mentioned, the ICache is time-shared every other cycle. We investigate two ICache fetch widths. In the *double fetch width* case, the fetch width is changed to 8 instructions every other cycle (compared to 4 instructions every cycle in the unshared case). The time-averaged effective fetch bandwidth (ignoring branch effects) remains unchanged in this case. In the *original structure fetch width* case, we leave the fetch width to be the same. In this case the effective per-core fetch bandwidth is halved. Finally, we also investigate a banked architecture, where cores can fetch 4 instructions every cycle, but only if their desired bank is allocated to them that cycle.

In the *double fetch width* case, sharing results in a wider instruction fetch path, wider multiplexors and extra instruction buffers before decode for the instruction front end. We have modeled this area increase and we also assume that sharing increases the access latency by 1 cycle. The double fetch width solution would also result in higher power consumption per fetch. Furthermore, since longer fetch blocks are more likely to include taken branches out of the block, the fetch efficiency is somewhat reduced. We also evaluate two cases corresponding to a shared instruction cache with an unchanged fetch width – one with the access time extended by a cycle and another when it remains unchanged.

Based on modeling with CACTI, in the baseline case each ICache takes up $1.15mm^2$. In the double fetch width case, the ICache has double the bandwidth (BITOUT=256), and requires $1.16mm^2$. However, instead of 8 ICaches on the die, there are just four of them. This results in a core-area savings of 9.8%. In the normal fetch width case (BITOUT=128), sharing results in core-area savings of 9.9%.

4.2 DCache sharing

Even though the DCaches occupy a significant area, DCache sharing is not an obvious candidate for sharing because of its relatively high utilization. In our DCache sharing experiments, two adjacent cores share a 64KB, 2-way set-associative L1 DCache. Each core can issue memory instructions only every other cycle.

Sharing entails lengthened wires that increase access latency slightly. This latency may or may not be able to be hidden in the pipeline. Thus, we evaluate two cases – one where the access time is lengthened by one cycle and another where the access time remains unchanged.

Based on modeling with CACTI, each dual-ported DCache takes up $2.59mm^2$ in the baseline processor. In the shared case, it takes up the area of just one cache for every two cores, but with some additional wiring. This results in core-area savings of 22.09%.

4.3 Crossbar sharing

As shown in Section 3, the crossbar occupies a significant fraction (13%) of the die area. The configuration and complexity of the crossbar is strongly tied to the number of cores, and hence we also study how crossbar sharing can be used to free up die area. This study also forms a subset of the larger study that is required for area-efficient interconnect designs for chip multiprocessors.

Crossbar sharing involves two adjacent cores sharing an input port to the L2 cache's crossbar interconnect. This halves the number of rows (or columns) in the crossbar matrix resulting in linear area savings. Crossbar sharing entails that only one of the two conjoined cores can issue a request to a particular L2 cache bank in a given cycle. Again, we assume a baseline implementation where one of the conjoined cores can issue requests to a bank every odd cycle, while the other conjoined core can issue requests only on even cycles. There would also be some overhead in routing signal and data to the shared input port. Hence, we assume the point-to-point communication latency will be lengthened by one cycle for the conjoined core case. Figure 10 shows conjoined core pairs sharing input ports to the crossbar.

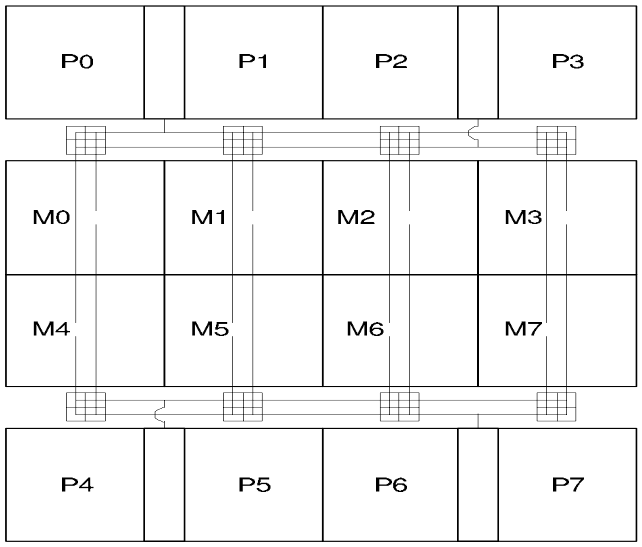


Figure 3. *Die floorplan with crossbar sharing.*

Crossbar sharing results in halving the area occupied by the interconnect and results in 6.43% die area savings. This is equivalent to 1.38 times the size of a single core.

Note that this is not the only way to reduce the area occupied by the crossbar interconnect. One can alternatively halve the number of wires for a given point-to-point link to (approximately) halve the area occupied by that link. This would, though, double the transfer latency for each connection. In section 6, we compare both these approaches and show that this performs worse than our port-sharing solution.

Finally, if the DCache and ICache are already shared between two cores, sharing the crossbar port between the same two cores is very straightforward since the cores have already been joined together before reaching the crossbar.

4.4 FPU sharing

Processor floorplans often have the FPU on one side and the integer datapath on the other side. So, FPU sharing can be enabled by simply mirroring adjacent processors without significant disruption to the floorplan. Wires connecting the FPU to the left core and the right core can be interdigitated, so no additional horizontal wiring tracks are required (see Figure 2). This also does not significantly increase the length of wires in comparison the the non-conjoined case.

In our baseline FPU sharing model, each conjoined core can issue floating-point instructions to the fully-pipelined floating-point sub-units only every other cycle. Based on our design experience, we believe that there would be no operation latency increase when sharing pipelined FPU sub-units between the cores. This is because for arithmetic operations the FP registers remain local to the FPU. For transfers and load/store operations, the routing distances from the integer datapath and caches to the FPU remain largely unchanged (see Figure 2). For the non-pipelined sub-units (e.g., divides and square root) we assume alternating three cycle scheduling windows for each core. If a non-pipelined unit is available at the start of its three-cycle window, the core may start using it, and has the remainder of the scheduling window to communicate this to the other core. Thus, when the non-pipelined units are idle, each core can only start a non-pipelined operation once every six cycles. However, since operations have a known long latency, there is no additional scheduling overhead needed at the end of non-pipelined operations. Thus, when a non-pipelined unit is in use, another core waiting for it can begin using the non-pipelined unit on the first cycle it becomes available.

The FPU area for EV5 is derived from published die photos, scaling the numbers to 0.07 micron technology and then subtracting the area occupied by the FP register file. The EV5 FPU takes up $1.05mm^2$ including the FP register file. We estimate the area taken up by a 5ERP, 4EWP, 32-entry FP register file using *register-bit equivalents* (rbe). The total area of the FPU (excluding the register file) is $0.72mm^2$. Sharing results in halving the number of units and results in area savings of 6.1%.

We also consider a case where each core has its own copy of the divide sub-unit, while the other FPU sub-units are shared. We estimated the area of the divide sub-unit to be $0.0524mm^2$. Total area savings in that case is 5.7%.

4.5 Summary of sharing

To sum up, ICache sharing results in core-area savings of 9.9%, DCache sharing results in core-area savings of 22%, FPU sharing saves 6.1% of the core-area, and sharing the input ports to the crossbar can result in a savings of 1.4 cores. Statically deciding to let each conjoined core access a shared hardware structure only every other cycle provides an upper-bound on the possible degradation. As our results in Section 6 indicate, even these conservative assumptions lead to relatively small performance degradation and hence reinforce the argument for conjoined-core chip multiprocessing.

5 Experimental Methodology

Benchmarks are simulated using SMTSIM, an execution-driven simulator that simulates an out-of-order, simultaneous multithreading processor [14]. SMTSIM executes unmodified, statically linked Alpha binaries. The simulator was modified to simulate the various chip multiprocessor (conjoined as well as conventional) architectures.

Several of our evaluations are done for various numbers of threads ranging from one through a maximum number of available processor contexts. Each result corresponds to one of three sets of eight benchmarks, where each data point is the average of several permutations of those benchmarks.

Program	Description	FF Dist (in millions)
bzip2	Compression	5200
crafty	Game Playing:Chess	100
eon	Computer Visualization	1900
gzip	Compression	400
mcf	Combinatorial Optimization	3170
perl	PERL Programming Language	200
twolf	Place and Route Simulator	3200
vpr	FPGA Circuit Placement and Routing	7200
applu	Parabolic/Elliptic Partial Diff. Eqn.	1900
apsi	Meteorology:Pollutant Distribution	4700
art	Image Recognition/Neural Networks	6800
equake	Seismic Wave Propagation Simulation	19500
facerec	Image Processing: Face Recognition	13700
fma3d	Finite-element Crash Simulation	29900
mesa	3-D Graphics Library	9000
wupwise	Physics/Quantum Chromodynamics	58500

Table 2. *Benchmarks simulated.*

Table 2 shows the subset of the SPEC CPU2000 benchmark suite that was used. The benchmarks are chosen such that out of the 8 CINT2000 benchmarks, half of them (*vpr,crafty,eon,twolf*) have a dataset of less than 100MB while the remaining half have datasets bigger than 100MB. Similarly, for CFP2000 benchmarks, half of them (*wupwise,applu,apsi,fma3d*) have datasets bigger than 100MB while the remaining half have datasets of less than 100MB. We also perform all our evaluations for mixed workloads which are generated using 4 integer benchmarks (*bzip2,mcf,crafty,eon*) and 4 floating-point benchmarks (*wupwise, applu, art, mesa*). Again, the subsetting was done based on application datasets.

All the data points are generated by evaluating 8 workloads for each case and then averaging the results. A workload consisting of n threads is constructed by selecting the benchmarks using a sliding window (with wraparound) of size n and then shifting the window right by one. Since there are 8 distinct benchmarks, the window selects eight distinct workloads (except for cases when the window-size is a multiple of 8, in those cases all the selected workloads have identical composition). All of these workloads are run, ensuring that each benchmark is equally represented at every data point. This methodology for workload construction is similar to that used in [13].

We also perform evaluations using the parallel benchmark *water* from the SPLASH benchmark suite and use the STREAM benchmark for crossbar evaluations. We change the problem size of STREAM to 16384 elements. At this size, when running eight copies of STREAM, the working set fits into the L2-cache and hence it acts as a worst-case test of L1-L2 bandwidth (and hence crossbar interconnect). We also removed the timing statistics collection routines.

The Simpoint tool [12] was used to find good representative fast-forward distances for each SPEC benchmark. Early simpoints are used. Table 2 also shows the distance to which each benchmark was fast-forwarded before beginning simulation. For *water*, fast-forwarding is done just enough so that the parallel threads get forked. We do not fast forward for STREAM.

All simulations involving n threads are preceded by a warmup of $10 \times n$ million cycles. Simulation length was 800 million cycles. All the SPEC benchmarks are simulated using *ref* inputs. All the performance results are in terms of throughput. We also performed all our evaluations using the weighted speedup metric [13] and observed no significant difference in our analyses or conclusions.

6 Simple Sharing

This section examines the performance impact of conjoining cores assuming simple time-slicing of the shared resources on alternate cycles. More intelligent sharing techniques are discussed in the next section.

In this section, we show results for various threading levels. We schedule the workloads statically and randomly such that two threads are run together on a conjoined-core pair only if one of them cannot be placed elsewhere. Hence, for the given architecture, for 1 to 4 threads, there is no other thread that is competing for the shared resource. If we have 5 runnable threads, one of the threads needs to be put on a conjoined-core pair that is already running a thread. And so on. However, even if there is no other thread running on the other core belonging to a conjoined-core pair, we still assume, in this section, that accesses can be made to the shared resource by a core only every other cycle.

6.1 Sharing the ICache

Results are shown as performance degradation relative to the the baseline conventional CMP architecture. Performance degradation experienced with ICache sharing comes from three sources: increased access latency, reduced effective fetch bandwidth, and inter-thread conflicts. Effective fetch bandwidth can be reduced even if the fetch width is doubled because of the decreased likelihood of filling an eight-wide fetch with useful instructions, relative to a four-wide fetch.

Figure 4 shows the performance impact of ICache sharing for varied threading levels for SPEC-based workloads. The results are shown for a fetch width of 8 instructions and assuming that there is an extra cycle latency for ICache access due to sharing. We assume the extra cycle is required since in the worst case the round-trip distance to read an ICache bit has gone up by two times the original core width due to sharing. We observe a performance degradation of 5% for integer workloads, 1.2% for FP workloads and 2.2% for mixed workloads. The performance degradation does not change significantly when the number of threads is increased from 1 to 8.

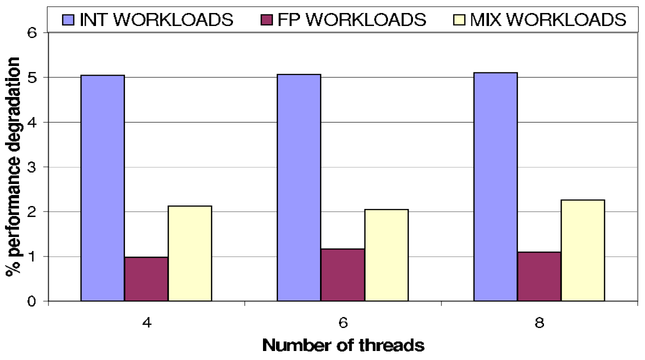


Figure 4. *Impact of ICache sharing for various threading levels.*

This indicates that inter-thread conflicts are not a problem for this workload and these caches. The SPEC benchmarks are known to have relatively small instruction working sets.

To identify the main cause for performance degradation on ICache sharing, we also show results assuming that there is no extra cycle increase in the latency. Figure 5 shows the 8-thread results for both integer and floating-point workloads. Performance degradation becomes less than 0.25%. Two conclusions can be drawn from this. First, the extra latency is the main reason for degradation on ICache sharing (note that the latency does not introduce a bubble in the pipeline – the performance degradation comes from the increased branch mispredict penalty due to the pipeline being extended by a cycle). The integer benchmarks are most affected by the extra cycle latency, being more sensitive to the branch mispredict penalty.

Increasing fetch width to 8 instructions ensures that the potential fetch bandwidth remains the same for the sharing case as the baseline case, but it increases the size of the ICache (relative to a single ICache in the base case) and results in increased power consumption. This is because doubling the output width doubles both the number of sense amps and the data output lines being driven, and these structures account for much of the power in the original cache. Thus, we also investigate the case where fetch width is kept the same. Hence, only up to 4 instructions can be fetched every other cycle (effectively halving the per-core fetch bandwidth). Figure 5 shows the results for 8-thread workloads. As can be seen, degradation jumps up to 16% for integer workloads and 10.2% for floating-point workloads. This is because at effective fetch bandwidth of 2 instructions every cycle (per core), the execution starts becoming fetch limited.

We also investigate the impact of partitioning the ICache vertically into two equal sized banks. A core can alternate accesses between the two banks. It can fetch 4 instructions every cycle but only if their desired bank is available. A core has access to bank 0 one cycle, bank 1 the next, etc., with the other core having the opposite allocation. This allows both threads to access the cache in some cycles. It is also possi-

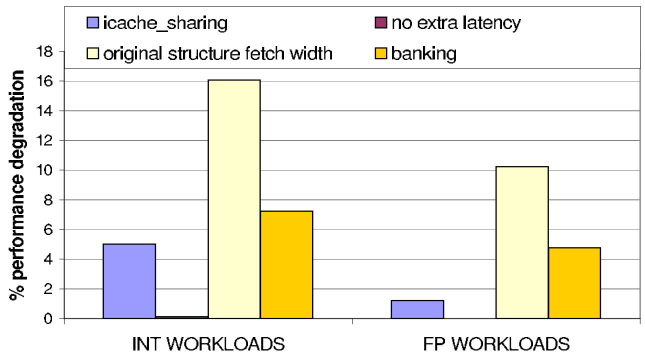


Figure 5. *ICache sharing when no extra latency overhead is assumed, cache structure bandwidth is not doubled, and cache is doubly banked.*

ble for both threads to be blocked in some cycles. However, bandwidth is guaranteed to exceed the previous case (ignoring cache miss effects) of one 4-instruction fetch every other cycle, because every cycle that both threads fail to get access will be immediately followed by a cycle in which they both can access the cache.

Figure 5 shows the results. Degradation goes down by 55% for integer workloads and 53% for FP workloads due to overall improvement in fetch bandwidth.

6.2 DCache sharing

Similar to the ICache, performance degradation due to DCache sharing comes from: increased access latency, reduced cache bandwidth, and inter-thread conflicts. Unlike the ICache, the DCache latency has a direct effect on performance, as the latency of the load is effectively increased if it cannot issue on the first cycle it is ready.

Figure 6 shows the impact on performance due to DCache sharing for SPEC workloads. The results are shown for various threading levels. We observe a performance degradation of 4-10% for integer workloads, 1-9% for floating point workloads and 2-13% for mixed workloads. Degradation is higher for integer workloads than floating point workloads for small numbers of threads. This is because the typically higher ILP of the FP workloads allows them to hide a small increase in latency more effectively. Also, inter-thread conflicts are higher, resulting in increased performance degradation for higher numbers of threads.

We also studied the case where the shared DCache has the same access latency as the unshared DCache. Figure 7 shows the results for the 8-thread case. Degradation lessens for both integer workloads as well as floating-point workloads, but less so in the case of FP workloads as conflict misses and cache bandwidth pressure remain.

6.3 FPU sharing

FPUs may be the most obvious candidates for sharing. For SPEC CINT2000 benchmarks only 0.1% of instructions are floating point while even for CFP2000 benchmarks, only 32.3% of instructions are floating-point instructions [2]. Also, FPU bandwidth is a performance bottleneck only for specialized applications.

We evaluated FPU sharing for integer workloads, FP workloads, and mixed workloads, but only present the FP and mixed results (Figure 8) here. The degradation is less than 0.5% for all levels of threading, even in these cases.

One reason for these results is that the competition for the non-pipelined units (divide and square root) is negligible in the SPEC benchmarks. To illustrate code where non-pipelined units are more heavily used, Figure 9 shows the

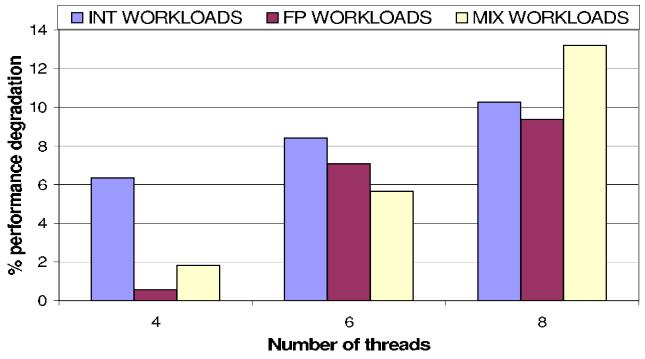


Figure 6. *Impact of Dcache sharing for various threading levels.*

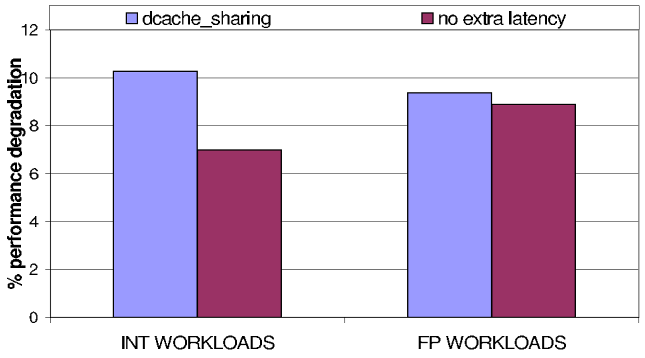


Figure 7. *DCache sharing when no extra latency overhead is assumed.*

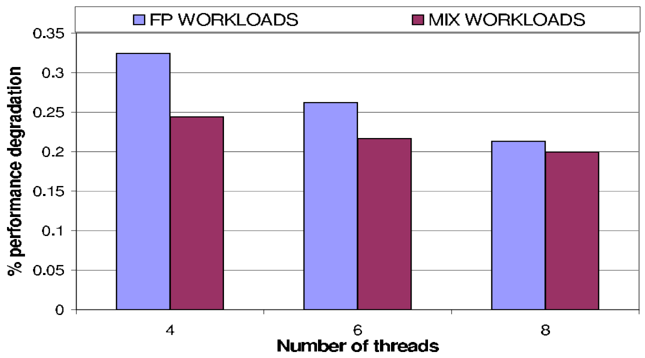


Figure 8. *Impact of FPU sharing for various threading levels.*

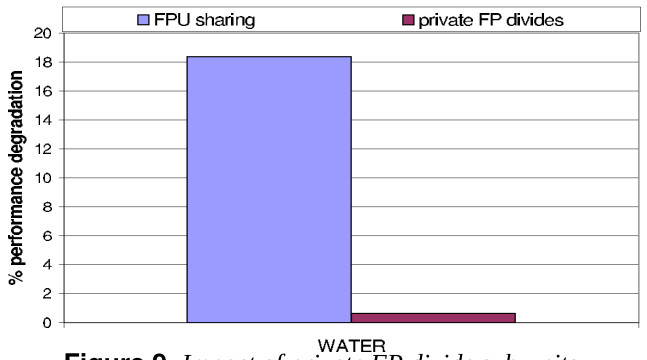


Figure 9. *Impact of private FP divide sub-units.*

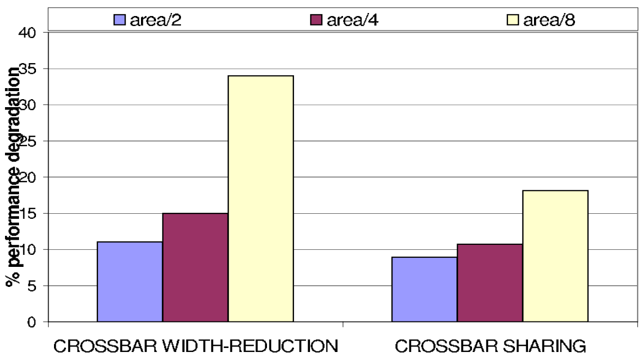


Figure 10. *Reducing crossbar area through width reduction and port sharing.*

performance of *water* (which has a non-trivial number of divides) running eight threads. It shows performance with a shared FP divide unit vs. unshared FP divide units. In this case, unless each core has its own copy of the FP divide unit, performance degradation can be significant.

6.4 Crossbar sharing

We implement the L1-L2 interconnect as a blocking fully-connected matrix crossbar, based on the initial Piranha design. As the volume of traffic between L1 and L2 increases, the utilization of the crossbar goes up. Since there is a single path from a core to a bank, high utilization can result in contention and queueing delays.

As discussed in section 4, the area of the crossbar can be reduced by decreasing the width of the crossbar links or by sharing the ports of the crossbar, thereby reducing the number of links. We examine both techniques. Crossbar sharing involves the conjoined cores sharing an input port of the crossbar. Figure 10 shows the results for eight copies of the STREAM benchmark. It must be noted that this is a component benchmark we have tuned for worst-case utilization of the crossbar. The results are shown in terms of performance degradation caused for achieving certain area savings. For example, for achieving crossbar area savings of 75% (*area/4*), we assume that the latency of every crossbar link has been doubled for the *crossbar sharing* case while the latency has been quadrupled for the *crossbar width reduction* case.

We observe that crossbar sharing outperforms crossbar width reduction in all cases. Even though sharing results in increased contention at the input ports, it is the latency of the links that is primarily responsible for queuing of requests and hence overall performance degradation.

We also conducted crossbar exploration experiments using SPEC benchmarks. However, most of the benchmarks do not exercise L1-L2 bandwidth much, resulting in relatively low crossbar utilization rates. The performance degradation in the worst case was less than 5% for an area reduction factor of 2.

6.5 Simple sharing summary

Note that for all the results in this section, we assume that the shared resource is accessible only every other cycle even if the other core on a conjoined-core pair is idle. This was done to expose the factors contributing to overall performance degradation. However, in a realistic case, if there is no program running on the other core, the shared resources can be made fully accessible to the core running the program and hence there would be no (or much smaller) degradation. Thus, for the above sharing cases, the degradation values for threading levels of four are overstated. In fact, the performance degradation due to conjoining will be minimal for light as well as medium loads.

This section indicates that, even in the absence of sophisticated sharing techniques, conjoined-core multiprocessing is a reasonable approach. Optimizations in the next section make it even more attractive. It might be argued that this is simply evidence of the over-provisioning of our baseline design. There are two reasons why that is the wrong conclusion. First, our baseline is based on real designs, and is not at all aggressive compared to modern processor architectures. Second, real processors are over-provisioned – to some extent that is the point of this study. Designers provision the CPU for the few important applications that really stress a particular resource. What this research shows is that we can maintain that same level of provisioning for any single thread, without multiplying the cost of that provisioning by the number of cores.

7 Intelligent Sharing of Resources

The previous section assumed a very basic sharing policy and hence gave an upper bound on the degradation for each kind of sharing. In this section, we discuss more advanced techniques for minimizing performance degradation.

7.1 ICache sharing

In this section, we will focus on that configuration that minimized area, but maximized slowdown — the four-wide

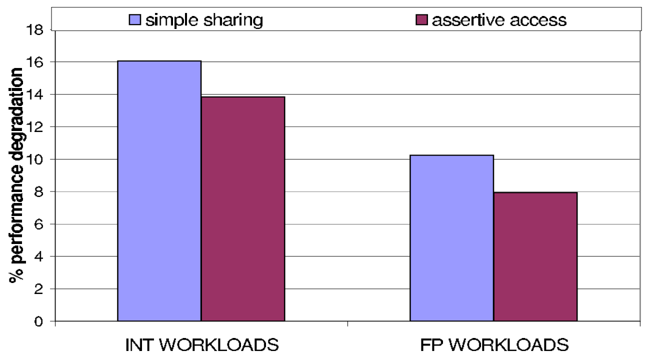


Figure 11. *ICache assertive access results when the original structure bandwidth is not doubled.*

fetch shared ICache, assuming an extra cycle of latency. In that case, both access latency and fetch bandwidth contribute to the overall degradation. We propose two techniques for minimizing degradation in that case. Most of these results would also apply to the other configurations of shared ICache, taking them even closer to zero degradation.

7.1.1 Assertive ICache Access

Section 6 discussed sharing such that the shared resource gets accessed evenly irrespective of the access needs of the individual cores. Instead, the control of a shared resource can be decided assertively based on the resource needs.

We explore *assertive ICache access* where, whenever there is an L1 miss, the other core can take control of the cache after miss detection. We assume that a miss can be detected and communicated to the other core in 3 cycles. The control would start getting shared again when the data returns. This does not incur any additional latency since the arrival cycle of the data is known well in advance of its return.

Figure 11 shows the results for *assertive icache access*. Like all graphs in this section, we show results for eight threads, where contention is highest. We observe a 13.7% improvement in the degradation of integer workloads and an improvement of 22.5% for floating point workloads. Performance improvement is because of improved effective fetch bandwidth. These results are for eight threads, so there is no contribution from threads that are not sharing an ICache. A minor tweak to *assertive access* (for ICache as well as DCache and FPU) can ensure that the shared resource becomes a private resource when the other core of the conjoined pair is idle.

7.1.2 Fetch combining

Most parallel code is composed of multiple threads, each executing code from the same or similar regions of the shared executable (possibly synchronizing occasionally to ensure they stay in the same region). Hence, it is not uncommon for two or more threads to be fetching from the same address in a particular cycle.

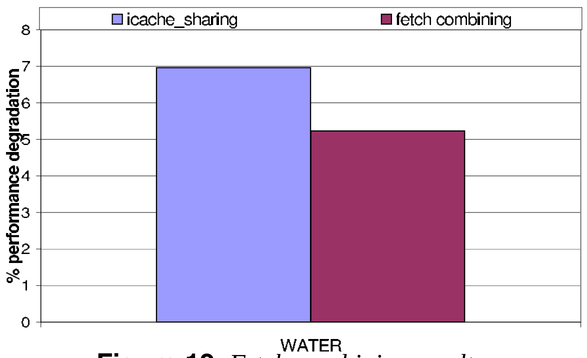


Figure 12. *Fetch-combining results.*

In a conjoined-core architecture with shared ICache, this property can be exploited for improving overall fetch bandwidth. We propose *fetch combining* – when two threads running on the same conjoined-core pair have the same nextPC cache index, then they both can return data from the cache that cycle. The overhead for fetch combining is minimal, under the following assumptions. We assume that the fetch units are designed so that, in the absence of sharing (no thread assigned to the alternate core), one core can fetch every cycle. Thus, each core has the ability to generate a nextPC cache index every cycle, and to consume a fetch line every cycle. In sharing mode, however, only one request is filled. Thus, in sharing mode with fetch combining, both cores can present a PC to the ICache, but only the PC associated with the core with access rights that cycle is serviced. However, if the presented PCs are identical, the alternate core also reads the data presented on the (already shared) output port and bus. This is simplified if there is some decoupling of the branch predictor from the fetch unit. If a queue of nextPC cache indices is buffered close to the ICache, we can continue to present new PCs to the cache every cycle, even if it takes more than a cycle for the result of the PC comparison to get back to the core.

We find that the frequency of coincident indices is quite high – this is because once the addresses match, they tend to stay synched up until the control flow diverges.

Figure 12 shows the performance of fetch combining for *water* running eight threads. We observed a 25% reduction of performance degradation. Note that fetch combining is appropriate for other multithreading schemes like SMT, etc.

7.2 DCache sharing

Performance loss due to DCache sharing is due to three factors – inter-thread conflict misses, reduced bandwidth and increased latency (if applicable). We propose two techniques for minimizing degradation due to DCache sharing.

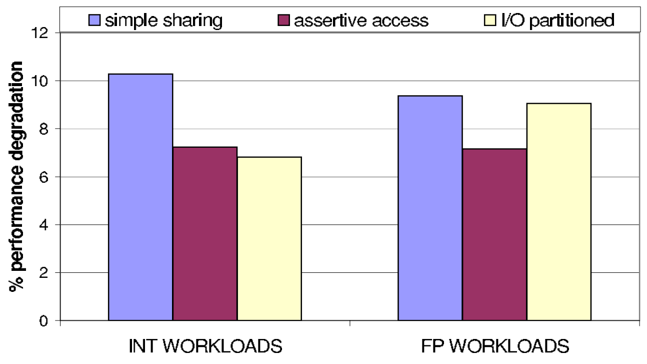


Figure 13. *Effect of assertive access and static assignment.*

7.2.1 Assertive DCache Access

Assertive access can also be used for the shared DCaches. Whenever there is an L1 miss on some data requested by a core, if the load is determined to be on the right path, the core relinquishes control over the shared DCache. There may be some delay between detection of L1 miss and the determination that the load is on the right path. Once the core relinquishes control, the other core takes over full control and can then access the DCache whenever it wants. The timings are the same as with the ICache assertive access. This policy is still somewhat naive, assuming that the processor will stall for this load (recall, these are in-order cores) before another load is ready to issue – more sophisticated policies are possible, and are the subject of further investigation.

Figure 13 shows the results. Assertive access leads to 29.6% improvements in the degradation for integer workloads and 23.7% improvements for floating point workloads. Improvements are due to improved data bandwidth.

7.2.2 I/O partitioning

The Dcache interface consists of two R/W ports. In the basic DCache sharing case, the DCache (and hence both the ports) can be accessed only every other cycle. Instead, one port can be statically assigned to each of the cores and that will make the DCache accessible every cycle.

Figure 13 shows the results comparing the baseline sharing policy against static port-to-core assignment. We observed a 33.6% reduction in degradation for integer workloads while the difference for FP workloads was only 3%. This outperforms the cycle-slicing mechanism, particularly for integer benchmarks, for the following reason: when load port utilization is not high, the likelihood (with port partitioning) of a port being available when a load becomes ready is high. However, with cycle-by-cycle slicing, the likelihood of a port being available that cycle is only 50%.

7.3 Symbiotic assignment of threads

Previous techniques involved either using additional hardware for minimizing performance impact or scheduling accesses to the shared resources intelligently. Alternatively, high-level scheduling of applications can be done such that "friendly" threads run on conjoined cores. Symbiotic scheduling has been shown previously to result in significant benefits on an SMT architecture [13] and involves co-scheduling threads to minimize competition for shared resources.

Units Shared	Perf. Degradation		Core Area
	Int Aps	FP Aps	Savings
Crossbar+FPU	0.97%	1.2%	23.1%
Crossbar+FPU+ICache	4.7%	3.9%	33.0%
Crossbar+FPU+DCache	6.1%	6.8%	45.2%
ICache+DCache	11.4%	7.6%	32.0%
Crossbar+FPU+ICache+DCache	11.9%	8.5%	55.1%

Table 3. *Results with multiple sharings.*

Since conflict misses are a significant source of performance degradation for DCache sharing, we evaluated the impact of scheduling applications intelligently on the cores instead of random mapping. Intelligent mapping involved putting programs on a conjoined-core pair that would not cause as many conflict misses and hence lessen the degradation. For symbiotic scheduling with 8 threads, we found the degradation decreased by 20% for integer workloads and 25.5% for FP workloads.

8 A Unified Conjoined Core Architecture

We have studied various combinations of FPU, crossbar, ICache, and DCache sharing. We assumed a shared doubly-banked ICache with a fetch-width of 16 bytes (similar to that used in Section 6.1), I/O partitioned shared DCache (similar to that used in Section 7.2.2), a fully-shared FPU and a shared crossbar input port for every conjoined-core pair. Sharing ICache, DCache, as well as the crossbar is assumed to have one cycle extra overhead. We assume that each shared structure can be *assertively accessed*. Assertive access for the I/O partitioned dual-ported DCache involves accessing the other port (the one not assigned to the core) assertively. Figure 3 shows the resulting area savings and performance for various sharing combinations. We map the applications to the cores such that "friendly" threads run on the conjoined cores where possible. All performance numbers are for the worst case when all cores are busy with threads.

The combination with all four types of sharing results in 38.1% core-area savings (excluding crossbar savings). In absolute terms, this is equivalent to the area occupied by 3.76 cores. *If crossbar savings are included, then the total area saved is equivalent to 5.14 times the area of a core.* We observed a 11.9% degradation for integer workloads and 8.5% degradation for floating-point workloads. Note that the degradation is significantly less than the sum of the individual degradation values that we observed for each kind of sharing. This is because a stall due to one bottleneck often either tolerates or obviates a stall due to some other bottleneck.

Another attractive configuration utilizes only FPU and crossbar sharing. This configuration provides a 23.1% reduction in core area while degrading performance by around 1% in the worst case with all cores busy, and provides the highest marginal utility for sharing. This configuration also has the advantage of being simpler to implement than configurations that share caches.

The results in Table 3 show that conjoined-core architectures can give superior computational efficiency over conventional non-conjoined cores. The area savings they give can be used to provide either reduced die area and hence increased yield, or can be leveraged to provide a significant increase in performance by implementing more cores in the same area.

Finally, besides providing an area efficiency advantage, conjoining can also result in more power-efficient computation. Since memory cells can be engineered to have low leakage, leakage power is primarily a function of the amount of high-performance logic. Thus by sharing FPUs and/or the peripheral logic of caches, the number of logic circuits and hence the leakage power of a multiprocessor can be significantly reduced. Moreover, dynamic power per instruction can also be reduced since the crossbar interconnect lengths that computation must traverse can be reduced by reducing the area of the cores.

9 Conclusions

This paper examines conjoined core multiprocessing, selectively targeting opportunities to share resources on an otherwise statically partitioned chip multiprocessor. In particular, we seek to achieve area savings, dynamic power reduction, and leakage reduction by sharing resources that have sufficient bandwidth and/or capacity to service multiple cores. We add the additional constraint that the sharing is topologically feasible with minimal impact to a conventional core layout.

This paper examines sharing of the floating point units, the crossbar network ports, and the first-level ICache and DCache. We show that, given a set of novel optimizations that reduce the negative impacts of this sharing, we can reduce area requirements by more than 50%, while achieving performance within 9-12% of conventional cores without conjoining. Alternatively, by only sharing floating point units and crossbar ports, core area can be reduced by more than 23% while achieving performance within 2% of conventional cores without conjoining.

These gains are a combination of the inherent advantage of sharing resources provisioned for worst-case utilization, and the application of new sharing policies that allow high bandwidth access to these resources without additional complexity.

Acknowledgments

The authors would like to thank Partha Ranganathan for his contributions to the paper, as well as the anonymous referees. The research was funded in part by NSF grant CCR-0311683 and a grant from the Semiconductor Research Corporation.

References

[1] International Technology Roadmap for Semiconductors 2003, http://public.itrs.net.

[2] Measuring processor performance with SPEC2000- a white paper, Intel Corporation. 2002.

[3] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[4] J. Burns and J.-L. Gaudiot. Area and system clock effects on smt/cmp processors. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, page 211. IEEE Computer Society, 2001.

[5] J. Collins and D. Tullsen. Clustered multithreaded architectures – pursuing both IPC and cycle time. In *Proceedings of IPDPS*, Apr. 2004.

[6] R. Dolbeau and A. Seznec. CASH: Revisiting hardware sharing in single-chip parallel processor. IRISA Report 1491, Nov. 2002.

[7] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. In *Computer*, volume 30, pages 79–85, 1997.

[8] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[9] F. Karim, A. Nguyen, S. Dey, and R. Rao. On-chip communication architecture for oc-768 network processors. In *Proceedings of the 2001 Design and Automation Conference*, 2001.

[10] V. Krishnan and J. Torrellas. A clustered approach to multithreaded processors. In *Proceedings of the International Parallel Processing Symposium*, pages 627–634, Mar. 1998.

[11] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *International Symposium on Microarchitecture*, Dec. 2003.

[12] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. In *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, Dec. 2003.

[13] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[14] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[15] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.