Partitioning Multi-Threaded Processors with a Large Number of Threads [*]

Ali El-Moursy[⋆], Rajeev Garg[⋆], David H. Albonesi[†] and Sandhya Dwarkadas[⋆]

[⋆]Departments of Electrical and Computer Engineering and of Computer Science, University of Rochester
[†]Computer Systems Laboratory, Cornell University
elmours@ece.rochester.edu, {garg,sandhya}@cs.rochester.edu, albonesi@csl.cornell.edu

Abstract

Today's general-purpose processors are increasingly using multithreading in order to better leverage the additional on-chip real estate available with each technology generation. Simultaneous Multi-Threading (SMT) was originally proposed as a large dynamic superscalar processor with monolithic hardware structures shared among all threads. Intel's Hyper-Threaded Pentium 4 processor partitions the queue structures among two threads, demonstrating more balanced performance by reducing the hoarding of structures by a single thread. IBM's Power5 processor is a 2-way Chip Multiprocessor (CMP) of SMT processors, each supporting 2 threads, which significantly reduces design complexity and can improve power efficiency.

This paper examines processor partitioning options for larger numbers of threads on a chip. While growing transistor budgets permit four and eight-thread processors to be designed, design complexity, power dissipation, and wire scaling limitations create significant barriers to their actual realization. We explore the design choices of sharing, or of partitioning and distributing, the front end (instruction cache, instruction fetch, and dispatch), the execution units and associated state, as well as the L1 Dcache banks, in a Clustered Multi-Threaded (CMT) processor. We show that the best performance is obtained by restricting the sharing of the L1 Dcache banks and the execution engines among threads. On the other hand, significant sharing of the front-end resources is the best approach.

When compared against large monolithic SMT processors, a CMT processor provides very competitive IPC performance on average, 90-96% of that of partitioned SMT while being more scalable and much more power efficient. In a CMP organization, the gap between SMT and CMT processors shrinks further, making a CMP of CMT processors a highly viable alternative for the future.

1 Introduction

In recent years, the exploitation of higher levels of instruction-level parallelism (ILP) by microarchitects has been replaced with the pursuit of both ILP and thread-level parallelism (TLP). Simultaneous Multi-Threaded (SMT) processors are designed to exploit both ILP and TLP within a dynamic superscalar processor core. As originally proposed by Tullsen [31], the hardware resources of an SMT processor are shared among all threads, permitting significantly greater resource utilization and higher instruction throughput.

However, many commercial multi-threaded processor implementations, such as the IBM Power5 [23] and Intel Pentium 4 [12], are not implemented as a single monolithic SMT core. Rather, hardware resources are *partitioned* among the threads. The Power5 uses a Chip Multiprocessor (CMP) of SMT processors, each of which is limited to two active threads. With this approach, a single, modest SMT core is designed and verified and then duplicated, which greatly reduces design complexity. Additional benefits may be a higher clock rate and lower power dissipation as the hardware structures and internal buses of each core are greatly simplified compared to that in a monolithic four-thread core. Intel's Hyper-Threaded Pentium 4 processor shares all execution units among all threads, but partitions some queue resources between the two active threads. This has been shown to reduce the hoarding of structures by a single thread [25], thereby providing more balanced cycle-level performance.

These initial implementations have been limited to two threads per processor core. As higher levels of chip integration permit more threads to be supported on a single die, the way in which hardware resources are partitioned among the threads needs to be re-examined. The CMP of SMT processors (CMP+SMT) approach has the disadvantage of statically partitioning resources among subsets of threads. The use of a larger number of cores in a CMP resulting in more simultaneous thread execution capability increases the likelihood that some threads are under-utilizing their resources while other threads could use more. This

[*]This work was supported in part by NSF grants CCR-0219848, ECS-0225413, CNS-0411127, INT-0117667, and EIA-0080124; by DARPA/IPTO under AFRL contract F29601-00-K-0182; by two IBM Faculty Partnership Awards; and by equipment and/or financial grants from Compaq, IBM, Intel, and Sun.

may significantly reduce overall per-cycle throughput compared to a monolithic approach. Intel's Hyper-Threading approach in the Pentium 4 [12] has the advantage of sharing execution resources among all threads, but the implementation complexity grows dramatically with more than two threads. Furthermore, the micro-op queues are partitioned but the schedulers (issue queues) are not. The schedulers are shared among threads and the only constraint is that the number of slots allocated to a given thread is not allowed to exceed a certain threshold. With more threads, much larger schedulers would be required. Thus, alternatives to these approaches must be pursued in order to yield efficient yet viable implementations with larger numbers of threads.

In this paper, we compare the per-cycle performance and energy efficiency of a variety of partitioned multi-threaded processors for four and eight threads per die. We choose a single monolithic SMT processor as a baseline against which to compare more viable approaches. In addition to the aforementioned CMP+SMT approach, we explore a partitioned SMT approach (P-SMT), a CMP of P-SMT processors (CMP+P-SMT), and a relatively new approach, a *Clustered Multi-Threaded (CMT)* processor. P-SMT is similar to Intel's Hyper-Threading in that the issue queue resources used by a single thread are constrained, but with the additional complexity reduction of explicitly partitioning the issue queue. The CMT approach partitions/clusters the execution units and potentially the data cache as well. Although extensive research has been conducted on single-threaded clustered processors [3, 9, 13, 24, 27, 28, 35], only recently has the idea of a CMT been proposed [11, 20]. The primary difference between the P-SMT and CMT approaches is that the former assigns threads to execution units at issue time, while in the more highly partitioned CMT processor, this assignment is done at dispatch time by steering each instruction to a particular cluster. The tradeoff is between the implementation complexity of many shared execution units versus the utilization of these resources.

Our results show that the best performance is obtained by restricting the sharing of the L1 Dcache banks and the execution engines among threads. On the other hand, significant sharing of the front-end resources is the best approach. The CMT approach, with appropriate sharing of the front-end, execution, and data cache resources, yields competitive, and sometimes even superior, cycle-level performance compared to other partitioned machines, as well as competitive energy efficiency. Finally, the last partitioning option that we explore is a CMP of CMT processors, each of which has fewer clusters than a single CMT organization. The scalability of the front-end and the back-end in this organization makes it an attractive alternative for future highly-threaded processors.

The rest of this paper is organized as follows. In the next section, we discuss the overall microarchitecture of the machines that we evaluate, followed by our methodology in Section 3. As CMTs are relatively new, in Section 4, we explore CMT design alternatives and propose new optimizations, including a partitioned L1 Dcache option. Our comparative results are presented in Section 5. Section 6 discusses related work, and we conclude in Section 7.

2 Partitioning Multi-Threaded Processors

There are many different options for partitioning processor resources among threads. Although not exhaustive, our work tries to capture the vast majority of the options that are most likely to be profitable.

We consider partitioning the three main microarchitectural components: the front-end, the execution engine, and the L1 DCache. The front-end includes the L1 ICache banks, the Fetch Queue, rename, and dispatch. The execution engine is comprised of the issue queues, the register file, and the functional units, while the L1 DCache consists of the Load-Store Queue and the data cache banks. Within these three major sub-systems, we consider options for partitioning the various queues and the L1 D-cache among the threads, and grouping the execution resources.

Figure 1 shows the more conventional multi-threaded microarchitectures that we explore in this paper. The SMT option is a monolithic machine similar to that proposed by Tullsen et al. [31, 32] without any level of partitioning. The major machine resources, the Fetch Queue (FQ), the Integer Issue Queue (IIQ), the Floating point Issue Queue (FIQ), the Integer and Floating point Functional Units (IFUs and FPUs), the Load-Store Queue (LSQ), and the caches, are shared among all threads. Instructions are fetched based on the ICOUNT mechanism as proposed in [32]. The L1 caches are highly banked to reduce inter-thread port conflicts.

In the partitioned SMT microarchitecture (P-SMT), all machine queues (FQ, IIQ, FIQ, and LSQ) are partitioned among all threads, and all other hardware resources are shared by all threads. The fetch policy of our P-SMT design is the same as in the SMT machine. The primary advantage of partitioning queues in the P-SMT option is that it prevents one thread from consuming so many resources so as to starve other threads.

For both the SMT and P-SMT designs, the shared functional units make it challenging to design a high speed and power-efficient machine with four or more threads. The large number of shared FUs required to exploit ILP in each of these threads significantly increases bypass complexity. A Chip Multi-Processor (CMP) of multi-threaded processors (either SMT or P-SMT – see Figure 1) circumvents these problems by equally partitioning resources among two or more processors, but this creates a resource sharing problem. Threads have access to only the resources on the

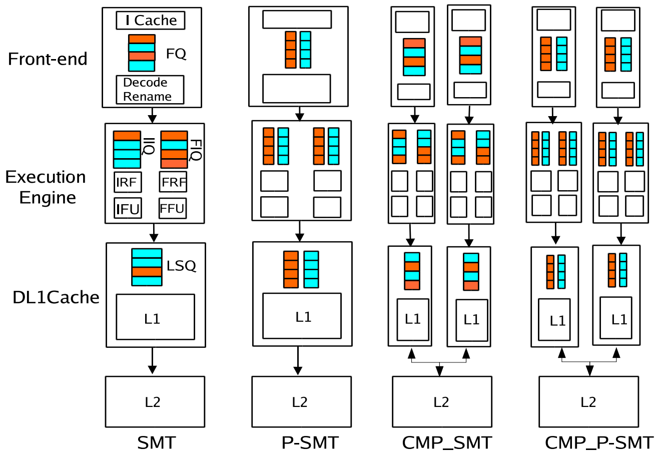

Figure 1. Conventional multi-threaded machine organizations

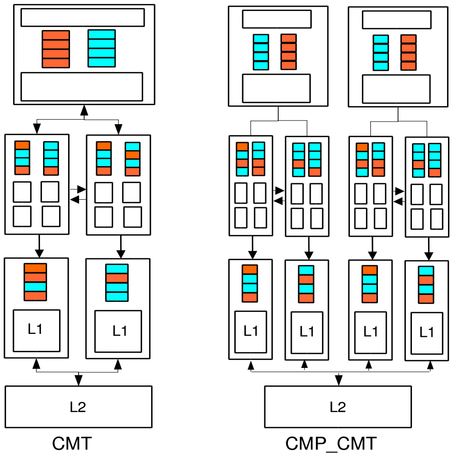

Figure 2. Clustered multi-threaded machine organizations

processor on which it was scheduled; idle resources (such as unused FUs) on another processor cannot be accessed. With higher levels of integration supporting more threads, it becomes necessary for clock speed and power reasons to create even more processors. If fewer than the maximum number of supported threads are running, then there may be many unnecessarily idle resources.

Clustered Multi-Threaded (CMT) processors support multiple threads in a clustered processor organization [3, 9, 11, 13, 20, 24, 27, 28, 35]. A CMT with a single front-end, two execution clusters, and two L1 DCache clusters is shown in Figure 2. Like a CMP, clustering avoids large monolithic structures that do not scale well (such as the large group of connected FUs in SMT and P-SMT designs), which increases clock speed, reduces power consumption, and increases scalability. Unlike a CMP, however, all of the resources in a CMT are available to all threads as communication links connect the front-end to all clusters, as well as the clusters themselves. This permits better resource utilization than CMPs.

With greater levels of integration, however, it becomes necessary to implement more clusters in order to maintain a scalable design. Thus, the maximum distance (in cycles) between clusters increases. This makes it more profitable to restrict each thread to a subset of the clusters rather than spread them across all clusters, as the added communication cost quickly overrides any ILP improvement; moreover, this reduces inter-thread resource contention, within clusters as well as over communication links. Thus, at higher levels of integration, it eventually becomes attractive to implement a CMP of CMTs (Figure 2) rather than a single large CMT.

Because CMTs are relatively new, we explore options for sharing front-end, execution, and L1 Dcache resources among threads in Section 4, before comparatively analyzing the different partitioning options in Section 5. First, we discuss our methodology.

3 Methodology

3.1 Simulation Infrastructure and Machine Configurations

Our simulator is based on Simplescalar-3.0 [8] for the Alpha AXP instruction set with the Wattch [7] power extensions. Like the Alpha 21264 [16], the register update unit (RUU) is decomposed into integer and floating point issue queues and register files, and a reorder buffer (ROB). The memory hierarchy is modeled in significant detail, including accounting for bus contention between multiple L1 DCaches, or multiple L1 DCache banks, and the unified L2 cache, for the CMP and CMT options.

Support for multithreading has been added by replicating the fetch control, rename, and ROB per thread. For the P-SMT design, per-thread FQ, IIQ, FIQ, and LSQ queues are implemented as discussed in the previous section. Because the threads in our study are independent applications, the CMP option in our simulator simply splits the front-end, execution, and L1 Dcache resources into a number of independent sections corresponding to the number of processors (SMTs, P-SMTs, or CMTs). Each processor in this context runs the same number of threads, and threads are randomly assigned to processors. To make a fair comparison, we ensure that threads are assigned to shared processors in a CMP exactly as they are assigned to shared clusters in a CMT.

For the clustered microarchitecture, we implemented Remote Access Window (RAW) structures [35] for waking up the remote consumer of a physical register. Unlike in [35], however, the physical register file serves as both the local register file and the Remote Access Buffer (RAB); no separate RAB is used. We assume a ring interconnect in which each cluster is directly connected to two other neighboring clusters via two pairs of uni-directional links in each direction. These links are required for load and store communication as well as to bypass register results among clusters. Our simulations show that a single pair of links significantly degrades performance, while performance tails off beyond two pairs. Our simulator accounts for the latency of these links (one cycle per hop) as well as contention for their use. The front-end is assumed to be co-located with cluster 0, and the distribution of instructions to the issues queues takes additional cycles based on the location of the cluster. For the ICOUNT fetch policy, per thread counters are required to prioritize the threads. We model the propagation delay from the cluster to the front-end to update these counters. (As in [32], instructions in the Fetch Queue are also considered in our ICOUNT scheme.) We also use this information to gate any thread that occupies at least $1/n$ of the total issue queue resources (across all clusters), where n is the number of threads. In all cases, the communication latency is calculated as the number of hops between the source cluster and the front-end, plus any added delay due to contention.

In our CMT model, once individual threads are assigned to particular clusters and to particular cache banks (we explore policies for both of these in Section 4), each thread is steered according to the heuristics in [3]. For instructions other than loads and stores, the mechanism of [9] that steers an instruction (and its destination register) to the cluster that produces most of its operands is used, using a criticality predictor [14, 33] to give a higher priority to the cluster that produces the critical source operand (with a provision for load balancing as is done in [9]).

Loads and stores are assigned to the cluster whose L1 DCache handles the corresponding memory address. A bank predictor [34] is used if the effective address is not known at rename time [3]. Once the effective address is computed, the request is sent to the correct cache bank via the interconnect if there was a misprediction, data is fetched from the cache bank, and returned to the requesting cluster. As the LSQ is distributed among the clusters along with the L1 DCache banks, a dummy slot is created in the other clusters whose banks are assigned to the particular thread [35]. Subsequent loads from this thread that are behind the dummy slot in these clusters are prevented from proceeding, because there is an earlier store with an unresolved address that could potentially cause conflicts. Once the effective address is computed, the information is sent to the LSQs associated with the banks across which a thread is distributed and the dummy slots in the corresponding LSQs except one are removed. This multicast operation increases the traffic on the interconnect for the register and cache data (which we model).

Table 1 lists the configuration parameters used in the evaluations. For all the machine configurations and experiments, the same total hardware of Table 1 is used unless explicitly mentioned. Figure 3 shows the performance change of increasing the size of the execution engine, including register files, issue queues and functional units, for SMT machines with two, four, eight, and 16 integer functional units (*SMT2, SMT4, SMT8, and SMT16*, respectively). Performance steadily improves as the total number of execution engines is increased. Comparing *SMT2* to *SMT16*, the improvement is about 3.25X for 4 threads and about 4X for 8 thread workloads. Although perhaps justified in terms of performance improvement, as we show later, the large *SMT16* configuration is very energy inefficient. This is one reason that we also explore partitioned SMT options, in addition to the CMT machine and various CMP options.

There are a total of eight L1 ICache banks to allow fetching from multiple threads in the same cycle. Similarly, the L1 DCache consists of a total of eight line-interleaved banks, effectively providing eight read and write ports, 1 per bank, in the absence of bank conflicts. In the base CMT architecture, a subset of the banks is associated with a cluster, with additional communication latency for accessing banks associated with a different cluster. In the SMT and P-SMT processors, the latency of the centrally located, multi-banked, L1 DCache is assumed equal to that for accessing a single bank in the CMT. This optimistic latency assumption favors the SMT and P-SMT configurations over the CMT.

One key partitioning parameter in our comparative analysis concerns the size of the execution resources that are grouped. We define this as the number of integer FUs that are grouped together, either within a SMT or P-SMT processor, or in a cluster in a CMT organization. Each of these groups has the appropriate number of FP FUs and issue queue and register file resources according to how the total resources in Table 1 are divided. Although we permit larger sizes in SMT and P-SMT processors for comparison purposes, we evaluate CMT processors with sizes of two or four. We found that a cluster size of one creates excessive communication in a CMT, while a larger size will limit clock speed and increase power dissipation as technology scales.

In our results, we designate different machine configurations according to the number of processors, the size of the grouped execution resources, and for the CMT, the number of clusters. The machine *CMPx_* has *x* processors, processors *SMTy* and *P-SMTy* have *y* integer FUs, and processor *CMTz_y* has *z* clusters of size *y*. Thus, the organization *CMPx_CMTz_y* has *x* CMT processors, each of which has *z* clusters of size *y* (*y* integer FUs).

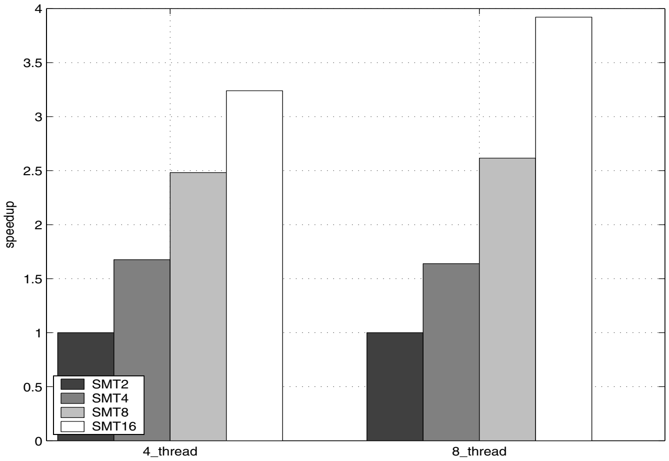

Figure 3. Execution engine size in SMT machines

Table 1. Simulator parameters

Branch predictor	comb of bimodal and 2-level
Bimodal predictor entries	2048
Level 1 table entries	1024
Level 2 table entries	4096
BTB entries, associativity	2048, 2-way
Branch mispredict penalty	10 cycles
Fetch policy	ICOUNT.8.32
Fetch width	32
Fetch queue size	32 per thread
Integer Issue queue size	160
FP Issue queue size	160
Register Access Window size	160
Load/Store queue size	384
Issue width	24
Dispatch and commit width	32
Integer Physical Registers	512
FP Physical Registers	512
Reorder Buffer Size	512 per thread
Integer FUs	16
FP FUs	8
L1 ICache	64KB, 2-way
L1 DCache	16KB, 2-Way, 8 banks, line-interleaved
L1 Dache Latency	2 cycles
L2 Cache	2MB, 8-way
L2 Cache latency	20
TLB (each, I and D)	128 entries, 8KB page size, fully associative, per thread
Cluster interconnect latency	1 cycle per hop
Communication Buffer size	15 per cluster per direction
Memory latency	100 cycles

Table 2. Benchmark classification

Benchmark Classification	Benchmarks Included
com_i	bzip, gcc, gzip, mcf, parser, perlbmk, twolf, vpr
com_f	art, equake
ilp_f	applu, galgel, lucas, mesa, mgrid, swim

Table 3. Multi-threaded workloads

Workload Name	Benchmarks Included
com_8_i	bzip, gcc, gzip, mcf, parser, perlbmk, twolf, vpr, galgel
mix_8_f	applu, mgrid, swim, equake, mesa, lucas, art, galgel
com_8_if	bzip, perlbmk, gzip, art, parser, vpr, twolf, equake
mix_8_if	mesa, parser, swim, twolf, art, gcc, lucas, mgrid
com_4_i	bzip, mcf, perlbmk, vpr
mix_4_if	gzip, applu, gcc, mesa
ilp_4_f	applu, lucas, swim, mgrid
mix_4_f	art, equake, galgel, mesa

3.2 Benchmarks and Multi-Threaded Workloads

Table 2 lists the SPEC2000 benchmarks used in our simulations, classified according to whether they are from the integer (*_i*) or floating point (*_f*) suites, and according to whether they are limited by communication (*com*) or the number of resources for exploiting ILP (*ilp*). We determined these classifications experimentally by running each benchmark individually and observing the change in IPC as the number of clusters is increased. The *ilp* benchmarks experience a gain in performance with more clusters, while the performance of the *com* benchmarks either stays constant or degrades. The former is due to the ability to make use of more execution resources or L1 Dcache banks (or both), while the latter is due to inter-cluster register to register communication, or accesses to remote L1 Dcache banks (or both). In any event, the *ilp* benchmarks tend to want to spread among more clusters than the *com* benchmarks.

Table 3 lists the multi-threaded workload mixes used to evaluate the different architectural configurations. The order in which the benchmarks are listed represents their adjacency during execution. The classification is similar to Table 2, except that we can have a mix of *ilp* and *com* (*mix_*) and integer and floating point (*if*) benchmarks, and either four or eight thread workloads.

Each individual benchmark was compiled with gcc with the -O4 optimization and run with its reference input set, fast-forwarding for 2 billion instructions, and then running for the next 100 million instructions. We therefore report the resulting IPC from executing 100 million instructions from each thread in the workload mix.

3.3 Metrics

As a performance metric, we chose the geometric mean of the relative IPC ratings of the n threads:

$$\sqrt[n]{\prod_{n} \frac{IPC_{new}}{IPC_{old}}}.$$

Since the metric used is a relative measure, the use of a geometric mean avoids the issue of which configuration is placed in the numerator or in the denominator [21]. The geometric mean equally weighs a performance improvement in one thread and an identical performance degradation in another simultaneously executing one. The harmonic mean would overly penalize a performance degradation, while the arithmetic mean would overly reward a performance improvement. The geometric mean avoids skewing the results in either direction when presenting relative performance.

For energy, we report the average energy per instruction normalized to that of the *SMT16* organization. This is an appropriate metric given that we run the exact same instructions in all configurations for a given workload.

4 CMT Optimizations

To date, there have been few efforts that explore optimizations for CMT processors. In this section, we explore alternatives for assigning threads to execution clusters, for organizing the L1 DCache banks, and partitioning the front end.

4.1 Thread to Cluster Assignment

In this section, we examine static and dynamic schemes for assigning threads to clusters. The manner in which the distribution is done preserves locality (i.e., threads are distributed to neighboring clusters) while taking the number of threads and the individual thread characteristics into account. We explore a range of possible static distributions of threads to clusters, from clusters being exclusively assigned to threads to being shared by multiple threads. Restricting the number of clusters assigned per thread reduces average communication cost and conflicts for per-cluster resources (as they are shared among fewer threads), but prevents resources from being available to all threads.

Figure 4 gives results for various static cluster assignment (CLA) schemes for the *CMT8_2* and *CMT4_4* configurations for the various four and eight thread workloads. For each workload, the results are normalized to the performance achieved by *CMT8_2* for that workload. We first note that in all cases, distributing the threads among all clusters (*CMT8_2_CLA8* and *CMT4_4_CLA4*) produces the worst performance. It is always better to assign threads to a subset of the clusters, thereby restricting communication and inter-thread contention for cluster resources. With four threads, the best static approach is to separate the threads completely from one another, assigning them to unique clusters (one cluster each for *CMT4_4* and two clusters each for *CMT8_2*). The same is true for the *com* workloads with eight threads, while the *mix* workloads perform best with two threads sharing each cluster. This is because of two factors. First, the *ILP* threads generally require more resources than provided in a single cluster for *CMT4_4* and two clusters for *CMT8_2*. The *com* benchmarks on the other hand, do not require significant resources, and thus the mixing of *com* and *ilp* threads on a cluster permits progress to be made by the *ilp* thread while the *com* thread is stalled due to communication.

As workload behavior may vary over time, we also developed a dynamic scheme that varies the thread-to-cluster assignment. Based on the static assignment results, we consider all but the *CMT8_2_CLA8* option to be potential candidates. After every 1M cycles, we execute each of these candidate thread-distribution options for 1K cycles and record the performance. After five such instrumentations, we pick the thread-to-cluster assignment that was the best most often, choosing the larger number of clusters in the case of a tie. The dynamic scheme closely matched the best static scheme in all cases, even exceeding its performance by about 10% in two cases. However, our results do not make a compelling case for the dynamic approach, as assigning threads to separate clusters works virtually just as well as the dynamic scheme in almost all cases, as also concluded by Latorre et al. [20].

4.2 L1 DCache Bank Assignment & Partitioning

In the CMT microarchitectures studied thus far, the L1 DCache banks are evenly distributed among the clusters as proposed by Zyuban [35]. The LSQ is also split across the different clusters. We explore the sharing of banks among all threads, in which case the banks are organized in a line-interleaved manner, as well as a private approach in which a given bank is assigned to a single thread independent of the thread distribution across the clusters.

The obvious tradeoff in the number of banks that are assigned to each thread is that between greater cache capacity and greater inter-cluster communication, as load operations become distributed among more clusters as more banks are assigned to threads. In terms of shared versus private banks, there is greater inter-thread interference with shared banks, but less efficient use of overall cache resources with private banks.

There are several other more subtle differences between these L1 DCache options. The bank misprediction rate, and thus the effectiveness of instruction steering, is dependent

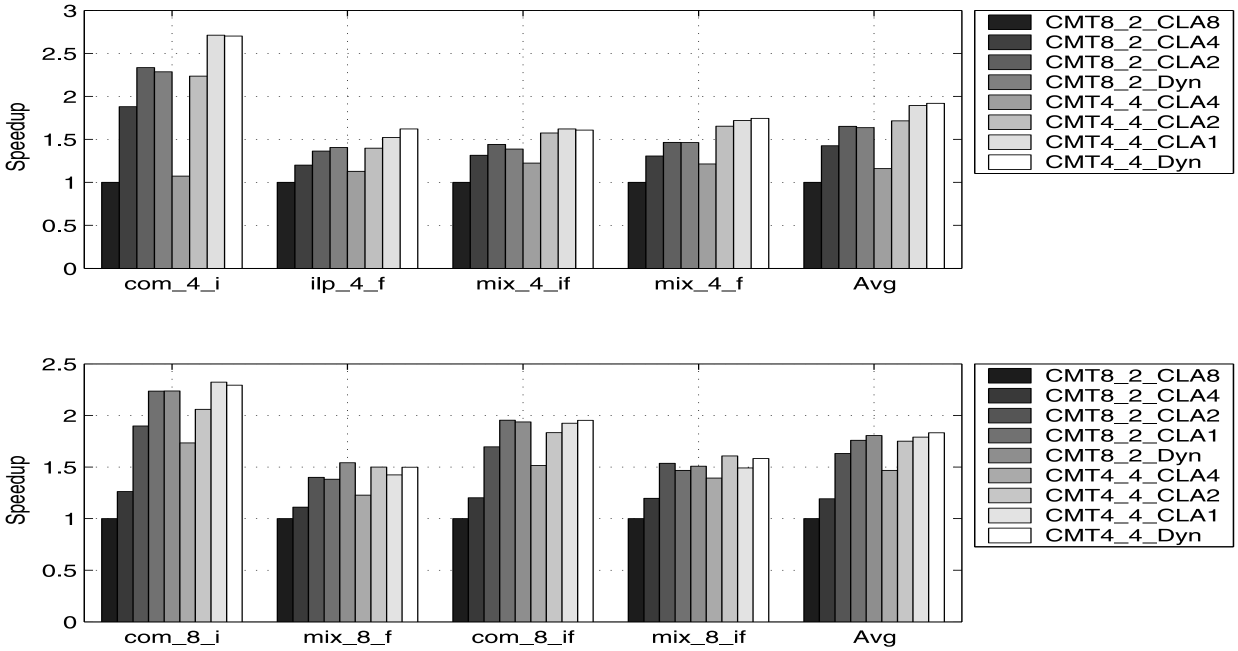

Figure 4. Static & dynamic thread to cluster assignment

on the number of banks: with fewer banks assigned to a thread, the lower the bank mispredict rate, which results in fewer corrective L1 DCache accesses. This is one advantage of limiting the degree of cache bank sharing among threads, or using fewer, private banks. A second advantage is that fewer LSQ entries are occupied by dummy slots as the number of per-thread banks is reduced, reducing the occurrence of a full LSQ. This also leads to less inter-cluster traffic as the notification of a completing store has to be multicast to fewer clusters. This is one advantage of the Partitioned option over the Decentralized one.

We first gain an understanding of the sensitivity of the L1 DCache size and bank assignment in an SMT architecture for our workloads. This permits us to isolate the effects of thread conflicts within banks and cache capacity constraints from the CMT-specific constraints discussed above. Figure 5 shows the performance of varying the total number of banks, and the number assigned per thread when using eight banks, for *SMT16* averaged over the four and eight thread workloads. Performance relative to the single bank result in each case steadily improves as the total number of banks is increased to eight. With eight banks and four threads, each thread should be given a minimum of two banks; this is the reason that there is one less bar for the four thread. Although eight banks are necessary for good SMT performance, restricting the number of banks per thread has only a small impact on performance. These results indicate that it is reasonable to consider restricting the number of L1 DCache banks assigned to threads in a CMT machine.

Figure 6 shows a four cluster CMT that illustrates the partitioning options that we evaluate for the L1 DCache[1]. In the Centralized scheme, the L1 Dcache in its entirety (all banks) and the LSQ are co-located with cluster 0 (as is the L2 cache). Load and store instructions originating from other clusters must traverse the inter-cluster interconnect to send the request, and the result returns over the interconnect as well. A Decentralized organization spreads the banks evenly over the different clusters. This is the option that we have been exploring thus far. Finally, Partitioned is an in-between option that assigns banks to between two and $n/2$, inclusive, of the n clusters. An additional degree of freedom that we explore is the assignment of threads to banks. In the Decentralized and Partitioned options, a particular L1 Dcache bank may be assigned to more than one thread (shared) or restricted to one particular thread only (private). In this section, we explore the tradeoff between the Decentralized and Partitioned options with shared and private banks.

Figure 7 compares the results for these different schemes for each of our workloads for the *CMT8_2* organization. Unlike the SMT where assigning all threads to all of the eight banks gives good results, the Centralized scheme for the CMT never performs best. The reason is the added communication overhead of accessing cluster 0 for all load and store operations significantly increases load latency and also increases network traffic. This is shown in Table 4, which gives various statistics related to the different cache

[1] Note that this does not accurately reflect the layout; the clusters are arranged so that the ring interconnect, not shown here, provides one-cycle latency between adjacent clusters, even the "end" ones. Also, we consider eight clusters in our results but four are shown here for space reasons.

Table 4. Cache statistics over all 8 thread workloads

Cache organization	LSQ occupancy	Dummy STORES	Bank prediction accuracy
Decentr, 8 banks/thrd	83.05	104.5	68.15 %
Decentr, 4 banks/thrd	92.25	50.47	73.64 %
Decentr, 2 banks/thrd	88.76	16.27	84.94 %
Decentr, 1 banks/thrd	89.84	0	100 %
Centralized	98.76	0	N/A
2 Partitions, 1 parts/thrd	95.75	0	N/A
4 Partitions, 1 parts/thrd	93.88	0	N/A

Cache organization	load latency (cycles)	DL1Miss rate	Communication buffer occupancy
Decentr, 8 banks/thrd	17.73	6.92 %	6.73
Decentr, 4 banks/thrd	16.23	6.92 %	5.48
Decentr, 2 banks/thrd	14.47	6.98 %	3.62
Decentr, 1 banks/thrd	13.39	7.38 %	1.38
Centralized	17.71	6.86 %	4.84
2 Partitions, 1 parts/thrd	17.12	6.86 %	4.59
4 Partitions, 1 parts/thrd	14.92	6.96 %	2.72

options. As expected, bank prediction rates improve with fewer banks/thread and the total number of dummy stores decreases. Communication network traffic (measured as the number of buffered messages per cycle in the system) is reduced with the Partitioned option compared to Decentralized due to the elimination of dummy stores and associated multicasts. The average load latency increases, however, due to the increase in the average distance between the clusters and the bank partitions. As expected, the miss rates do not change significantly between schemes with the same total number of banks per thread. The net effect is a small difference between the best Decentralized and Partitioned approaches. However, the Partitioned option does not require bank prediction or dummy stores; we therefore choose it due to its simplicity and expected robustness over a variety of workloads. Our conclusion for the sharing of L1 Dcache banks is similar to that for execution clusters, i.e., sharing should be restricted in order to achieve the best performance.

4.3 Partitioned Versus Shared Front-end

In this section, we examine the effect of sharing the front-end in clustered multi-threaded processors given that the back-ends are privately assigned to the threads. Figure 8 shows the relative performance of shared versus private front-ends for different workloads and back-end options. Partitioning the front-end into four clusters clearly degrades performance. For four threads, the performance loss is more pronounced as each front-end is occupied by a single thread, which greatly reduces its utilization. With four threads, moving from four to two front-ends has the greatest impact on the mix workloads (a 20%-30% improvement) due to their orthogonal machine resource requirements.

Similar results are observed with eight threads although the differences are less pronounced. Overall, although the com_4_i and ilp_4_f workloads have slightly lower performance with one compared to two front-ends, a single, unified, front-end is the best performing option, and eliminates a crossbar connection to the back-ends.

5 Comparative Analysis of Partitioning Options

Figure 9 compares the performance of various SMT options (monolithic, partitioned, with and without flushing [30]), CMT, and various CMP options relative to the SMT machine, while Figure 10 provides relative energy per instruction results. The first five bars are uniprocessor machines, while the remaining ones are various CMP options.

In comparing the uniprocessor machines, we first note that P-SMT provides a large performance boost over SMT for both 4 and 8 thread workloads. As noted previously, providing per-thread queue resources prevents a subset of the threads from hoarding machine resources and degrading overall performance. Even with the use of flushing [30], which squashs the instuctions of threads that experience DL2Cache misses in order to free resources so that other simultaneously executing threads can utilize them, P-SMT still outperforms SMT. (Note that flushing would have little impact if used with P-SMT due to the use of separate queues.) This is because flushing only addresses the L2 miss source of issue queue clog, whereas partitioned queues by their design address all sources. In addition, flushing wastes fetch bandwidth when instructions are flushed from the pipeline whereas partitioned queues incur no such penalty.

The cycle-level performance of the CMT machines is very competitive with that of P-SMT. The *CMT4_4_Dyn* organization even comes within 90-96% of the cycle level performance of *P-SMT16*, yet the CMT design will surely be simpler to implement and achieve a higher clock rate. Its energy is also slightly less than 40% of *SMT16* and less than 50% of that of *P-SMT16*. Flushing causes extra energy to be consumed due to the need to fetch, decode, and dispatch the same instructions after flushing is invoked.

The large energy reductions on the CMT architecture relative to the SMT machine are mainly the result of reductions in the register files and result bus. The issue queues and LSQs experience a noticeable energy reduction as well. These queues and the register files have essentially been partitioned into 8 banks and the energy savings come from the access of a much smaller structure. It could be argued that the SMT design could also use a banked organization. However, this would come at a cost of additional complexity as well as some additional energy for the long wires across banks that are avoided in the case of clustering. The

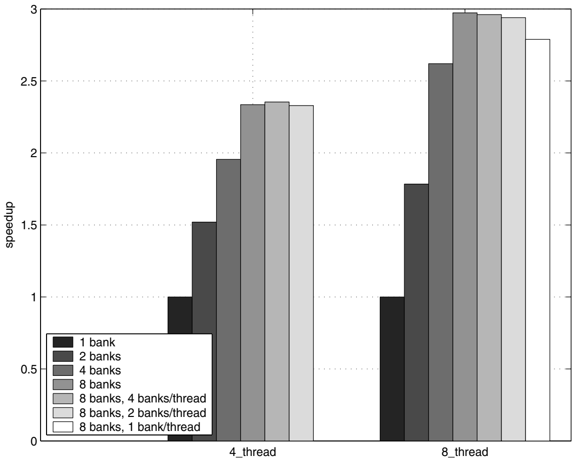

Figure 5. Cache bank assignment in an SMT machine

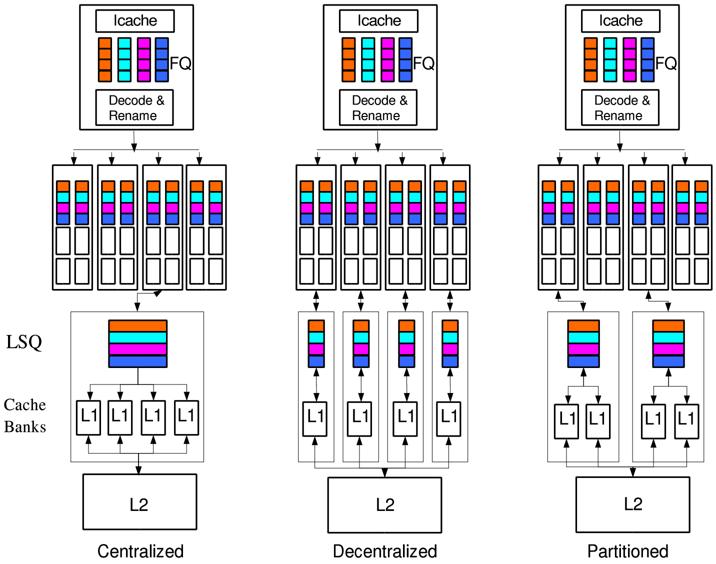

Figure 6. CMT L1 DCache options

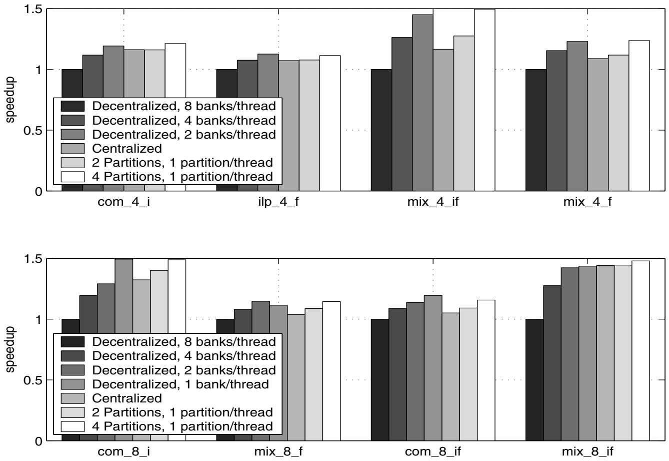

Figure 7. Cache bank assignment and cache partitioning in a CMT machine

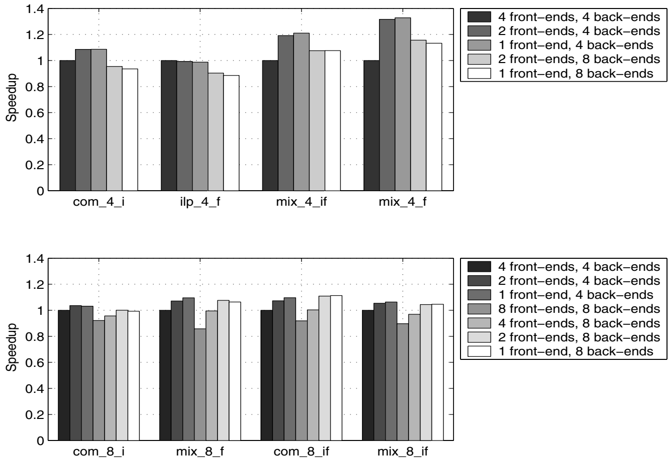

Figure 8. Front-end partitioning with private back-end in a CMT machine

result bus in each cluster of the CMT is much shorter and lightly loaded compared to that for the SMT architecture. Some of the energy reduction in the result bus is compensated for by the *Communication* component, i.e., the energy in the cluster interconnects. However, since communication is point-to-point and occurs only when dependencies exist across clusters, while results are always broadcast on the result bus in the SMT design, this additional energy is much smaller than that saved in the result bus.

As expected, the result bus and register file energy are similar for both SMT and P-SMT, but the issue queue (IQ) energy *increases* despite the partitioning in P-SMT. This is due to the interconnection network required between the queues and the shared FUs (which is not required in either SMT or CMT), the contribution of which we lump into the IQ category.

Previously, we showed how *CMT4_4_Dyn* outperforms *CMT8_2_Dyn*. The difference in clock power due to fewer clusters also manifests itself in a similar energy difference as shown in Figure 10. The *CMT8_2_Dyn* has the advantage, however, of having a smaller cluster size, and therefore would be expected to have either a faster clock or a shallower intra-cluster pipeline. These factors are not taken into account in our results.

The CMP options that we evaluate include two and four processor SMT and P-SMT organizations, an eight single-threaded (ST) processor configuration, and two processor CMT organizations. Four processor CMT options are not considered as this would require twice the number of FUs as the other options.

We first observe that the performance gap between the P-SMT and CMT options closes considerably for the CMP organizations (but the energy gap closes as well). The reason is that a major advantage of P-SMT has been diminished: its ability to assign threads to any of the FUs. Thus, there is a non-negligible performance drop from *P-SMT16* to *CMP2_P-SMT8*, while the performance of the CMT organizations (which have the same cluster size but half the number of clusters compared to the uniprocessor CMT options) remains steady. Recall that the CMT cluster and cache assignment policies restrict threads to a few clusters. Therefore, creating a CMP of fewer cluster CMTs has very little impact. The energy of the SMT and P-SMT options drops significantly in a 2-processor CMP due to significant reductions in register file, result bus, and IQ energy. Yet CMT is compellingly better than CMP2_P-SMT8 when considering the complexity in implementing an out-of-order processor with eight integer ALUs (compared to four with CMT4_4).

The remaining three CMP organizations, consisting of four processor SMT and P-SMT configurations, and eight single-threaded processors are much more implementable than the previous SMT and P-SMT options. However, their performance is significantly less than the CMT options. The flexibility advantage of SMT and P-SMT over CMT is non-existent at this point; in fact, the CMT organizations have *more* flexibility as each thread has access to more FUs via the inter-cluster network. Furthermore, both front-end and back-end resources are now highly divided among threads with four and eight processor CMPs. Thus, our results make the case for CMT4_4, which outperforms CMP4_P-SMT4 by a wide margin for our workloads.

Finally, we note that the uniprocessor CMT options and dual processor of smaller CMTs achieve almost identical performance and energy efficiency. For a coarse-grain threaded workload environment like that modeled here, the choice is one of hardware and software implementation complexity. More pronounced differences may occur for parallel workloads, but this is the subject of future work.

In summary, a Clustered Multi-Threaded organization with a single front end, clusters of four integer units, a Partitioned L1 DCache, and appropriate thread distribution heuristics, can achieve 90-96% of the cycle-level performance of a partitioned SMT processor with 16 integer units shared among all threads. The CMT is much more realizable, more energy efficient, and is likely to have either a higher clock or lower internal latencies. The CMPs of more realistic P-SMT and SMT configurations fall far short of the CMT performance. Thus, for future chips supporting four and eight thread workloads, a uniprocessor CMT or a dual processor of simpler CMTs are very attractive options.

6 Related Work

As feature sizes shrink and wires become slower relative to logic, future microprocessors will tend to be more communication bound than computation bound [1], and it will not be possible to increase the number of threads in a SMT processor further. Zyuban and Kogge [35] proposed a multicluster architecture to reduce power consumption in superscalar processors. A multicluster architecture is power-efficient, tackles the wire delay problem, and is scalable. The only drawback of such an architecture is the intercluster communication cost. Various groups [2, 3, 4, 6, 9, 10] have studied cluster assignment mechanisms for one thread to reduce the overhead of inter-cluster communication. We extend their ideas to a clustered architecture with multiple threads.

Krishnan and Torrellas [18] investigated a hybrid SMT/CMP approach of multiple SMT processor cores on a chip. Their results showed that limiting the level of SMT allowed performance benefits comparable to full-scale SMT over the pure CMP approach while retaining its shorter cycle time advantage. Hetrogeneous multi-core architectures for multi-threaded workloads have been studied by Kumar et al. [19]. They also proposed dynamic thread to core assignment policies. Recently, Collins and Tullsen [11] evaluated the effect of clustering the front-end, execution engine, and register file on the overall performance of a multithreaded processor. Cache banking and clustering was not evaluated. Their results showed that multithreading can reduce the negative IPC impact of clustering. Raasch and Reinhardt [25] studied the performance impact of partitioning hardware resources in SMT processors and showed that partitioning of the storage structures avoided starvation due to resource contention, while the sharing of issue bandwidth and functional units contributed to SMT's improved throughput. Latorre et al. [20] have also studied clustered multithreaded processors. Their work is the closest to our proposed CMP_CMT architecture except that simultaneous use of individual front-ends or back-ends by multiple threads is not considered. Also, they don't study processors with more than four threads, cache partitioning options, or detailed comparisons with other partitioned multi-threaded alternatives. [3, 26] investigated cache designs for clustered microarchitectures. Balasubramonian et al. [3] used dummy stores for memory disambiguation, whereas Racunas and Patt [26] used a centralized partition assignment table for maintaining information about data assignment to partitions. We chose the former approach of dummy stores in our design to avoid the extra hardware complexity.

Tullsen et al. [31] proposed simultaneous multithreading (SMT) to attack both horizontal and vertical waste. They showed a 52% average speedup as compared to a four-processor, single-chip multiprocessor with comparable ex-

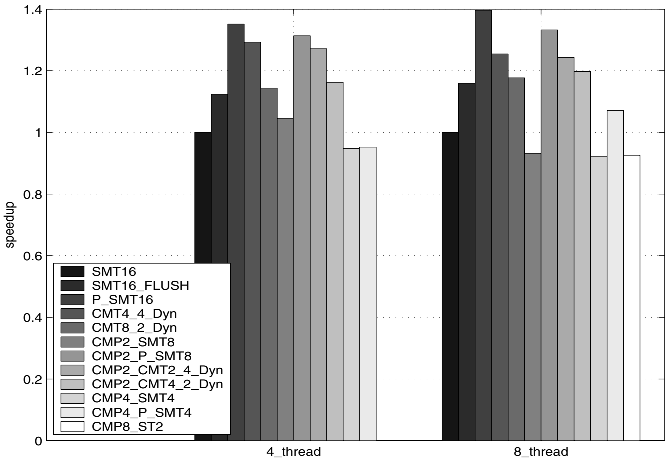

Figure 9. Comparison of the performance of the different machine organizations

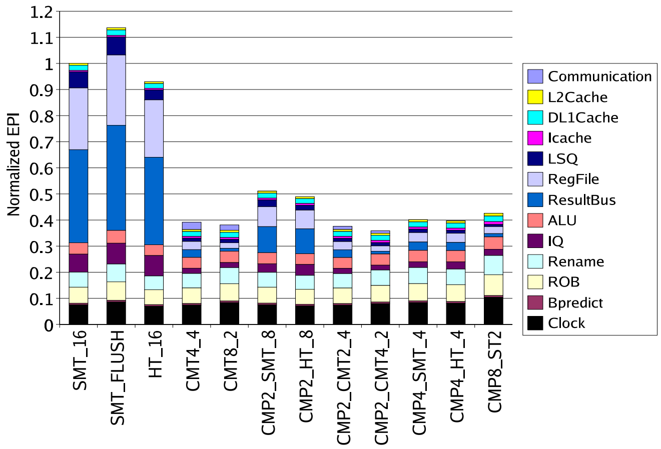

Figure 10. Average normalized energy (EPI) over all the 8-thread workload mixes for each of the architectural configurations normalized to the monolithic SMT architecture

ecution resources. Intel's Hyper-Threaded Pentium 4 processor [17, 22] is a SMT processor with support for two threads, which was analyzed by Tuck and Tullsen [29]. Huh et al. [15] studied the space of CMP organizations. Two commercial CMPs include IBM's Power5 [23], a CMP of SMT processors, and the experimental Piranha system [5] developed at Compaq. The latter integrates eight simple Alpha processor cores along with a two-level cache hierarchy onto a single chip. Our work includes these design points in a comparative evaluation of CMTs and CMPs with and without SMT support for up to 8 contexts.

7 Conclusions and Future Work

With higher levels of integration supporting more on-chip threads, the way in which the resources in multi-threaded processors are partitioned needs to be examined. In this paper, we compare the performance and energy efficiency of several partitioning options in a 4 to 8-thread multi-threaded architecture, ranging from chip multiprocessing (CMP) to simultaneous multi-threading (SMT), for a fixed amount of total resources. Our analysis shows that the best performance is obtained by partitioning (and restricting the sharing of) the L1 data cache banks and execution units among threads, but allowing significant sharing of the front-end resources. The corresponding clustered multi-threaded (CMT) architecture is highly competitive with unrealizable SMT processors, achieving 90-96% of the cycle-level performance of a partitioned SMT (which improves on the base SMT), while dissipating about 50% of its energy. Adopting a dual-processor CMP approach closes the performance gap between the SMT and CMT processors, as the major advantage of the former, flexibility in the assignment of threads to FUs, diminishes. Further dividing the resources for SMT processors among four processors creates more realizable organizations, but their performance falls far short of the CMT alternatives. We find that a monolithic CMT and a dual processor CMP of CMT processors are very attractive options for future designs.

We have recently incorporated support in our infrastructure for parallel applications and are exploring support for database and server-oriented benchmarks. They are the subject of our ongoing work.

References

[1] V. Agarwal, M. S. Hrishikesh, S. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–260, June 2000.

[2] A. Aletá, J. M. Codina, A. González, and D. Kaeli. Instruction Replication for Clustered Microarchitectures. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 326–336, December 2003.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 275–286, June 2003.

[4] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 337–347, December 2000.

[5] L. A. 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*, pages 282–293, June 2000.

[6] R. Bhargava and L. K. John. Improving Dynamic Cluster Assignment for Clustered Trace Cache Processors. In *Proceedings of the*

30th Annual International Symposium on Computer Architecture, pages 264–274, June 2003.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[8] D. Burger and T. Austin. The Simplescalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[9] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Cluster Assignment Mechanisms. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pages 132–142, January 2000.

[10] M. Chu, K. Fan, and S. Mahike. Region-based Hierarchical Operation Partitioning for Multicluster Processors. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 300–311, June 2003.

[11] J. D. Collins and D. M. Tullsen. Clustered Multithreaded Architectures – Pursuing Both IPC and Cycle Time. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pages 76–86, April 2004.

[12] I. Corporation. IA-32 Intel Architecture Optimization: Reference Manual. *http://www.intel.com/design/pentium4/manuals*, 2004.

[13] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time through Partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 149–159, December 1997.

[14] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[15] J. Huh, S. W. Keckler, and D. Burger. Exploring the Design Space of Future CMPs. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, pages 199–210, September 2001.

[16] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[17] D. Koufaty and D. T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

[18] V. Krishnan and J. Torrellas. A Clustered Approach to Multithreaded Processors. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 627–634, March 1998.

[19] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, pages 64–7, June 2004.

[20] F. Latorre, J. González, and A. González. Back-end Assignment Schemes for Clustered Multithreaded Processors. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing*, pages 316–325, June 2004.

[21] K. Luo, J. Gummaraju, and M. Franklin. Balancing Thoughput and Fairness in SMT Processors. In *International Symposium on Performance Analysis of Systems and Software*, pages 164–171, January 2001.

[22] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[23] R. Merritt. IBM Weaves Multithreading into Power5. *EE Times*, 2003.

[24] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 206–218, June 1997.

[25] S. Raasch and S. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proceedings of the 12th International Conference of Parallel Architectures and Compilation Techniques*, pages 15–26, September 2003.

[26] P. Racunas and Y. N. Patt. Partitioned First-Level Cache Design for Clustered Microarchitectures. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 22–31, June 2003.

[27] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 138–148, December 1997.

[28] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[29] N. Tuck and D. M. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proceedings of the 12th Annual International Symposium on Parallel Architectures and Compilation Techniques*, pages 26– 34, September 2003.

[30] D. Tullsen and J. Brown. Handling Long-latency Loads in a Simultaneous Multithreading Processor. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 318–327, December 2001.

[31] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[32] D. Tullsen, S. Eggers, H. Levy, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[33] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic Prediction of Critical Path Instructions. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 185–196, January 2001.

[34] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 42–53, May 1999.

[35] V. Zyuban and P. Kogge. Inherently Lower-Power High-Performance Superscalar Architectures. *IEEE Transactions on Computers*, 50(3):268–285, March 2001.