Dynamic Capacity-Speed Tradeoffs in SMT Processor Caches*

Sonia López[1], Steve Dropsho[2], David H. Albonesi[3],
Oscar Garnica[1], and Juan Lanchares[1]

[1] Departamento de Arquitectura de Computadores y Automatica,
U. Complutense de Madrid, Spain
[2] School of Computer and Communication Science, EPFL, Switzerland
[3] Computer Systems Laboratory, Cornell University, USA

Abstract

Caches are designed to provide the best tradeoff between access speed and capacity for a set of target applications. Unfortunately, different applications, and even different phases within the same application, may require a different capacity-speed tradeoff. This problem is exacerbated in a Simultaneous Multi-Threaded (SMT) processor where the optimal cache design may vary drastically with the number of running threads and their characteristics.

We propose to make this capacity-speed cache tradeoff dynamic within an SMT core. We extend a previously proposed globally asynchronous, locally synchronous (GALS) processor core with multi-threaded support, and implement dynamically resizable instruction and data caches. As the number of threads and their characteristics change, these adaptive caches automatically adjust from small sizes with fast access times to higher capacity configurations. While the former is more performance-optimal when the core runs a single thread, or a dual-thread workload with modest cache requirements, higher capacity caches work best with most multiple thread workloads. The use of a GALS microarchitecture permits the rest of the processor, namely the execution core, to run at full speed irrespective of the cache speeds. This approach yields an overall performance improvement of 24.7% over the best fixed-size caches for dual-thread workloads, and 19.2% for single-threaded applications.

1. Introduction

Simultaneous Multi-Threading (SMT) [18, 19] is a widely used approach to increase the efficiency of the processor core. In SMT, multiple threads simultaneously share many of the major hardware resources, thereby making use of resources that may lie partially unused by a single thread. SMT processors have the significant advantage of dynamically trading off instruction-level parallelism (ILP) for thread-level parallelism (TLP). That is, hardware resources that are partially unoccupied due to insufficient single-thread ILP can be filled by instructions from a second thread. This leads to a significant boost in instruction throughput over a single threaded processor with only a modest increase in hardware resources.

* This research was supported in part by Spanish Government Grant TIN2005-05619, National Science Foundation Grant CCF-0304574, an IBM Faculty Partnership Award, a grant from the Intel Research Council, and by equipment grants from Intel and IBM.

Despite this advantage, large variations in runtime load cause greater pipeline inefficiencies in an SMT processor. These load fluctuations arise due to increased variation in workload phase behavior, due to individual application phase differences coupled with variations in the number of actively running threads. There are several reasons why the number of running threads may differ from the maximum supported in the SMT hardware:

- A server will not always operate at 100% load, leaving some processors running with fewer than the maximum number of threads;
- Lock contention in systems such as databases can restrict the number of threads that can run simultaneously on the processor core;
- A parallel program will have parallel portions running many threads, and serial portions in which only a single thread runs.

Thus, a significant challenge for SMT processor core microarchitects is devising a single core microarchitecture that is near optimal under a variety of single and multiple thread operating conditions. Whereas a single thread, and some dual-thread workloads, benefits from a streamlined pipeline with small, fast, hardware structures, such an organization would yield sub-optimal performance for many multi-threaded workloads. Rather, larger structures would yield a better tradeoff between hardware resource complexity and operating speed.

To address this disparity, we propose a phase-adaptive cache hierarchy whose size and speed can dynamically adapt to match varying SMT workload behavior. We implement a dual-threaded core, and begin with a small, fast cache that works well for single threaded, and some dual-threaded, workloads with modest working sets. The addition of a second running thread often puts more demand on cache capacity, and this greater demand is met by dynamically *upsizing* the cache. We adopt the *Accounting Cache* design of [8] for our resizable caches.

Because the upsized cache has a higher access time, we also adopt a Globally Asynchronous, Locally Synchronous (GALS) design style, in particular, the *Multiple Clock Domain (MCD)* approach of [16] to decouple the adaptive caches from the execution core. Unlike [16], our MCD processor supports SMT and a different domain organization. Since the integer and floating point domains are at fixed frequency, and there is negligible interaction between them (and thus negligible synchronization cost), they in effect serve as a single "execution core" domain. This execution core operates at full speed while the front-end and load/store domains adapt their frequencies to the resizing of the caches.

Our results demonstrate that adaptive caches are very effective at reacting to single and dual-threaded workload phase behavior. As expected, larger cache configurations are more prominent for the dual-thread workloads due to the higher cache pressure of running two simultaneous threads. The cache control algorithm often dynamically chooses more optimal smaller, faster caches for single-threaded workloads, and in several cases different cache configurations run for non-trivial periods of execution, demonstrating the benefit of phase-adaptive multi-threaded workload adaptation. The result is an average 24.7% performance improvement over an aggressive baseline synchronous architecture with fixed cache designs for two-threaded workloads, and 19.2% for single-threaded applications.

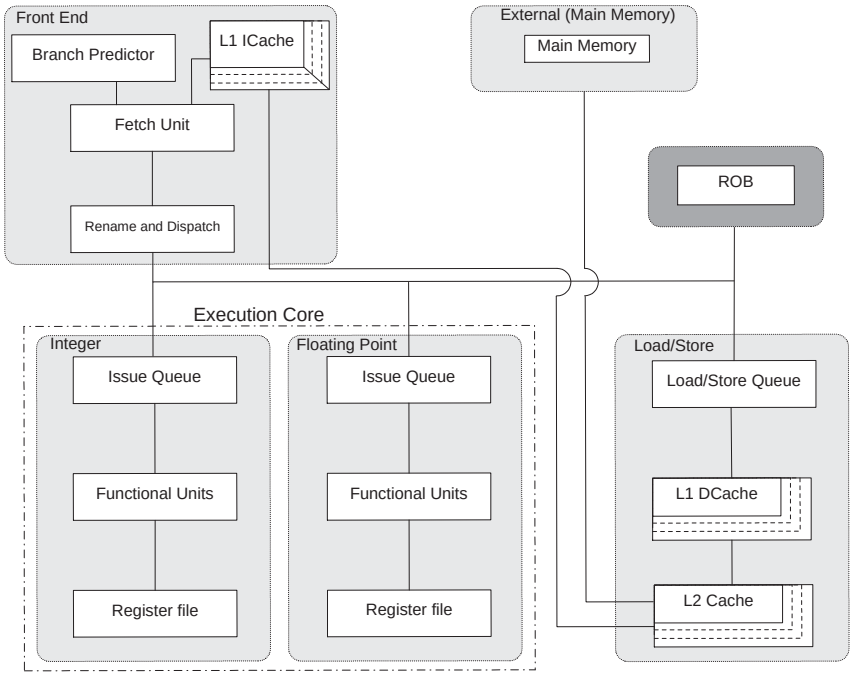

Fig. 1. Adaptive MCD microarchitecture. Boxes with multiple borders indicate resizable structures.

The rest of this paper is organized as follows. The next section discusses the adaptive SMT MCD microarchitecture, including the adaptive cache organizations. Section 3 briefly reviews the Accounting Cache algorithm for resizing the caches and modifying their domain frequencies. Our simulation infrastructure and benchmarks are described next, followed by our results. Finally, we discuss related work in Section 6, and present our conclusions in Section 7.

2. Adaptive SMT MCD Microarchitecture

The base MCD microarchitecture highlighted in Figure 1 has five independent clock domains, comprising the front end (L1 ICache, branch prediction, rename and dispatch); integer processing core (issue queue, register file and execution units); floating-point processing core (issue queue, register file and execution units); load / store unit (load / store queue, L1 DCache and unified L2 cache); and ROB (Reorder Buffer). The integer and floating-point domains have fixed structures and run at fixed frequency at all times. Since there is little interaction between them (and thus their interface introduces negligible synchronization cost), they are effectively one fixed-frequency execution core domain.

The ROB lies in its own independent domain running at all times at full frequency. Since the speed of both dispatch and commit depends on the operating speed of the ROB, decoupling the ROB in this manner permits the front and back-end domains to be independently adapted [21]. The dynamic frequency control circuit within the two adaptive domains (front end and load / store) is a PLL clocking circuit based on industrial circuits [7, 10]. The lock time in our experiments is normally distributed with a mean time of 15μs and a range of 10–20μs. As in the XScale processor [7], we assume that a domain is able to continue operating through a frequency change. Main memory operates at the same fixed base frequency as the processing core and is also non-adaptive.

Data generated in one domain and needed in another must cross a domain boundary, potentially incurring synchronization costs. The MCD simulator models synchronization circuitry based on the work of Sjogren and Myers [17]. It imposes a delay of one cycle in the consumer domain whenever the distance between the edges of the two clocks is within 30% of the period of the faster clock. Both superscalar execution (which allows instructions to cross domains in groups) and out-of-order execution (which reduces the impact of individual instruction latencies) tend to hide synchronization costs, resulting in an average overall slowdown of less than 3% [15]. Further details on the baseline MCD model, including a description of the inter-domain synchronization circuitry, can be found in prior papers [12, 14–16] . We extend this model with SMT (dual-thread) support, the details of which are provided in Section 4.

In this paper, we focus on the varying caching needs of SMT workloads. We therefore make the L1 ICache in the front end domain adaptive, as well as the L1 DCache and L2 cache of the load / store domain. This *adaptive SMT MCD architecture* has a base configuration using the smallest cache sizes running at a high clock rate. For applications that perform better with larger storage, caches can be upsized with a corresponding reduction in the clock rate of their domain. In this study, the three non-adaptive domains, integer, floating point, and main memory, run at the base frequency of 1.0 GHz. Only the front end and load / store domains make frequency adjustments. The L1 DCache and L2 cache are resized in tandem.

Having adaptable structures and a variable clock means that structures may be safely oversized [9]. The greater capacity (and lower domain frequency) is used only if a workload attains a net benefit; Section 5 demonstrates that this occurs more often under dual-thread conditions. Workloads that do not require the extra capacity, such as many single-threaded workloads, configure to a smaller size and run at a higher frequency. This approach permits the tradeoff between per-domain clock rate and complexity to be made for each workload phase.

Implementing adaptive structures incurs two static performance penalties. First, the degree of pipelining in each domain matches the high frequency that supports the smallest dynamic configuration. When the clock frequency is lowered to accommodate the additional delay of an upsized structure, the resulting stage delay imbalance results in a design that is over-pipelined with respect to the new frequency. This results in a longer branch mis-predict penalty compared with a tuned synchronous design. In our study, the adaptive SMT MCD microarchitecture incurs one additional front end cycle and two additional integer cycles for branch mispredictions. Second, we have found that the

resizable structures should be optimized in terms of sub-banking for fast access at the smallest size. However, for modularity, the larger configurations consist of duplicates of this baseline structure size. These larger configurations may have a different sub-bank organization than the same size structure that has been optimized for a non-adaptive design. We model this cost as additional access latency for the larger configurations.

2.1. Resizable Caches

In the load / store domain, the adaptive L1 DCache and L2 cache are up to eight-way set associative, and resized by ways [2, 8]. This provides a wide range of sizes to accommodate a wide variation in workload behavior. The base configuration (smallest size and highest clock rate) is a 32 KB direct-mapped L1 DCache and a 256 KB direct-mapped L2 cache. Both caches are upsized in tandem by increasing their associativity. We restrict the resizing to one, two, four, and eight ways to reduce the state space of possible configurations (shown in Table 1).

We use version 3.1 of the CACTI modeling tool [20] to obtain timings for all plausible cache configurations at a given size. The *Optimal* columns in Table 1 list the configurations that provide the fastest access time for the given capacity and associativity, without the ability to resize. The number of sub-banks per way in the *Adapt* columns were chosen by adopting the fastest configuration of the minimal-size structure and then replicating this configuration at higher levels of associativity to obtain the larger configurations. This strategy ensures the fastest clock frequency at the smallest configuration, but may not produce the fastest possible configuration when structures are upsized. Since CACTI configures a 32 KB direct-mapped cache as 32 sub-banks, each additional way in the adaptive L1 DCache is an identical 32 KB RAM. The reconfigurable L2, similarly, has eight sub-banks per 256 KB way. In contrast, the number of sub-banks in an optimal fixed L1 varies with total capacity, and the optimal L2 structure has four sub-banks per way for all sizes larger than the minimum.

In the front end domain, the L1 ICache is adaptive; the configurations are shown in Table 2. The cache adapts by ways as with the L1 DCache, but with associativities of one, two, three, and four.

Table 1. Adaptive L1 DCache and L2 Cache configurations. The *Size* column refers to the size of the *A* partition selected by the cache control algorithm, as discussed in Section 3. The column *Adapt* provides the number of sub-banks per way for each adaptive cache configuration, while *Optimal* gives the number that produces the fastest access time at that size and associativity.

	L1 DCache		**Sub-banks**		**L2 Cache**		**Sub-banks**	
Configuration	**Size**	**Assoc**	**Adapt**	**Optimal**	**Size**	**Assoc**	**Adapt**	**Optimal**
D0	32 KB	1	32	32	256 KB	1	8	8
D1	64 KB	2	32	8	512 KB	2	8	4
D2	128 KB	4	32	16	1 MB	4	8	4
D3	256 KB	8	32	4	2 MB	8	8	4

Table 2. Adaptive L1 ICache configurations.

	I-cache, dynamic		
Configuration	**Size**	**Assoc**	**Sub-banks**
I0	16 KB	1	32
I1	32 KB	2	32
I2	48 KB	3	32
I3	64 KB	4	32

3. Phase Adaptive Cache Control Algorithm

To control the reconfigurable caches, we employ the *Accounting Cache* algorithm previously applied to improving cache energy efficiency [8]. Due to the fact that the smaller configurations are subsets of the larger ones, the algorithm is able to collect statistics for all possible configurations simultaneously. This permits the calculation of the number of hits and misses that *would have* occurred over a given span of time for any of the possible configurations.

In a four-way set associative Accounting Cache there are four possible configurations, as shown in Figure 2. In this example, the *A* partition can be one, two, three, or four ways. The *B* partition is the remaining portion. The *A* partition is accessed first. If the data block is found, it is returned. Otherwise a second access is made to the *B* partition, which causes blocks to be swapped in the *A* and *B* partitions. Note that the *B* partition is not considered a lower level of cache. In the adaptive MCD architecture, all three caches (L1 ICache, L1 DCache, and combined L2 Cache) have their own *A* and *B* partitions. An access to an L1 cache accesses its *A* partition first; on a miss, the access is sent to the *B* partition and simultaneously to the L2 *A* partition. A hit in the L1 *B* partition squashes the in-flight L2 access.

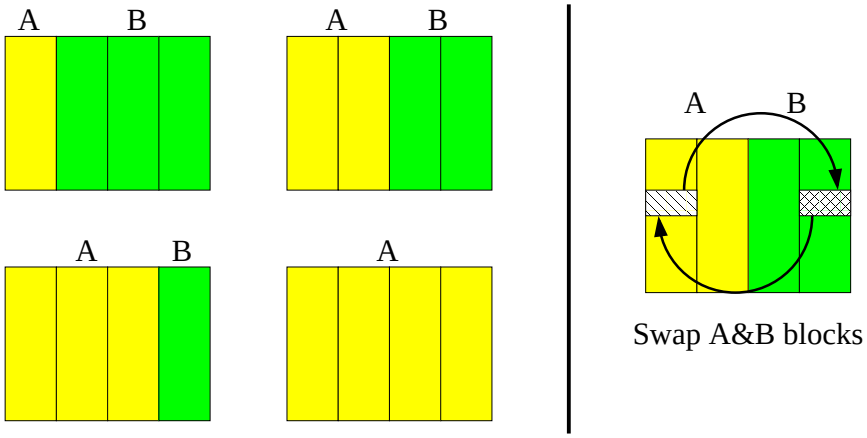

Fig. 2. Partitioning options for a four-way Accounting Cache.

A cache with a small *A* partition runs at a higher frequency than one with a larger *A* partition. (The *B* partition access latency is an integral number of cycles at the clock rate dictated by the size of the *A* partition.) At runtime, the cache control algorithm attempts to continually maintain the best balance between the speed of an *A* access and the number of slower *B* accesses.

As described in detail in previous work [8], the Accounting Cache maintains full most-recently-used (MRU) state on cache lines. Simple counts of the number of blocks accessed in each MRU state are sufficient to reconstruct the precise number of hits and misses to the *A* and *B* partitions *for all possible cache configurations, regardless of the current configuration*. The control algorithm resets the counts at the end of every 15K instruction interval, choosing a configuration for the next interval that would have minimized total access delay cost in the interval just ended. This interval length is comparable to the PLL lock-down time. Thus, during a frequency change we always run with a permissible configuration: downsizing at the beginning of the change when speeding up the clock, upsizing at the end of the change when slowing down. The control algorithm circuitry requires 10K equivalent gates and 32 cycles per cache configuration decision [9].

4. Evaluation Methodology

The simulation environment is based on the SimpleScalar toolset [5] with MCD processor extensions [16]. The modifications include splitting the Register Update Unit (RUU) into a Reorder Buffer and separate issue queues and physical register files for both the integer and floating point domains. The time management code has also been re-written to emulate separate clocks for each domain, complete with jitter, and to account for synchronization delays on all cross-domain communication.

For this study, we extended this model to support an SMT core. The SMT processor extensions include independent program counters for each thread; thread IDs for caches and predictor history tables; and per-thread ROBs and queues. Table 3 contains a summary of the simulation parameters. These have been chosen, in general, to match the characteristics of the Alpha 21264, but with additional resources for two threads.

Table 4 provides timing parameters for adaptive L1 and L2 caches, as well as the clock domain frequencies for each configuration. The four configurations of each of the load/store (D0-D3) and fetch (I0-I3) domains are shown. Listed for each configuration are the frequency of the domain and the cache access times (in cycles) at that frequency. The first access time is for *A* partition accesses and the second is the *B* partition access. For comparison, the baseline fully synchronous processor (described in detail below) runs at a frequency of 1.0 GHz and has an L1 DCache access time of two (pipelined) cycles, L2 access time of 12 (pipelined) cycles, and an L1 ICache access time of one cycle. Note that larger adaptive cache configurations have over double the access latency (in ns) of the fully synchronous baseline design. However, if for a given interval, the larger cache could have reduced the number of misses of an application sufficiently to compensate for this extra delay on every access, then the reconfiguration algorithm will upsize the cache for the next interval.

Our workload consists of nine programs from the SPEC2000 suite, which we combine into nine dual-thread workloads, shown in Table 6. Table 5 specifies the individual benchmarks along with the instruction windows and input data sets.

We chose a fully synchronous baseline processor based on a prior study [9]. In that study on *single thread* performance, an exhaustive search of the design space was made to determine the best overall baseline fully synchronous design for a set of 32 applications (of which the benchmarks in this study are a subset). The study ran 1024 config-

Table 3. Architectural parameters for simulated processor.

Fetch queue (per thread): 16 entries
Issue queue (per thread): 32 Int, 32 FP
Branch predictor: Combined gshare & 2-level PAg
Level 1 1024 entries, history 10
Level 2 4096 entries
Bimodal predictor size 2048
Combining predictor size 4096
BTB 4096 sets, 2-way
Branch mispredict penalty: 10 front-end + 9 integer cycles
Decode, issue, and retire widths: 8, 11, and 24 instructions
Memory latency: 80 ns (1st access), 2 ns (subsequent)
Integer ALUs: 6 + 1 mult/div unit
FP ALUs: 4 + 1 mult/div/sqrt unit
Load/store queue (per thread): 32 entries
Reorder buffer (per thread): 256 entries

Table 4. Cache latencies (in cycles) and domain frequency for each cache configuration.

Load/Store Domain				
Configuration	D0	D1	D3	D4
Frequency	1.59 GHz	1.00 GHz	0.76 GHz	0.44 GHz
L1 DCache Latency (A/B)	2/7	2/5	2/2	2/-
L2 Cache Latency (A/B)	12/42	12/27	12/12	12/-

Front End Domain				
Configuration	I0	I1	I3	I4
Frequency	1.62 GHz	1.14 GHz	1.12 GHz	1.10 GHz
L1 ICache Latency (A/B)	2/3	2/2	2/2	2/-

Table 5. SPEC2000 benchmarks, input datasets used, and simulation windows.

Benchmark	**Datasets**	**Simulation window**
Integer		
`bzip2`	source 58	100M–200M
`gzip`	source 60	100M–200M
`parser`	ref	100M–200M
`vpr`	ref	190M–290M
Floating-Point		
`art`	ref	300M–400M
`equake`	ref	100M–200M
`galgel`	ref	100M–200M
`mesa`	ref	100M–200M
`mgrid`	ref	100M–200M

uration combinations on each of the 32 applications and required 160 CPU months to simulate. The result for a single threaded processor was that best overall performance is achieved with a direct-mapped L1 ICache, L1 DCache, and L2 cache with sizes 64 KB, 32 KB, and 256 KB, respectively. To create a fully synchronous SMT competitor to

Table 6. SPEC2000 dual-thread workloads.

Integer
bzip2-vpr, parser-gzip, vpr-parser
Floating-Point
art-galgel, equake-galgel, mesa-mgrid
Combined Integer and Floating-Point
galgel-bzip2, gzip-equake, vpr-art

the adaptive SMT MCD architecture, we made the L1 instruction cache two-way set-associative to support SMT and reduced its access time to one cycle from two cycles (we determined a faster L1 ICache access is more important than additional cache capacity), and doubled the L1 DCache and L2 cache (to 64 KB and 512 KB, respectively) made each two-way set-associative. We did not increase the access times from the original baseline processor of [9] even though we increased the capacity and/or associativity (we actually decrease the L1 ICache access time while increasing its associativity). The access times are one cycle, two cycles, and 12 cycles for the L1 ICache, L1 DCache, and L2 cache, respectively.

5. Performance Results

In this section, we compare the performance of the SMT MCD microarchitecture with adaptive caches with that of the baseline fully synchronous design described in Section 4, for both dual-thread and single-thread SPEC2000 workloads. First, however, we demonstrate the variety of cache behavior observed in SMT machines, by comparing the best-performing cache configurations selected by the Accounting Cache algorithm for single and dual-thread workloads.

Figure 3 shows the percentage of time each application spends in a particular load / store domain cache configuration when run in isolation (non-SMT) on the adaptive MCD SMT processor, while Figure 4 shows the corresponding results for the dual-thread workloads. (We do not discuss the instruction caches since the control algorithm always selects the configuration with the smallest *A* partition. The reason is the front end is generally more sensitive to the clock frequency rather than L1 ICache capacity.) As the first figure shows, most applications exhibit a dominant behavior requiring a particular cache configuration. Exceptions are `art`, `mgrid`, and `vpr` which use multiple configurations a noticeable fraction of the time. While four of the benchmarks prefer the larger *A* partition configurations, five gravitate to the smaller *A* partition sizes.

In contrast, as shown in Figure 4, two-thirds of the dual-thread workloads highly favor the largest *A* partition cache configuration, due to the additional pressure on the caches with two threads. A noteworthy counter-example is the SMT pair `art_galgel` in which the preference of `art` for the smaller *A* partition configurations dominates the larger *A* partition preference of `galgel`. Even though `galgel` needs a large L1 *A* partition to avoid downstream accesses, the large fraction of memory accesses in `art` make it the bottleneck of the pair and the control algorithm adapts appropriately. The configuration changes of the SMT pair `galgel_bzip2` are quite extreme. This pair runs for 75% of the time using the largest 256 KB *A* partition configuration (*D3*), but once `galgel` finishes, the algorithm selects the preferred configuration for `bzip2` (*D0*).

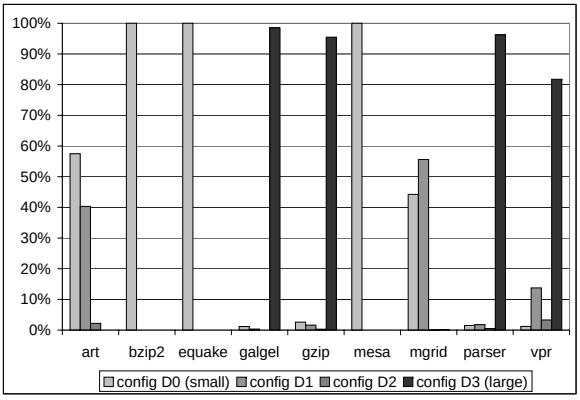

Fig. 3. Percentage of time individual SPEC2000 benchmarks spend in each load / store domain cache configuration.

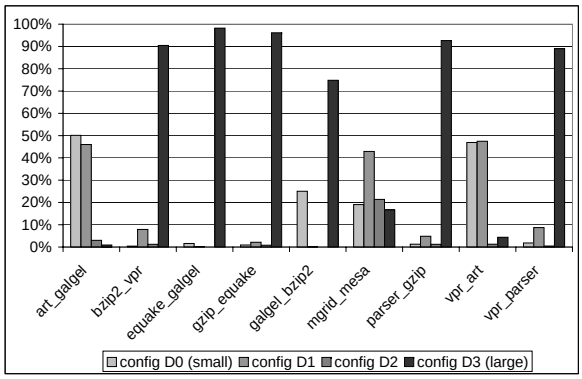

Fig. 4. Percentage of time dual-thread workloads spend in each load / store domain cache configuration.

These results demonstrate the wide ranging cache behavior observed in a simple SMT processor design that supports either one or two threads (the behavior would vary even more widely for a four-thread machine). While single-thread workloads largely favor smaller, faster L1 *A* cache partitions (there are exceptions of course), the two-thread workloads more often select larger *A* partitions for best performance. Interestingly, for some dual-thread workloads containing an application that favors a large cache, the control algorithm gravitates towards the small fast *A* partition configurations. The performance sensitivity of both applications to memory latency is the deciding factor that determines the chosen configuration. Finally, significant phase behavior is observed in several workloads that choose several different configurations during execution.

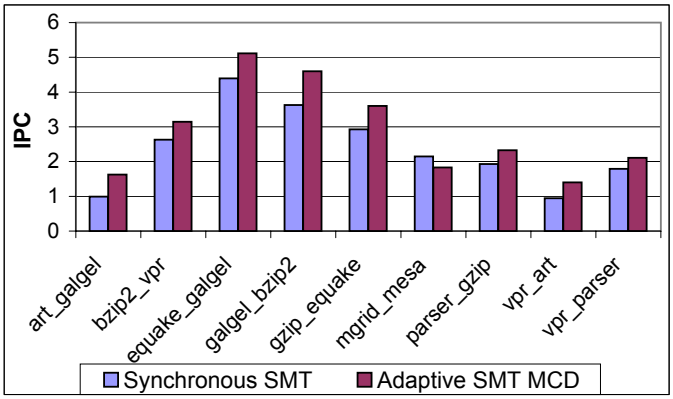

Fig. 5. IPC comparison of dual-thread workloads for the fully synchronous SMT baseline (left bars) and SMT MCD with adaptive caches (right bars).

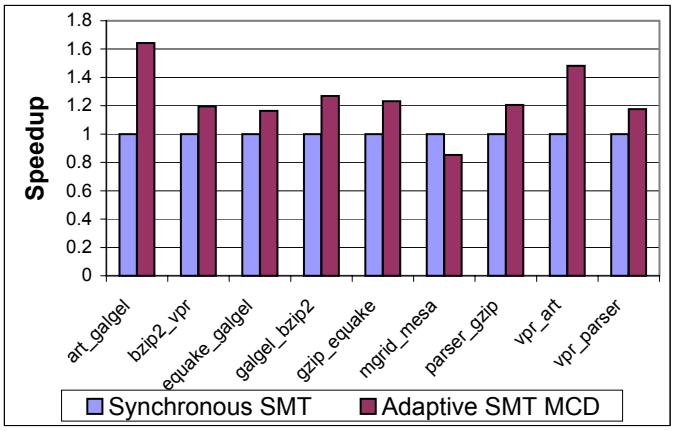

Fig. 6. Speedup of the adaptive design over the baseline with dual-thread workloads.

Figures 5, 6, 7, and 8 show how this phase varying cache behavior translates into a significant performance advantage for the adaptive SMT MCD configuration. The first figure compares the IPCs of the adaptive and fixed cache approaches for the dual-thread workloads, while the second figure graphs the corresponding speedup of the adaptive approach. Figures 7 and 8 show the same results for single-thread SPEC2000 applications. In all dual-thread cases, except `mgrid_mesa`, the adaptive organization outperforms the conventional approach, with `art_galgel` improving by 64%. The average dual-thread performance improvement is 24.7%. The overall improvement for the single SPEC2000 applications is almost as significant: an overall performance improvement of 19.2%.

The slight degradation seen in `mgrid` is due to an approximation made in the Accounting Cache cost estimation function. Significant temporal overlap in adjacent memory requests have a true delay cost that is lower than the heuristic estimates (the

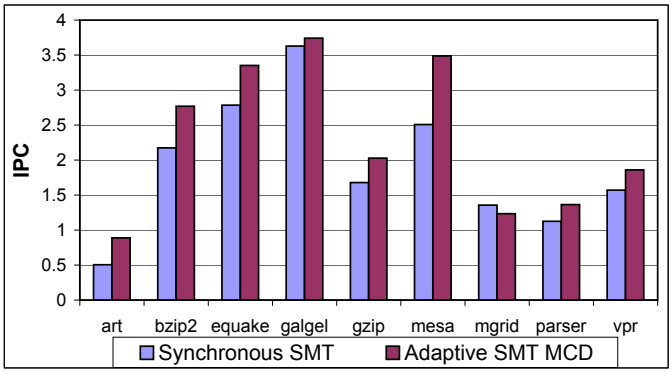

Fig. 7. IPC comparison of individual SPEC2000 applications for the fully synchronous SMT baseline (left bars) and SMT MCD with adaptive caches (right bars).

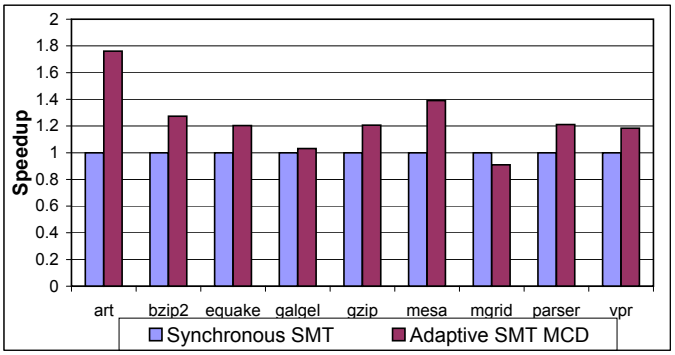

Fig. 8. Speedup of the adaptive design over the baseline with individual SPEC2000 applications.

Accounting Cache can only estimate the degree of parallelism between accesses). Including more accurate temporal overlap measurements in the cost estimation would improve the adaptive SMT MCD performance on `mgrid` and `mgrid_mesa`. Still, these performance losses pale in comparison to the significant gains observed with the other workloads using the SMT MCD organization with phase adaptive caches.

6. Related Work

Our work builds on prior efforts in both adaptive caches and GALS processor microarchitectures.

In terms of the former, Albonesi [1] first proposed the idea of trading off clock speed for hardware complexity within a single-threaded processor core, including the L1 DCache. Unlike our approach, the global clock within the synchronous processor is slowed down whenever the cache is upsized. Thus, only those applications whose performance is data-cache-bound show signficant benefit. Albonesi also proposed Selective Cache Ways in which energy is saved in single-threaded processors by disabling

ways for application working sets that fit comfortably in a subset of the cache [2]. Our approach, by contrast, explores improving performance on an SMT processor.

Balasubramonian *et al.* [3] describe a reconfigurable data cache hierarchy whose latency adjusts with the configuration. Their cache organization uses a similar A/B organization as our own, but they use a different control algorithm. Powell *et al.* [13] devise a variable latency data cache. By predicting which way in a set associative cache holds the requested data, an access can be as fast as in a direct-mapped cache.

Both of these approaches assume a single threaded processor. In addition, they require additional cache hardware support for, for instance, variable cache latency. Our approach is to decouple the clock domains and adapt the frequencies within the cache clock domains along with the cache configurations. This approach maintains a constant latency (in terms of cycles), but adds the complexity of decoupled domains and asynchronous interfaces. If GALS designs are adopted for other reasons, such as lower clock overhead, energy efficiency, or increased tolerance to process variations, then our approach can be easily integrated. Our work also has the additional contribution of exploring the varying cache demands in an SMT processor, due in part to differences in the number of running threads.

Multiple clock domain architectures [4, 11, 16] permit the frequency of each domain to be set independently of the others. Semeraro *et al.* [14] adjust frequencies automatically at run time to reduce energy in domains that are not on the critical path of the instruction stream. These approaches attempt to downsize hardware structures in single-threaded workloads for energy efficiency. In contrast, we explore the integration of adaptive caches within MCD architectures built around SMT processors to improve performance.

Our work highly leverages the Accounting Cache research of [8] and the adaptive GALS processor design of [9]. The latter effort uses the Accounting Cache within an MCD processor design much like we do. We deviate from this work in that we investigate for the first time the use of adaptive caches within SMT processors, particularly, the benefits of adapting the caches to workloads that vary widely due in part to differences in the number of running threads.

In terms of adaptive SMT processors, [6] is perhaps the closest effort to our own. In this paper, the allocation of shared hardware resources to threads is dynamically managed at runtime by the hardware. While this effort focuses on the issue queues and register files, ours is concerned with the caches. In addition, [6] uses fixed size resources and allocates a certain number of entries to each thread. Our approach is to resize the hardware resources (caches in our case) to fit varying SMT cache behavior, and we add the ability to change the size/frequency tradeoff through the use of an MCD processor.

7. Conclusions

The advent of SMT processors creates significantly more variability in the cache behavior of workloads, due in part to the differing number of threads that may be running at any given time. We propose to integrate adaptive caches within an SMT MCD processor microarchitecture. Our approach is to decouple the execution core from the adaptive cache domains, to permit full speed operation of the execution core as the cache domain frequencies are dynamically altered to match the cache configurations.

We demonstrate significant differences in the configurations chosen by the cache algorithm under single and dual-thread workloads, and we also observe cases of strong phase behavior where multiple cache configurations are chosen for non-trivial periods of execution. Overall, a 24.7% improvement is realized for dual-thread workloads by implementing adaptive cache organizations and an MCD design style within an SMT processor. In addition, a 19.2% performance improvement is observed for single-threaded applications.

References

1. D. H. Albonesi. Dynamic IPC/Clock Rate Optimization. In *25th Intl. Symp. on Computer Architecture*, June 1998.
2. D. H. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *32nd Intl. Symp. on Microarchitecture*, Nov. 1999.
3. R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *33rd Intl. Symp. on Microarchitecture*, Dec. 2000.
4. L. Bengtsson and B. Svensson. A Globally Asynchronous, Locally Synchronous SIMD Processor. In *3rd Intl. Conf. on Massively Parallel Computing Systems*, Apr. 1998.
5. D. Burger and T. Austin. The Simplescalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, U. Wisc.–Madison, June 1997.
6. F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez. Dynamically Controlled Resource Allocation in SMT Processors. In *37th Intl. Symp. on Microarchitecture*, Dec. 2004.
7. L. T. Clark. Circuit Design of XScale^{TM} Microprocessors. In *2001 Symposium on VLSI Circuits, Short Course on Physical Design for Low-Power and High-Performance Microprocessor Circuits*, June 2001.
8. S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *11th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2002.
9. S. Dropsho, G. Semeraro, D. H. Albonesi, G. Magklis, and M. L. Scott. Dynamically Trading Frequency for Complexity in a GALS Microprocessor. In *37th Intl. Symp. on Microarchitecture*, Dec. 2004.
10. M. Fleischmann. LongRun^{TM} Power Management. Technical report, Transmeta Corporation, Jan. 2001.
11. A. Iyer and D. Marculescu. Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors. In *29th Intl. Symp. on Computer Architecture*, May 2002.
12. G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. G. Dropsho. Profile-Based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In *30th Intl. Symp. on Computer Architecture*, June 2003.
13. M. Powell, A. Agrawal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing Set-Associative Cache Energy Via Selective Direct-Mapping and Way Prediction. In *34th Intl. Symp. on Microarchitecture*, Dec. 2001.
14. G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture. In *35th Intl. Symp. on Microarchitecture*, Nov. 2002.
15. G. Semeraro, D. H. Albonesi, G. Magklis, M. L. Scott, S. G. Dropsho, and S. Dwarkadas. Hiding Synchronization Delays in a GALS Processor Microarchitecture. In *10th Intl. Symp. on Asynchronous Circuits and Systems*, Apr. 2004.

16. G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *8th Intl. Symp. on High-Performance Computer Architecture*, Feb. 2002.
17. A. E. Sjogren and C. J. Myers. Interfacing Synchronous and Asynchronous Modules Within A High-Speed Pipeline. In *17th Conf. on Advanced Research in VLSI*, Sept. 1997.
18. D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *22nd Intl. Symp. on Computer Architecture*, June 1995.
19. D. Tullsen, S. Eggers, H. Levy, J. Emer, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23rd Intl. Symp. on Computer Architecture*, May 1996.
20. S. J. E. Wilton and N. P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE J. of Solid-State Circuits*, May 1996.
21. Y. Zhu, D. H. Albonesi, and A. Buyuktosunoglu. A High Performance, Energy Efficient, GALS Processor Microarchitecture with Reduced Implementation Complexity. In *Intl. Symp. on Performance Analysis of Systems and Software*, March 2005.