Adaptive Cache Memories for SMT Processors

Sonia Lopez*, Oscar Garnica†, David H. Albonesi‡,
Steven Dropsho§, Juan Lanchares† and Jose I. Hidalgo†
* Department of Computer Engineering, Rochester Institute of Technology, Rochester, NY, USA
slaeec@rit.edu
† Department of Computer Architecture, Universidad Complutense de Madrid, Madrid, Spain
{ogarnica,julandan,hidalgo}@dacya.ucm.es
‡ Computer Systems Laboratory, Cornell University, Ithaca, NY, USA
albonesi@csl.cornell.edu
§ Google Inc., Zurich, Switzerland
stevendropsho@google.com

***Abstract*—Resizable caches can trade-off capacity for access speed to dynamically match the needs of the workload. In Simultaneous Multi-Threaded (SMT) cores, the caching needs can vary greatly across the number of threads and their characteristics, offering opportunities to dynamically adjust cache resources to the workload. In this paper we propose the use of resizable caches in order to improve the performance of SMT cores, and introduce a new control algorithm that provides good results independent of the number of running threads. In workloads with a single thread, the resizable cache control algorithm should optimize for *cache miss* behavior because misses typically form the critical path. In contrast, with several independent threads running, we show that optimizing for *cache hit* behavior has more impact, since large SMT workloads have other threads to run during a cache miss. Moreover, we demonstrate that these seemingly diametrically opposed policies can be simultaneously satisfied by using the harmonic mean of the per-thread speedups as the metric to evaluate the system performance, and to smoothly and naturally adjust to the degree of multithreading.**

I. Introduction

SIMULTANEOUS Multi-Threading (SMT) [1], [2] designs enable multiple threads to simultaneously share many of the major hardware resources, thereby making use of resources that may lie partially unused when running a single thread. However, the threads sharing the resources *compete* for those resources. Depending on the needs of each thread, this competition might cause thread resource starvation; that is, one thread may monopolize the resources, not allowing the others to progress through the pipeline. This *fairness* problem has tipically been addressed by avoiding resource monopolization due to long latency operations. Static resource allocation policies [3], [4] split critical processor resources among all threads, ensuring that no thread monopolizes a resource. Dynamic resource allocation uses a pool of common resources that are shared among all active threads and a resource allocation policy that dynamically assigns resources to threads according to their requirements. In [5], [6], [7] the allocation of fetch bandwidth is carefully controlled, since this influences the sharing of resources further down the pipeline. A different approach is Dynamically Controlled Resource Allocation (DCRA) [8] that explicitly controls shared SMT resources such as the issue queue and register file.

We propose the use of *phase-adaptive reconfigurable caches* in a Globally Asynchronous, Locally Synchronous (GALS) design that, in conjunction with a cache control strategy, reduces the average latency of cache operations for the active threads. Our approach tackles the problems related with long-latency operations *at the source* by reducing the average latency of cache accesses. Our approach makes size/frequency cache tradeoffs to fit varying SMT cache behavior through the use of an MCD processor. Therefore our technique is orthogonal to previously proposed fetch and dynamic resource allocation policies.

Using a GALS design approach, we place the reconfigurable caches into an independent clock domain within which frequency can change in conjunction with the cache configuration. The configuration for any given period of execution is established by a control algorithm that makes a decision based on the cache behavior of the different active threads.

Our work builds on prior efforts in both phase-adaptive re-sizable caches and GALS processor microarchitectures in order to improve performance or save power consumption in single-threaded cores [9], [10], [11], [12], [13], [14]. Of these, only our previous work [14] addresses SMT workloads. We demonstrate that the cache control strategy of [14] is not as effective for dual and four thread SMT workloads as it is for single thread ones. If we take into account fairness (the harmonic mean of the per thread speedups [15]), the performance of the four thread workloads degrades significantly.

The intuition behind the limited scalability of this

prior strategy is that it is constructed on the assumption that cache misses are on the critical path of a thread's computation. Thus, the original strategy attempts to minimize the total access time to reduce the cost of the cache misses. However, when there are multiple active threads in an SMT core, the overall performance is affected less from cache misses because other threads can run in the shadow of the miss. In this scenario, a better cache control strategy is one that selects cache configurations that greedily maximize the near term cache access rate to favor threads that use the cache efficiently. Since the number of active threads on a given core may vary at runtime, the desired control strategy should behave effectively without regard to the degree of multithreading, both minimizing the total access time when there are few threads, and maximizing the access rate when there are many threads. We propose and evaluate such an approach in this paper.

To demonstrate the effectiveness of our proposed cache control strategy, we implement a quad-threaded core in the Simplescalar simulator. The core is optimized to run with a small, fast cache that can adjust to greater demands by dynamically *upsizing*. We adopt the *Accounting Cache* design of [13] for our resizable caches, but implement a new cache control algorithm that better balances multi-threaded needs compared to the original algorithm designed for the single threaded case.

Our technical contributions above our previous work are: (1) A detailed explanation of the behavior of our previous control algorithm (which we call *AMAT*) in architectural and mathematical terms; (2) We demonstrate that *AMAT* is not effective for dual and four thread SMT workloads; (3) An explanation of the difficulties presented with adaptive caches as the number of threads varies; (4) The introduction of the *HAMAT* algorithm; and (5) A detailed comparison, both mathematical and simulation-based, between the two algorithms. We demonstrate that the new *HAMAT* algorithm performs much more consistently than *AMAT* as the SMT load changes, and that it achieves strong speedups over a conventional fixed cache.

The rest of this paper is organized as follows. Section II discusses the adaptive microarchitecture, including the adaptive cache organizations. Section III presents the adaptive cache control algorithms. Our simulation infrastructure and benchmarks are described next, followed by our results, and finally our conclusions in Section VI.

II. Adaptive SMT MCD Microarchitecture

The *adaptive SMT Multiple Clock Domain (MCD) microarchitecture* highlighted in Fig. 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

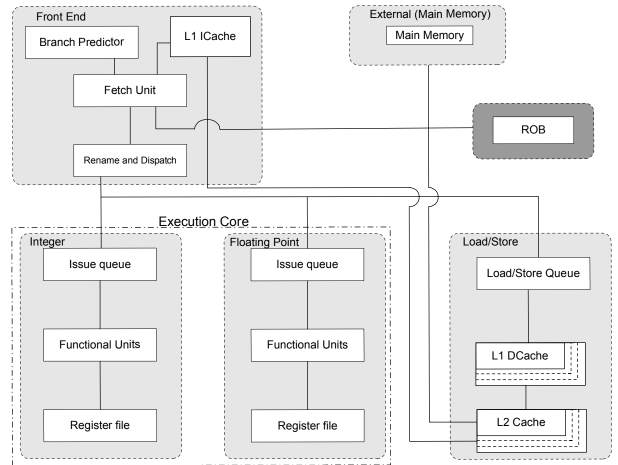


Fig. 1: Adaptive SMT MCD microarchitecture. Boxes demarcated by dotted lines (L1 DCache and L2 Cache) illustrate the adaptive capability of these structures.

(load / store queue, L1 DCache and unified L2 cache); and ROB (Reorder Buffer). The load / store domain varies its frequency based on the cache configuration. The other domains run at fixed frequency at all times and since there is little interaction between them (and thus their interface introduces negligible synchronization cost), they are effectively one fixed-frequency execution core domain. External main memory operates at the same fixed base frequency as the processing core and is also non-adaptive.

The focus of this study is the load / store domain having reconfigurable L1/L2 caches; moreover, only the L1 DCache and L2 cache of the load / store domain are adapted under the direction of control algorithms that we introduce later. This adaptive SMT MCD architecture has a base configuration that uses small cache sizes running at a high clock rate, but the caches can be upsized with a corresponding reduction in the clock rate of the load / store domain. In this study, all the non-adaptive domains – front end, integer, floating point, and main memory – run at a base frequency of 1.0 GHz. The L1 DCache and L2 cache are resized in tandem with the frequency of the load / store domain varied accordingly. The dynamic frequency control circuit within the load / store domain is a PLL clocking circuit based on industrial circuits [16], [17]. The lock time in our experiments is normally distributed with a mean time of 15 μs and a range of 10–20 μs. As in the XScale processor [16], we assume that a domain is able to continue operating through a frequency change.

Data generated in one domain and needed in another must cross a domain boundary, potentially incurring synchronization costs. Our SMT MCD simulator models synchronization circuitry based on the work of Sjogren and Myers [18]. 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. Further details on the baseline MCD model,

including a description of the inter-domain synchronization circuitry, can be found in prior papers [9], [19], [20], [21]. We have extended this model to include SMT support, the details of which are provided in Section IV.

In the load / store domain, the adaptive L1 DCache and L2 cache are up to eight-way set associative, and reconfigured by ways [13], [22]. This adaptive cache is divided into two partitions, *A* and *B*, each one containing a subset of the total ways. The number of ways contained in each partition depends on the cache configuration, and ranges from one to eight. We restrict the resizing to 1/7, 2/6, 4/4, and 8/0 ways in the *A*/*B* partitions to reduce the state space of possible configurations to four options: D0, D1, D2 and D3, respectively. 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.

Table I shows the characteristics of the *A* partition of each configuration. We use version 3.1 of the CACTI modeling tool [23] to obtain timings for all plausible cache configurations at a given size. The *Optimal* columns in Table I 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, which we found from our earlier work to be critical for good performance, 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 our evaluation, we account for the additional access latency incurred due to this sub-optimal sub-banking for the configurations with larger A partitions (D1, D2, D3).

Because of its design, a cache with a small *A* partition runs at a higher frequency than one with a larger *A* partition and 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.

III. Phase Adaptive Cache Control Algorithms

To control the reconfigurable caches, we employ an Accounting Cache design similar to that in [13] but tailored to an SMT processor. With this approach, since the smaller configurations are proper subsets of the larger ones, a single set of counters can be used to collect statistics (i.e., hits and misses) that identify the performance of every possible cache configuration during the monitored period. This permits the calculation of the number of hits and misses that *would have occurred* over that span of time for any of the possible configurations.

As described in detail in [13], 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 optimized the interval that just ended (i.e., the assumption is the next interval will be similar).

A. Minimizing Aggregate Cache Access Delay

The original Accounting Cache design uses a phase adaptive control algorithm that configures the cache to minimize the *total access delay* for the set of references made over the interval [13]. Implicit in this algorithm is the treatment of the set of references as a unit of work that must be completed before the next unit of work can begin. Under this model of the workload, minimizing the total delay of each unit of work results in minimizing total execution time. This model precisely describes the behavior for a single-threaded application where control flow and data dependencies limit overall ILP, and where long memory stalls are on the critical path and stall the pipeline.

Dividing the total access delay by the number of references we arrive at the arithmetic mean of the cache access delay for a given configuration, which we term the *Arithmetic Mean Access Time* ($AMAT^i$) of the cache at configuration *i*. Let there be a set of N references with reference r requiring t_r^i time at configuration *i* (cache configuration dependent). Then $AMAT^i$ is defined as:

$$AMAT^i = \frac{1}{N} \sum_{r \in Refs} t_r^i \tag{1}$$

The total access time at configuration *i* can be expressed as,

$$\sum_r t_r^i = hits_A^i * cost_A^i + hits_B^i * cost_B^i + \\ + misses * cost_{misses}$$

where $hits_A^i$ and $hits_B^i$ are the number of hits on the *A* and *B* partitions, $cost_A^i$ and $cost_B^i$ are the cost (in time, normalized to the frequency of the load / store domain at configuration *i*) to access the *A* and *B* partitions, and *misses* and $cost_{misses}$ are the number of, and the cost of,

TABLE I: Adaptive L1 DCache and L2 Cache configurations. The *A Size* and *A Assoc* columns refer to the size and associativity of the *A* partition selected by the cache control algorithm, as discussed in Section III. The column *Adapt* provides the number of sub-banks per way for the adaptive cache, while *Optimal* gives the number that produces the fastest access time at that size and associativity.

	L1 DCache		**Sub-banks/Way**		**L2 Cache**		**Sub-banks/Way**	
Configuration	**A Size**	**A Assoc**	**Adapt**	**Optimal**	**A Size**	**A 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

misses to both partitions. Hence, measuring the number of hits and misses to the *A* and *B* partitions for all possible cache configurations permits the estimation of *AMAT* for all possible configurations.

The control algorithm circuitry to maximize the total access delay (which avoids doing the division by N) requires 5K equivalent gates. A complete reconfiguration decision requires approximately 32 cycles, based on binary addition trees and the generation of a single partial product per cycle [12].

B. Maximizing Cache Access Rate

With SMT, if one thread stalls due to cache misses, other threads may likely make use of the available resources of the processor. Thus, this algorithm that minimizes the total delay of the set of accesses over an interval is, in fact, prioritizing for the slower, least cache-efficient threads to the detriment of the faster, more cache-efficient threads.

A more circumspect control algorithm should take into account the degree of SMT in the system and optimize for total access delay when the degree of SMT is low (e.g., one or two threads) but exploit thread-level parallelism to maximize *access rate* when the degree of SMT is high.

First, we define needed terms. Let n_j be the number of references of the thread *j* and $t^i_{r_j}$ be the required time for each reference of the thread *j* under configuration *i*. Then $N = \sum_j n_j$ and the *Arithmetic Mean Access Time* of the thread *j* under configuration *i*, $AMAT^i_j$, is

$$AMAT^i_j = \frac{1}{n_j} \sum_{r_j} t^i_{r_j} \tag{2}$$

It can easily be shown that

$$\begin{aligned} AMAT^i = \frac{1}{N} \sum_{r} t^i_r = \frac{1}{N} \sum_{j} \sum_{r_j} t^i_{r_j} = \\ + \frac{1}{N} \sum_{j} n_j \cdot AMAT^i_j \end{aligned} \tag{3}$$

where $AMAT^i_j$ can be calculated as

$$\begin{aligned} AMAT^i_j = hits^i_{jA} * cost^i_A + hits^i_{jB} * cost^i_B + \\ + misses_j * cost_{misses} \end{aligned}$$

In this case, $hits^i_{jA}$ and $hits^i_{jB}$ are the number of hits of thread *j* on the *A* and *B* partitions, and $misses_j$ is the number of misses of thread *j*.

The *Arithmetic Mean Access Rate* for a thread j of the cache at configuration *i*, $AMAR^i_j$, is

$$AMAR^i_j = \frac{1}{AMAT^i_j} \tag{4}$$

The *Average Access Rate* of a cache at configuration *i* (AAR^i) is the number of memory access per unit of time and thread, weighted according to the number of accesses per thread:

$$AAR^i = \frac{1}{N} \sum_{j} \frac{n_j}{AMAT^i_j} \tag{5}$$

Those threads with a higher number of accesses should have a higher weight in AAR^i. Hence, the *Average Access Rate* for a cache configuration *i* is the weighted access rate across all threads for that configuration. In this way, the pattern access of the set of threads is taken into account.

The cache configuration chosen is the one with the maximum *AAR*. In other words, the cache configuration is the one that maximizes the access rate for the actual access pattern. Equivalently, if we choose to minimize the reciprocal of the *AAR* then we have the *Harmonic Mean* of the per thread weighted arithmetic mean access times in cache configuration *i*, $HAMAT^i$:

$$HAMAT^i = \frac{1}{\frac{1}{N} \sum_{j \in Threads} \frac{n_j}{AMAT^i_j}} \tag{6}$$

Interestingly, the harmonic mean is naturally related to maximizing the access rate. In the case of having only one thread, $N = n_0$, $AMAT^i = AMAT^i_0$, and $HAMAT^i = AMAT^i$.

To implement this algorithm as per Eq. (5) (again ignoring the division by N), per thread accesses must be captured by the Accounting Cache which requires customization of the Accounting Cache design to the SMT environment. To do so, the total MRU state of the original design is increased by the number of simultaneous threads that can run. For a four thread SMT processor, implementing per-thread counters increases the overhead from 0.3% to 1.2% of the total cache [12]. In addition, four times the original number of calculations are needed to generate the arithmetic mean delay for

each thread. However, adding four additional circuits at 5K gates apiece maintains the total calculation time of 32 cycles per configuration decision for the arithmetic mean values. The four required reciprocal calculations can leverage the capability of the arithmetic unit via PALcode-type mechanisms.

On each cache access, we update the MRU and miss counters and after 15K instructions we select the cache configuration with the lowest value of *HAMAT*.

IV. Evaluation Methodology

The simulation environment is based on the SimpleScalar toolset [24] with MCD processor [9] and SMT extensions. The time management code has been rewritten to emulate separate clocks for each domain, complete with jitter, and to account for synchronization delays on all cross-domain communication.

The SMT processor extensions include independent program counters for each thread; thread IDs for queues, caches and predictor history tables; and per-thread ROBs. Our fetch policy is *ICOUNT2.8* from [2], i.e., up to eight instructions are fetched from each of up to two threads per cycle. Table II contains a summary of the simulation architectural parameters. These have been chosen to match the characteristics of the Alpha 21264, but with additional resources for four threads.

TABLE II: Architectural parameters for simulated processor

Processor Configuration
Fetch queue : 16 entries
Issue queue : 32 Int, 32 FP
Load/store queue : 32 entries
Physical register file (per th.): 100 integer, 100 FP
Reorder buffer (per th.): 256 entries
Decode, issue, and retire widths: 8, 11, and 24 instructions
Integer ALUs: 6 + 1 mult/div unit
FP ALUs: 4 + 1 mult/div/sqrt unit
Number of threads fetch: 2
Fetch (per th.), Issue and Commit width: 8, 11, 24 instr.
Branch Predictor Configuration
Combined gshare & 2-level PAg
Level 1 1024 entries, history 10
hspace1emLevel 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
Memory Configuration
Static L1 Data Cache: 128KB, 4-way set associative
Static Instruction Cache: 32KB, 2-way set associative
Static L2 Unified Cache: 1MB, 4-way set associative
Static L1 Data Cache latency: 2 ns
Static Instruction Cache latency: 1 ns
Static L2 Unified Cache latency: 15 ns
Main Memory latency: 80 ns (1st access), 2 ns (subsequent)

Table III provides timing parameters for adaptive L1 and L2 caches, as well as the clock domain frequencies for each configuration. The four configurations (D0-D3) of the load / store domain are shown. Listed for each configuration are the frequency of the domain and the cache access times, also known as latency (in cycles) at that frequency. The first access time is for *A* partition accesses and the second for *B* partition access. For comparison, the baseline 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 15 (pipelined) cycles. Note that larger adaptive cache configurations have over double the access latency (in ns) of the baseline design. Thus, the control algorithms only upsize the *A* partition if the greater capacity reduces misses sufficiently to compensate for this extra delay on every access.

TABLE III: Cache latencies (in cycles) and domain frequency for each cache configuration

Load/Store Domain				
Configuration	D0	D1	D2	D3
Frequency (GHz)	1.59	1.00	0.76	0.44
L1DCache Lat.(A/B)	2/7	2/5	2/2	2/-
L2Cache Lat.(A/B)	12/42	12/27	12/12	12/-

Our workloads consists of combinations of fifteen programs from the SPEC2000 suite. Table IV specifies the individual benchmarks along with the instruction windows and input data sets. We combine these individual programs into fourteen dual thread and eleven quad thread workloads, shown in Table V and Table VI.

In this paper, we measure performance improvements with respect to a baseline conventional (non-adaptive) fully synchronous processor whose architectural parameters have been chosen based on simulations we have conducted. These simulations used the baseline non-adaptive configuration in [12] as a starting point to select the configuration that achieved best performance across all workloads. This best conventional processor has a 128 KB four-way set associative L1 DCache with a two-cycle latency, and a 1024 KB four-way set associative L2 cache with a 15 cycle latency.

TABLE IV: SPEC2000 benchmarks, input datasets used, and simulation windows

Benchmark	Datasets	Simulation window
Integer		
bzip2	source 58	100M–600M
crafty	ref	1000M–1500M
cc1	166.i	2000M–2500M
gzip	source 60	100M–600M
mcf	ref	1000M–1500M
parser	ref	100M–600M
twolf	ref	1000M–1500M
vpr	ref	190M–690M
Floating-Point		
art	ref	300M–800M
equake	ref	100M–600M
galgel	ref	100M–600M
lucas	ref	100M–600M
mesa	ref	100M–600M
mgrid	ref	100M–600M
swim	ref	1000M–1500M

TABLE V: SPEC2000 dual-thread workloads

Integer
vpr-bzip2, cc1-bzip2 crafty-vpr
Floating-Point
equake-art, equake-galgel lucas-galgel
Combined Integer and Floating-Point
bzip2-art, galgel-bzip2 gzip-galgel, mcf-lucas, mesa-twolf, mgrid-bzip2 vpr-art, vpr-swim

TABLE VI: SPEC2000 four-thread workloads

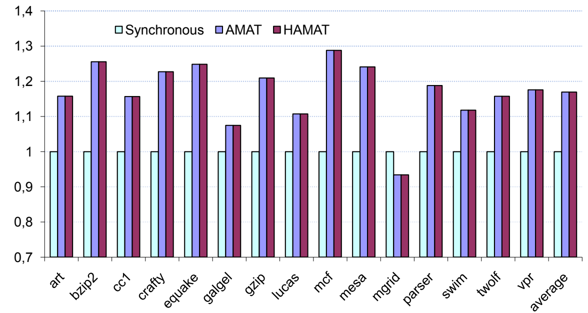
Integer
bzip2-cc1-gzip-mcf twolf-mcf-vpr-crafty
Floating-Point
art-equake-galgel-lucas mesa-mgrid-swim-art mesa-equake-swim-art
Combined Integer and Floating-Point
art-bzip2-equake-cc1 twolf-mesa-mgrid-swim, cc1-bzip2-gzip-equake, swim-vpr-art-crafty galgel-gzip-mcf-lucas

V. Performance Results

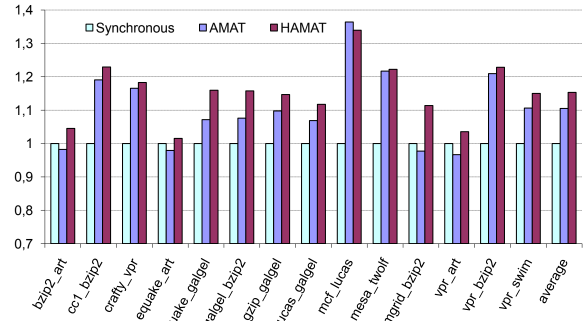
In this section, we compare the performance of the SMT MCD microarchitecture with adaptive caches with that of the baseline fully synchronous design described in Section IV, for single, dual, and quad thread workloads. We compare the performance of the cache control algorithms, *AMAT* and *HAMAT*, described in Section III. For the dual and quad thread workloads, the reported speedup results use the harmonic mean of the per thread speedups in the mix. This metric rewards balancing speedup improvements across all threads in the mix and penalizes those cases in which one thread monopolizes resources [15], thereby taking into account the fairness of the execution.

Fig. 2 shows the speedups from resizing the caches for the single, dual and quad thread workloads. The three bars show the baseline synchronous processor (speedup of 1.0 by definition), the *AMAT* algorithm, and the *HAMAT* algorithm. For the single thread workloads, the two control algorithms behave the same, so their performance is identical, as shown in Fig. 2(a) and proven in Section III. The adaptive caches outperform the synchronous processor in all applications except `mgrid`, and achieve an average performance improvement of 16.9% over all applications. The low performance of `mgrid` is explained in more detail at the end of this section.

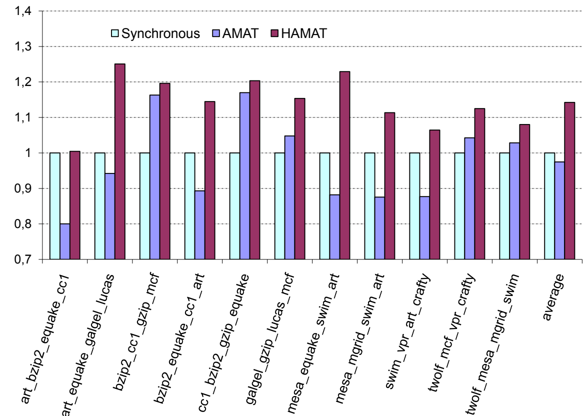
The dual thread runs in Fig. 2(b) show the *AMAT*



(a) Single thread workload



(b) Dual thread workload



(c) Quad thread workload

Fig. 2: Speedups using cache reconfiguration *AMAT* and *HAMAT* control algorithms relative to the best synchronous baseline processor.

algorithm performing below that of the baseline in a number of cases, while *HAMAT* outperforms both the synchronous baseline processor and the *AMAT* algorithm in all instances. In particular, *AMAT* has difficulty when threads are combined with `art` (`bzip2_art`, `equake_art`, and `vpr_art`). The benchmark `art` has a large number of cache misses which cause the *AMAT* algorithm to upsize the caches. While upsizing effectively reduces the total run-time when `art` is run alone (Fig. 2(a)), with two threads, the slower access time of a larger cache on every access significantly degrades the thread paired with `art`, to the detriment of the overall speedup. *HAMAT* better balances the needs of the two threads and performs as well or better than *AMAT* on every mix. We illustrate this in Table VII, which shows the speedup for each thread in each

TABLE VII: Per thread speedup for the dual thread workloads.

	AMAT Alg.		HAMAT Alg.	
	Th. 1	Th. 2	Th. 1	Th. 2
bzip2_art	0.9228	1.0501	1.0699	1.0216
cc1_bzip2	1.1739	1.2082	1.2079	1.2516
crafty_vpr	1.2040	1.1292	1.2236	1.1445
equake_art	0.8910	1.0869	1.0242	1.0068
equake_galgel	1.0638	1.0793	1.1501	1.1695
galgel_bzip2	1.0083	1.1535	1.0788	1.2490
gzip_galgel	1.1344	1.0629	1.1936	1.1035
lucas_galgel	1.0584	1.0794	1.0876	1.1491
mcf_lucas	1.3802	1.3486	1.3802	1.30098
mesa_twolf	1.2246	1.2095	1.2284	1.2164
mgrid_bzip2	0.9481	1.0085	1.0746	1.1547
vpr_art	0.8891	1.0591	1.0538	1.0175
vpr_bzip2	1.1908	1.2284	1.2092	1.2481
vpr_swim	1.1462	1.0688	1.2047	1.002

TABLE VIII: Per thread speedup for the four thread workloads.

	AMAT Alg.				HAMAT Alg.			
	Th. 1	Th. 2	Th. 3	Th. 4	Th. 1	Th. 2	Th. 3	Th. 4
art_bzip2_equake_cc1	1.1375	0.7745	0.9309	0.5697	1.2983	1.0986	1.3271	0.6457
art_equake_galgel_lucas	0.9329	0.9063	0.7211	1.4612	1.0610	1.3175	0.9854	2.0736
bzip2_cc1_gzip_mcf	1.1804	1.1893	1.1869	1.09964	1.2177	1.2104	1.2249	1.1358
bzip2_equake_cc1_art	0.8363	0.7696	1.0097	1.0065	1.1876	1.0982	1.1456	1.1507
cc1_bzip2_gzip_equake	1.1964	1.1708	1.1828	1.1314	1.2160	1.2089	1.2206	1.1692
galgel_gzip_lucas_mcf	1.0472	1.1319	1.0029	1.0182	1.1554	1.2412	1.0968	1.1302
mesa_equake_swim_art	0.8465	0.8486	0.8315	1.0264	1.2751	1.3187	1.1559	1.1808
mesa_mgrid_swim_art	0.7994	0.9044	0.8049	1.0305	1.1301	1.1264	1.0455	1.1574
swim_vpr_art_crafty	0.8111	0.8214	1.0102	0.8913	1.0518	1.0889	1.1664	0.9693
twolf_mcf_vpr_crafty	1.0492	1.0289	1.0227	1.0708	1.0905	1.0718	1.0630	1.3070
twolf_mesa_mgrid_swim	1.0615	1.0140	1.0644	0.9779	1.1182	1.0851	1.1214	1.0034

dual-threaded workload for both algorithms. We shade those cases with per-thread speedup below one, i.e., that thread performs worse than the same thread in the same workload executed on the baseline processor. For *AMAT*, those threads paired with `art` and `mgrid` degrade in performance relative to the baseline. However, all threads experience a performance improvement using HAMAT since it balances the needs of both threads and it does not unduly benefit one at the expense of the other. Despite the three cases of degradation, the average performance improvement for *AMAT* is significant (11.2%), yet *HAMAT* performs much better, reaching 16.2% on average.

The differences between the two algorithms are accentuated in the quad thread workloads shown in Fig. 2(c). Under a higher SMT load, *AMAT* performs worse than the synchronous baseline processor in many cases. On the other hand, *HAMAT* outperforms both the synchronous baseline processor and the *AMAT* algorithm for all workloads, averaging 14.2% over the set of quad thread workloads. From Table VIII, we observe many shaded cells for the *AMAT* algorithm, one for every thread that performs worse than its counterpart in the conventional baseline, and far fewer for *HAMAT*. Just three threads perform worse than the baseline for *HAMAT*, two marginally worse. In the one case where performance degrades non-trivially, overall performance still improves due to the speedups for the other threads.

The differences in cache configuration decisions between the two algorithms are shown in Fig. 3 as the overall *average A partition associativity* across the runs of single, dual, and quad thread workloads. Again, the algorithms are identical for single thread workloads, as are the configuration decisions. However, with dual and quad thread workloads, *HAMAT* tends to choose smaller cache configuration (that is, those with lower associativity). The smaller A partition cache configuration runs at a higher clock rate which improves the access rate to the cache for cache-efficient threads, possibly at the expense of a thread that would normally select a larger cache if run as a single thread (e.g., `art`).

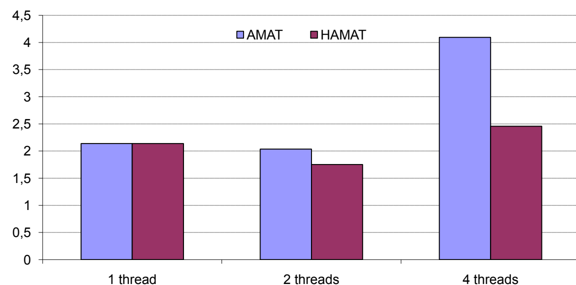


Fig. 3: Average A partition associativity across all benchmarks using *AMAT* and *HAMAT* algorithms.

Fig. 4 summarizes the differences in speedup between the two algorithms. The performance of *AMAT* falls sharply beyond two threads, while *HAMAT* balances per thread needs with overall throughput to maintain its benefits nearly independent of the number of threads.

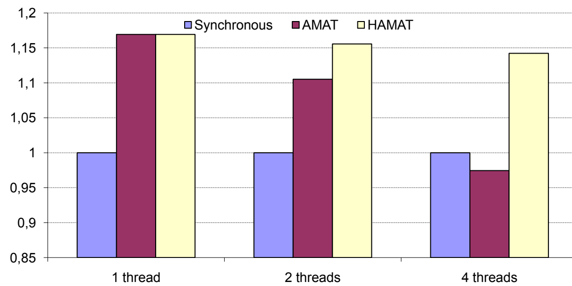


Fig. 4: Average speedup across all benchmarks usinusingg *AMAT* and *HAMAT* over the best synchronous baseline processor.

VI. Conclusions

Compared to a single-threaded processor, an SMT processor increases the variability in cache demand due to varying numbers of threads, in addition to phases within the threads themselves. Adaptive caches offer an attractive solution for dynamically tuning the configuration to the instantaneous needs of the workload and to ensure a fair competition for the resources.

We show that the control of adaptive caches under a heavy SMT workload differs significantly than when running only one or two threads. Under light SMT loads, caches should be configured to minimize the impact of misses through judicious upsizing for additional capacity, whereas under heavy SMT loads, the rate of access to the cache should be maximized through judicious downsizing for additional speed. Our hybrid algorithm gives consistent performance improvements over a wide range of SMT workloads, specifically, 17%, 16%, and 14.2% on one, two, and four thread workloads, respectively, over the best fixed-size cache design. The algorithm gives much more consistent results than the arithmetic mean algorithm for varying numbers of threads. We also show that the new algorithm treats all the running threads in a more fair manner, attending to the needs of the entire workload.

Acknowledgment

This work has been supported by Spanish Government grants TIN2008-00508 and MEC Consolider Ingenio CSD00C-07-20811 of the Spanish Council of Science and Technology.

References

[1] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *Proc. 22nd Int'l Symp. on Computer Architecture*, Jun. 1998, pp. 533–544.

[2] D. M. Tullsen, S. J. Eggers, H. M. Levy, J. S. Emer, J. L. Lo, and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," in *Proc. 23rd Int'l Symp. on Computer Architecture*, May 1996, pp. 191–202.

[3] D. T. Marr, F. Binns, D. Hill, G.Hinton, D. Koufaty, J. Miller, and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," in *Intel Technology Journal*, vol. 6, no. 1, Feb. 2002, pp. 1–12.

[4] S. Raasch and S. Reinhardt, "The Impact of Resource Partitioning on SMT Processors," in *Proc. 12th Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2003, pp. 15–25.

[5] D. M. Tullsen and J. A. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreading Processor," in *Proc. 34th Ann. IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 2001, pp. 318–327.

[6] A. El-Moursy and D. H. Albonesi, "Front-End Policies for Improved Issue Efficiency in SMT Processors," in *Proc. 8th Int'l Symp. on High-Performance Computer Architecture*, Feb. 2003, pp. 31–42.

[7] F. J. Cazorla, E. Fernández, A. Ramírez, and M. Valero, "Improving Memory Latency Aware Fetch Policies for SMT Processors," in *Proc. 5th Int'l Symp. on High-Performance Computing*, Oct. 2003, pp. 70–85.

[8] F. J. Cazorla, A. Ramírez, M. Valero, and E. Fernández, "Dynamically Controlled Resource Allocation in SMT Processors," in *Proc. 37th Ann. IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 2004, pp. 171–182.

[9] 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 *Proc. 8th Int'l Symp. on High-Performance Computer Architecture*, Feb. 2002, pp. 29–40.

[10] A. Iyer and D. Marculescu, "Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors," in *29th Int'l. Symp. on Computer Architecture*, May 2002.

[11] D. H. Albonesi, "Dynamic IPC/Clock Rate Optimization," in *Proc. 25th Int'l Symp. on Computer Architecture*, Jun. 1998, pp. 282–292.

[12] S. Dropsho, G. Semeraro, D. H. Albonesi, G. Magklis, and M. L. Scott, "Dynamically Trading Frequency for Complexity in a GALS Microprocessor," in *Proc. 37th Ann. IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 2004, pp. 157–186.

[13] 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 *Proc. 11th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2002, pp. 141–152.

[14] S. Lopez, S. Dropsho, D. H. Albonesi, O. Garnica, and J. Lanchares, "Dynamic Capacity-Speed Tradeoffs in SMT Processor Caches," in *Proc. 4th Int'l Conf. on High Performance and Embedded Architectures and Compilers*, Jan. 2007, pp. 136–150.

[15] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Throughput and Fairness in SMT Processors," in *Proc. Int'l. Symp. on Performance Analysis of Systems and Software*, Jan. 2001, pp. 164–171.

[16] L. T. Clark, "Circuit Design of XScale^{TM} Microprocessors," in *Proc. Symp. on VLSI Circuits, Short Course on Physical Design for Low-Power and High-Performance Microprocessor Circuits*, Jun. 2001.

[17] M. Fleischmann, "LongRun^{TM} Power Management," Transmeta Corporation, Tech. Rep., Jan. 2001.

[18] A. E. Sjogren and C. J. Myers, "Interfacing Synchronous and Asynchronous Modules Within A High-Speed Pipeline," in *Proc. 17th Conf. on Advanced Research in VLSI*, Sep. 1997, pp. 47–61.

[19] 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 *Proc. 10th Int'l Symp. on Asynchronous Circuits and Systems*, Apr. 2004, pp. 159–169.

[20] 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 *Proc. 30th Int'l Symp. on Computer Architecture*, Jun. 2003, pp. 14–25.

[21] 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 *Proc. 35th Ann. IEEE/ACM Int'l Symp. on Microarchitecture*, Nov. 2002, pp. 356–367.

[22] D. H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," in *Proc. 32nd Ann. IEEE/ACM Int'l Symp. on Microarchitecture*, Nov. 1999, pp. 248–259.

[23] S. J. E. Wilton and N. P. Jouppi, "CACTI: An Enhanced Cache Access and Cycle Time Model," in *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, May 1996, pp. 677–688.

[24] D. Burger and T. Austin, "The Simplescalar Tool Set, Version 2.0," U. Wisc.–Madison, Tech. Rep. CS-TR-97-1342, June 1997.