Fetch Unit Design for Scalable Simultaneous Multithreading (ScSMT)

JUAN C. MOURE, DOLORES I. REXACHS, EMILIO LUQUE

Computer Architecture and Operating Systems Group,
University Autónoma of Barcelona. Barcelona, 08193. SPAIN
E-mail: (juancarlos.moure, dolores.rexachs)@uab.es, e.luque@cc.uab.es

Abstract. Continuous IC process enhancements make possible to integrate on a single chip the resources required for simultaneously executing multiple control flows or threads, exploiting different levels of thread-level parallelism: application-, function-, and loop-level. Scalable simultaneous multithreading combines static and dynamic mechanisms to assemble a complexity-effective design that provides high instruction per cycle rates without sacrificing cycle time nor single-thread performance.

This paper addresses the design of the fetch unit for a high-performance, scalable, simultaneous multithreaded processor. We present the detailed microarchitecture of a clustered and reconfigurable fetch unit based on an existing single-thread fetch unit. In order to minimize the occurrence of fetch hazards, the fetch unit dynamically adapts to the available thread-level parallelism and to the fetch characteristics of the active threads, working as a single shared unit or as two separate clusters. It combines static and dynamic methods in a complexity-efficient way. The design is supported by a simulation-based analysis of different instruction cache and branch target buffer configurations on the context of a multithreaded execution workload. Average reductions on the miss rates between 30% and 60% and peak reductions greater than 200% are obtained.

1 Introduction

IC process enhancements allow integrating thousands of million transistors on a single chip. This trend has important implications in the design of high-performance parallel systems, since it makes possible to simultaneously execute multiple control flows or threads into a single chip, exploiting the thread-level parallelism (TLP) into the workload.

Two different approaches for exploiting TLP have arisen. A Chip-Multiprocessor (CMP) [4] is a static, highly distributed design that exploits moderate amounts of the instruction-level parallelism (ILP) on a fixed number of threads. On the contrary, a Simultaneous Multithreaded (SMT) processor [10,11] uses dynamic mechanisms and policies to exploit all the available ILP of a varying number of threads. SMT, by using both TLP and ILP interchangeably, can exploit the on-chip computation and memory bandwidth more effectively and provide larger instruction per cycle (IPC) rates, even on single-thread workloads. However, it may penalize clock frequency due to its complex, tightly coupled design. Scalable simultaneous multithreading (ScSMT) [7] combines static (CMP) and dynamic (SMT) mechanisms to assemble a complexity-effective [8] design that provides high IPC rates without sacrificing cycle time nor single-thread performance.

Figure 1 outlines the different parts of the ScSMT processor. Each thread has access to a subset of the processor resources, the thread's cluster, as in a CMP, but a thread may also access some of its neighbor's resources when the owner thread does not need them. In this way, fast intra-cluster execution, which is expected to be very frequent, is not sacrificed by infrequent, slow inter-cluster execution. This paper addresses the design of the ScSMT processor's fetch unit.

Instruction cache (i-cache) misses stall the fetch flow of a program for several cycles, reducing the rate of instruction delivery to the execution core. Branch direction and target prediction allows fetching instructions following a branch without waiting for the condition and/or the target address to be computed. The Branch Target Buffer (BTB) is a simple branch target prediction mechanism that keeps the most recent target for each branch and replaces the predicted target on mispredictions [6]. Also, conditional branch predictors use branch history tables and prediction counters that are trained to predict conditional branch directions. A conditional branch or BTB misprediction makes the fetch unit to supply invalid instructions for several cycles, reducing useful fetch bandwidth. Finally, taken branches make the dynamic instruction sequence, which may exist in the cache, to be broken into several fetch blocks that must be fetched sequentially. This problem of noncontiguous instruction fetching introduces holes in the fetch flow, also decreasing fetch bandwidth.

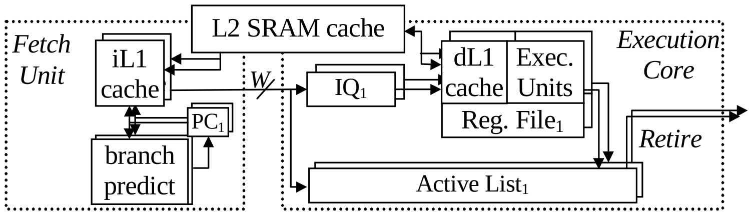


Figure 1. Scalable Simultaneous Multithreaded (ScSMT) processor

When we try to augment the issue rate of a superscalar processor the total penalty of fetch hazards increases more than linearly, making the fetch unit to become a performance bottleneck. Simultaneously fetching from several threads, however, permits scaling the penalty of fetch hazards and achieving a higher instruction fetch rate. A pure distributed design (CMP) partitions fetch resources among *t* threads, scaling fetch penalties by *t*, but since the total amount of memory devoted to the cache and prediction tables is also partitioned, the rate of fetch hazards increases. On the contrary, dynamically sharing the i-cache and prediction tables makes a more efficient use of the resources and often reduces fetch stalls, although sometimes excessive negative thread interference may provide even worse results than a static partition.

This paper analyzes several fetch unit topologies to minimize the increase of i-cache and BTB miss rate generated by running several simultaneous threads. Although multithreading may hide a large part of the miss penalties, the increase in L1 cache-miss traffic due to the combination of higher i-cache miss rates and higher execution rates may saturate the processor's L1-L2 bandwidth, limiting the potential performance gain of multithreading. Also, since we do not know a misprediction has occurred until the correct target is already available, it is not easy to hide the entire target misprediction penalty, especially for indirect branches, unless we have lots of ready threads.

We study the effect of thread interference on the cache and BTB, and analyze several organizations trying to minimize cache and BTB hazards. Based on these results, we propose a reconfigurable, two-cluster fetch unit for the ScSMT processor based on the Alpha 21264 processor [5]. The fetch unit adapts dynamically to the number and characteristics of the executing threads, working as a single shared unit or as two separate clusters in order to minimize the occurrence of fetch hazards. Average reductions on the cache miss rate between 30% and 60% and peak reductions greater than 200% are obtained. Similar results are achieved for branch target prediction.

Section 2 describes and discusses a detailed proposal of the ScSMT fetch unit. Section 3 presents related work. Section 4 describes the methodology used in our simulation-based study. Sections 5 and 6 analyze the effect of thread interference on the cache and BTB, respectively, providing the qualitative and quantitative results supporting the design presented in section 2 and the thread allocation policy proposed in section 7. We conclude summarizing the paper contribution.

2 Designing a ScSMT Fetch Unit Microarchitecture

In this section we describe a detailed proposal of the ScSMT fetch unit based on the Alpha 21264 microarchitecture [5]. Section 2.1 describes the Alpha fetch unit and outlines the major block-level changes made to accommodate multithreading. Our design is supported by qualitative arguments, leaving the discussion of quantitative decisions to sections 5 and 6, after we present our simulation results. Section 2.2 discusses in detail the new scheduling issues introduced by multithreading. Section 2.3 analyzes the area and cycle time cost of our proposal.

2.1 ScSMT Fetch Unit Description

The Alpha processor has an issue width of 4 integer instructions per cycle. Its 64K instruction cache is logically organized as two ways of 512 cache lines per way. However, it is physically organized as 4096 fetch blocks (four blocks per line), each containing four sequential instructions and including a field predicting the next fetch block and way to be fetched (i.e., the BTB is integrated with the cache). Each cycle, the predicted fetch block is accessed and, in case of a way or next-block misprediction, a second access is required. This pseudo-associative design combines the low miss rate of a truly 2-way associative cache/BTB with the fast access time of a direct-mapped organization.

A two-bit hysteresis counter is attached to each next-block field to improve the BTB prediction accuracy for indirect branch targets. The target of subroutine returns, however, is predicted using a Return Address Stack, RAS, which stores the last *n* return addresses [9]. Finally, the conditional branch predictor combines a 1K-entry local history table indexing a 1K-entry local prediction table with a 4K-entry global history table, using a 4K-entry selection table.

Figure 2 shows a block-level description of our ScSMT fetch unit. Our experimental results (sections 5 and 6) show that more than 8 threads can increase cache miss rate too much, so we assume a maximum of 8 threads. This value is enough to reach maximum performance since a single thread has typical IPC rates between 0.5 and 3 and the maximum issue rate of the processor is 4. In order to minimize cache/BTB hazards on multithreaded execution, we divide the cache/BTB into two clusters that can be reconfigured to work as a single cache/BTB. The unified configuration minimizes cache/BTB miss rate for single-thread execution, since it allows exploiting all the available fetch resources. Sometimes, this organization also minimizes miss rates for dual-thread execution. The two-cluster configuration can hold up to four different threads per cluster. Results from sections 5 and 6 will show that a dual cache/BTB minimizes miss hazards for many combinations of two threads and for almost all combination of 3-6 threads. Adding four-cluster reconfiguration capability would reduce hazard rates when executing 7 or 8 simultaneous threads, but the area and cycle time cost (studied in section 2.2) would hamper overall performance.

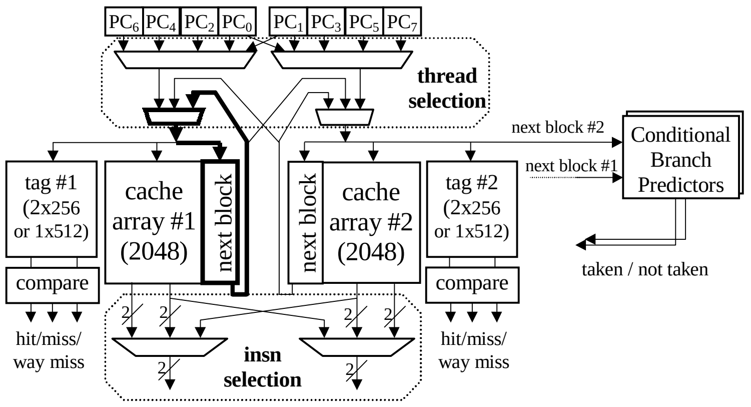


Figure 2. ScSMT fetch unit microarchitecture based on the Alpha 21264 processor

Reconfiguration simply requires changing the interpretation of cache tags and modifying the generation of cache addresses and tag comparisons. Therefore, cache contents are lost each time the fetch unit is reconfigured. For example, one thread addressing the entire cache uses a full 12-bit block index obtained either from the PC or from the next-block field of a previous access. The block index is sent to the cluster indicated by its higher order bit (it also indicates the cache way) and the two 512-entry tag tables are used in parallel for checking way and cache misses.

If the cache is logically split, then only 11 bits are used for accessing the cluster assigned to the thread, and the higher order bit indicates the logical way accessed. The tag table of the assigned cluster is used as two 256-entry tables, and provides two tags for checking way/cache misses.

We statically partition the conditional branch prediction tables by half and provide two independent units supporting a prediction rate of two conditional branches per cycle (one per cluster). Since very few RAS entries per thread (16 or less) are enough to neglect return target mispredictions, we provide per-thread RASes, as proposed in [10,11], which affects overall cost very slightly.

2.2 Thread and Instruction Selection

To minimize the occurrence and impact of fetch hazards we propose three levels of scheduling: thread allocation, thread selection, and instruction selection. *Thread allocation* selects either the unified or the two-cluster configuration and decides which threads are assigned to each cluster. A description of the proposed policy is left to section 7, after we present our simulation results.

Thread selection determines, cycle-by-cycle, which thread accesses each cluster (see figure 2). It is aimed at hiding fetch hazard penalties. For example, threads with pending cache misses are not considered for cache/BTB access. Also, if two or more threads in the same cluster alternate cache accesses, the computed next PC can be used instead of the next-block prediction, avoiding target misprediction penalties (except for indirect jumps, computed too late). Tullsen et al. [11] analyze the use of thread selection policies to maximize processor performance and propose using the number of instructions in the waiting queues of the execution unit. An alternative approach for thread selection is using confidence estimation for branch predictions [3]. An in-depth analysis of these issues, though, is out of the scope of this paper.

The clustered configuration allows reading two fetch blocks and two next-block predictions per cycle, duplicating fetch bandwidth. *Instruction selection* is a lower-level scheduling to select, each cycle, which four instructions from the eight fetched instructions will be sent to the decode stage. Part of the fetch hazards not overlapped by thread selection and part of the fetch block fragmentation due to taken branches may be hidden by instruction selection. Figure 2 shows the proposed selection logic, which combines aligned 2-instruction sub-blocks from the two clusters. Compared to the SMT proposal [11], this arrangement is less flexible, since it does not permit some merging combinations, but, instead, it reduces logic and delay.

2.3 Cost and Cycle Time Issues

The Alpha 21264 implementation is optimized for high clock frequencies and, in order to preserve high single-thread performance, we take care that the proposed modifications do not increase cycle time and do not require too much additional chip area. Since the total size of the cache and prediction tables is maintained, the area cost of clustering only includes duplicating address decoders and tag comparators, and adding a few registers, counters, multiplexers, and control logic.

The critical path of the cache/BTB access stage in the Alpha design is the computation of the next-block address (highlighted in figure 2), which is needed to start the following fetch cycle. Tag comparison, branch prediction, and instruction decode are outside the critical fetch loop [5]. Thread selection can be put outside the critical path by assuming that some information (for example, detecting a cache miss) will not be on time for taking the current decision.

Considering the worse case for our analysis, we assume the Alpha implements a single memory bank for storing the 4096 next-block fields. Then, the only addition to the critical path is a third entry in the multiplexer selecting the next-block access (highlighted in figure 2), which is necessary to reconfigure the clustered cache as a unified cache and its effect on cycle time can be neglected.

3 Related Work

Some other authors have approached the problem of thread interference in a multithreaded processor. Here we present the most significant related work and highlight the basic differences between their approach and ours.

Eickemeyer et al. [2] use trace-driven simulation to study cache interference on the context of coarse-grained multithreading, which executes a single thread and switches to other thread on L2 cache misses. They found little increase in miss rate when sharing the instruction cache by up to 6 threads, even for small caches (8Kbytes). However, coarse-grained multithreading stresses the i-cache less than SMT, since it permits each thread to perform a relatively long and uninterrupted sequence of accesses to the cache. Moreover, the workload used, with some of the threads executing the same code, benefits from instruction prefetching. Finally, they recognize the problem of memory bandwidth limitations as more threads are added.

Tullsen et al. [10,11] identify the design of the fetch unit for a SMT processor as the most performance-critical issue. In their experiments they find that the wasted issue cycles due to instruction cache misses grows from 1% at one thread to 14% at 8 threads in a multiprogrammed workload. They compare a shared, multi-ported, direct-mapped instruction cache with using eight correspondingly smaller caches, for a varying number of threads. They find the shared cache provides better results for less than 6 threads. However, they do not consider mixed approaches, like two caches each one shared by four threads. They also found a small degree of BTB and branch prediction interference, mainly due to the positive synergy of using threads from the same application.

Hammond et. al. [4] analyze two design alternatives for the L1 cache of a CMP. First, 8 separate 16KB, 2-way caches are used for an 8-thread design. Then, a multi-banked, 128KB cache is shared by the threads, assuming a 2-cycle latency due to the size of the cache and the bank interconnection complexity. They find the average memory access time to the primary cache alone to go up from 1.1 to 5.7, mostly due to extra queueing delays at the contended banks, with an overall performance drop of 24%. However, they do not report the different influence of the instruction and data cache.

4 Experimental Methodology

We simulate the SPECint95 benchmarks to quantify i-cache and BTB misses on different ScSMT fetch unit organizations (table 1). The SPECfp95 benchmarks generally present high instruction and branch target locality, imposing few problems to the fetch unit. Using the *SimpleScalar* [1] simulation tools we have built a MT simulator supporting the concurrent execution of several programs. The multithreaded fetch trace is generated by interleaving instructions from different threads in a perfect round-robin fashion. Benchmarks have been compiled using the DEC C compiler (–O4) for the Alpha architecture. Following the approach in [9], we report measures from the simulation of a representative 100-million-instruction segment of each program that avoids its unrepresentative startup behavior and any warm up effects (see in table 1 the amount of instructions skipped per program). We start simulation one million instructions before the execution segment to warm up the cache and prediction tables. Preliminary analysis and simulation have been done to find each benchmark's execution segment so that results does not change significantly with respect to executing the complete program. Reported cache and BTB miss rates are per executed instruction.

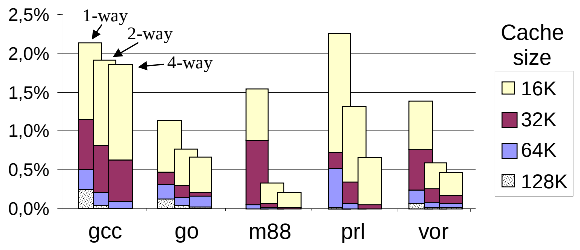
Table 1. SPECint95 benchmarks with their simulation parameters

Benchmark	input data	Skipped	Benchmark	input data	skipped
compress	*reference*	1900 M	***li***	*reference*	750 M
gcc	*cccp.i*	100 M	***m88ksim***	*reference*	300 M
go	*reference*	20 M	***perl***	*scrabbl.pl*	600 M

ijpeg	*penguin.ppm*	100 M	***vortex***	*reference*	2700 M

5 Analysis of Thread Interference on the Instruction Cache

This section studies how to reduce the rate of i-cache misses by accurately selecting an appropriate configuration. Figure 3 presents i-cache miss rates on single-thread execution for size and associativity values closed to the Alpha 21264 i-cache topology (cache lines are 8-instruction long). The table in figure 3 classifies benchmarks depending on its instruction working set, small or large, and its sensitivity to conflict misses. Benchmarks with large working sets benefit from increasing cache size while conflict-sensitive benchmarks benefit significantly from increasing cache associativity. Benchmarks with very small working sets present very low miss rates and are not shown in figure 3.



Benchmark	Working Set	Conflict-Sensitive
compress	*small*	no
gcc	*large*	no
go	*large*	no
ijpeg	*small*	no
li	*small*	no
m88ksim	*medium*	yes
perl	*large*	yes
vortex	*large*	yes

Figure 3. Instruction-cache miss rate for varying size and associativity

Figure 4.a shows the average cache miss rate for all combinations of two different benchmarks in three situations: executing one thread and then the other **(Sngl:** single-thread), multithreaded execution with split caches **(Sp)**, and multithreaded execution with a shared cache **(Sh**). Due to multithreading, miss rates more than double even with 4-way associative caches. This fact must prevent us from neglecting the problem of increasing miss rates. In general, shared caches provide lower miss rates, but direct-mapped caches often generate excessive thread conflicts that make split caches better. For a given pair of threads, *li* and *cmp*, even though both have very small working sets, a shared direct-mapped cache generate a miss rate of 3.2%, in comparison to a 0.14% miss rate with split caches. Since this behavior is intolerable, we discard direct-mapped caches as a suitable option for a ScSMT processor (in contrast, this option was selected for SMT [11]). If fast cache access is what is needed, then an Alpha-like pseudo-associative design could be used.

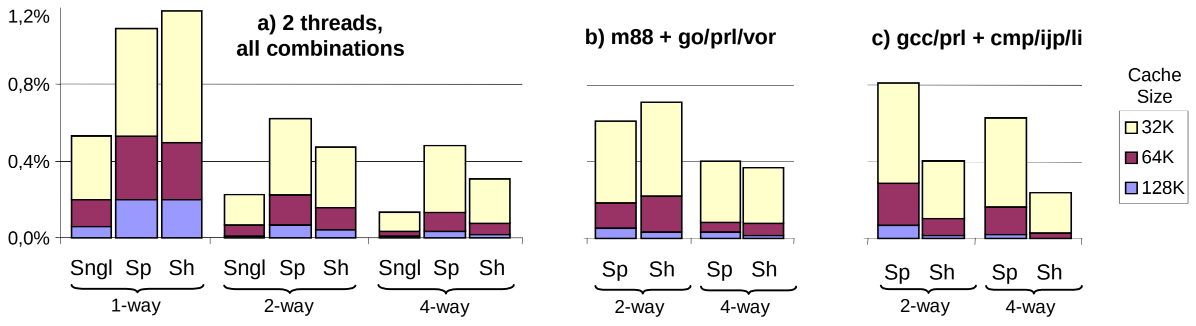


Figure 4. I-cache miss rate for some combinations of 2 threads. (**Sngl**: single-thread, **Sp**: split, **Sh**: shared)

In order to compare the behavior of threads with different characteristics, figures 4.b and 4.c report results for selected combinations of benchmarks. The most conflict-sensitive thread (*m88*) combined with other conflict-sensitive threads (*go*, *prl* and *vor*) produces so many conflicts that a 2-way associative shared cache increases miss rate around 20% compared to using split caches (for 32K and 64K sizes). On the contrary, combining threads with large working sets (*gcc*, *prl*) and threads with small working sets (*cmp*, *ijp*, *li*) minimizes thread interference while the thread with a large working set may benefit from a larger cache. In this case, separate caches increase more than 100% the miss rate compared to sharing the cache (for all cache sizes and associativities).

We have performed exhaustive simulations with 3 to 8 simultaneous threads fetching instructions on all possible cache organizations, including asymmetric organizations (for example, three clusters of sizes 32K, 16K and 16K). Asymmetric organizations help very little, only when the number of threads is not a power of two, and for very particular cases, so we do not consider them anymore. For simulating several combinations of 8 threads we include a second instance of the programs, shifting its execution window to avoid systematic conflicts. Figure 5.a and 5.b shows results for the most significant organizations with 4 and 8 threads.

In general, the best option is a compromise between sharing and splitting the cache. For example, with 4 threads it is better to partition the cache into two split caches and let two threads share each of the split caches (labeled *Sh2* in figure 5). This solution provides an average advantage with respect to a larger shared cache or smaller split caches of 30% and 60% respectively. This organization, however, admits three different thread allocation possibilities. In these cases we report minimum, maximum and average miss rates for all possible placements. The issue of thread allocation is important, since a bad selection may represent an average miss rate increase between 30% and 60% with 4 threads, and peak increases higher than 200%. It is also important to notice that, even if we choose the optimal cache organization and thread allocation, the increase in miss rate due to multi-threading is higher than the number of threads.

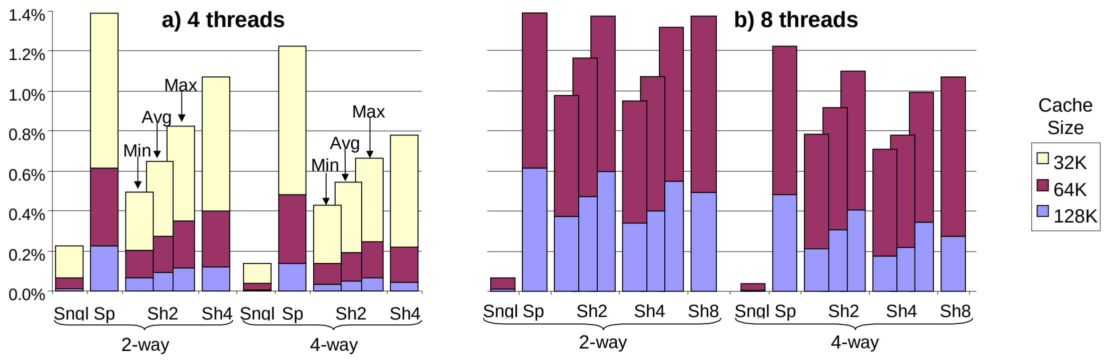


Figure 5. I-cache miss rate with 4/8 threads.(**Sngl**: single-thread, **Sp**: split, **Sh2/4/8**: 2/4/8 threads per cache)

6 Analysis of Thread Interference on the Prediction of Branch Target Addresses

This section analyses the best BTB organization to minimize the frequency of target mispredictions in the presence of multiple simultaneous threads. Figure 6 presents BTB misprediction rates on single-thread execution for size and associativity values closed to the Alpha 21264 BTB topology. Again, benchmarks with very low misprediction rates are not shown. Figures 6.a and 6.b distinguish between direct and indirect branches.

For direct branches, the behavior of benchmarks is practically the same as it was for the i-cache. Benchmarks with a large instruction working set also present a large branch working set (*gcc*, *go*, ..), and benchmarks sensitive to cache conflict misses are also sensitive to BTB conflicts (*m88*, *prl* and *vor*). For indirect jumps, reducing the BTB size or associativity provokes little increase in the target misprediction rate (even on a direct-mapped, 128-entry BTB). This fact indicates there are very few indirect jumps per program. Therefore, for indirect jumps, we can ignore thread interference on a shared BTB and the effect of reducing BTB size on a split organization.

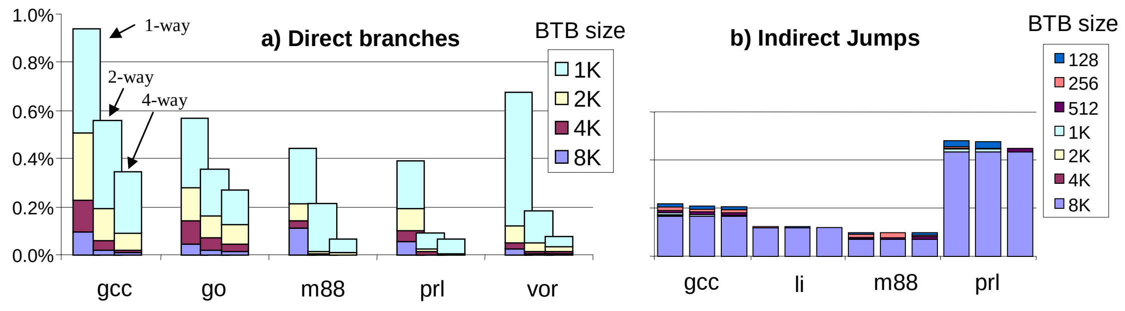


Figure 6. Branch target misprediction rates for varying size and associativity.

Figure 7.a shows the average (direct branch) target misprediction rate for all combinations of 2 different benchmarks on single-thread execution and multithreaded execution with all possible BTB organizations (split and shared). Due to multithreading, misprediction rates increase between 65% and 75%. Shared BTBs are almost always the best configuration, with average improvements between 20% and 100% (improvements grow when BTB associativity is increased). As in the case of the i-cache, the combination of a thread with a large branch working set and a thread with a small working set gives the best results (figure 7.b), with an increase respect to single-thread execution between 10% and 30%. Also, the combination of threads with large branch working sets gives the larger increases with respect to single-thread execution, between 90% and 120% (see figure 7.c).

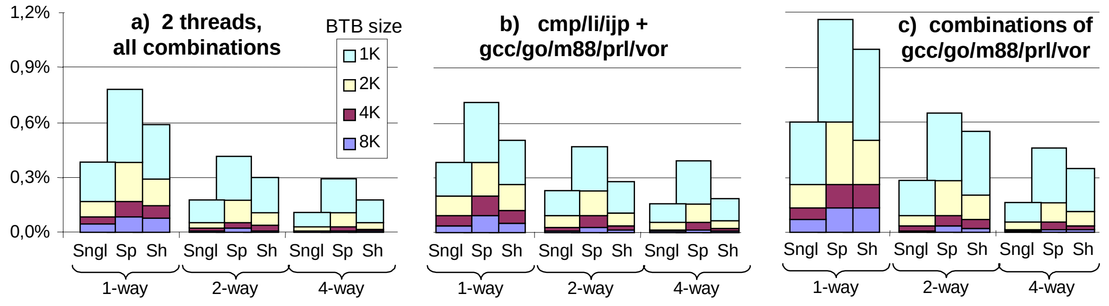


Figure 7. Average direct-branch target misprediction rate for some combinations of two threads for varying BTB size and associativity. (**Sngl**: single-thread execution, **Sp**: split BTBs, **Sh**: shared BTB)

Figure 8 shows results for the most significant BTB organizations with 4 threads. Due to multithreading, misprediction rates multiply by 3-5. In the average, a large BTB shared by all 4 threads or two smaller BTBs, each shared by 2 threads, provide similar results. Both outperform a split BTB by 40%-140%. Although the SH2 configuration outperforms the SH4 configuration for some thread combinations by 5-15%, it suffers an average degradation between 20% and 50% if the best thread allocation is not selected.

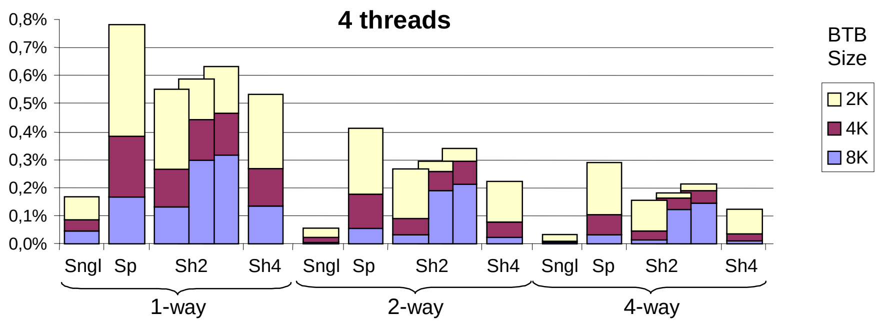


Figure 8. Average target misprediction rate for all combinations of 4 threads on a BTB with varying size and associativity. (**Sngl**: single-thread execution, **Sp**: split BTB, **Sh2/4**: each single BTB is shared by 2/4 threads)

7 Thread Allocation Policy

Results from the two previous sections have shown that selecting either a unified or a two-cluster configuration and deciding which threads are assigned to each cluster, could provide miss rate reductions ranging 30% to 200%. In this section we present a *thread allocation* policy that dynamically adapts to the number and characteristics of the active threads in order to reduce cache/BTB misses. Since we have found that the cache and BTB behavior of a program are very similar, we assume that the best configuration for the cache also performs well for the BTB.

We propose a dynamic, software/hardware implementation of thread allocation. Hardware counters measure the per-thread IPC and cache/BTB miss rates and provide an associated trap mechanism, which is activated when a sum/subtraction of counters reaches a programmable threshold value. A system thread (with supervisor privileges) handles the trap and, based on the contents of the counters, may decide to reconfigure the fetch unit or to swap one or two threads from one cluster to another. The same system handler will be invoked when a new thread is created or finishes. Additionally, static information taken from previous executions or provided by the compiler could be used by the thread allocation policy.

For one thread, the cache/BTB is always configured as a single unit. The two-cluster configuration is activated when adding a thread increases miss rate excessively. Reconfiguring the cache is a costly operation and must be done only if a low (or high) multithreaded degree is expected to stay for a sufficiently large period of time. On a two-cluster configuration, the best performance is obtained when combining threads with large working sets (i.e., high miss rates) and threads with small working sets (low miss rates). Therefore, the allocation policy places the two threads with higher cache miss rates (which we called the two *primary* threads) into two different clusters. Also, it must prevent excessive thread interference due to associativity conflicts.

The use of the hardware counters for the dynamic identification of primary threads and their placement into different clusters must be controlled by an aging mechanism to avoid ping-pong effects when more than two threads alternate the higher miss rates. Also, to reduce the overhead of dynamic thread allocation, thresholds must be chosen so that scheduling traps are not generated too frequently. If the primary threads are clearly detected, but miss rates or the overall IPC reach a given critical threshold, then we may try swapping non-primary threads between clusters to avoid potential associativity conflicts. If this strategy does not work then the problem is likely due to capacity misses and we must reduce the number of executing threads to improve the overall processor performance. In this case, the best option is to swap out one of the primary threads.

8 Summary

On the average, multithreading increases the instruction cache and BTB miss rates more than linearly with respect to the number of threads. In addition, if the cache/BTB topology and thread placement is selected without care, the degradation may increase, on the average, between 30% and 90%, depending on the size, associativity, and number of threads. Finally, there can be particular cases where thread interference on a shared cache may generate cache trashing (this fact lead us to neglect the option of a shared direct-mapped cache). In order to adapt dynamically to the changing characteristics of a multithreaded workload and minimize cache/BTB miss rates, we propose a clustered and reconfigurable organization controlled by three levels of thread scheduling. We describe the proposed microarchitecture and show that it does not increase cycle time and does not require too much additional chip area with respect to a high clock-frequency single-thread processor. This fact, combined with the reconfigurability of the fetch unit, assures both maximum single-thread performance and optimized multithread execution.

An alternative, brute-force approach to the problem is duplicating the i-cache and BTB size and increasing their associativity. However, technological trends indicate that wire delays will soon dominate gate delays. Therefore, since the access time of memories is dominated by wire delays, very fast clock frequencies can only be achieved using very small, direct-mapped memories [8]. Pipelining a large, high-associative shared cache increases the number of pipeline stages, which augments the penalty of branch mispredictions. Finally, software trends continue to be the generation of larger and larger code (higher-level object-oriented languages, compiler optimizations, EPIC architectures, ...), which also reduces instruction cache locality and puts pressure on the cache size.

Although out-of-order execution and multithreading may hide a large part of the cache/BTB miss penalty, there persists the problem of consuming too much L1-L2 bandwidth, which could become a performance bottleneck. The accurate analysis of the performance impact of fetch hazards, modeling all the sources of contention in the execution core (bandwidth and latencies of the data cache and execution units, and instruction queue sizes) and the pollution introduced by speculative execution, is an area for future work. Also, an in-depth analysis of cycle time issues between split and shared cache/BTBs could consider alternative organizations, like using multi-banked or pipelined modules.

Acknowledgements

We would like to thank Rafael García for many valuable discussions and numerous suggestions. This work has been supported by the CICYT under contract number TIC 98-0433 and partially supported by the “Comisionat per a Universitats i Recerca” (Grups de Recerca Consolidats 1999 SGR 00086)

9 References

1. Burger, D., Austin, T.M.: The SimpleScalar Tool Set. *Univ. Wisconsin-Madison Computer Science Department*, Technical Report TR-1342, 1997.
2. Eickemeyer, R. et al.: Evaluation of Multithreaded Uniprocessors for Commercial Application Environments. Proc. of the 23th *Int. Symp. on Computer Architecture* 1996, 203-212.
3. Grunwald, D., Klauser, A., Manne, S., Pleszkun, A.: Confidence Estimation for Speculation Control, Proc. of the 25th *Int. Symp. on Computer Architecture* 1998, 122-131.
4. Hammond, L., Nayfeh, B.A., Olukotun, K.: A Single-Chip Multiprocessor, IEEE *Computer,* Sept. 1997, 79-85.
5. Kessler, R.E.: The Alpha 21264 Microprocessor, IEEE *Micro,* March-April 1999, 24-36.
6. Lee, J., Smith, A.: Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, Jan. 1984, 6-22.
7. Moure, J., García, R., Rexachs, D., Luque, E.: Scalable Simultaneous Multithreading (ScSMT). *Parallel Computing: Fundamental & Applications. Proc. of the Int. Conf. ParCo99*, Imperial College Press, 2000, 615-622.

8. Palacharla, S., Jouppi, N.P., Smith, J.E.: Complexity-Effective Superscalar Processors, Proc. of the 24th *Int. Symp. on Computer Architecture* 1997, 206-218.

9. Skadron, K., Ahuja, P., Martonosi, M., Clark, D.: Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms. Proc. of the 31st *Int. Symp. on Microarchitecture* 1998, 259-271.

10. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous Multithreading: Maximizing On-Chip Parallelism, Proc. of the 22nd *Int. Symp. on Computer Architecture* 1995, 392-403.

11. Tullsen, D.M. et al.: Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor, Proc. of the 23rd *Int. Symp. on Computer Architecture* 1996, 105-116.