TÜBİTAK

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2013) 21: 714 – 733

doi:10.3906/elk-1110-35

Low-cost and power-efficient thread collision detection scheme for shared caches in a real-time multithreaded embedded processor

Emre ÖZER*
ARM Ltd., Cambridge, UK

Received: 17.10.2011 • **Accepted:** 12.03.2012 • **Published Online:** 03.05.2013 • **Printed:** 27.05.2013

Abstract: This paper addresses an important issue in a real-time multithreaded embedded processor where several active hardware threads share the critical resources such as caches in the processor. Thread interferences or collisions could lead to severe performance degradations on the real-time threads. Although the cache interference issue on real-time multithreaded processors has been studied before, no cost-effective and simpler hardware solutions were proposed to maintain the single thread performance of the real-time thread while still letting the low-priority threads progress. A novel technique called collision tag is proposed to address these issues. Progressively, the collision tag scheme can be reduced into a more power-efficient form called collision bit vector, with almost no impact in the overall performance of the real-time multithreaded embedded processor.

Key words: Microarchitecture, real-time systems, multithreading, energy-efficient computing, embedded processors and caches

1. Introduction

In a multithreaded processor core, several active threads having their own hardware contexts share the same resources, such as pipelines, execution units, caches, translation lookaside buffers (TLBs), and branch target buffers (BTBs). A typical multithreaded processor core is depicted in Figure 1, where the shared resources are highlighted. Each thread has its own hardware context (i.e. program counter and register file per thread), while the rest of the microarchitecture resources are shared. This means that multiple threads can be active and share the processor pipelines simultaneously. Hence, the operating system (OS) does not need to perform context switches to share the processor. In a multithreaded processor, the single-thread performance can be severely affected by thread interferences or collisions in the shared resources. If the multithreaded processor is a general-purpose processor system in which all threads have the same priority, then each thread is expected to be given an equal share of the shared resources.

On the other hand, a multithreaded processor that is used in environments with real-time constraints has a mix of high-priority (HP) and low-priority (LP) threads sharing the critical processor resources. A HP thread can be a real-time thread that has a certain deadline to complete a given task. If the HP thread is not complete before the deadline, the system's quality of service will be poor in a soft real-time system, such as live video streaming, or it could be a catastrophic situation in a hard real-time system, such as failure in an antilock brake system. In this paper, the focus is soft real-time embedded processors, where missing deadlines reduce the quality of service rather than leading to catastrophic failures.

*Correspondence: emre.ozer@arm.com

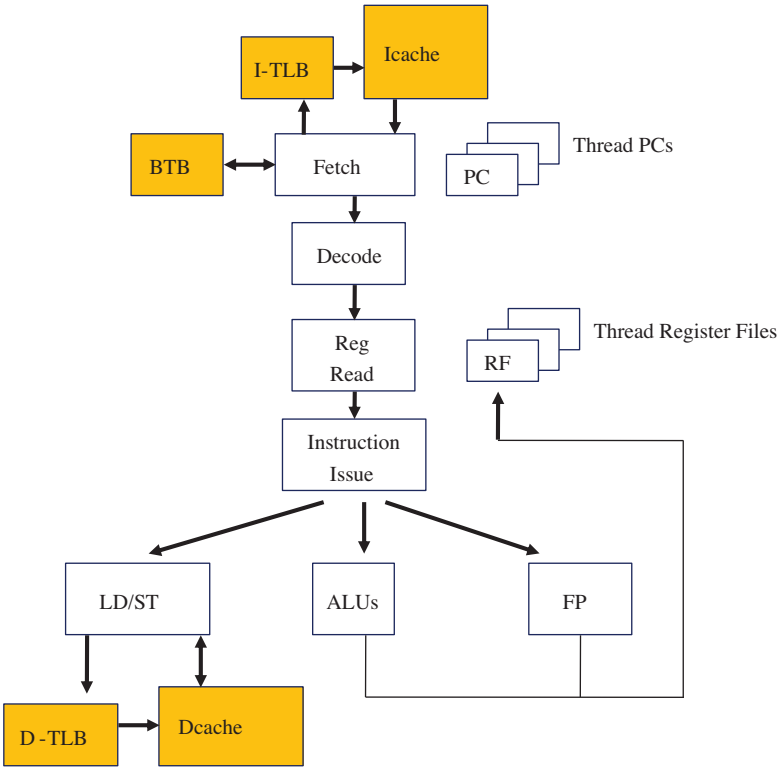

Figure 1. Typical multithreaded processor core.

Caches, in particular level-1 (L1) caches, are extremely critical for high performance because a cache miss could cause dozens of processor cycles to bring the required data from the next level of the cache or memory. Collisions in a shared cache among different simultaneous threads in a soft real-time embedded processor may degrade the performance of higher-priority threads. For example, a LP thread misses in the cache and brings the cache line from the next level of memory to the cache, but evicts one of the cache lines that could belong to a HP thread. This situation occurs because the cache replacement policy selects a victim entry without taking into consideration the thread priorities. In this situation, a higher-priority thread can be evicted from the cache.

HP thread entries in the shared cache must be protected against evictions caused by lower-priority threads. A brute-force technique to protect the HP entries is to force the replacement policy to evict the LP cache lines. If there is no LP thread entry, then a HP thread entry is evicted using the underlying replacement policy. Hence, the HP threads are minimally delayed by the LP ones giving the best single HP thread performance. The downside of this brute-force scheme is that the LP threads can hardly utilize the shared caches and may have extremely long delays in bringing the data all the way from the next level of memory, because they very often miss the shared cache and bring the required data from the off-chip memory. Although this can be tolerated in a multithreaded processor, the overall throughput can be negatively impacted because the LP threads cannot progress effectively.

The main assumption of the brute-force policy is that every LP-to-HP interthread conflict is pathological, meaning that the eviction of every HP thread entry by the LP thread has a negative impact on the performance of the HP thread. This may not be true for every LP-to-HP interthread conflict. Not every evicted HP thread entry is needed by the processor. Thus, this policy may be overly restrictive and can be relaxed so that certain LP-to-HP interthread evictions are allowed under some control. Thus, simple but low-cost hardware schemes are proposed to relax the brute-force cache interference handling policy. First, 'collision prevention by cache

locking', which adapts the traditional cache locking to prevent thread collisions in a shared cache, is analyzed. Next, a novel scheme called 'collision tag', which keeps intuitive information about the evicted HP thread line by the LP thread, is proposed. Later, the collision tag scheme evolves into a simpler, less costly, and more power-efficient scheme called 'collision bit vector'.

The structure of the paper is as follows: Section 2 discusses the related work on the multithreaded processors in general, as well as the cache interference problem. Section 3 describes a novel low-cost collision tag scheme. Section 4 presents the experimental framework along with the performance evaluations, and also motivates the evolution from collision tag to the power-efficient collision bit vector. Finally, Section 5 concludes the paper with a discussion of the results and other potential applications of the proposed schemes.

2. Related work

Cache interference in general-purpose experimental and commercial simultaneous multithreading (SMT) [1] processor models was studied in [2,3]. Both studies argued that the cache interference in the SMT processor is reduced with a better OS job scheduler. Although reducing cache interferences at the OS level is promising in general-purpose computing systems, they reduce the predictability in real-time constrained embedded systems.

Predictability and resource sharing in SMT processors were discussed in [4–6]. Moreover, SMT techniques were applied to soft real-time embedded processor systems [7–12], as well as in hard real-time systems [13]. None of these studies investigated low-cost and power-efficient cache interference or collision detection hardware schemes.

3. Thread collision detection schemes

Various thread collision detection schemes, including the novel collision tag and collision bit vector schemes, are discussed in the following sections. The description of these schemes focuses on a shared cache structure, although they can also be used in other shared table structures in the processor such as TLBs and BTBs.

3.1. Collision prevention with cache locking

A simple way of protecting the HP thread entries in the caches can be achieved using a technique called cache locking [14–16]. Cache locking prevents locked cache entries from being evicted from the cache and thereby provides predictability in the embedded systems having caches. It can be implemented on a granularity of cache lines, cache ways, or the entire cache. Herein, we use cache locking on a per cache line basis by adding a lock bit to each entry in the shared cache. When an entry is allocated to a HP thread, the lock bit is set. The next time around, the replacement policy is steered to select a victim entry by checking the lock bits as well as other replacement maintenance information when a LP thread entry needs to be allocated in the table. If a lock bit is set for a cache line, the line belongs to a HP thread. If there is no LP thread entry (i.e. all of the lock bits are 1), then a HP thread entry is evicted using the underlying replacement policy (e.g., least recently used (LRU) or random). This policy is named 'HP always locked' or HPAL.

In the HPAL policy as shown in Figure 2, each cache line has a lock bit associated with it. Initially, all of the bits are 0. When a HP thread line is allocated in the cache, this bit is set. It is only reset for a cache line when an LP thread cache line replaces the HP one. This only happens when all of the cache ways in the set have the HP thread lines. The cache replacement algorithm could be any algorithm such as the LRU, random, or round-robin. Thus, the HPAL is independent of the underlying replacement policy. The replacement algorithm does not select the locked lines to evict unless all of the lines in the set are locked. In that particular case, the algorithm will select 1 line, ignoring the locks.

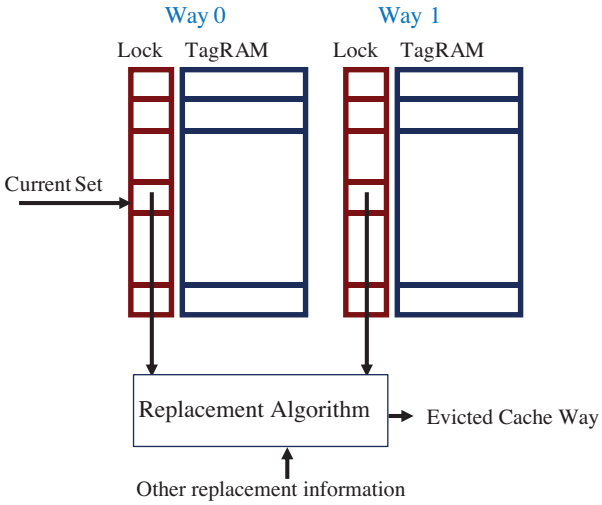

Figure 2. Implementation of the HPAL scheme.

The main drawback of the HPAL policy is that it is too strict for the LP threads when allocating cache lines. This leads to low LP thread performance and also a poor overall throughput of the multithreaded processor because the LP threads cannot progress due to the higher number of cache misses.

3.2. Collision tag scheme

The progress of the LP thread(s), and therefore the aggregate throughput of the multithreaded processor core, can be improved while keeping the performance degradation of the HP thread at a minimum. Thus, we develop a more dynamic policy that can consider these. The premise of the dynamic scheme is that it allows the LP threads to evict the HP thread cache lines. It uses an extra storage called 'collision tag' to keep the address tag information of the HP line, which is evicted by a LP cache access. Hence, the dynamic scheme is called 'collision tag'. The collision tag storage can be implemented on a per-way or per-set basis. The per-set basis is opted for because it requires less area. Figure 3 depicts the collision tag scheme implemented on a per-set basis. It uses a lock bit per-cache line and a collision tag per-cache set. In order to distinguish the owner of the cache line (i.e. whether it belongs to the HP or LP thread), 1 bit is added to each cache line, called the HP bit. Unlike the HPAL policy, the lock bit alone is not sufficient to identify an HP thread cache line. Initially, all of the HP bits are 0, and when a HP thread allocates a cache line, the HP bit is set for that cache line. When a LP request misses in the cache and needs to evict a cache line, its thread priority (e.g., 0 for LP and 1 for HP) is compared to the HP bit of the to-be-evicted cache line in order to detect the case of a LP thread evicting a HP thread entry.

The mechanism of how the collision tag works is as follows: initially, all of the lock bits and HP bits are set to 0. Unlike the HPAL scheme, the lock bit is not set to 1 when a HP thread cache line is allocated. This means that the HP thread lines can be evicted from the cache by the LP threads steered by the replacement algorithm. When a LP thread evicts a HP thread cache line, the tag of the evicted HP cache line is read and stored in the collision tag entry of the cache set. Whenever a HP thread allocates a cache line, its tag is always compared with the tag stored in the corresponding collision tag entry to identify whether this HP cache line has been returned to the cache. Hence, this cache line should be prevented from being evicted again because it is more than likely frequently used data. When there is a match in the collision tag entry, the lock bit of the cache line is set to 1 to prevent further eviction of this HP cache line by the LP threads.

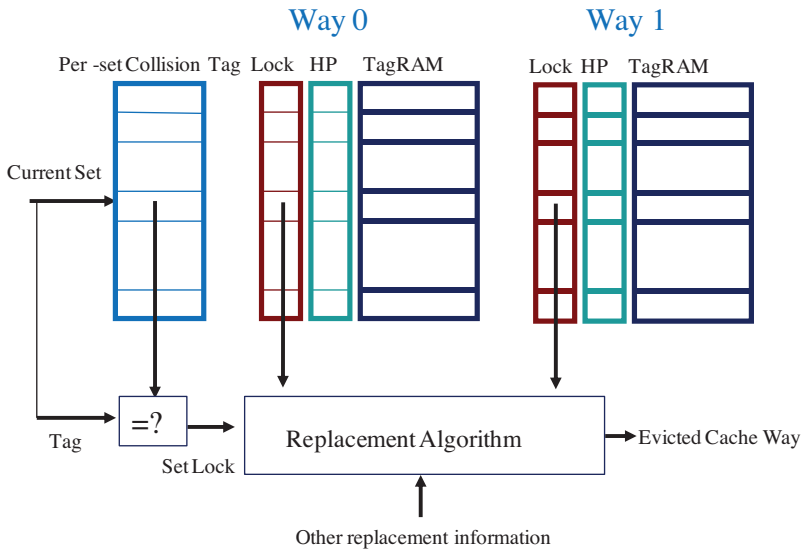

Figure 3. Implementation of the collision tag scheme.

In summary, the collision tag scheme allows the HP cache lines to be evicted for the first time but prevents them from being evicted from the cache when they return to the cache by locking them. The precise detection of their return is accomplished by saving the tag of the evicted HP line in the collision tag allocated for each cache set. Although there is only 1 collision tag entry for each cache set, the individual lock bits allow for locking multiple HP cache lines that return to the cache after being evicted in the past. The collision tag scheme provides considerable improvements in the LP thread performance and the overall throughput of the multithreaded processor core.

The cache replacement logic needs to check not only the lock bits but also the HP bits and the collision tag comparison match results to make a decision as to which cache line to evict. This does not have any delay overhead in the decision process if the eviction decision is made at the time of the cache line allocation, when the cache line is brought from the next level of memory. In this case, there is no delay overhead because the eviction decision can be overlapped with the time, while the cache waits for the cache line to come from the next level of the memory.

4. Experimental framework and results

4.1. Simulation framework

A cycle-accurate simulation of a SMT implementation of an ARMv7 architecture-compliant ARM processor core model is conducted using a trace-driven simulation, as shown in Figure 4. The cycle-accurate simulator is an in-house simulator developed in ARM, which simulates only applications. Note that this is not a full-system simulator, and so it does not run the OS in the simulator and therefore cannot model the OS effects. The front end of the trace-driven simulation is the instruction set simulator that reads the binaries of the applications and datasets, and each application spits out instruction traces into a pipe, which are then consumed by the trace-driven ARMv7 architecture-compliant SMT processor microarchitecture simulator.

The real-time SMT processor microarchitecture is modeled as a multiple-thread soft real-time multi-threaded core. Similar to the studies in [5] and [8], there is a single real-time or HP thread and the rest of the threads are LP ones. The performance of the real-time thread should be compromised minimally while increasing the throughput of the multithreaded processor by simultaneously running LP (nonreal-time threads

or tasks). The HP thread has priority using the fetch, decode, and issue resources over the LP thread(s) in the SMT model. The simulated SMT processor core microarchitecture parameters are shown in the Table.

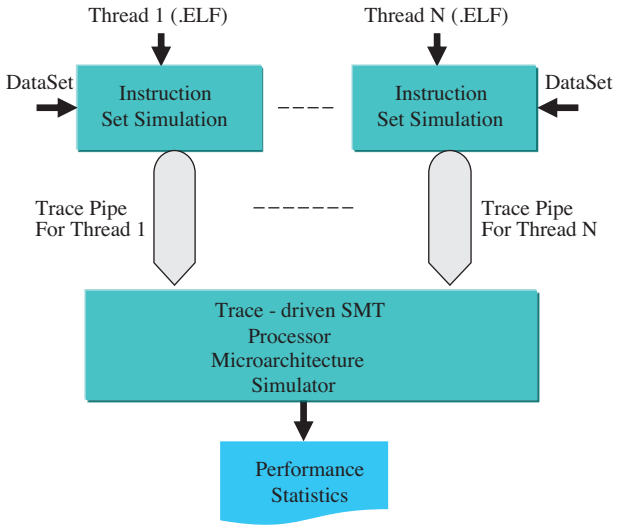

Figure 4. Trace-driven simulation framework.

Table. Simulated SMT processor microarchitecture parameters.

Parameters	Details
Processor type	In-order superscalar
Issue width	Dual-issue
Fetch bandwidth	Two 32-bit or four 16-bit instruction fetch per cycle
Decode bandwidth	2 instructions per cycle
# of threads	2 and 4
On-chip L1 instruction cache	4-way 8 KB with 32 B cache line size 1-cycle hit time
On-chip L1 data cache	4-way 8 KB with 32 B cache line size 1-cycle hit time
Instruction and data TLB sizes	32-entry fully associative
Bulk memory access time	60 core cycles
Branch predictor	Shared 2K global branch predictor Shared
	512-entry BHB Global branch history register per thread
Return address stack	8-entry per thread

The focus of this study is to gain insight into the collision behavior in a shared table structure in an SMT processor core. Hence, the aforementioned microarchitectural techniques are applied to the shared data cache (Dcache) in the processor core, the collision behaviors of multiple threads are observed, and the results are discussed in the following subsections. Having said that, the schemes proposed in this paper can also be applicable in other shared table structures in the processor, such as the instruction cache, level-2 (L2) cache, TLBs, and BTBs. The Dcache is selected because it is the most collision-sensitive shared table structure in modern SMT processors [1,2]. Two models of multithreading are modeled: the dual-thread and quadruple-thread models. In the dual-thread model, 1 thread is HP (i.e. real-time thread) and the other is LP. Similarly, 1 thread is HP (real-time thread) and the other 3 are all LP threads in the quadruple-thread model.

4.2. Experimental methodology

A set of applications from the EEMBC benchmark suite [17], covering a wide range of embedded applications including consumer, automotive, networking and multimedia, is used in the experimental framework. The EEMBC benchmark suite contains dozens of benchmarks from various market segments.

It requires a prohibitive amount of simulation time to run every possible benchmark permutation if we use all of the benchmarks in the EEMBC suite. For example, to study the dual-thread mode, all of the dual permutations of the benchmark suite must be simulated. For example, 400 benchmark couples must be simulated per experiment if 20 benchmarks are selected from the suite. This gets even worse for the quadruple-thread mode, where 160,000 benchmark-quadruples must be simulated per experiment. This is clearly infeasible in terms of simulation time as it may take days or weeks to get the simulations results for 1 experiment.

In order to conduct a feasible simulation study, a methodology is required to select a reasonable number of benchmarks from the suite using a meaningful criterion. A meaningful criterion can be to measure the Dcache miss rate of each benchmark in the suite, since the focus of the paper is to observe the collision behavior in the shared Dcache. This is meaningful because the Dcache can be stressed more aggressively when several memory-intensive benchmarks are running simultaneously.

Hence, each benchmark is run alone in the simulator to collect its Dcache miss rate. The benchmarks are sorted in descending order of their Dcache miss rates. Next, the benchmarks with the highest Dcache miss rates are selected for the study. In order to perform the simulations within a reasonable timeframe, the 9 benchmarks with the highest Dcache miss rates are identified as follows:

- Three applications (TCP_Dataset1, TCP_Dataset2 and TCP_Dataset3) from Networking™ 2.0 [18].
- Two applications (fast Fourier transform and inverse fast Fourier transform) from AutoBench™ 1.1 [19].
- Two applications (RGB_2_CMYK_conversion and RGB_2_YIQ_conversion) from ConsumerBench™ 1.1 [20].
- One application (Bit_Allocation) from TeleBench™ 1.1 [21].
- One application (mpeg2 decoder) from DENBench™ 1.0 [22].

For the 9 benchmarks in the dual-thread mode, 81 benchmark couples (self-coupling is allowed) must be simulated per experiment and 6561 benchmark quadruples must be simulated for the quadruple-thread mode. The criterion to decide which thread is the real-time or HP thread is straightforward: the first benchmark in each permutation is assumed to be the HP thread, and the others are all LP threads. Hence, each benchmark becomes an HP thread in an equal share per experiment. For example, each benchmark becomes an HP thread 9 times in 81 simulations per experiment. A simulation run for a couple or quadruple stops when the HP thread completes.

As said before, the soft real-time SMT processor core has 1 real-time or HP thread, and all of the other threads are LP threads. The objective of this study, which is the subject of the following experiments, is to minimize the impact on the progress of the real-time thread in the presence of the other LP threads, and at the same time to improve the SMT processor throughput by executing as many instructions from the LP threads as possible.

4.3. Performance results

The baseline model used for the comparison is the exact configuration of the same multithreaded processor model, but the cache model does not distinguish the cache accesses based on the thread priority (i.e. it does not use any cache locking or any other mechanism when replacing the cache lines).

Figures 5 and 6 show the speedup of the HP thread in the HPAL and collision tag schemes with respect to the baseline model for the dual and quadruple threads, respectively. The HP thread speedup results are presented for each benchmark, which represents the HP thread. For example, there are 81 thread couplings for the dual-thread model for 9 benchmarks, and each benchmark becomes a HP thread for the 9 of the total couplings. The average of the 9 speedup results for each HP thread is calculated and depicted in Figures 5 and 6.

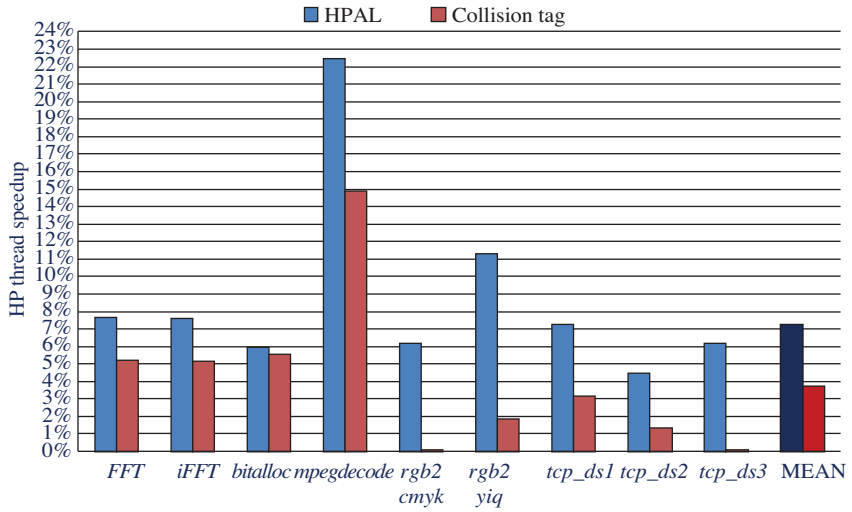

Figure 5. HP thread speedup relative to the baseline in the dual-thread model.

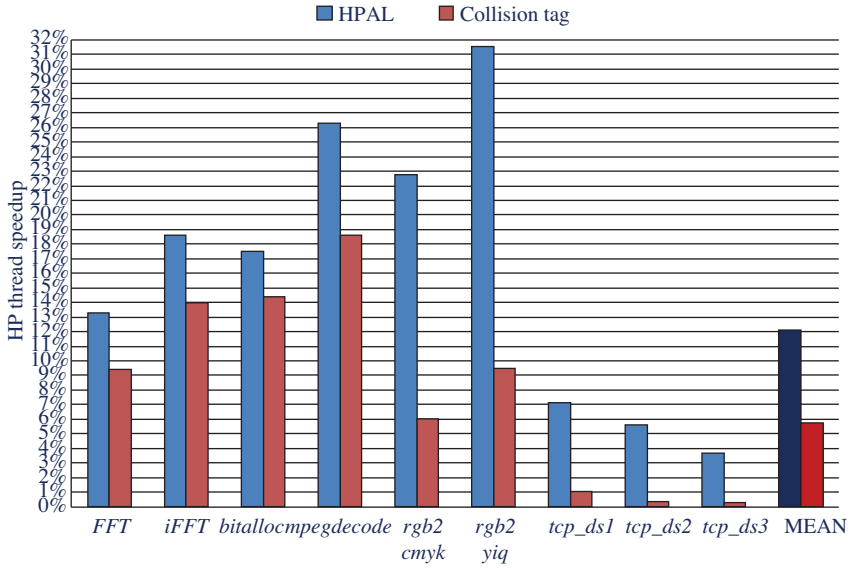

Figure 6. HP thread speedup relative to the baseline in the quadruple-thread model.

As stated before, the HPAL scheme favors the HP thread by fully protecting it from the interference of the LP thread. Thus, the HPAL scheme performs better than the collision tag in terms of pure HP thread performance. Overall, the HPAL improves the HP thread performance by over 7% while the collision tag remains a little below 4%, as shown in Figure 5. The HPAL boosts the performance of *mpeg2decode* and *rgb2yiq* in particular, as these 2 benchmarks are sensitive to the cache interference by the LP thread. Thus, they enjoy a significant boost in the HP thread performance when the HP thread is protected by the HPAL.

There is a steady improvement in the HP thread's execution time as the number of LP threads increases, as shown in Figure 6. More LP threads increase the probability of evicting a HP cache line from the cache. Since the HPAL is overly protective of the HP lines, the HP thread observes a considerable improvement in its execution time. On the other hand, the HP thread speedup rate in collision tag is slower than that of the HPAL. This is because the collision tag scheme protects only a certain number of the evicted HP cache lines rather than protecting all of the HP cache lines. Thus, this hurts the HP thread performance. Overall, the HPAL improves the HP thread performance by over 12% while the collision tag remains a little below 6%. With respect to the dual-thread model, the HP thread speedup jump from 7% to 12% is quite noticeable. This is because the HP thread performance in the baseline model deteriorates with the extra 3 LP threads slowing down the HP thread in the quadruple model. Thus, the relative speedup from the dual to quadruple model rises considerably for the HPAL scheme.

Figures 7 and 8 show the slowdown in the LP thread(s) relative to the performance of the LP threads in the baseline model. The metric to measure the performance of the LP threads are their aggregate interference protection criteria (IPC) of LP threads. The LP thread slowdown is calculated by comparing their aggregate IPC to the aggregate LP thread IPC in the baseline model. Essentially, this metric presents how much the LP threads can progress under a HP thread. The LP thread performance is always better in the baseline model because the baseline does not favor the HP threads over the LP ones, and so the LP threads can progress more freely. The slowdown in the LP thread is significantly large in the HPAL because it completely sacrifices their performance in favor of the HP thread, as shown in Figure 7. This is particularly striking in *mpeg2decode*,

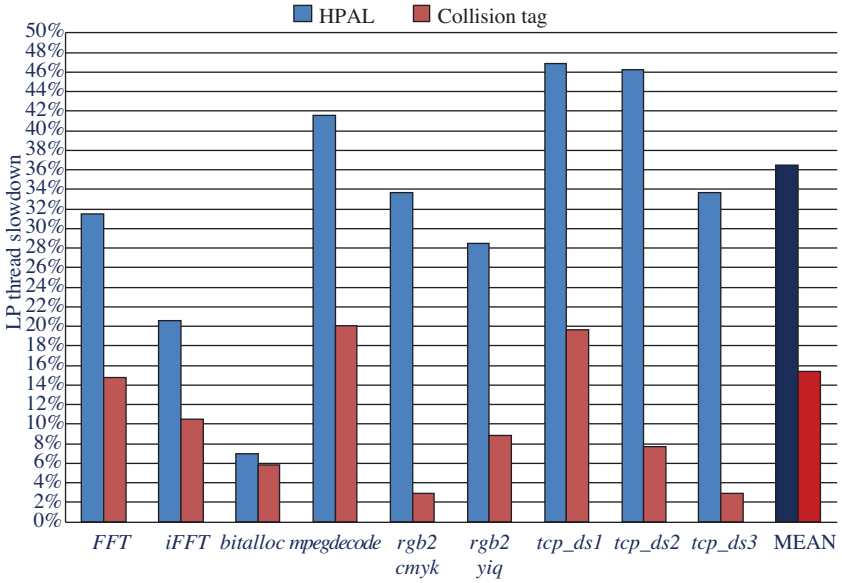

Figure 7. LP thread slowdown relative to the baseline in the dual-thread model.

rgb2cmyk, *rgb2yiq*, *tcp_ds1*, *tcp_ds2*, and *tcp_ds3*. On the other hand, collision tag treats the LP thread more fairly, and therefore there is no noticeable spike in the LP thread slowdown. Overall, the LP thread slowdown relative to the baseline remains under 16% in collision tag where this is little over 36% in the HPAL. In the quadruple-thread model, there is improvement in the overall performance of the LP threads, both for the HPAL and collision tag, since there are 3 extra LP threads running along with the HP thread, which increases the likelihood of the LP thread progress. As seen in Figure 8, the collision tag scheme visibly improves the LP thread performance from the dual to quadruple threads. On the other hand, the HPAL exhibits more variable behavior, e.g., *mpeg2decode* and all 3 *tcp* benchmarks improve with more LP threads, while the LP threads rarely progress in the other benchmarks. Overall, the slowdown in LP threads is reduced to 10% in collision tag, while it stays around 29% in the HPAL.

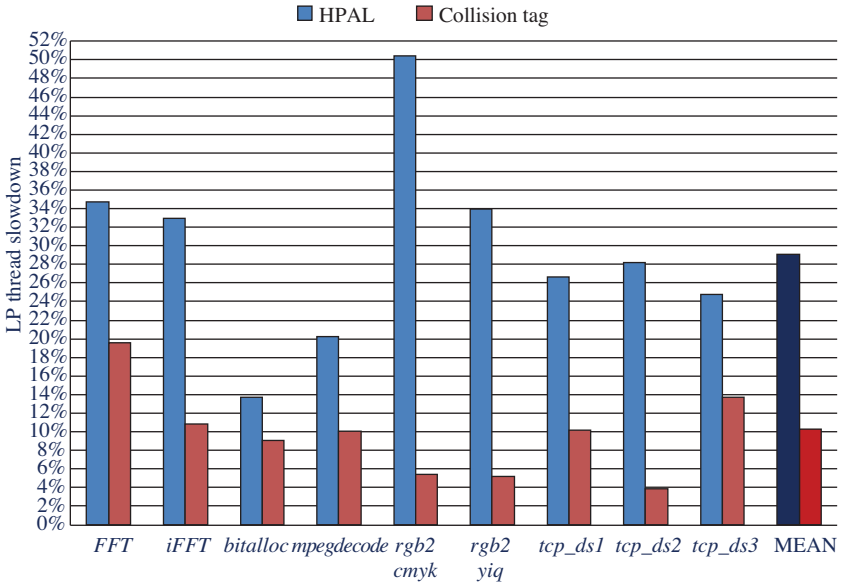

Figure 8. LP thread slowdown relative to the baseline in the quadruple-thread model.

Finally, the total throughputs of the dual and quadruple thread models are shown in Figures 9 and 10. The throughput is measured as the aggregate number of instructions retired per cycle of the core, including both the HP and LP threads. The IPC improvement in percentage for both schemes is measured with respect to the baseline model (harmonic mean is used when averaging the total IPC numbers). The total throughput of the HPAL over all of the benchmarks is worse than that of collision tag because of the limited progress in the LP thread(s), except for *bitalloc*, *mpeg2decode*, *rgb2cmyk*, and *rgb2yiq*. The HPAL speeds up *mpeg2decode*, *rgb2cmyk*, and *rgb2yiq* at the cost of low performance in the LP thread. The total IPC value is better in these benchmarks because a faster HP thread means a reduced total execution time, which improved the IPC. In *bitalloc*, the total throughput improves even though the HP thread speedup is moderate. This is because its LP thread performance is comparably higher than *mpeg2decode*, *rgb2cmyk*, and *rgb2yiq*. The quadruple-thread model in Figure 10 improves the total IPC when compared to the dual-thread model because the overall LP thread progress is much better due to more LP threads than in the single LP thread progress in the dual-thread model. Overall, the total throughput in collision tag is slightly less than the baseline in both thread models, and it surpasses the overall throughput of the HPAL scheme.

4.3.1. Summary

Overall, the HPAL policy gives the best HP thread performance by its extreme protective behavior of the HP thread lines. The HPAL policy represents the upper limit for the performance of the HP thread. On the other hand, the collision tag scheme provides more balanced overall performance by improving the performance of the LP threads while keeping the performance degradation of the HP thread under control.

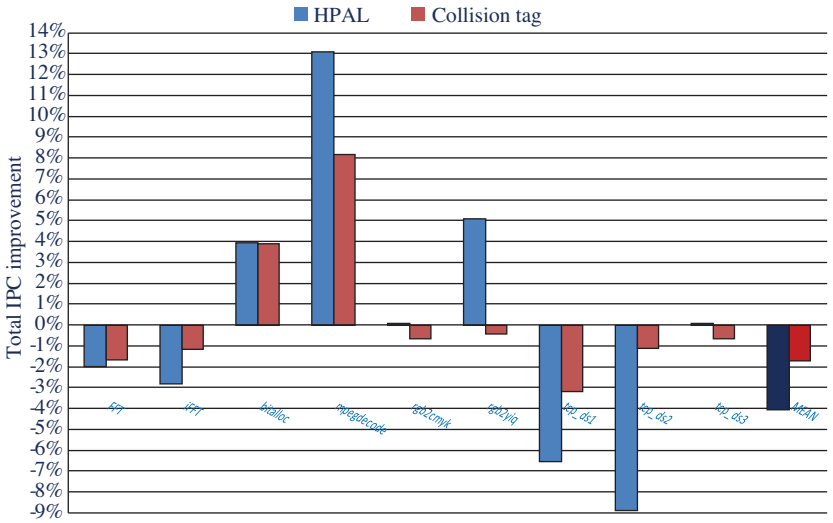

Figure 9. Total IPC improvement relative to the baseline in the dual-thread model.

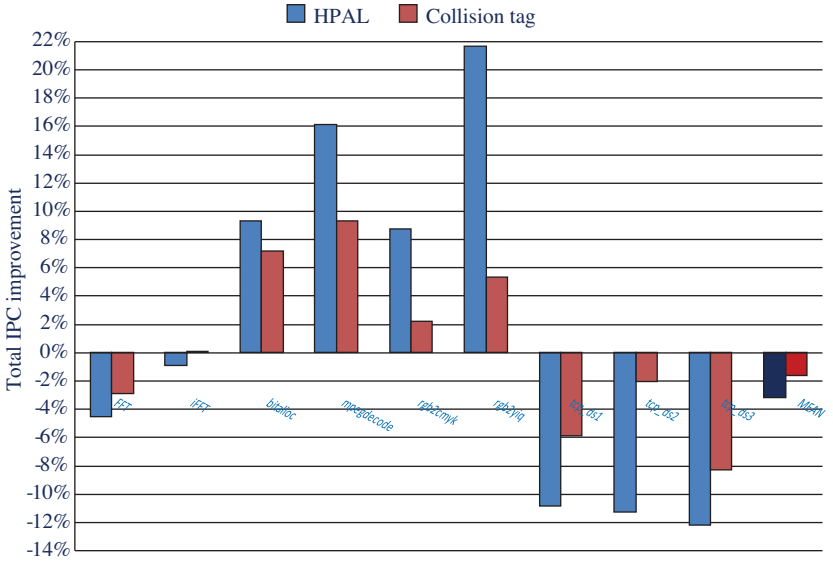

Figure 10. Total IPC improvement relative to the baseline in the quadruple-thread model.

4.4. Collision bit vector scheme

The collision tag contains the full tag information about the HP thread cache lines in order to detect precisely the return of the formerly evicted HP cache lines. However, the precise detection of full collision tag matches may not be necessary because detecting the collision tag matches less precisely only causes false positives, i.e. some HP cache lines are locked unnecessarily by taking them as formerly evicted HP cache lines that return.

In fact, such unnecessary locks improve the performance of the HP thread at the cost of lower performance in the LP thread(s).

Thus, storing only a few least significant bits from the full collision tag, instead of storing the full collision tag, is also tried and demonstrated. The HP thread speedup results averaged over all of the simulation runs are shown for 1 least-significant bit of the collision tag, 4 least-significant bits of the collision tag, 8 least-significant bits of the collision tag, and the full collision tags in Figure 11 for the dual-thread model. The HPAL policy (the first bar) is also shown for comparison. As predicted earlier, the HP thread performance storing only narrower collision tags is slightly higher than storing the full collision tag. In fact, using as narrow as 1 least-significant bit from the full tag provides the best speedup because it gets the highest number of false positives.

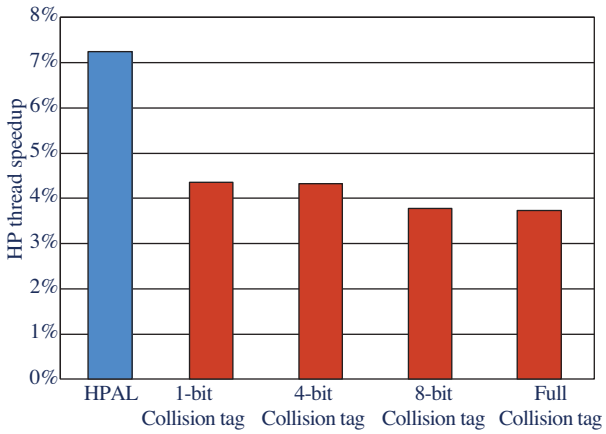

Figure 11. HP thread speedup for different widths of collision tags for the dual-thread model (bigger is better).

Now, let us look at the LP thread slowdown as shown in Figure 12. The 1 bit from the collision tag behaves worse than the full tag policy. This is expected because the false positives now hurt the performance

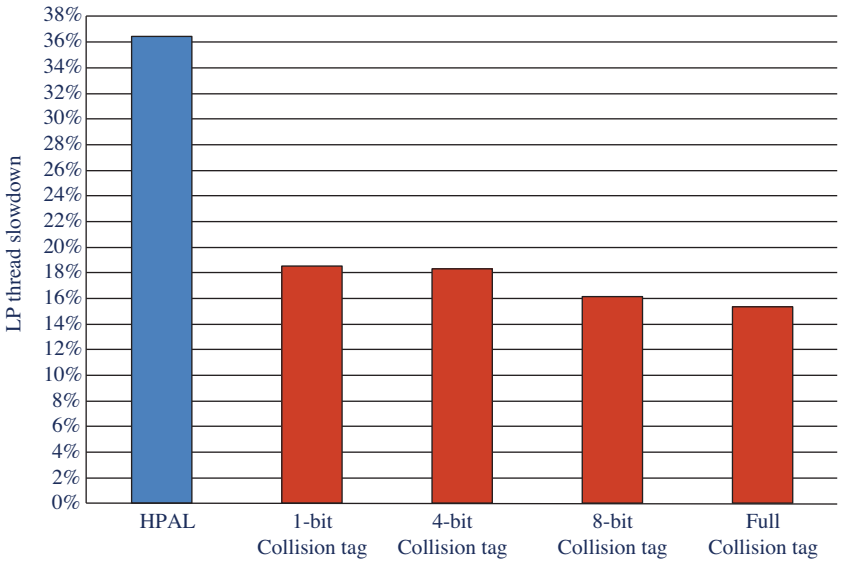

Figure 12. LP thread slowdown for different widths of collision tags for the dual-thread model (smaller is better).

of the LP threads. In any case, the LP thread slowdown difference between the 2 extreme schemes is only 3 percentage points. Finally, Figure 13 shows the total IPC improvement. The percentage IPC improvement rates are very close in both schemes, where the full collision tag scheme is only 0.3 percentage points better than storing a single bit from the collision tag.

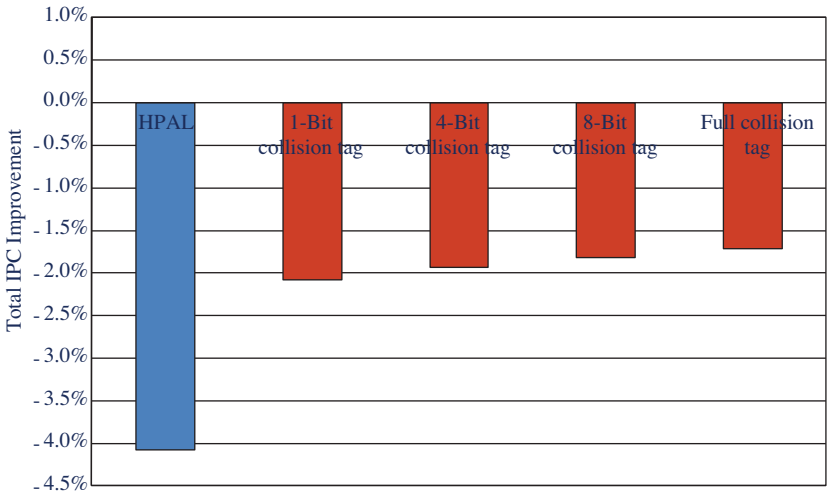

Figure 13. Total IPC improvement for different widths of collision tags for the dual-thread model (less negative is better).

As observed in these 3 graphs, the difference in the performance numbers is negligibly small between storing a single bit from the collision tag and the full-bit collision tag policies. However, storing a single bit from the collision tag has an energy-saving advantage over the full bit because of the reduction in the storage array size.

Thus, we propose to store only the least-significant bit from the collision tag and call it the collision bit vector scheme, as shown in Figure 14, where the full collision tag is reduced to a single bit vector. The operation

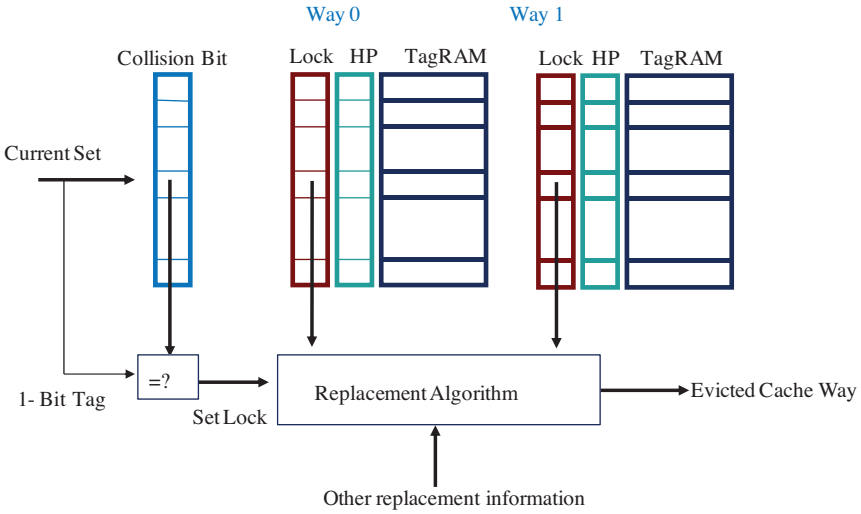

Figure 14. Collision bit vector.

of the collision bit vector scheme is exactly the same as the collision tag scheme. When a HP thread cache line is evicted, only the least-significant bit of its tag is stored in the collision bit vector location in the cache set, rather than storing the full tag in the collision tag location as in the collision tag scheme. This scheme is more power-efficient than the collision tag scheme because the 23-bit collision tag entry per cache set is now reduced to 1 bit. At every cache miss, an extra 1-collision bit comparison is needed rather than an extra 23-bit tag lookup. Thus, the collision bit vector power overhead is negligibly small.

Figures 15–20 summarize the comparison of the 3 metrics (i.e. HP thread speedup, LP thread slowdown, and total IPC improvement) for the HPAL, collision tag, and collision bit vector policies as the number of threads increases from 2 to 4 or as the number of LP threads increases from 1 to 3. The main observation here is that as the number of LP threads increases, the gap in the HP thread performance between the HPAL and collision bit vector widens in favor of the HPAL. As there are more LP threads in the system, the HP thread is more vulnerable to performance attacks by the LP threads. Hence, the HPAL performs quite well by providing unconditional protection to the HP thread cache lines. It is also true that more LP threads provide better LP thread performance in both policies. Even the LP thread performance is improved in the HPAL policy because other LP threads are able to progress when some LP threads stall due to the overprotection mechanism. As the number of LP threads increases, the gap in the IPC degradation of the LP thread gets smaller. The overall effect of these 2 factors is that the gap in the aggregate IPC degradation becomes smaller as the number of LP threads increases.

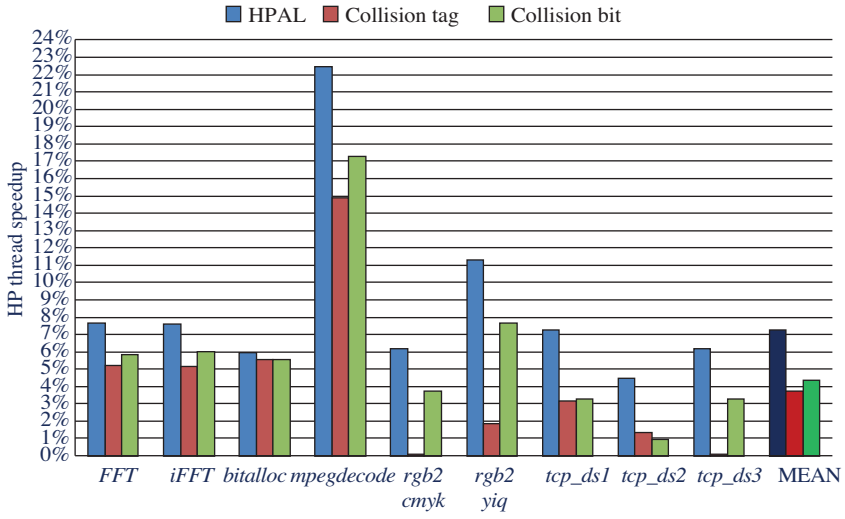

Figure 15. HP thread speedup comparing the HPAL, collision tag, and collision bit schemes relative to the baseline in the dual-thread model.

4.4.1. Power advantage of the collision bit vector

Clearly, the collision bit vector scheme saves more power than the collision tag scheme. In order to quantify the power saving benefits of the collision bit vector scheme, the power overheads of the collision tag and bit vector schemes are measured. The power overhead is calculated relative to the baseline tag lookup power consumption. In the baseline model, each cache access involves looking up all of the tag arrays. For example, 4 tag lookups are required for a 4-way set-associative cache per-cache access. The power overhead of the collision tag scheme on top of the regular tag lookups is the lock and HP bit lookups per way, storing the tag into the collision tag array in the case of a LP thread evicting the HP thread cache line and comparing the tag data in the collision

tag array with the tag portion of the HP thread line address if the HP thread is doing a line-fill to the cache. On the other hand, the power overhead of the collision bit vector scheme on top of the regular tag lookups is the lock and HP bit lookups per way, storing the 1-bit into the collision bit vector in the case of a LP thread evicting the HP thread cache line and comparing the 1-bit tag data in the collision bit vector with the 1-bit tag portion of the HP thread line address if the HP thread is doing a line-fill to the cache.

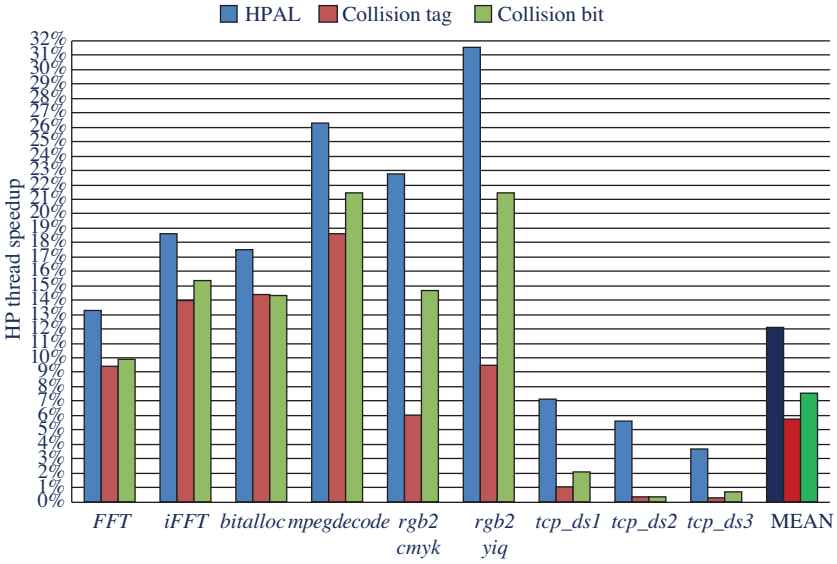

Figure 16. HP thread speedup comparing the HPAL, collision tag, and collision bit schemes relative to the baseline in the quadruple-thread model.

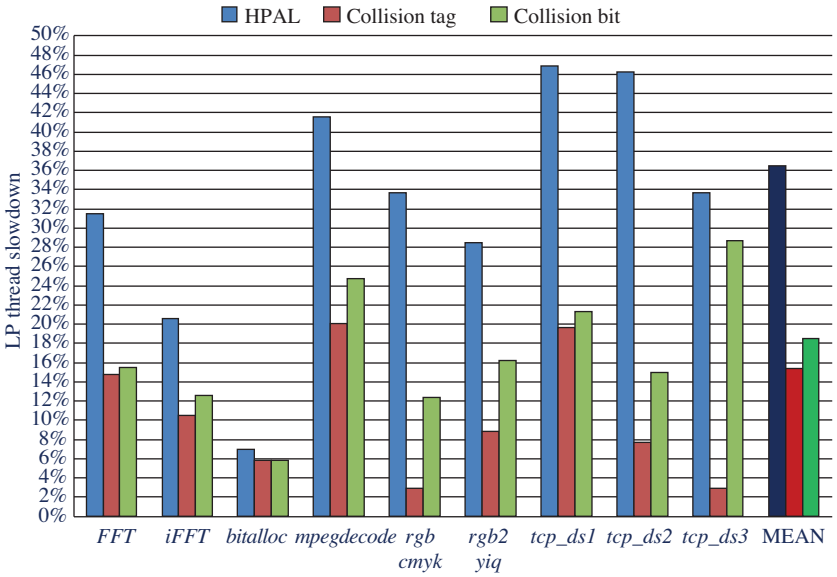

Figure 17. LP thread slowdown comparing the HPAL, collision tag, and collision bit schemes relative to the baseline in the dual-thread model.

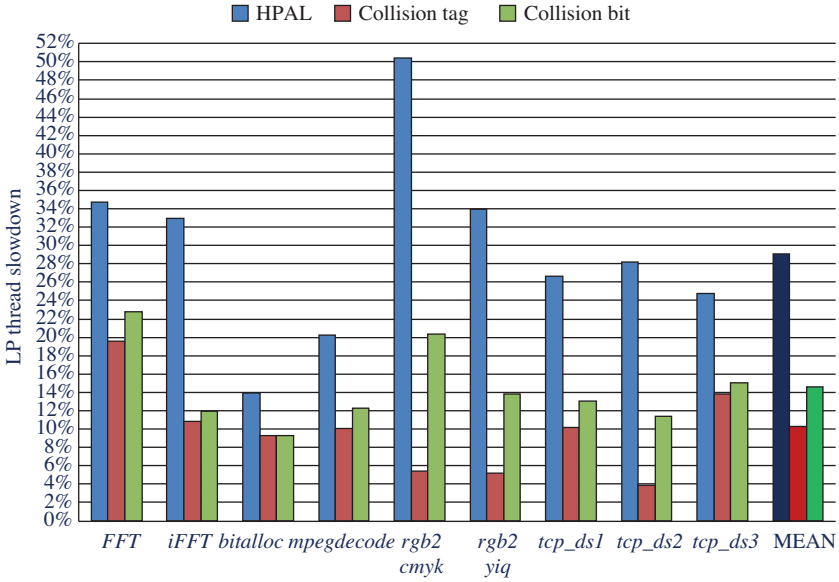

Figure 18. LP thread slowdown comparing the HPAL, collision tag, and collision bit schemes relative to the baseline in the quadruple-thread model.

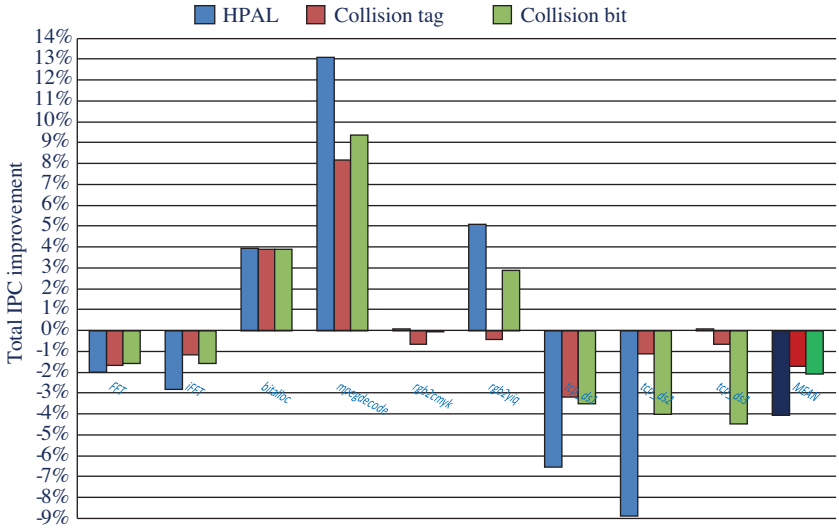

Figure 19. Total IPC improvement comparing the HPAL, collision tag, and collision bit schemes relative to the baseline in the dual-thread model.

The industry-standard ARM embedded memory compilers [23] are used to create different tag array and bit vector models. The ARM embedded memory compilers model the delay, area, and power (dynamic and leakage) of the static random-access memories (SRAMs) accurately, and have been used in commercial system-on-chip designs. System designers can select from high-density, high-speed, and low-power SRAMs to optimize their design for speed, power, and/or area. ARM SRAM memory compilers are available in various foundries and 65 process variants from 250 nm to 28 nm. In this paper, the IBM 32 nm low power foundry and technology library are used to build the SRAMs. The user can parameterize the SRAM by providing the number of rows and the number of bits per row, the operating frequency, design corners, etc., and the compiler synthesizes the

SRAM and generates the register transfer language code of the SRAM array/peripheral logic along with the power, area, and access delay statistics.

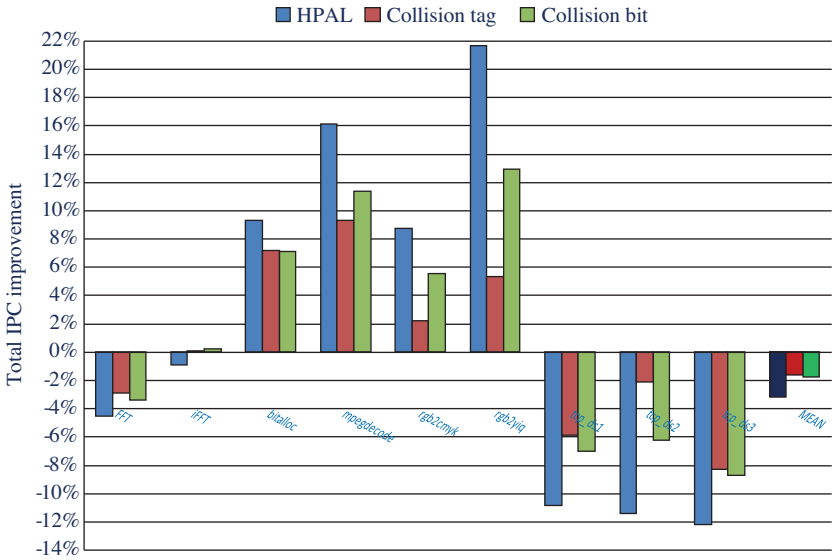

Figure 20. Total IPC improvement comparing the HPAL, collision tag, and collision bit schemes relative to the baseline in the quadruple-thread model.

The number of tag bits for a 4-way 8 KB data cache (the same data cache used in the performance study in previous sections) using a 32 B cache line size in a processor using 32-bit addressing is 21 bits. Hence, the tag SRAM array size is 64 × 21 bits per way. An array of 64 × 2 bits is required to represent the lock and HP bits per way for both the collision tag and bit vector schemes. The collision tag scheme needs an extra collision tag array of the same size of 64 × 21 bits per set, whereas the collision bit vector scheme only needs an extra 1-bit vector of 64 × 1 bits per set. Thus, 3 different SRAM array models are created: 1) a tag array of 64 × 21 bits (for regular tags and the collision tag), 2) an array of 64 × 2 bits (for the lock and HP bits), and 3) a bit vector of 64 × 1 bits (for the 1-bit collision vector). When comparing the power overheads of both schemes, the total power (dynamic + leakage) is used since the memory compiler generates both. The tag lookup power overheads of both schemes relative to the baseline model are shown in Figure 21. The power overhead of the collision tag scheme is about 12%, while this overhead is only 6.5% in the collision bit vector scheme. These results are intuitive because the collision bit vector saves 20 bits in the collision tag array per set. The area savings from a 64 × 21-bit array to a 64 × 1-bit vector are translated into a 5% savings in the cache tag lookup power consumption.

4.4.2. Dual-mode hybrid scheme

The logical deduction from the results above is to design a more dynamic policy by exploiting the best of the HPAL and collision bit vector policies, as shown in Figure 22. When the number of LP threads is low in the SMT processor, the collision bit vector policy is used by enabling the collision bit and HP bit vectors. As more LP threads become active in the processor, the monitor logic within the processor core decides to disable the collision bit and HP bit vectors and switch to the HPAL mode from the collision bit vector; hence, the dual-mode hybrid scheme. Switching to the HPAL mode is a straightforward step and can be performed as

follows: as soon as a HP cache line is allocated in the cache, its corresponding HP bit and lock bit are set simultaneously (i.e. the HP and lock bits have to be the same all of the time).

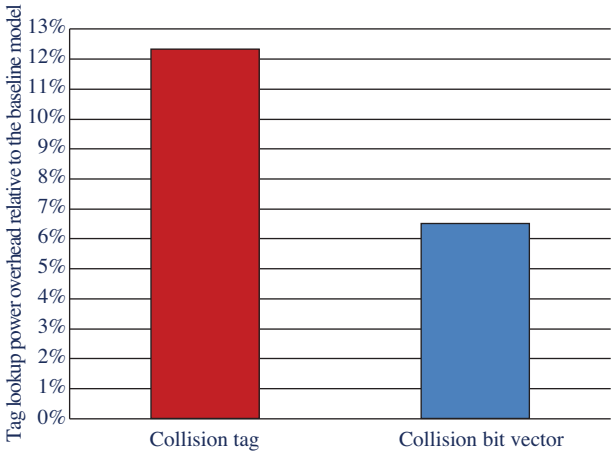

Figure 21. Tag lookup power overhead of the proposed schemes relative to the baseline model.

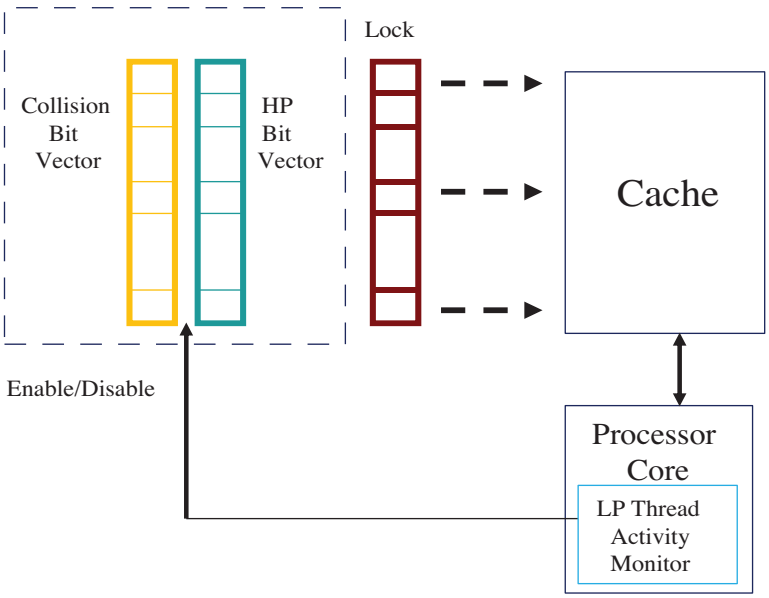

Figure 22. HPAL collision bit vector hybrid policy.

5. Conclusion

Two simple hardware schemes for detecting thread interferences in a shared cache of a real-time multithreaded embedded processor have been presented. First, we have proposed a novel scheme called collision tag, which keeps intuitive information about the evicted real-time thread cache line by the LP thread. Next, the collision bit vector scheme has been proposed to provide a low-cost and more power-efficient solution evolving from the collision tag scheme. These 2 schemes improve the overall throughput of the multithreaded processor by allowing LP threads to progress without impacting the criticality of the real-time thread, and the collision bit vector scheme provides area and power advantages over the collision tag scheme without comprising the performance of the real-time multithreaded processor core.

Although the schemes have been evaluated in the context of a L1 cache, they can be applied to other levels of shared caches in the system. In fact, the schemes can also be applied to the processors that are not multithreaded. For example, a multicore system could run the real-time thread in one of the cores and the rest of the cores could run the LP tasks, but all of the cores share a L2 cache. In this particular case, the collision tag/bit vector and dual-mode hybrid policies can efficiently deal with thread collisions in the shared L2 cache.

References

[1] D. Tullsen, S.J. Eggers, H.M. Levy, "Simultaneous multithreading: maximizing on chip parallelism", Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 392–403, 1995.

[2] J. Kihm, A. Settle, A. Janiszewski, D.A. Connors, "Understanding the impact of inter-thread cache interference on ILP in modern SMT processors", Journal of Instruction Level Parallelism, Vol. 7, pp. 1–28, 2005.

[3] V. Čakarević, P. Radojković, J. Verdú, A. Pajuelo, F.J. Cazorla, M. Nemirovsky, M. Valero, "Characterizing the resource-sharing levels in the UltraSPARC T2 processor", Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009.

[4] S.E. Raasch, S.K. Reinhardt, "Applications of thread prioritization in SMT processors", Proceedings of Multithreaded Execution, Architecture and Compilation Workshop, 1999.

[5] G.K. Dorai, D. Yeung, "Transparent threads: resource sharing in SMT processors for high single-thread performance", Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques, pp. 30–41, 2002.

[6] F.J. Cazorla, P.M. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramirez, M. Valero, "Predictable performance in SMT processors", Proceedings of the 1st Conference on Computing Frontiers, pp. 433–443, 2004.

[7] N. Yamasaki, I. Magaki, T. Itou, "Prioritized SMT architecture with IPC control method for real-time processing", 13th IEEE Real Time and Embedded Technology and Applications Symposium, pp. 12–21, 2007.

[8] R. Jain, C.J. Hughes, S.V. Adve, "Soft real-time scheduling on simultaneous multithreaded processors", Proceedings of the 23rd IEEE Real-Time Systems Symposium, pp. 134–145, 2002.

[9] A. El-Haj-Mahmoud, A.S. Al-Zawawi, A. Anantaraman, E. Rotenberg, "Virtual multiprocessor: an analyzable, high-performance microarchitecture for real-time computing", Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 213–224, 2005.

[10] E. Özer, A. Reid, S. Biles, "Low-cost techniques for reducing branch context pollution in a soft real-time embedded multithreaded processor", 19th International Symposium on Computer Architecture and High Performance Computing, pp. 37–44, 2007.

[11] E. Özer, S. Biles, "Thread priority-aware random replacement in TLBs for a high-performance real-time SMT processor", Proceedings of the 12th Asia-Pacific Computer Systems Architecture Conference, pp. 376–386, 2007.

[12] E. Özer, R. Dreslinski, T. Mudge, S. Biles, K. Flautner, "Energy-efficient simultaneous thread fetch from different cache levels in a soft real-time SMT processor", Proceedings of the 8th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, pp. 12–22, 2008.

[13] J. Mische, I. Guliashvili, S. Uhrig, T. Ungerer, "How to enhance a superscalar processor to provide hard real-time capable in-order SMT", 23rd International Conference on Architecture of Computing Systems, pp. 2–4, 2010.

[14] A. Arnaud, I. Puaut, "Dynamic instruction cache locking in hard real-time systems", IEEE Proceedings of the 14th International Conference on Real-Time and Network Systems, 2006.

[15] ARM, "ARM 940T^{TM} Technical Reference Manual", http://www.infocenter.arm.com, 2000.

[16] ARM, "ARM1136JF-S^{TM} and ARM1136J-S^{TM} Technical Reference Manual", http://www.infocenter.arm.com, 2009.

[17] EEMBC Benchmark Suite, http://www.eembc.org/, 2012.

[18] EEMBC, "Networking™ 2.0 Benchmarks", http://www.eembc.org/benchmark/networking2_sl.php, 2012.

[19] EEMBC, "AutoBench™ 1.1 Benchmarks", http://www.eembc.org/benchmark/automotive_sl.php, 2012.

[20] EEMBC, "ConsumerBench™ 1.1 Benchmarks", http://www.eembc.org/benchmark/consumer_sl.php, 2012.

[21] EEMBC, "TeleBench™ 1.1 Benchmarks", http://www.eembc.org/benchmark/telecom_sl.php, 2012.

[22] EEMBC, "DENBench™ 1.0 Benchmarks", http://www.eembc.org/benchmark/digital_entertainment_sl.php, 2012.

[23] ARM, "ARM Embedded Memory Compilers", http://www.arm.com/products/physical-ip/embedded-memory-ip/sram.php, 2012.