Energy-Efficient Simultaneous Thread Fetch from Different Cache Levels in a Soft Real-Time SMT Processor

Emre Özer[1], Ronald G. Dreslinski[2], Trevor Mudge[2], Stuart Biles[1], and Krisztián Flautner[1]

[1] ARM Ltd., Cambridge, UK
[2] Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, US
emre.ozer@arm.com, rdreslin@umich.edu, tnm@eecs.umich.edu, stuart.biles@arm.com, krisztian.flautner@arm.com

Abstract. This paper focuses on the instruction fetch resources in a real-time SMT processor to provide an energy-efficient configuration for a soft real-time application running as a high priority thread as fast as possible while still offering decent progress in low priority or non-real-time thread(s). We propose a fetch mechanism, *Fetch-around*, where a high priority thread accesses the L1 ICache, and low priority threads directly access the L2. This allows both the high and low priority threads to simultaneously fetch instructions, while preventing the low priority threads from thrashing the high priority thread's ICache data. Overall, we show an energy-performance metric that is 13% better than the next best policy when the high performance thread priority is 10x that of the low performance thread.

Keywords: Caches, Embedded Processors, Energy Efficiency, Real-time, SMT.

1 Introduction

Simultaneous multithreading (SMT) techniques have been proposed to increase the utilization of core resources. The main goal is to provide multiple thread contexts from which the core can choose instructions to be executed. However, this comes at the price of a single thread's performance being degraded at the expense of the collection of threads achieving a higher aggregate performance. Previous work has focused on the techniques to provide each thread with a fair allocation of shared resources. In particular, the instruction fetch bandwidth has been the focus of many papers, and a round-robin policy with directed feedback from the processor [1] has been shown to increase fetch bandwidth and overall SMT performance.

Soft real-time systems are systems which are not time-critical [2], meaning that some form of quality is sacrificed if the real-time task misses its deadline. Examples include real audio/video players, tele/video conferencing, etc. where the sacrifice in quality may come in the form of a dropped frame or packet.

An soft real-time SMT processor is asymmetric in nature that one thread is given higher priority for the use of shared resources, which becomes the real-time thread, and the rest of the threads in the system are low-priority threads. In this case, implementing thread fetching with a round-robin policy is a poor decision. This type of policy will degrade the performance of the high priority (HP) thread by lengthening its execution time. Instinctively, a much better solution would be to assign the full fetch bandwidth to the HP thread at every cycle, and the low priority (LP) threads can only fetch when the HP thread stalls for data or control dependency, as was done in as done in [3], [4] and [5]. This allows the HP thread to fetch without any interruption by the LP threads. On the other hand, this policy can adversely affect the performance of the LP threads as they fetch and execute instructions less frequently. Thus, the contribution of the LP threads to the overall system performance is minimal.

In addition to the resource conflict that occurs for the fetch bandwidth, L1 instruction cache space is also a critical shared resource. As threads execute they compete for the same ICache space. This means that with the addition of LP threads to a system, the HP thread may incur more ICache misses and a lengthened execution time. One obvious solution to avoid the fetch bandwidth and cache space problems would be to replicate the ICache for each thread, but this is neither a cost effective nor power efficient solution. Making the ICache multi-ported [6,7] allows each thread to fetch independently. However, multi-ported caches are known to be very energy hungry and do not address the cache thrashing that occurs. An alternative to multi-porting the ICache, would be to partition the cache into several banks and allow the HP and LP threads to access independent banks [8]. However, bank conflicts between the threads still needs to be arbitrated and cache thrashing still occurs.

Ideally, a soft real-time SMT processor would perform the best if provided a system where the HP and LP threads can fetch simultaneously and the LP threads do not thrash the ICache space of the HP thread. In this case the HP thread is not delayed by the LP thread, and the LP threads can retire more instructions by fetching in parallel to the HP thread. In this paper, we propose an energy-efficient SMT thread fetching mechanism that fetches instructions from different levels of the memory hierarchy for different thread priorities. The HP thread always fetches from the ICache and the LP thread(s) fetch directly from the L2. This benefits the system in 3 main ways: a) The HP and LP threads can fetch simultaneously, since they are accessing different levels of the hierarchy, thus improving LP thread performance. b) The ICache is dedicated to the use of the HP thread, avoiding cache thrashing from the LP thread, which keeps the runtime low for the HP thread. c) The ICache size can be kept small since it only needs to handle the HP thread. Thus reducing the access energy of the HP thread providing an energy-efficient solution.

Ultimately, this leads to a system with an energy performance that is 13% better than the next best policy with the same cache sizes when the HP thread has 10x the priority of the LP thread. Alternatively, it achieves the same performance while requiring only a quarter to half of the instruction cache space. The only additional hardware required to achieve this is a private bus between the fetch engine and the L2 cache, and a second instruction address calculation unit.

The organization of the paper is as follows: **Section 2** gives some background on fetch mechanisms in multi-threaded processors. **Section 3** explains the details of how multiple thread instruction fetch can be performed from different cache levels. **Section 4** introduces the experimental framework and presents energy and performance results. Finally, **Section 5** concludes the paper.

2 Related Work

Static cache partitioning allocates the cache ways among the threads so that each thread can access its partition. This may not be an efficient technique for L1 caches in which the set associativity is 2 or 4 way. The real-time thread can suffer performance losses even though the majority of the cache ways is allocated to it. Also, the dynamic partitioning [9] allocates cache lines to threads according to its priority and dynamic behaviour. Their efficiency comes at a hardware complexity as the performance of each thread is tracked using monitoring counters and decision logic, which increases the hardware complexity and may not be affordable for cost-sensitive embedded processors.

There have been fetch policies proposed for generic SMT processors that dynamically allocate the fetch bandwidth to the threads so as to efficiently utilize the instruction issue queues [10,11]. However, these fetch policies do not address the problem in the context of attaining a minimally-delayed real-time thread in a real-time SMT processor.

There also have been some prior investigations on soft and hard real-time SMT processors. For instance, the HP and LP thread model is explored in [3] in the context of prioritizing the fetch bandwidth among threads. Their proposed fetch policy is that the HP thread has priority for fetching first over the LP threads, and the LP threads can only fetch when the HP thread stalls. Similarly, [4] investigates resource allocation policies to keep the performance of the HP thread as high as possible while performing LP tasks along with the HP thread. [12] discusses a technique to improve the performance by keeping its IPC of HP thread in an SMT processor under OS control. A similar approach is taken by [13] in which the IPC is controlled to guarantee the real-time thread deadlines in an SMT processor. [14] investigates efficient ways of co-scheduling threads into a soft real-time SMT processor. Finally, [15] presents a virtualized SMT processor for hard real-time tasks, which uses scratchpad memories rather than caches for deterministic behavior.

3 Simultaneous Thread Instruction Fetch Via Different Cache Levels

3.1 Real-Time SMT Model

Although the proposed mechanism is valid for any real-time SMT processor supporting one HP thread and many other LP threads, we will focus on a dual-thread real-time SMT processor core supporting one HP and one LP thread.

Figure 1a shows the traditional instruction fetch mechanism in a multi-threaded processor. Only one thread can perform an instruction fetch at a time. In a real-time SMT processor, this is prioritized in a way that the HP thread has the priority to perform the instruction fetch over the LP thread. The LP thread performs instruction fetch only when the HP thread stalls. This technique will be called *HPFirst*, and is the baseline for all comparisons that are performed.

3.2 Fetch-Around Mechanism

We propose an energy-efficient multiple thread instruction fetching mechanism for a real-time SMT processor as shown in Figure 1b. The HP thread always fetches from the ICache and the LP thread directly fetches from the L2 cache. This is called the *Fetch-around* instruction fetch mechanism because the LP thread fetches directly from L2 cache passing around the instruction cache. When the L2 instruction fetch for LP thread is performed, the fetched cache line does not have to be allocated into the ICache and it is brought through a separate bus that connects the L2 to the core and is directly written into the LP thread Fetch Queue in the core.

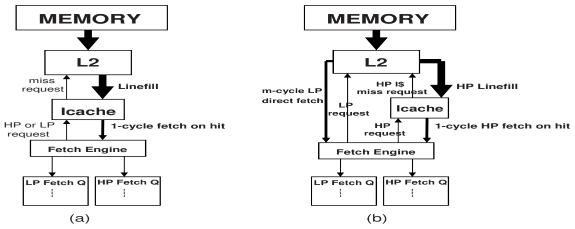

Fig. 1. Traditional instruction fetch in a multi-threaded processor (a), simultaneous thread instruction fetch at different cache levels in a soft real-time SMT processor (b)

This mechanism is quite advantageous because the LP thread is a background thread and an m-cycle direct L2 fetch can be tolerated as the HP thread is operating from the ICache. This way, the whole bandwidth of the ICache can be dedicated to the HP thread. This is very beneficial for the performance of the HP thread as the LP thread(s) instructions do not interfere with the HP thread, and therefore no thrashing of HP thread instructions occurs.

The *Fetch-around* policy may also consume less energy than other fetch policies. Although accessing the L2 consumes more energy than the L1 due to looking up additional cache ways and larger line sizes, the *Fetch-around* policy only needs to read a subset of the cache line (i.e. instruction fetch width) on a L2 I-side read operation from a LP thread. Another crucial factor for cache energy reduction is that the LP thread does not use the ICache at all, and therefore does not thrash the HP thread in the ICache. This will reduce the traffic of the HP thread to the L2 cache, and provide a higher hit rate in the more energy efficient ICache. Furthermore, the energy consumed by allocating L2 cache lines into the ICache is totally eliminated for the LP thread(s). Since the number of HP thread instructions is significantly larger than the LP, the energy savings of the HP thread in the ICache outweighs that of the LP threads increase in L2 energy.

In addition to its low energy consumption capability, the *Fetch-around* policy has the advantage of not requiring a large ICache for an increased number of threads. Since the ICache is only used by the HP thread, additional threads in the system put no more demands on the cache, and the performance remains the same as single threaded version. It is possible that a fetch policy such as round-robin may need twice the size of the ICache to achieve the same HP thread performance level as the *Fetch-around* policy in order to counteract the thrashing effect. Thus, the *Fetch-around* policy is likely to reduce the ICache size requirements, and therefore the static and dynamic ICache energy.

It takes approximately m-cycles (i.e. the L2 access time) to bring the LP thread instructions to the core from L2. This effectively means that the LP thread is fetched at every m cycles. One concern is the cost of the direct path between the L2 and ICache. This path does not have to be an L2 cache line size in width since the bus connects directly to the core and only need deliver the fetch width (2 instructions).

4 Energy and Performance Results

4.1 Experimental Framework

We have performed a cycle-accurate simulation of an SMT implementation of an ARMv7 architecture-compliant processor using the EEMBC benchmark suite [16]. We have used 24 benchmarks from the EEMBC benchmark suite covering a wide range of embedded applications including consumer, automotive, telecommunications and DSP. We run all possible dual-thread permutations of these benchmarks (i.e. 576 runs). A dual-thread simulation run completes when the HP thread finishes its execution, and then we collect statistics such as total IPC, degree of LP thread progress, HP thread speedup and etc. We present the average of these statistics over all runs in the figures.

The simulated processor model is a dual-issue in-order superscalar dual-thread SMT processor core with 4-way 1KB Icache, 4-way 8KB Dcache, and 8-way 16KB L2 cache. The hit latency is 1 cycle for L1 caches and 8 cycles for the L2 cache, the memory latency is 60 cycles and the cache line size is 64B for all caches. There is a 4096-entry global branch predictor with a shared branch history buffer and a replicated global branch history register for each thread, 2-way set associative 512-entry branch target buffer, and 8-entry replicated return address stack for each thread. The ICache delivers 2 32-bit instructions to the core per instruction fetch request. We used two thread fetch select policies: *Fetch-around* and *HPFirst*. *HPFirst* is the baseline fetch policy in which only one thread can fetch at a time, and the priority is always given to the HP thread first. There are two decoders in the decode stage that can decode up to instructions, and the HP thread has the priority over the LP thread to use the two decoders. If the HP thread instruction fetch queue is empty, then the LP thread instructions, if any, are decoded. Similarly, the HP thread has the priority to use the two issue slots. If it can issue only 1 instruction or cannot issue at all, then the LP thread is able to issue 1 or 2 instructions.

Most of the EEMBC benchmarks can fit into 2-to-8KB instruction cache. Thus, we deliberately select a very small instruction cache size (i.e. 1KB) to measure the effect of instruction cache stress. The L2 line size is 512 bits and the L1 instruction fetch width is 64 bits. From L2 to L1 ICache, a line size of 512 bits (i.e. 8 64 bits) are allocated on an ICache miss. ICache contains 4 banks or ways, and each bank consists of 2 sub-banks of 64 bits, so 8 sub-banks of 64 bits comprise a line of 512 bits. When an ICache linefill is performed, all sub-banks tag and data banks are written. We model both ICache and L2 cache as serial access caches meaning that the selected data bank is sense-amplified only after a tag match.

4.2 Thread Performance

We have measured 2 metrics to compare these fetch policies:

1. Slowdown in terms of execution time of the highest priority thread relative to itself running on the single-threaded processor,
2. Slowdown in terms of CPI of the lowest priority thread. As the HP thread has the priority to use all processor resources,

Sharing resources with other LP threads lengthens the HP thread execution time, and therefore we need to measure how the HP thread execution time in the SMT mode compares against its single-threaded run. In the single-threaded run, the execution time of the HP thread running alone is measured. Ideally, we would like not to degrade the performance of the HP thread but at the same time we would like to improve the performance of the LP thread. Thus, we measure the slowdown in LP thread CPI under SMT for each configuration with respect to their single-threaded CPI. The CPI of the LP thread is measured when it runs alone.

Table 1 shows the percentage slowdown in HP thread execution time relative to its single-threaded execution time. Although the ICache is not shared among threads in *Fetch-around*, the slowdown in the HP thread by about 10% occurs due to inter-thread interferences in data cache, L2 cache, branch prediction tables and execution units. On the other hand, the HP thread slowdown is about 13% in *HPFirst*. Since *Fetch-around* is the only fetch policy that does not allow the LP thread to use the ICache, the HP thread has the freedom to use the entire ICache and does not encounter any inter-thread interference.

Table 1 also shows the progress of the LP thread under the shadow of the HP thread measured in CPI. The progress of the LP thread is the slowest in *Fetch-around* as expected because the LP thread fetches instructions from L2, which is 8-cycles away from the core. *HPFirst* has better LP thread performance as LP

Table 1. Percentage slowdown in HP thread, and the progress of the LP thread

	Single-thread	**HPFirst**	**Fetch-around**
Percentage slowdown in HP	N/A	12.7%	9.5%
The LP CPI	1.6	3.8	5.1

thread instructions are being fetched from the ICache in a single cycle access. However, this benefit comes at the price of evicting HP thread instructions from the ICache due to interthread interference and increasing the HP thread runtime.

4.3 Area Efficiency of the Fetch-Around Policy

We take a further step by increasing the ICache size from 1KB to 2KB and 4KB for *HPFirst* and compare its performance to *Fetch-around* using only a 1KB instruction cache. Table 2 shows that *Fetch-around* using only a 1KB instruction cache still outperforms the other policies having 2 and 4KB ICache sizes. In addition to *Fetch-around* and *HPFirst* fetch policies, we also include the round-robin (RR) fetch policy for illustration purposes where the threads are fetched in a round-robin fashion even though it may not be an appropriate fetch technique for a real-time SMT processor. Although some improvement in HP thread slowdown (i.e. drop in percentage) is observed in these 2 policies when the ICache size is doubled from 1KB to 2KB, and quadrupled to 4KB, it is still far from being close to 9.5% in *Fetch-around* using 1KB ICache. Thus, these policies suffer a considerable amount of inter-thread interference in the ICache even when the ICache size is quadrupled. Table 3 supports this argument by showing the HP thread instruction cache hit rates. As the ICache is only used by the HP thread in *Fetch-around*, its hit rate is exactly the same as the hit rate of the single-thread model running only the HP thread. On the other hand, the hit rates in *HPFirst* and *RR* are lower than *Fetch-around* because both policies observe the LP thread interfering and evicting the HP thread cache lines. These results suggest that *Fetch-around* is much more area-efficient than the other fetch policies.

Table 2. Comparing the HP thread slowdown of *Fetch-around* using only 1KB instruction cache to *HPFirst* and *RR* policies using 2KB and 4KB instruction caches

Fetch-around 1K	**HPFirst 2K**	**HPFirst 4K**	**RR 2K**	**RR 4K**
9.5%	12.3%	11.7%	17.7%	17.2%

Table 3. HP Thread ICache hit rates

HPFirst	**Fetch-around**	**RR**
98.6%	97.6%	95.4%

4.4 Iside Dynamic Cache Energy Consumption

For each fetch policy, the dynamic energy spent in the Iside of the L1 and L2 caches is calculated during instruction fetch activities. We call this *Iside dynamic cache energy*. We measure the Iside dynamic cache energy increase in each fetch policy relative to the Iside dynamic energy consumed when the HP thread runs alone. We use Artisan 90nm SRAM [17] library to model tag and data RAM read and write energies for L1I and L2 caches.

Table 4. Percentage of Iside cache energy increase with respect to the HP thread running in single-threaded mode for 1KB instruction cache

HPFirst	**Fetch-around**	**RR**
75.2%	47%	75%

Table 4 shows the percentage energy increase in the Iside dynamic cache energy relative to the energy consumed when the HP thread runs alone. Although accessing the L2 consumes more power than the L1 due to looking up more ways and reading a wider data width (i.e. 512 bits), *Fetch-around* consumes less L2 energy than normal L2 I-side read operations by reading only 64-bits (i.e. instruction fetch width) for the LP threads. *Fetch-around* also reduces the L2 energy to some degree as the LP thread does not thrash the HP thread in the ICache, reducing the HP thread miss rate compared to *HPFirst*. This smaller miss rate translates to less L2 accesses from the HP thread, and a reduction in L2 energy. Besides, *Fetch-around* also eliminates ICache tag comparisons and dataRAM read energy for the LP thread. And further saves ICache line allocation energy by bypassing the ICache allocation for the LP thread. *Fetch-around* consumes the least amount of energy among all fetch policies at the expense of executing fewer LP thread instructions. This fact can be observed more clearly if the individual energy consumption per instruction of each thread is presented.

Table 5. Energy per Instruction (uJ)

uJ/Inst	**HPFirst**	**Fetch-around**	**RR**
HP Thread	34.3	28.8	34.3
LP Thread	55.3	72.6	47.8

Table 5 presents the energy consumption per HP and LP threads separately. *Fetch-around* consumes the least amount of energy per HP thread instruction even though the execution of an LP thread instruction is the most energy-hungry among all fetch policies. As the number of HP thread instructions dominate the number of LP thread instructions, having very low energy-per-HP-instruction causes the *Fetch-around* policy to obtain the lowest overall Iside cache energy consumption levels. *HPFirst* and *RR* have about the same energy-per-HP-instruction while *RR* has lower energy-per-LP-instruction than *HPFirst*. *RR* retires more LP thread instructions than *HPFirst*, and this behavior (i.e. *RR* retiring a high number of low-energy LP thread instructions and *HPFirst* retiring a low number of high-energy LP thread instructions) brings the total Iside cache energy consumption of both fetch policies to the same level.

4.5 Energy Efficiency of the Fetch-Around Policy

The best fetch policy can be determined as the one that gives higher performance (i.e. low HP thread slowdown and low LP thread CPI) and lower Iside cache energy consumption, and should minimize the product of the thread performance and Iside cache energy consumption overheads. The thread performance overhead is calculated as the weighted mean of the normalized HP Execution Time and LP Thread CPI as these two metrics contribute at different importance weights or degrees of importance into the overall performance of the real-time SMT processor. Thus, we introduce two new qualitative parameters called *HP thread degree of importance* and *LP thread degree of importance*, which can take any real number. When these two weights are equal, this means that the performance of both threads is equally important. If the HP thread degree of importance is higher than the LP thread degree of importance, the LP thread performance is sacrificed in favor of attaining higher HP thread performance. For a real-time SMT system, the HP thread degree of importance should be much greater than the LP thread degree of importance. HP Execution Time, LP Thread CPI, and Iside Cache Energy are normalized by dividing each term obtained in SMT mode by the equivalent statistic obtained when the relevant thread runs alone. The Iside Cache Energy is normalized to the Iside cache energy consumption value when the HP thread runs alone. These normalized values are always greater than 1 and represent performance and energy overhead relative to the single-thread version.

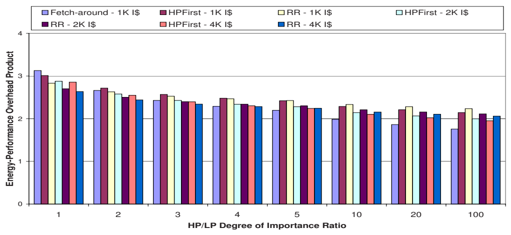

Fig. 2. Comparison of the energy-performance overhead products

Figure 2 presents the energy-performance overhead products for all fetch policies using 1KB instruction cache. The x-axis represents the ratio of the HP thread degree of importance to the LP thread degree of importance. In addition to this, the figure shows the overhead product values for *HPFirst* and *RR* policies using 2KB and 4KB instruction caches. When the ratio is 1, both threads are equally important, and there is no real advantage of using *Fetch-around* as it has the highest energy-performance overhead product. When the ratio becomes about 3, *Fetch-around* has lower overhead product than the other two policies using the same size ICache. In fact, it is even slightly better than *HPFirst* using 2KB ICache. When the ratio is 5 and above, not only *Fetch-around* is more energy-efficient than *HPFirst* and *RR* using the same ICache size but also better than *HPFirst* and *RR* using 2KB and 4KB ICaches. When it becomes 10, *Fetch-around* is 13% and 15% more efficient than *HPFirst* and *RR* for the same ICache size. When the ratio ramps up towards 100, the energy-efficiency of *Fetch-around* increases significantly. For instance, it becomes from 10% to 21% more efficient that the other two policies with equal and larger ICaches when the ratio is 100.

5 Conclusion

We propose a new SMT thread fetching policy to be used in the context of systems that have priorities associated with threads, i.e. soft real-time applications like real audio/video and tele/video conferencing. The proposed solution, *Fetch-around*, has high priority threads access the ICache while requiring low priority threads to directly access the L2 cache. This prevents the low priority threads from thrashing the ICache and degrading the performance of the high priority thread. It also allows the threads to simultaneously fetch instructions, improving the aggregate performance of the system. When considering the energy performance of the system, the *Fetch-around* policy does 13% better than the next best policy with the same cache sizes when the priority of the high performance thread is 10x that of the low priority thread. Alternatively, it achieves the same performance while requiring only a quarter to half of the instruction cache space.

References

1. Tullsen, D., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: Maximizing on-chip parallelism. In: Proceedings of the 22nd Annual Intl. Symposium on Computer Architecture (June 1995)
2. Brandt, S., Nutt, G., Berk, T., Humphrey, M.: Soft real-time application execution with dynamic quality of service assurance. In: Proceedings of the 6th IEEE/IFIP International Workshop on Quality of Service (May 1998)
3. Raasch, S.E., Reinhardt, S.K.: Applications of thread prioritization in smt processors. In: Proceedings of Multithreaded Execution, Architecture and Compilation Workshop (January 1999)
4. Dorai, G.K., Yeung, D.: Transparent threads: Resource sharing in smt processors for high single-thread performance. In: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (2002)
5. Yamasaki, N.: Responsive multithreaded processor for distributed real-time processing. Journal of Robotics and Mechatronics, 44–56 (2006)
6. Falcón, A., Ramirez, A., Valero, M.: A low-complexity, high-performance fetch unit for simultaneous multithreading processors. In: Proceedings of the 10th Intl. Conference on High Performance Computer Architecture (February 2004)

7. Klauser, A., Grunwald, D.: Instruction fetch mechanisms for multipath execution processors. In: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (November 1999)
8. Burns, J., Gaudiot, J.L.: Quantifying the smt layout overhead, does smt pull its weight? In: Proc. Sixth Int'l Symp. High Performance Computer Architecture (HPCA) (January 2000)
9. Suh, G., Devadas, S., Rudolph, L.: Dynamic cache partitioning for simultaneous multithreading systems. In: The 13th International Conference on Parallel and Distributed Computing System (PDCS) (August 2001)
10. Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L.: Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In: Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA) (May 1996)
11. El-Moursy, A., Albonesi, D.H.: Front-end policies for improved issue efficiency in smt processors. In: Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA) (February 2003)
12. Cazorla, F.J., Knijnenburg, P.M., Sakellariou, R., Fernández, E., Ramirez, A., Valero, M.: Predictable performance in smt processors. In: Proceedings of the 1st Conference on Computing Frontiers (April 2004)
13. Yamasaki, N., Magaki, I., Itou, T.: Prioritized smt architecture with ipc control method for real-time processing. In: 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS 2007), pp. 12–21 (2007)
14. Jain, R., Hughes, C.J., Adve, S.V.: Soft real-time scheduling on simultaneous multithreaded processors. In: Proceedings of the 23rd IEEE Real-Time Systems Symposium (December 2002)
15. El-Haj-Mahmoud, A., AL-Zawawi, A.S., Anantaraman, A., Rotenberg, E.: Virtual multiprocessor: An analyzable, high-performance microarchitecture for real-time computing. In: Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2005) (September 2005)
16. EEMBC, `http://www.eembc.com`
17. Artisan, `http://www.arm.com/products/physicalip/productsservices.html`