Achieving Predictable Performance in SMT Processors by Instruction Fetch Policy

Caixia Sun, Yongwen Wang, and Jinbo Xu

School of Computer, National University of Defense Technology,
Changsha 410073, Hunan, P.R. China
cxsun@nudt.edu.cn

Abstract. With the applications in embedded systems increasingly complex, future embedded processors will resemble current high performance general purpose processors. Simultaneous multithreading (SMT) is a good choice in embedded processors for its good cost-performance tradeoff. However, in SMT processors, the execute time of a thread is unpredictable. The unpredictability is an undesirable feature in embedded systems. In order to apply SMT architecture to embedded processors, the problem of performance unpredictability must be addressed. Among the current researches, a noted one is done by Cazorla et al (we call it Cazorla policy). However, Cazorla policy achieves predictable performance for a time critical thread by shared resources reservation, which weakens the advantage of resources sharing in SMT processors.

In this paper, we propose a novel instruction fetch policy called APP (Achieving Predictable Performance) to control the performance of a time critical thread in SMT processors. Simulation results show that APP can achieve predictable performance for the time critical thread as effectively as Cazorla policy does. Furthermore, APP can make full use of shared resources more effectively to optimize the performance of other co-scheduled threads and overall throughput. Compared with Cazorla policy, overall throughput obtained by APP is increased by 4.9% on average and the performance of other co-scheduled threads is increased by 17.6%.

Keywords: Simultaneous Multithreading, Instruction Fetch Policy, Predictable Performance.

1 Introduction

Applications in embedded systems are increasingly complex, which places an increasingly demand on the performance of embedded processors. To meet these growing demands, future embedded processors will resemble current high performance general purpose processors. How to make general purpose processors suitable for the embedded systems are studying [1–5].

Embedded processors differ from general purpose processors in their concentration on low cost. That is, embedded processors hope to obtain as much performance as possible from each resource. Hence, simultaneous multithreading

W. Xu et al. (Eds.): NCCET 2013, CCIS 396, pp. 186–197, 2013.

(SMT) [6–8] architecture is a good option for embedded processors. In SMT processors, multiple threads share hardware resources, and a good cost-performance trade-off can be achieved.

However, co-scheduled threads in SMT processors compete for shared resources. Different threads have different competition abilities. When a thread is co-scheduled with different threads, its performance will be varied.

Figure 1 shows IPC (Instructions Per Cycle) of crafty (a benchmark from SPEC2000) when it runs alone and is co-scheduled with different threads. For multithreaded workloads, ICOUNT [7] fetch policy is used. We can see that the performance of crafty varies with the workload it is executed in.

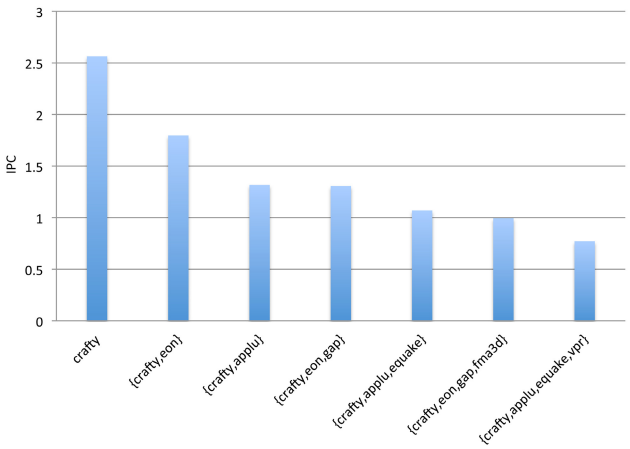

Fig. 1. IPC of crafty for different workloads

As a consequence, in SMT processors, the execute time of a thread is unpredictable. The unpredictability is an undesirable feature in embedded systems. In order to apply SMT architecture to embedded processors, the problem of performance unpredictability must be addressed.

There are few researches that address this problem. Among the current researches, a noted one is done by Cazorla et al. [9–13]. They proposed a hardware mechanism to run a given thread at a desired speed. We call this mechanism Cazorla policy. In Cazorla policy, shared resources are reserved for the time critical thread, and other co-scheduled threads can not occupy the reserved resources, consequently guaranteeing the time critical thread can achieve desired performance. In Cazorla policy, shared resources are allocated explicitly and co-scheduled threads can not compete for resources freely, leading to that the performance of other threads and overall throughput are affected. We will discuss it in detail in section 2.

In this paper, we propose a novel instruction fetch policy to control the performance of a time critical thread in SMT processors. Different from Cazorla policy, our policy allocates shared resources implicitly by fetch control. The goal

of our policy is to ensure that the time critical thread can achieve desired performance regardless of the workload it is executed in, at the same time to make full use of shared resources to maximize the performance of other threads and overall throughput.

The rest of the paper is organized as follows. Section 2 introduces Cazorla policy, and discusses it. In Section 3, we detail our new fetch policy and describe how to implement predictable performance for a particular thread by fetch control. Section 4 presents the methodology and Section 5 illustrates the results. Finally, concluding remarks are given in Section 6.

2 Cazorla Policy

Give a workload of N threads and a time critical thread in this workload. The time critical thread is called High Priority Thread (HPT) and other threads are called Low Priority Threads (LPTs) [12]. The goal of Cazorla policy is to ensure that HPT runs at a given target IPC that represents X% of IPC_{alone}. IPC_{alone} is the IPC of HPT when it would run alone on the machine [12].

Cazorla policy is a dynamic resources allocation mechanism, which dynamically adjusts the reserved resources for HPT according to its real performance. Cazorla policy employs two phases:

During the first phase, the sample phase, the processor runs in single-thread mode. HPT runs alone for a certain time. As a result, IPC_{alone} of HPT can be obtained. The target IPC would be achieved correspondingly.

During the second phase, the tune phase, the amount of shared resources dedicated to HPT is varied according to the real IPC of HPT. If the real IPC is lower than the target one, increase the amount of resources deserved for HPT. Otherwise, the amount of resources given to HPT is decreased.

Cazorla policy can implement predictable performance for HPT. However, there are two problems.

- Firstly, HPT achieves the desired performance by shared resources reservation. The reserved resources can only be used by HPT. Co-scheduled threads cannot compete for resources freely. Consequently, the advantage of resources sharing in SMT processors is weakened.
- Secondly, when Cazorla policy adjusts resources allocation every time, the amount of physical registers (integer and floating point) and issue queues (integer, floating point and load/store) given to HPT is changed at the same time. For example, if the real IPC of HPT is lower than the target one, physical registers and issue queues are all increased by a certain amount. In fact, the reason that HPT is incapable of achieving desired performance maybe lack integer registers and integer issue queue, and there is no need to increase the amount of other resources. As a result, resource under-use exists in Cazorla policy.

3 Achieving Predictable Performance by Instruction Fetch Policy

In this section, we introduce our instruction fetch policy. To be simple, in the next of our paper, we call the proposed policy APP (Achieving Predictable Performance). Same to Cazorla policy, the goal of APP is also to ensure that HPT achieves the target IPC, and to implement performance predictability. At the same time, APP tries to make full use of shared resources to maximize the performance of other co-scheduled threads and overall throughput.

3.1 Basic Idea

The basic idea of our policy is to compare the real IPC and the target IPC for HPT, and to adjust the fetch priority of HPT based on the comparison result.

We use ICOUNT2.8 as the default fetch policy. That is, co-scheduled threads are ordered by ICOUNT, the number of threads that can fetch in one cycle is 2, and the maximum number of instructions fetched per thread in one cycle is 8. Furthermore, we define two new policies: PHPT and PLPT.

- **PHPT:** Prioritizing HPT. That is to say, the HPT has the highest fetch priority, and LPTs are ordered by ICOUNT.
- **PLPT:** Prioritizing LPTs. That is to say, the HPT has the lowest fetch priority, and LPTs are ordered by ICOUNT.

Let IPC_{dsr} denote the target IPC of HPT, IPC_{real} denote the real IPC of HPT, and DIPC denote the difference between IPC_{dsr} and IPC_{real}. D_{IPC} is given by equation (1).

$$D_{IPC} = \frac{IPC_{real} - IPC_{dsr}}{IPC_{dsr}} \times 100\% \tag{1}$$

Our policy switches between PHPT2.8, ICOUNT2.8 and PLPT2.8 according to the value of D_{IPC}. If D_{IPC} is smaller that a threshold defined as $Th_{PHPT2.8}$, which means that the HPT has not achieved its desired performance, we use PHPT2.8 to accelerate the execution of HPT. If D_{IPC} is bigger than a threshold defined as $Th_{PLPT2.8}$, PLPT2.8 is used to maximize the performance of LPTs. Otherwise, if D_{IPC} is between $Th_{PHPT2.8}$ and $Th_{PLPT2.8}$, ICOUNT2.8 is used, allowing all co-scheduled threads to compete for shared resources freely and increasing overall throughput.

When PHPT2.8 is used to accelerate the execution of HPT, HPT may not obtain desired performance yet, especially when the target IPC is very high. The reason is that PHPT2.8 fetches instructions from two threads in one cycle. Although HPT has the highest priority, LPTs can still fetch instructions and occupy shared resources, which may cause that HPT has not enough resources to achieve desired performance. So we define a new threshold $Th_{PHPT1.8}$. When D_{IPC} is smaller than $Th_{PHPT1.8}$, PHPT1.8 is used to prevent LPTs from occupying more resources. PHPT1.8 means that only one thread (that is, HPT) can fetch instructions in a cycle.

Figure 2 shows how APP adjusts fetch policy according to the value of D_{IPC}.

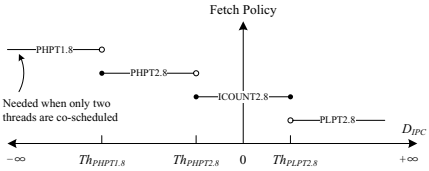

Fig. 2. The relationship between fetch policy and D_{IPC} in APP

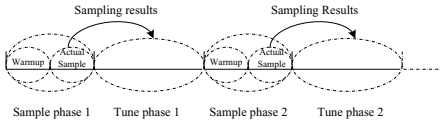

Fig. 3. Sample phase and tune phase in an alternate fashion

3.2 Implementation

To switch between different fetch policies and adjust the fetch priority of the HPT, D_{IPC} must be known, that is, IPC_{dsr} and IPC_{real} are needed. Just as done in [12], we represent IPC_{dsr} as X% of IPC when the HPT runs alone on the machine. Assume that the OS has some goals and decides X% for the HPT. So the hardware needs to know IPC of the HPT when it runs alone, that is, IPC_{alone}. To get IPC_{alone} dynamically, we employ two phases: sample phase and tune phase, just as done in [12].

During the sample phase, HPT runs alone for a certain time. The sample phase is divided two periods: the first period is called warm up period, which is used to remove the pollution by the LPTs from the shared resources and to increase the accuracy of IPC_{alone}. During the second period, IPC_{alone} is measured and IPC_{dsr} is achieved correspondingly.

During the tune phase, all threads are co-scheduled. Each cycle, IPC_{real} and D_{IPC} is re-calculated for HPT. The fetch priority of the HPT is adjusted according to the value of D_{IPC}.

A key point must be considered. Programs experience different phases in their execution in which their IPC varies significantly [14]. Hence, if we want to realize X% of the overall IPC for HPT, we need take into account this variable IPC. Our solution is to execute sample phase and tune phase in an alternate fashion, just as shown in Figure 3.

From the description above, other than switching threshold, three additional parameters are needed to be defined, which are: Lwarm-up, Lactual-sample and Ltune.

- $L_{warm-up}$**:** the length of the warm up period in the sample phase.
- $L_{actual-sample}$**:** the length of the actual sample phase.
- L_{tune}**:** the length of the tune phase.

4 Methodology

In this section, we give the simulator and benchmarks used in our experiments, the metrics employed to evaluate APP, and the values of parameters defined in APP.

4.1 Simulator

Execution is simulated on an out-of-order superscalar processor model derived from SMTSIM [15]. The simulator models all typical sources of latency, including caches, branch mispredictions, TLB misses, etc. It also carefully models execution down the wrong path between branch misprediction and branch misprediction recovery. The baseline configuration of our simulator is shown in Table 1.

Table 1. Baseline Configuration of the Simulator

Parameter	Value
Fetch	Width 8 instructions per cycle
Instruction Queues	64 int, 64 fp
Functional Units	6 int (4 load/store), 3 fp
Renaming Registers	100 int, 100 fp
Active List Entries	256 entries per thread
Branch Predictor	2K gshare
Branch Target Buffer	256 entries, 4-way associative
L1I cache, L1D cache	64KB, 2-way, 64-bytes lines, 1 cycle access
L2 cache	512KB, 2-way, 64-bytes lines, 10 cycles latency
L3 cache	4MB, 2-way, 64-bytes lines, 20 cycles latency
Main Memory Latency	100 cycles

4.2 Benchmarks

Table 2 summarizes the benchmarks used in our simulations. Generally, HPT would be multimedia applications. So we use MediaBench [16] (Denoted as B) as HPT. LPTs are still taken from the SPEC2000 suite [17]. SPEC2000 benchmarks are divided into two groups based on their cache behaviors: those experiencing more than 0.01 L2 cache misses per instruction, on average, over the simulated portion of the code are considered memory-intensive applications, called MEM (denoted as M), and the rest are called ILP (denoted as I), which have lower miss rates and higher inherent ILP (Instruction Level Parallelism).

Two kinds of workloads are simulated: BI and BM. In BI workloads, LPTs are all taken form ILP benchmarks. In BM workloads, LPTs are all MEM benchmarks. The number of threads included in a workload may be 2, 3 or 4. The simulation ends when HPT is finished.

4.3 Metrics

To quantify the efficiency of APP in achieving predictable performance, we use two metrics: Success Rate (SR) and Maximum Performance Variance (MPV).

If the real IPC of HPT is not less than the target one, APP is successful in achieving predictable performance. Otherwise, it fails. Success Rate is the proportion of success cases to all measured cases.

Table 2. Benchmarks

HPT	LPTs		
adpcm-encode, adpcm-decode, epic-encode, epic-decode, mpeg2-encode, mpeg2-decode	ILP	1	gzip, gap, eon, fma3d, mesa
		2	{crafty, eon}, {crafty, gzip}, {gzip, fma3d}, {eon, mesa}, {fma3d, mesa}
		3	{gzip, eon, gap}, {crafty, fma3d, gap}, {eon, fma3d, mesa}, {mesa, gap, crafty}, {gap, fma3d, mesa}
	MEM	1	twolf, vpr, swim, applu, lucas
		2	twolf, vpr, {twolf, swim}, {applu, lucas}, {vpr, lucas}, {equake, applu}
		3	{twolf, vpr, lucas}, {applu, swim, equake}, {twolf, lucas, swim}, {vpr, equake, applu}, {lucas, swim, applu}

Maximum Performance Variance is a metric for the failed cases. Supposed that X_{real} is the percentage of real IPC with respect to IPC_{alone}, and X is the target percentage. Define Performance Variance as equation (2):

$$V_{HPT} = X - X_{real} \tag{2}$$

For all failed cases, the maximum of VHPT is Maximum Performance Variance. If a policy has a Success Rate of 1, Maximum Performance Variance will be 0.

In addition, we will evaluate the performance of LPTs and overall throughput. IPC is used as the metric.

4.4 Choosing Parameter

Same to Cazorla policy, APP also employs two phases: sample phase and tune phase. When evaluating these two policies, the same length is used, as shown in Table 3.

Table 3. The Values of Phase Length

Parameter	$L_{warm-up}$	$L_{actual-sample}$	L_{tune}
Value	2^{16} cycles	2^{14} cycles	2^{22} cycles

Table 4. The Values of Switching Threshold

Parameter	$Th_{PHPT1.8}$	$Th_{PHPT2.8}$	$Th_{PLPT2.8}$
Value	-3%	-0.1%	0.1%

In APP policy, there are three parameters deciding when to switch fetch policy. Their values are shown in Table 4. These values are determined from plenty of experiments.

5 Results

We first show efficiency of APP in achieving predictable performance. Next, the total throughput and the performance of LPTs obtained by APP are compared to those under ICOUNT. At last, we compare APP policy with Cazorla policy.

5.1 Efficiency in Achieving Predictable Performance

Figure 4 shows SR and MPV results achieved by APP. On the x-axis, the target percentage of HPT is given, ranging from 10% to 90%. For each size of the workload (2, 3, or 4 threads), SR result is given on the left y-axis and MPV result is given on the right y-axis.

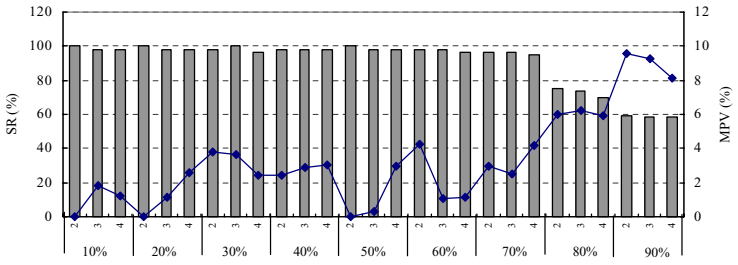

Fig. 4. SR and MPV results of APP

When the target percentage is not more than 70%, for all kinds of workloads, SR is very high, reaching 95% at worst case, and MPV is not more than 5%. However, after the target percentage reaches 80%, SR declines rapidly, and MPV reaches 9.6% for the worst case. The main reason is IPC_{alone} achieved during the actual sample phase can not exactly represent the IPC during the tune phase. If IPC_{alone} sampled is smaller than that of the tune phase, the real performance achieved in the tune phase will be lower than the target one. If IPC_{alone} is bigger, the real IPC will not always exceed the target IPC. For example, assume that IPC_{alone} is 4 and during the tune phase, the IPC of the HPT when it is runs alone is only 3.5. If the percentage is 90%, the desired IPC is 3.6. It is impossible that real IPC reaches 3.6 during the tune phase. But if the percentage is low, the real IPC can exceed the target one, and the final performance meets the target by the complementary effect.

SR is high when the target percentage is lower than 70%, but it does not reach 100%. Fortunately, from the further experiments, we found that using a target percentage higher than actually desired one by 5%, a success rate of 1 can always be achieved. This method is only effective when the target percentage is lower than 70%. When the target percentage is higher than 80%, to ensure that HPT can always achieve desired performance, running HPT alone may be a simple and effective way.

5.2 The Performance of LPTs and Overall Throughput Results

Figure 5 depicts the IPC of LPTs relative to ICOUNT and Figure 6 depicts the total IPC achieved by APP relative to ICOUNT. ICOUNT is a representative fetch policy orienting towards throughput maximization. The performance of LPTs and overall throughput are given as the percentage of those under ICOUNT.

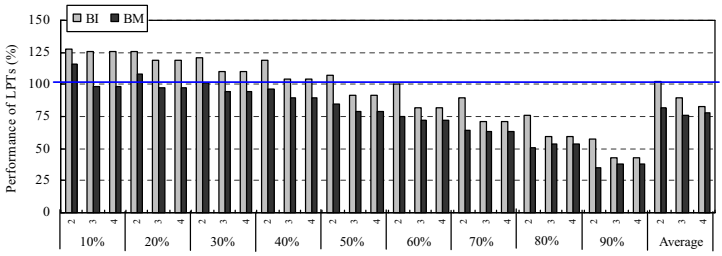

Fig. 5. Performance of LPTs relative to ICOUNT

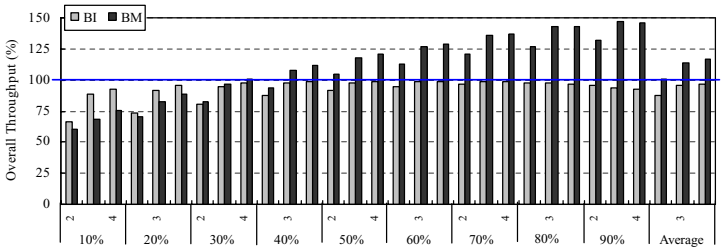

Fig. 6. Overall throughput relative to ICOUNT

For the workload of type BI, the total IPC achieved by APP is always lower than that achieved by ICOUNT. Especially, when the percentage is very slow or very high, the degradation is more severe. The main reason is ICOUNT orients towards throughput optimization of ILP workloads. However, in our policy, to achieve desired performance for the HPT, the fetch priority of HPT is forced to the highest or the lowest sometimes regardless of its real execution state. In fact, if HPT has the highest priority and occupies many resources, resources clogging may occur; if HPT can make forward progress by using the resources not required by LPTs but the HPT has the lowest priority, resources under-use happens. Compared to ICOUNT, the average degradation of overall throughput is not more then 7%.

For the workload of type BM, the total IPC becomes higher as the target percentage increases. When the percentage is very high, APP even outperforms ICOUNT. But for different sizes of workloads, the point at which APP begins to outperform ICOUNT is different. For BM2, BM3 and BM4, the points are 50%, 40% and 30%, respectively. That is to say, the smaller the size of workload is, the higher the percentage is. Now let us give the explanation. When ICOUNT is used, IPC of HPT in BM4 is the lowest, because the competition for shared resources is the most severe. By simulations, we find IPC of the HPT in BM4 under ICOUNT is only 28% of its full speed. However, for BM3 and BM2, the respective values are 33% and 45%. When the target percentage exceeds these values, IPC of the HPT achieved by APP will be higher than that obtained under ICOUNT. The increase of HPT in throughput will lead to the decrease of

LPTs in throughput. But HPT has higher inherent IPC, resulting in the total throughput is increased eventually. Compared to ICOUNT, overall throughput achieved by APP is increased by 10.5% on average.

As a whole, compared to ICOUNT, the performance of LPTs is decreased by not more than 15%, and overall throughput is even increased by 1.8% on average. It can be concluded that to achieve predictable performance for HPT, APP does not sacrifice the performance of LPTs and overall throughput severely.

5.3 Compared with Cazorla Policy

APP is same with Cazorla policy for that they all obtain the target IPC of HPT in sample phase and implement desired performance in tune phase. The difference is the way how to allocate shared resources to ensure that HPT achieves desired performance. Cazorla policy allocates shared resources explicitly by resources reservation for HPT, and APP allocates shared resources implicitly by instruction fetch policy.

In Figure 7, APP is compared with Cazorla policy in success rate and maximum performance variance. Whether in SR or MPV, the results of APP and Cazorla policy are very close. It is concluded that APP can achieve predictable performance for HPT as effectively as Cazorla policy does.

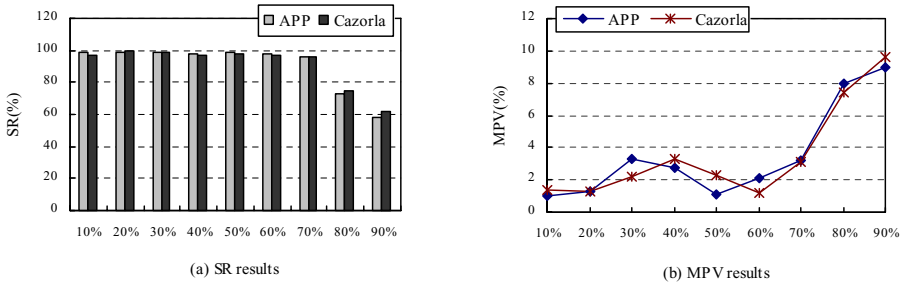

Fig. 7. SR and MPV results of APP and Cazorla policy

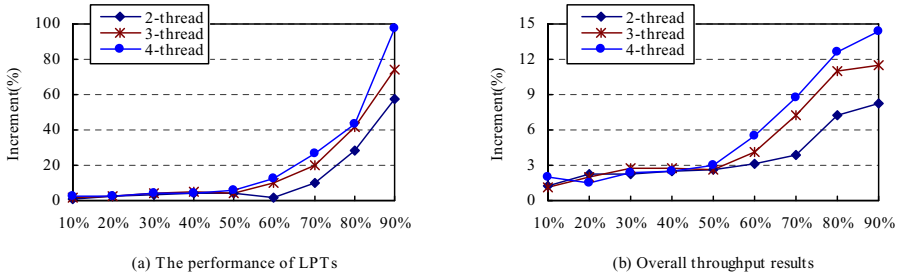

Fig. 8. The increment of APP in the performance of LPTs and overall throughput relative to Cazorla policy

Figure 8 shows the increment of APP in the performance of LPTs and overall throughput relative to Cazorla policy. Overall throughput is increased by 4.9% on average and the performance of LPTs is increased by 17.6%. The increment becomes higher as the target percentage increases. When the target percentage is 90%, for two-thread, three-thread and four-thread workloads, the performance of LPTs is increased by 57.7%, 74.5% and 97.7% respectively. Compared to Cazorla policy, APP can more effectively make full use of shared resources to optimize the performance of LPTs and overall throughput.

6 Conclusions

SMT processor is a good option in embedded systems for its good cost-performance trade-off. However, in SMT processors, the execute time of a thread is unpredictable. The unpredictability is an undesirable feature in embedded systems. In order to apply SMT architecture to embedded processors, the problem of performance unpredictability must be addressed.

In this paper, we propose a fetch policy called APP to control the performance of HPT in SMT processors. APP switches between PHPT, ICOUNT and PLPT according to the execution state of HPT, and implements predictable performance for HPT. Simulation results show that when the target percentage is not more than 70%, APP obtains a success rate of over 95%. For the failed cases, maximum performance variance is less than 5%. After the target percentage reaches 80%, SR will decline and MPV is 9.6% for the worst case.

APP is also compared with Cazorla policy, a noted mechanism to address the problem of performance unpredictability in SMT processors. APP can achieve predictable performance for HPT as effectively as Cazorla policy does. Furthermore, APP can more effectively make full use of shared resources to optimize the performance of LPTs and overall throughput. Compared with Cazorla policy, overall throughput is increased by 4.9% on average and the performance of LPTs is increased by 17.6%.

Acknowledgments. This work was supported by Natural Science Foundation of China under grant number 61103011, 61170045 and 61202126.

References

1. Dehnavi, M.M., Hassanein, W.: A Clustered SMT Architecture for Scalable Embedded Processors. In: Proc. PRWT 2006(2006)
2. Berekovic, M., Moch, S., Pirsch, P.: A scalable, clustered SMT processor for digital signal processing. SIGARCH Computer Architecture News 32(3), 62–69
3. Radojkovic, P., Girbal, S., et al.: On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-Critical Environments. In: 7th International Conference on High-Performance Embedded Architectures and Compilers, Paris, France, January 23-25 (2012)

4. Paolieri, M., Quiones, E., et al.: Hardware support for WCET analysis of hard real-time multicore systems. In: ISCA 2009, pp. 57–68 (2009)
5. Ungerer, T., Cazorla, F.J., et al.: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. IEEE Micro 30(5), 66–75 (2010)
6. Tullsen, D., Eggers, S., Levy, H.: Simultaneous multithreading: Maximizing on-chip parallelism. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, pp. 392–403 (June 1995)
7. Tullsen, D., Eggers, S., et al.: Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: Proceedings of the 23rd Annual International Symposium on Computer Architecture, PA, USA, pp. 191–202 (May 1996)
8. Eggers, S.J., Emer, J., et al.: Simultaneous Multithreading: A Platform for next-generation processors. IEEE Micro, 12–19 (September-October 1997)
9. Cazorla, F., Knijnenburg, P., et al.: QoS for high-performance SMT processors in embedded systems. IEEE Micro. Special Issue on Embedded Systems 24(4), 24–31 (2004)
10. Cazorla, F., Knijnenburg, P., et al.: Architectural support for real-time task scheduling in SMT processors. In: Proceedings of International Conference on Compilers, Architectures and Synthesis for Embedded Systems, California, USA, pp. 166–176 (September 2005)
11. Cazorla, F.J., Knijnenburg, P.M.W., Sakellariou, R., Fernández, E., Ramírez, A., Valero, M.: Feasibility of QoS for SMT. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 535–540. Springer, Heidelberg (2004)
12. Cazorla, F., Knijnenburg, P., et al.: Predictable performance in SMT processors. In: Proceedings of ACM International Conference on Computing Frontiers, pp. 171–182 (April 2004)
13. Cazorla, F., et al.: Enabling SMT for Real-Time Embedded Systems. In: Proceedings of the 12th European Signal Processing Conference (September 2004)
14. Sherwood, T., Calder, B.: Time varying behavior of programs, Tech. Report UCSDCS99-630, University. of California (August 1999)
15. Tullsen, D.: Simulation and modeling of a simultaneous multithreading processor. In: Proceedings of the 22nd Annual Computer Measurement Group Conference, San Diego, CA, USA, pp. 819–828 (December 1996)
16. MediaBench II Benchmark, `http://euler.slu.edu/~fritts/mediabench`
17. The standard performance evaluation corporation, WWW cite: `http://www.specbench.org`