FROCM: A Fair and Low-Overhead Method in SMT Processor*

Shuming Chen and Pengyong Ma

School of Computer Science and Technology,
National University of Defense Technology, Changsha. 410073 China
mpy9608@yahoo.com.cn
mapy@sohu.com

Abstract. Simultaneous Multithreading (SMT)[1][2] and chip multiprocessors (CMP) processors [3] have emerged as the mainstream computing platform in major market segments, including PC, server, and embedded domains. However, prior work on fetch policies almost focuses on throughput optimization. The issue of fairness between threads in progress rates is studied rarely. But without fairness, serious problems, such as thread starvation and priority inversion can arise and render the OS scheduler ineffective. The fairness research methods always disturb the threads running Simultaneous, such as single thread sampling [4]. In this paper, we propose an approach FROCM (Fairness Recalculate Once Cache Miss) to enhance the fairness of running multithreads in SMT processor without disturbing their running states. Using FROCM, every thread's $IPC_{approximately}$ is re-calculated in SMT processor Once Cache Miss, $IPC_{approximately}$ is the approximately value of IPC when the thread runs alone. Using $IPC_{approximately}$, the instructions' issue priority may be changed in due course. We can hold the Fairness value (Fn) higher. Fn is fairness metric defined in this paper, when it is equal to 1, it means utterly fair. Results show that using FROCM, we can hold the most of Fn larger than 0.95, and the throughput hasn't larger change. It needs less hardware to realize the FROCM, including 4 counters, 1 shifter and 1 adder.

1 Introduction

During the last two decades, different architectures were introduced to support multiple threads on a single die (chip). Simultaneous Multi-Threading (SMT) is the active one, in which instructions from multiple threads are fetched, executed and retired on each cycle, sharing most of the resources in the core. SMT processors improve performance by running instructions from several threads at a single cycle. Co-scheduled threads share some resources, such as issue queues, functional units, and on chip cache. The way of allocating shared resources among the threads will affect throughput and fairness. Prior work on fetch policies almost focuses on throughput optimization, such as ICOUNT2.8[1], PDG[5], and so on. The issue of fairness between threads in progress rates is studied rarely. But fairness is a critical

* This work is supported by National Natural Science Foundation of China(No. 60473079) and the Research Fund for the Doctoral Program of Higher Education (No. 20059998026).

R. Perrott et al. (Eds.): HPCC 2007, LNCS 4782, pp. 566–576, 2007.
© Springer-Verlag Berlin Heidelberg 2007

issue because the Operating System (OS) thread scheduler's effectiveness depends on the hardware to provide fairness to co-scheduled threads. Without fairness, serious problems, such as thread starvation and priority inversion, may arise and render the OS scheduler ineffective.

In multithread processor, we can achieve fairness by many methods. Currently, shared resources allocation is mainly decided by the instruction fetch policy, especially in SMT processor with VLIW architecture.

Ron Gabor and Shlomo Weiss research the Fairness and Throughput in Switch on Event Multithreading[6]. In coarse gain multithreads processors, there is only one thread running in processor simultaneously, every thread's IPC_{alone} (IPC of thread when executed alone) can be accounted easily. IPC_{SOE} (IPC of each thread when executed using Switch on Event with other threads) can get every clock cycle. Every thread's acceleration can be easily worked out. So they can control the processors' fairness according to the thread's acceleration. But in SMT processor, the IPC_{alone} can't be accounted by this method.

Caixia Sun, Hongwei Tang, and Minxuan Zhang proposed a Fetch Policy ICOUNT2.8-fairness with Better Fairness for SMT Processors [4][7]. They take fairness as the main optimization goal. In this policy, relative progress rates of all co-scheduled threads are recorded and detected each cycle. If the range of relative progress rates is lower than a threshold, fairness is met approximately and ICOUNT2.8 is used as the fetch policy. Relative progress rates are used to decide fetch priorities. The lower a thread's relative progress rate is, the higher its fetch priority is. Unfair situation is corrected by this way. But in this policy, in order to get IPC_{alone} dynamically, they employ two phases: sample phase and statistic phase. During the sample phase, the processor has to run in single-thread mode. It will disturb other running threads and the throughput will be reduced.

There are other methods to insure the fairness of system, for example static and dynamic resource partition[11][12], Handling long-latency load[13], and so on.

In this paper, we propose a method FROCM to correct unfairness without sample phase, all the threads run from beginning to end without interrupt. Every thread's $IPC_{approximately}$ is re-calculated in SMT processor Real-timely, $IPC_{approximately}$ is the approximately value of IPC when the thread runs alone. Using $IPC_{approximately}$, we can hold the Fn (fairness metric defined in this paper, when it is equal to 1, it means utterly fair) larger than a threshold. Results show that we can hold the most of Fn larger than 0.95.

The rest of the paper is organized as follows. Section 2 presents the methodology and gives the definition of system's fairness. In Section 3, we detail how to realize FROCM with less hardware in SMT processor with VLIW architecture. Section 4 illustrates the results. Finally, concluding remarks are given in Section 5.

2 Model Proposed and Fairness Definition

Fairness is very important in multi-user or multi-task system. No user would like to wait long respond time. In SMT processor, OS assigns more time slices to threads with higher priority, thus priority-based time slice assignment can work in a SMT

processor system as effectively as in a time-shared single-thread processor system. So the hardware must support fairness, otherwise priority inversion may arise. In this paper, we detail how to support fairness by hardware.

In order to judge whether the performance of multithreads in a SMT processor is fair, at first we define several conceptions.

IC[i]: the instruction counters of thread i.

IPC_{alone}: IPC of thread when executed alone.

$IPC_{alone}[i]$: the IPC_{alone} of thread i.

$IPC_{SMT}[i]$: IPC of thread i when executed in SMT processor with other threads.

$T_{alone}[i]$: the runtime of thread i when executed alone,

$$T_{alone}[i] = \frac{IC[i]}{IPC_{alone}[i]} . \tag{1}$$

$T_{SMT}[i]$: the runtime of thread i when executed in SMT processor with other threads,

$$T_{SMT}[i] = \frac{IC[i]}{IPC_{SMT}[i]} . \tag{2}$$

$IPC_{approximately}[i]$: when thread i executed in SMT processor with other threads, we calculate the approximately value of its IPC_{alone}, $IPC_{approximately}[i] \approx IPC_{alone}[i]$.

Deceleration[i]: the deceleration of thread [i]'s performance when executed in SMT processor. Because of sharing resource with other threads in SMT processor, it will cause resource confliction. The threads' runtime will enlarge, and then the performance will be decelerated.

$$Deceleration[i] = \frac{T_{alone}[i]}{T_{SMT}[i]} . \tag{3}$$

Apparently $Deceleration[i] \in [0, 1]$.

$Deceleration_{ratio}[i, j]$: the ration of performance deceleration when thread i and thread j run in SMT processor,

$$Deceleration_{ratio}[i, j] = Min(\frac{Deceleration[i]}{Deceleration[j]}, \frac{Deceleration[j]}{Deceleration[i]}), \tag{4}$$

Apparently $Deceleration_{ratio}[i, j] \in [0, 1]$.

Fn: The fairness value we defined in SMT processor,

$$Fn = \underset{\forall i, j}{Min}(Deceleration_{ratio}[i, j]) . \tag{5}$$

It is two threads' deceleration ratio that they deceleration value difference is maximal, because of $\forall (i, j), Deceleration_{ratio}[i, j] \in [0, 1]$, so $Fn \in [0, 1]$.

If we want that the system is fair, we should try our best to let the drop of every thread's performance equally. In other words, if we ensure the Fn approximate to 1, the system is fair. We can know from Equation 5:

$$
\begin{aligned}
& Fn = 1 \\
& \Rightarrow Deceleration_{ratio}[i, j] = 1 \\
& \Rightarrow \frac{Deceleration_{ratio}[i]}{Deceleration_{ratio}[j]} = 1 \\
& \Rightarrow \frac{T_{alone}[i]}{T_{SMT}[i]} = \frac{T_{alone}[j]}{T_{SMT}[j]} \\
& \Rightarrow \frac{IPC_{SMT}[i]}{IPC_{alone}[i]} = \frac{IPC_{SMT}[j]}{IPC_{alone}[j]} \\
& \Rightarrow \frac{IPC_{SMT}[i]}{IPC_{SMT}[j]} = \frac{IPC_{alone}[i]}{IPC_{alone}[j]}
\end{aligned}
\tag{6}
$$

In other words, if we insure that the ratio of random two threads' IPC in SMT processor is equal to the ratio of the IPC when they run alone, the system is fair. Because $IPC_{alone}[i]$ and $IPC_{alone}[j]$ are the character of thread i and j, they don't change when running in SMT processor. So if we get every thread's IPC_{alone}, we can control the system's fairness accurately.

Usually there are two ways to acquire the thread's IPC_{alone}, static method and dynamic sampling. Static method is that every thread run alone in processor at the beginning, then we can get the IPC_{alone} statistically. This way is unpractical in most of application. In dynamic sampling, the processor runs in single-thread mode. Each thread runs alone for a certain interval respectively. This method has disadvantage too, if the sample phase is too frequently or too long, it will degrade the throughput of the SMT processor. On the other contrary, if the sample phase is infrequently or too short, the statistical value will largely differ from the real IPC_{alone}. We can't get the accurate IPC_{alone}, and then we can't ensure the system's fairness

Now we analysis the threads' run model in SMT and uni-core processor. The IPC_{alone} is correlative with the issue cycle and the memory miss delay.

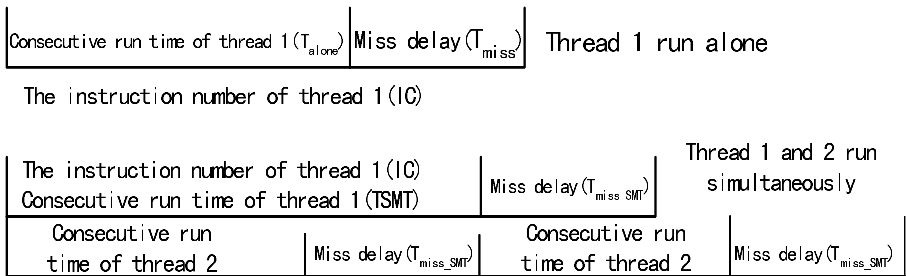

Fig. 1. The sequence of thread 1 run alone and with other thread simultaneously

As Fig. 1 shows, the time of thread 1 is divided into 2 parts per cache miss, the execute time(T_{alone}) and miss delay (T_{miss}). So we can get the IPC of thread 1.

$$IPC_{alone} = \frac{IC}{T_{alone} + T_{miss}}. \tag{7}$$

The T_{miss} is the processor's character, and it is a constant.

When thread 1 run in SMT processor with other threads, the run schedule as the bottom of Fig. 1 (suppose that the cache miss sequence as same as it when thread 1 run alone in uni-core processor.). When cache miss occurs, the instruction counter is same as which when thread run alone. T_{miss} is the character of the uni-core processor and we can acquire it at the beginning. So if we want get the IPC_{alone}, nothing but that we acquire the T_{alone}. As Fig. 1 show, T_{SMT} is larger than T_{alone} in evidence because of resource sharing, so it can't replace the T_{alone}. But in SMT processor, we can statistic the IC of thread 1, the IC is equal to the issue cycle for single issuing processor, that is to say IC=T_{alone}. But it doesn't fit for supper scalar or VLIW processor. In super scalar or VLIW processor, the IC or instruction issuing cycle is not equal to T_{alone}.

As Fig. 2 shows, in VLIW SMT processor, there are five instructions of thread 2 may be issued in time T0, because of share 8 execute units and thread 1 with high priority, only SUB and MPY were issued in T0. The remainder instructions ADD, ADD and MPY were issued in T1. If we counter the issue cycle, the issue cycle of these five instructions is 2. But when thread 2 run alone in uni-core processor, the execute time of these 5 instructions is only 1 cycle. So no matter how many clock cycle one execute packet issued, we only counter once. When thread run alone in uni-core processor with VLIW architecture, one cycle issue one execute packet (EP), so in SMT processor, the number of EP is equal to T_{alone}. EP = T_{alone}.

Substituting EP = T_{alone} into equation 7,

$$IPC_{\text{approximately}} = \frac{IC}{EP + T_{miss}} \tag{8}$$

In SMT processor, because of source shared by several threads, the Cache miss ratio may be increased. So in equation 8, we use $IPC_{approximately}$ instead of IPC_{alone}. If system ensures Cache partition is fair, then the increase ratio of threads Cache miss are close to each other. The decrease ratios of threads' IPC_{alone} are approximately, so the system is fair too.

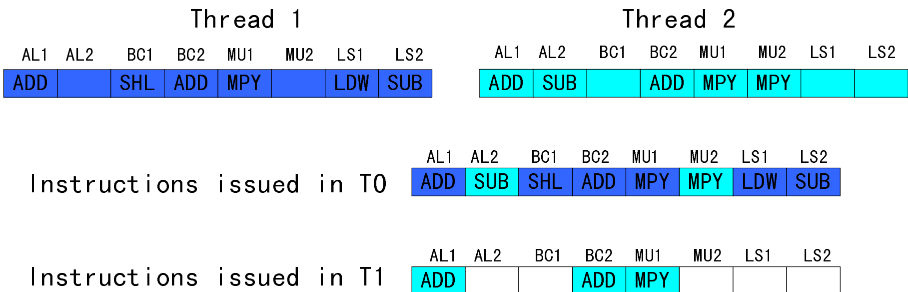

Fig. 2. Separate issue the instructions of thread 2

Substituting $IPC_{approximately}=IPC_{alone}$ into equation 6,

$$\frac{IPC_{SMT}[i]}{IPC_{SMT}[j]} = \frac{IPC_{approximately}[i]}{IPC_{approximately}[j]}. \quad (9)$$

If system ensure equation 9, It is a fair system.

3 Realization with Low-Hardware

Equation 8 shows that if we want acquire the approximate IPC_{alone} of thread: $IPC_{approximately}$, it needs a division unit. It needs lots hardware to realize a division unit, so we should avoid using it. YHFT DSP/900 is an instance of SMT processor. It is a two-ways SMT with 8 execute units shared. Because that it issues 8 instructions one cycle at most, so the $IPC_{approximately}$ value is a real number between 0 and 8, we use a integer register to store it, in order to enhance the precision, we enlarge both threads' $IPC_{approximately}$ to 8 times. It is a value between 0 and 64, so a 6-bits integer register is enough. We may use subtraction instead of division. When we subtract ($EP+T_{miss}$) from IC, if the remainder larger than 0, add 1 to $IPC_{approximately}$. Repetition until the remainder is less than 0. Fig. 3 shows the arithmetic flow of $IPC_{approximately}$.

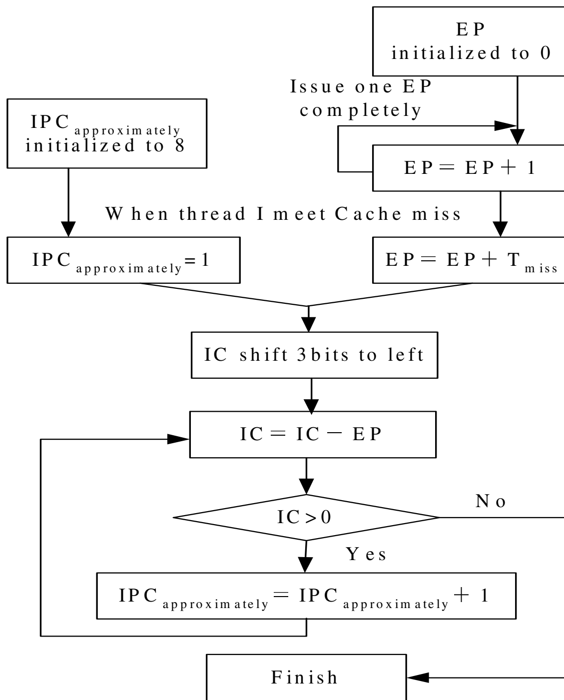

Fig. 3. FROCM arithmetic flow

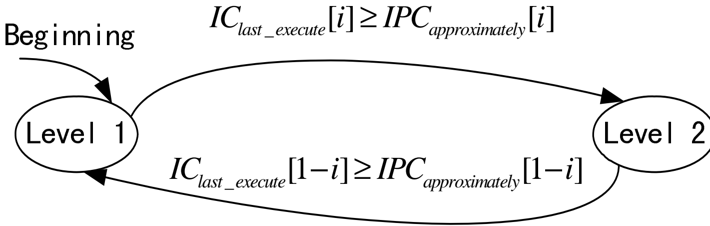

Fig. 4. Transition of thread[i] issue priority

For each thread, processor set a 6-bits register to record the number of issued instructions $IC_{last_execute}$ after the last modifying issuing priority. Once $IC_{last_execute}$ is larger than or equal to $IPC_{approximately}$, $IC_{last_execute} = IC_{last_execute} - IPC_{approximately}$, and then the thread reduces its instruction issued priority to level 2, enhance the other thread's instruction issued priority to level 1. Fig. 4 shows the transition of thread[i] issue priority.

For example, thread 0 and 1 run in SMT processor simultaneously. $IPC_{approximately}[0]$ is equal to 2.5, and $IPC_{approximately}[1]$ is equal to 3.7. After enlarge 8 times, the values are 20 and 30. After instructions issued in one clock cycle, the $IC_{last_execute}[0]$ is 24, and $IC_{last_execute}[1]$ is 29, this means that thread 0 run rapidly. $IC_{last_execute}[0] = IC_{last_execute}[0] - IPC_{approximately}[0] = 24 - 20 = 4$, and then thread 0 reduces its instruction issued priority to level 2, enhance the priority of thread 1 to level 1 simultaneously.

Now we analyze the cost of hardware.

Counter IC: in order to record the number of issued instructions per Cache miss. A 16-bits register is enough; it can record 2^{16}=65536 instructions.

Counter EP: in order to record the number of execute packet per Cache miss. It is less than IC, so a 16-bits register is enough.

Shifter: in order to enhance the precision, IC shift 3-bits to left.

Adder: use it to calculate $EP=EP+T_{miss}$ and IC=IC-EP.

Counter $IPC_{approximately}$: it is about 8 times of IPC_{alone}, because IPC_{alone} is a value between 0 and 8, so $IPC_{approximately}$ is between 0 and 64. A 6-bits register is enough.

Counter $IC_{last_execute}$: the number of issued instructions after last modifying issuing priority. In FROCM, it is less than $IPC_{approximately}$, so a 6-bits register is enough.

4 Results

4.1 Environment Establishing

In order to test the performance of FROCM, we realize RR(round robin) and FROCM of issuing priority in YHFT DSP/900[8][9] simulator respectively. Several program combination run in the simulator with both RR and FROCM respectively.

Now we introduce the architecture of YHFT DSP/900 at first. It is a two-ways SMT processor with VLIW architecture. Each thread hold private fetch instructions unit, register files, and so on. They share 8 execute units, Cache on chip, peripheral

function, and so forth. The pipeline is divided into 3 stages, fetch, dispatch and execute. Every clock, it can issue 8 instructions at most.

Fig. 5 is the diagram of YHFT DSP/900 core. The gray area in Fig. 5 is the shared resource. It has two levels Cache. Once level one data or Instruction Cache miss occurred, the delay T_{miss} is 5 clock cycles. The delay of level 2 Cache miss T_{miss} is 60 clock cycles.

Table 1 shows the character of YHFT DSP/900's memory system. It lists the architecture and the bandwidth of data bus. The delay of L2 and the delay of memory is the value in uni-core processor YHFT DSP/700. In other words, it is the delay when only one thread miss request.

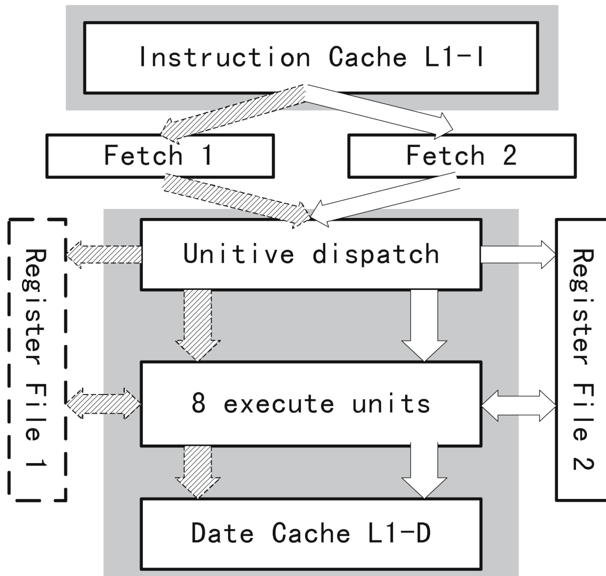

Fig. 5. The diagram of YHFT DSP/900 core

Table 1. The character of memory system

Fetch width	8×32bits instructions
Bandwidth between L1 I-Cache and L2	256bits
Bandwidth between L1 D-Cache and L2	128bits
L1 I-Cache	4KB, direct mapping, 512bits per line, single port, read miss allocation
L1 D-Cache	4KB, 2 ways, 256bits per line, two ports, read miss allocation
L2	64KB，4 ways, 1024bits per line, single port, read/write miss allocation
latency of L2	5 cycles
latency of memory	60 cycles

4.2 Test Benchmarks

The program is divided into 2 types. One type is the program, which don't aim at any type processor. It includes FFT, the decode/encode of ADPCM in Mibench, the decode/encode of G721 in MediaBench. The other type is the benchmarks that aim at DSP processor, these benchmarks has high parallel and includes dotp_sqr, fir, matrix, and fftSPxSP. When two threads run in YHFT DSP/900 simultaneously, the runtimes aren't equal to each other. In order to test the performance of FROCM exactly, we ensure there are two threads running in processor from beginning to end. When the first thread finished, it will restart again until the second thread finish too. Then we take the time that two threads finished respectively as the threads' runtime in SMT T_{SMT}.

Fig. 6 shows the Fn of various instructions issued priority, RR(round robin) and FROCM. Fig. 7 shows the throughput of system when they adopt RR and FROCM. In Fig. 7, we take the IPC as the system's throughput.

From Fig. 6 and 7, we can find that the fairness of system is enhanced 5% after adopting FROCM policy. The most of Fn are larger than 0.95 without dropping the system's throughput. From Fig. 6, we find the Fn of four program-combinations enhanced greatly. There are FFT+fftSPxSP, dotp_sqr+fir, matrix+fftSPxSP, G721_D+matrix. Because that dotp_sqr, fir, matrix and fftSPxSP are the assemble code, they have high parallel. When they run with other programs, there are many instructions may issued every clock cycle, more than 8 instructions sometime. So the conflict of execute units is frequently. If processor adopt RR policy, it always result in overbalance of execute units allocation, so the Fn is lower. After use FROCM policy, processor control threads run speed by the IPC_{alone}, and allocate the units fairly. So the Fn is enhanced largely.

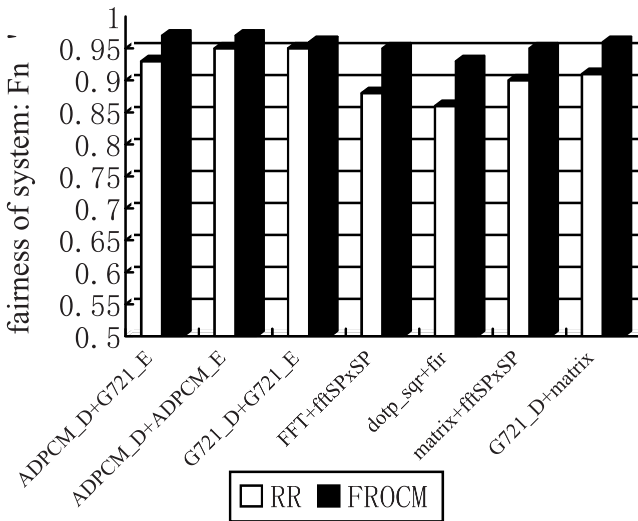

Fig. 6. Fairness for RR and FROCM

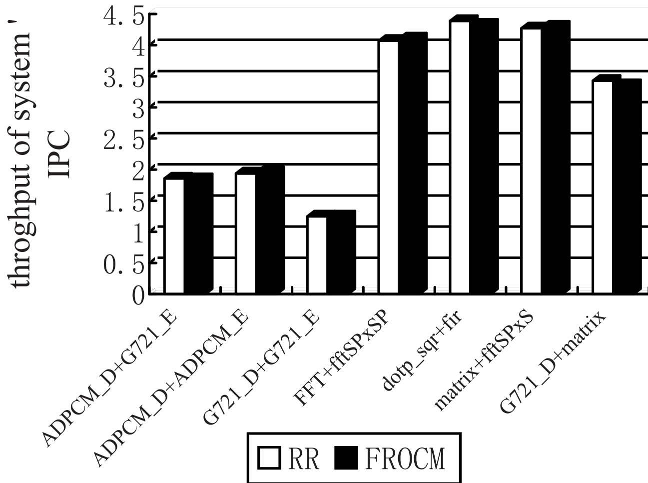

Fig. 7. Throughput for RR and FROCM

5 Conclusions

Because that we can't get the IPC of thread running alone in uni-core process when it runs with other threads in SMT processor, then the system's fairness is difficult to balance. This paper proposes a low and less-hardware fairness policy FROCM. In FROCM, the IPC_{alone} will be recalculated again once the thread meets a Cache miss, then the system will adjust the threads' priority depending on the IPC_{alone} and balance the processor's fairness. The FROCM policy has several advantages. The first is that it doesn't need sample phase; it doesn't interrupt the simultaneous running threads, so the throughput will not be dropped. The second is that it can recalculate the thread's IPC_{alone} at the real time, and then adjust the threads' priority. So it can control the fairness accurately. The last is the low-hardware. Such as a two-ways thread SMT processor with 8words VLIW architecture, it only needs two 16bits counter, two 6bits counter, one shifter and a 16bits adder. The simulations show that FROCM can ensure the most Fn of program-combination larger than 0.95, this indicates that it is an effective fairness policy.

References

1. Tullsen, D.M., Eggers, S., Emer, J., Levy, H., Lo, J., Stamm, R.: Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: In Proc. of ISCA-23, pp. 191–202 (1996)
2. Tullsen, D.M., Eggers, S., Levy, H.: Simultaneous multithreading: maximizing on-chip parallelism. In: ISCA 1998, pp. 533–544 (1998)
3. Olukotun, K., et al.: The Case for a Single-Chip Multiprocessor. In: Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 2–11 (1996)

4. Sun, C., Tang, H., Zhang, M.: Enhancing ICOUNT2.8 Fetch Policy with Better Fairness for SMT Processors. In: Jesshope, C., Egan, C. (eds.) ACSAC 2006. LNCS, vol. 4186, pp. 459–465. Springer, Heidelberg (2006)
5. El-Moursy, A., Albonesi, D.: Front-end policies for improved issue efficiency in SMT processors. In: Proc. HPCA-9 (2003)
6. Gabor, R., Weiss, S., Mendelson, A.: Fairness and Throughput in Switch on Event Multithreading. In: The 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2006. (2006)
7. Sun, C., Tang, H., Zhang, M.: A Fetch Policy Maximizing Throughput and Fairness for Two-Context SMT Processors. In: Cao, J., Nejdl, W., Xu, M. (eds.) APPT 2005. LNCS, vol. 3756, pp. 13–22. Springer, Heidelberg (2005)
8. Shuming, C., Zhentao, L.: Research and Development of High Performance YHFT DSP. Journal of Computer Research and Development, China 43 (2006)
9. Jiang-Hua, W., Shu-Ming, C.: MOSI: a SMT Microarchitecture Based On VLIW Processors. Chinese Journal of Computer 39 (2006)
10. Dan-Yu, Z., Peng-Yong, M., Shu-Ming, C.: The Mechanism of Miss Pipeline Cache Controller based on Two Class VLIW Architecture. Journal of Computer Research and Development, China (2005)
11. Raasch, S.E., Reinhardt, S.K.: The impact of resource partitioning on SMT processors. In: Proc. of PACT-12, p. 15 (2003)
12. Luo, K., Gummaraju, J., Franklin, M.: Balancing throughput and fairness in SMT processors. In: Proc. of the International Symposium on Performance Analysis of Systems and Software, pp. 164–171 (2001)
13. Tullsen, D.M., Brown, J.: Handling long-latency loads in a simultaneous multithreading processor. In: Proc. Of MICRO-34, pp. 318–327 (2001)