2005－MPS－55（2）
2005／6／28

Development of a Thread Scheduler for SMT Processor Architecture

Kaname Uchikura, Koichi Sasada, Mikiko Sato, Masanori Yamato, Norito Kato,
Hironori Nakajo and Mitaro Namiki
Tokyo University of Agriculture and Technology
2-24-16 Naka-cho, Koganei-shi, Tokyo, Japan

Abstract ***An SMT(Simultaneous MultiThreading) architecture processor aims to progress processor performance by executing parallel threads. However, the increasing cache misses caused by the capacity miss and the conflict miss in the shared cache memory. In this paper, we propose a thread scheduler based on a concept of thread affinity. Our proposed system observes performance of concerning threads with common cache and reschedules them. In addition, we have developed a strategy to choose the suitable thread number according to decreasing of cache hit ratio. As experimental results, the system with our thread scheduler performs up to 1.96 times higher with benchmark programs of RADIX sort in SPLASH-2.***

Keywords: thread scheduler, multithreaded architecture, system software

1 Introduction

Multithreaded architecture is gaining popularity as processor architecture progresses. One example of the Multithreaded architecture is **Simultaneous MultiThreading (SMT)**[1], which executes multiple threads in parallel while sharing hardware resources like arithmetic units to make efficient use of execution units and improve performance with user level threads controls.

Against this background, SMT processor has weakness. On the SMT processor, the increasing cache misses caused by sharing the cache memory bring performance degradation.

In this paper, we have proposed two types of thread schedulers. First, a previous scheduler is deciding the most appropriate the number of active threads to decrease cache misses. Second, a scheduler based on a concept of thread affinity chooses threads whose affinity are higher each other.

2 Goal

We develop two types of thread schedulers. It is a significant issue to decrease L1 cache miss ratio. Increasing L1 cache miss ratio has two causes. One is the capacity miss, other is the conflict miss. To remove the capacity miss, we develop a strategy to choose the suitable thread number according to decreasing cache hit ratio. While, to remove the conflict miss, we have developed a thread scheduler based on a concept of thread affinity. Our proposed system observes performance of concerning threads with common cache and reschedules them.

3 Designing of the Thread Scheduler

According to the pilot study researched beforehand, a cause of decreasing L1 cache hit ratio is divided into two problems. One is the capacity miss, other is the conflict miss. We develop the two types of thread schedulers to each of those miss problems. The former is **The method to decide on the number of threads (DT)**, the latter is **Scheduling with Thread Affinity (SA)**.

Table 1. The assumption to stop an AT and the number of stopped ATs

A cache hit ratio	80% less	80〜90%	90% more
The number of ATs (X)	$X * 1/2$	$X * 3/4$	$X + 1$

3.1 The Method to decide on the Number of Threads

We propose a thread scheduler which has a strategy to choose the suitable thread number according to decreasing cache hit ratio. We call this thread scheduler **The method to decide on the number of threads (DT)**.

Fig. 1 describes the processing flow of the DT. In (a), the DT gets the number of the L1 cache hit and the number of the L1 cache miss. Their value is counted in the processor.

In (b), it calculates the L1 cache hit ratio by using below mathematical formula (1).

Hit_Ratio : L1 cache hit ratio

Hit_num : The number of cache hits

$Miss\_num$: The number of cache misses

$$Hit\_Ratio = \frac{Hit\_num}{Hit\_num + Miss\_num} \times 100 \quad (1)$$

In (c), it finds out the number of the threads that the processor executed previous time.

Finally, in (d) it decides on the number of LTs along Table 1. If the L1 cache hit ratio decreases less than 80%, the DT assigns half LTs to ATs of previous executing LTs. If the L1 cache hit ratio decreases between 80% and 90%, the DT assign 3/4 LTs to ATs than previous executing LTs. While on the other hand, if the 1st cache hit ratio increases more than 90%, the DT assign some LTs plus 1 to ATs than previous executing LTs.

The DT has applied this configuration parameter in this time from the pilot study. However, most suitable configuration parameter depends on the feature of application programs. Therefore, those configuration parameters enable to change.

3.2 Scheduling with Thread Affinity

We propose a thread scheduler based on a concept of thread affinity. Our proposed system observes performance of concerning threads with common cache and reschedules them. We call this thread scheduler **Scheduling with Thread Affinity(SA)**.

```
(a)Getting the whole number of
  L1 cache hits and misses
(b)Calculation of formula 1
(c)X = the previous number of
  executed threads
(d)/*Combining based for Table 2*/
  if(Hit_ratio >= 90){
        X = X + 1;
  }
  else if(Hit_ratio >= 80){
     X = X * 3 / 4;
  }
  else if(Hit_ratio < 80){
     X = X * 2 / 4;
  }
```

Figure 1. The algorithm of the method to decide on the thread number

A *Thread Affinity* is a performance indicator between a thread and the other. When the SA gauges the *Thread Affinity*, a thread targets at another one. This another thread is called a *Target Thread*. If the SA gauges the *Thread Affinity* between a thread which is A and a *Target Thread* which is B, we define the *Thread Affinity* below mathematical formula 2. *A* and *B* are each of a thread. If *B* is *A*'s *Target Thread*, B is *target(A)*. The *share(A,B)* is the number of sharing *A*'s cache data and *B*'s. The *exclude(A,B)* is the number of excluding *A*'s cache data and *B*'s. *TA(A,B)* is the *Thread Affinity* between *A* and *B*.

$$TA(A,B) = \frac{share(A,B)}{share(A,B) + exclude(A,B)} \times 100 \quad (2)$$

Consequently the SA is able to know the *Thread Affinity* between a thread *A* and a thread *B*.

Moreover, by comparing *Hit_Ratio* shown in the mathematical formula 1 to *TA(A,B)*, we define the **good** or **bad** *Thread Affinity* below mathematical formula (3) and (4).

$$TA(A,B) \geqq Hit\_Ratio : \ Good\ Thread\ Affinity \quad (3)$$
$$TA(A,B) < Hit\_Ratio : \ Bad\ Thread\ Affinity \quad (4)$$

Every time the SA is called, the *Hit_Ratio* is at an average rate of total L1 cache hit ratio on the SMT processor, in contrast the *Thread Affinity* is localized L1 cache hit ratio on a couple of threads.

In Fig. 2 which shows module of 4 ATs, each of architecture threads has a *Target Thread*, *share* and

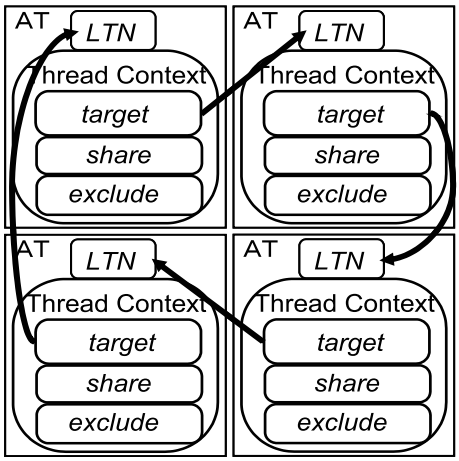

Figure 2. The relation between a thread and a target thread

exclude register in thread contexts. Each of thread has just one *Target Thread* register. By rights, each of threads provides some registers to store all *Target Threads* and should observe all of the *Thread Affinity*. For example, if the configuration of the SMT processor is 4 ATs, a thread combination is 6 ways. If 8 ATs, a thread combination is 28 ways.[1] However we have made the specification of the SA to use registers requisite minimum because the SMT processor is limited to thread contexts and registers. All of the data such as *Hit_num*, *Miss_num*, *share* and *exclude* are counted in the AT and kept into the AT's thread context.

Fig. 3 describes the processing flow of the SA. In the SA according to whether the thread affinity between a thread and its target thread is good or bad, it is assigned to an AT again or stored to a queue. A thread also is assigned to an AT if its target thread is assigned when the thread affinity between them is good and it also is stored to a queue if its target thread is stored. A thread is inversely stored to a queue if its target thread is assigned when the thread affinity between them is bad and it is inversely assigned to an AT if its target thread is stored to a queue. The SA should assign a couple of threads if they are good thread affinity and divide an AT from a queue if they are bad thread affinity. According to that processing flow, the possibility of executing threads which are good thread affinity will increase.

[1] ${}_4C_2 = 6, {}_8C_2 = 28$

```
A = LTN /*It has executed until now*/
B = target(A);/*It is A's target LTN*/
if(TA(A,B) >= Hit_num){
   if(B was stored to a queue){
      A also is stored to a queue
   }
   else{/*B was assigned to an AT*/
      A also is assigned to an AT
   }
 }
else{
   if(B was stored to a queue){
      A is assigned to an AT
   }
   else{/*B was assigned to an AT*/
      A is stored to a queue
   }
 }
```

Figure 3. The algorithm of the scheduler with Thread Affinity

4 Evaluation

In the evaluation, we use LU matrix decomposition(LU), Fast Fourier Transform(FFT) and RADIX sort(RADIX) from SPLASH-2 benchmark.[2] Each of them products 16 threads. The L1 cache size is 4KB and 32KB. The number of architecture threads is 4 and 8. The thread scheduler mode is DT, SA, and DT+SA.

We researched speed up ratio from an IPC applying the thread schedulers and an IPC applying none thread scheduler. Fig. 4 describes the speed up ratio of each benchmarks and ATs when the speed up ratio of none scheduler mode is defined as 1.00. In the case of L1 cache size 4KB, RADIX, 8 ATs, and DTSA, the thread scheduler performs up to 1.96 times higher than none thread scheduler. This is the best speed up ratio in all the executions.

In the case of L1 cache size 4KB, effects of the DT has appeared strongly. We have accomplished an improvement of an SMT processor performance. On applying the DT, both IPC of 4 ATs and IPC of 8 ATs have gone up to 1.60. In this case, the DT was able to perform better because the capacity miss have occurred frequently. It follows that an SMT processor should adopt preventing the decline of the L1 cache hit ratio rather than

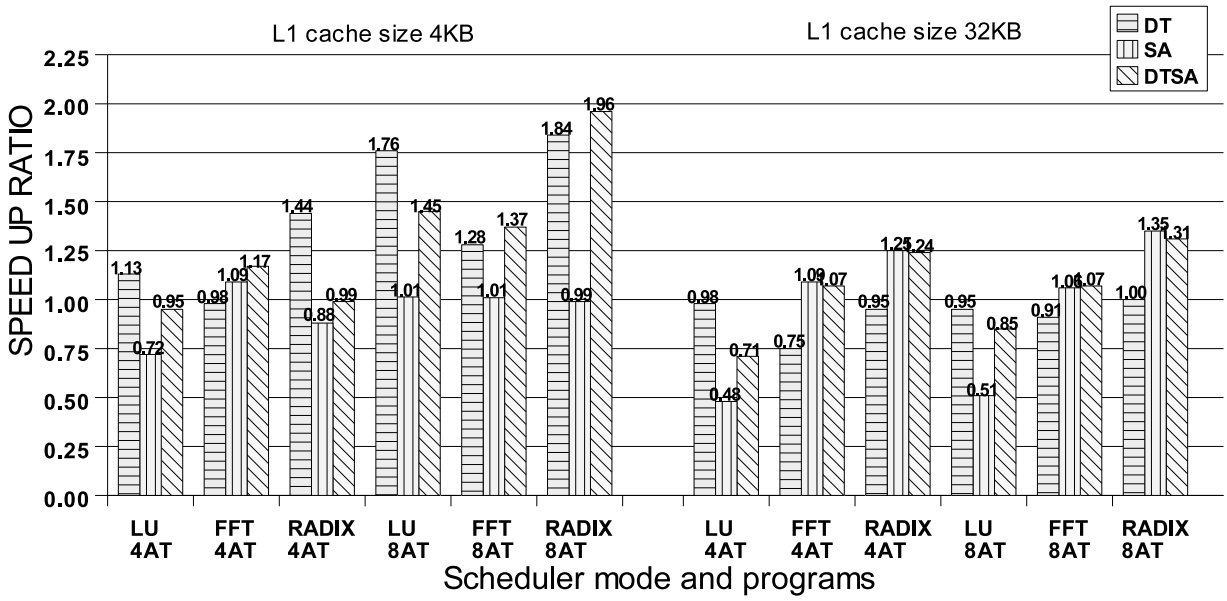

Figure 4. The speed up ratio by scheduler mode
*The speed up ratio of none scheduler mode is defined as 1.00 on each of benchmarks.

executing parallel threads to the full.

In the case of L1 cache size 32KB, effects of the SA have appeared strongly. On applying the SA, FFT and RADIX have shown performance notably better because they have accomplished combining a couple of thread which are good affinity by discerning good and bad affinity. If an SMT processor executes parallel threads with sharing much of cache data, its performance will progress better. However, LU has not shown performance better because a working set of the LU program has shared little.

The DT and the SA were able to accomplish high performance by overcoming weaknesses which are the capacity miss and conflict miss.

5 Conclusion

In this paper, we have proposed and developed a thread scheduler based on a concept of thread affinity. Our proposed system observes performance of concerning threads with common cache and reschedules them. In addition, we have developed a strategy to choose the suitable thread number according to decreasing of cache hit ratio. As experimental results, the system with our developed thread scheduler performs up to 1.96 times higher with benchmark programs of RADIX sort in SPLASH-2.

In future work, attempting to apply these thread scheduler; the DT and the SA to the other SMT processor and finding them to be effective.

References

[1] Dean M. Tullsen, Susan Eggers, and Henry M. Levy: Simultaneous multithreading: Maximizing on-chip parallelism, In *Proceedings of the22th Annual International Symposium on Computer Architecture*, pp. 392–403, 1995.

[2] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta: The SPLASH-2 Programs:Characterization and Methodological Consid-erations, In *Proceedings of the22th Annual International Symposium on Computer Architecture*, pp.24–36, 1995.