

Thread-Sensitive Instruction Issue for SMT Processors

Behnam Robotmili, Nasser Yazdani, Somayeh Sardashti
Router Laboratory, University of Tehran
beroy@ece.ut.ac.ir, yazdani@ut.ac.ir, s.sardashti@ece.ut.ac.ir

Mehrdad Nourani
University of Texas at Dallas
nourani@utdallas.edu

Abstract—Simultaneous Multi Threading (SMT) is a processor design method in which concurrent hardware threads share processor resources like functional units and memory. The scheduling complexity and performance of an SMT processor depend on the topology used in the fetch and issue stages. In this paper, we propose a thread sensitive issue policy for a partitioned SMT processor which is based on a thread metric. We propose the number of ready-to-issue instructions of each thread as priority metric. To evaluate our method, we have developed a reconfigurable SMT-simulator on top of the SimpleScalar Toolset. We simulated our modeled processor under several workloads composed of SPEC benchmarks. Experimental results show around 30% improvement compared to the conventional OLDEST_FIRST mixed topology issue policy. Additionally, the hardware implementation of our architecture with this metric in issue stage is quite simple

I. INTRODUCTION

Simultaneous multithreading (SMT) is a processor design approach which permits multiple hardware threads to share functional units simultaneously in each cycle [6][7]. SMT increases resource utilization by assigning unused resources to active threads [2]. It combines both thread-level parallelism (TLP) and instruction-level parallelism (ILP) [3][5].

An important part of the SMT architecture is the issue unit which chooses instructions for issue. There are some algorithms to decrease issue slot waste and issue proper instructions [7]. The scheduler complexity and performance depend on the topology used in the fetch and issue stages. Partitioning instruction queue (IQ) and allocating each partition to one of the active thread can properly achieve higher performance while reducing the complexity of the scheduler [3]. In addition, prioritizing threads in the issue stage is easier in the partitioned topology compared to the mixed topology proposed in previous SMT studies [2][6][7]. A few studies have focused on this topology and its corresponding issue policies [7].

In this paper, we propose a scheduling policy for SMT partitioned topology which uses thread metric. Our proposal includes a dynamic and real-time metric for our scheduler based on the number of ready-to-issue instructions of each thread called ready instruction count (RIC). Using this metric, programs in different threads are served according to their immediate need and urgency for CPU resources. To evaluate our method, we have developed a reconfigurable SMT simulator on top of SimpleScalar Tool Set. Experimental results show about 30% improvement in comparison with the conventional mixed topology issue policy (OLDEST_FIRST).

Manuscript submitted: 29 June 2004. Manuscript accepted: 23 Aug. 2004. Final manuscript received: 26 Aug. 2004.

The rest of the paper is organized as follows: Section 2

review related work on the SMT fetch and issue policies. SMT partitioned and mixed topologies are discussed in Section 3. Their proper issue policies are also described in this section. Explanation of our thread-sensitive issue policy for partitioned SMT topology comes in Section 4. The simulation environment and results are discussed in Section 5. Finally, Section 6 concludes the paper.

II. RELATED WORK

To increase resource utilization, SMT processors try to fetch instructions of threads to provide the best instructions throughput or instruction per cycle (IPC) by using fetch policies and algorithms [3][6][7]. These policies attempt to improve fetch throughput by partitioning the fetch unit among threads (fetch efficiency), improving the quality of the instructions fetched (fetch effectiveness) and eliminating conditions that block the fetch unit (fetch availability). Many algorithms are proposed to reach these goals. Tullsen et al. proposed their SMT architecture as an extension to Superscalar processors and studied fetch policies for their SMT processor [6]. Among proposed policies, ICOUNT provides the best throughput [6]. ICOUNT is a thread sensitive fetch algorithm which gives priority to threads with the least number of instructions in decode, dispatch and issue stages. Therefore, ICOUNT tries to choose threads which their instructions move faster in IQ [7].

Tullsen et al. also proposed some issue policies for their processor to prevent the issue waste. These policies are OLDEST_FIRST, OPT_LAST, SPEC_LAST and BRANCH_FIRST [7]. All of these methods are based on prioritizing some instructions in the issue stage without considering the attributes of the threads owning those instructions. In other words, proposed issue policies are thread-oblivious and lack several features which are discussed in the next section. To the best of our knowledge, no thread sensitive issue policy has been presented so far in the literature.

III. SMT TOPOLOGIES AND ISSUE POLICIES

A. SMT IQ Topologies

The mechanism of selecting ready instructions from the instruction queue (IQ) in the issue stage can significantly affect the performance of the processor. There are few studies focused on the SMT issue methods and algorithms. The reason is that in the primitive SMT architectures [6][7], the main bottleneck was fetching instructions. This prevents achieving high performance in general. However, in new

studies and real products, in addition to using high speed multi-port memories, a lot of work has been done to improve and optimize the SMT fetch stage [2][3][6]. Therefore, the instruction queue topology and issue policy can be more sensitive in SMT products. IQ topology in SMT processors is determined by the way instructions from different threads are placed inside the instruction queue and how ready instructions in the queue are selected for execution. IQ topologies used in SMT architectures can be classified as follows:

- **IQ Mixed Topology:** This topology used in the first proposed SMT architecture, was an extension to superscalar architecture, instructions from different threads are placed in an IQ without any specific consideration [6][7]. Each instruction in the instruction queue identifies its owner thread with a thread ID (TID). A *central issue scheduler* selects some of the ready instructions from IQ for execution in the functional units in each cycle. Even though this topology is easily derived from the superscalar architecture with low overhead, it is not so scalable. Adding more threads requires larger IQ which increases complexity of the superscalar scheduler since it is proportional to the IQ size [4]. This problem considerably limits the clock frequency.
- **IQ Partitioned Topology:** Another possible topology for IQ in SMT architecture is made by partitioning IQ into several segments (or sub-queues) each assigned to a thread. Each sub-queue is scheduled by a separate issue scheduler that performs superscalar scheduling for its corresponding thread [3]. The complexity and delay of each scheduler is significantly lower than the one in the mixed model because of the reduced length of each sub-queue. This leads to a shorter clock and higher working frequency.

Fig. 1 shows a general view of IQ mixed and partitioned topologies.

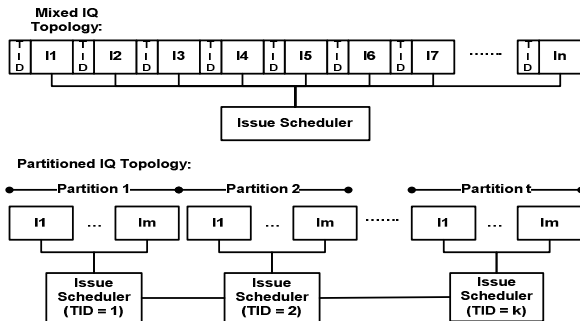


Fig. 1. SMT IQ topologies

In this figure, the number of active threads in the system is k . $TIDs$ are the thread IDs in mixed topology (A value between 1 and k), the size of IQ (IQ_{SIZE}) is n and the partition size in partitioned topology is m .

We assume the same queue size for both topologies which means that $m*k=n$. Because of linear dependency between IQ size and scheduler complexity, the delay and complexity of each partitioned scheduler are $1/k$ of the central scheduler used in the mixed topology.

B. SMT Issue Policies

In an SMT processor, issue logic selects instructions out of the ready instructions in IQ based on an issue policy. Issue policies used for the mixed topology mostly work based on prioritizing the ready instructions in the queue according to specific metrics [7]. The most common issue policy in this category is OLDEST_FIRST (OLDEST for short), similar to its original superscalar versions, selects the oldest ready instructions [7]. This policy is thread-oblivious because it doesn't consider the status and profile of the active threads. Other thread oblivious issue policies in this group are OPT_LAST, SPEC_LAST and BRANCH_FIRST. OPT_LAST and SPEC_LAST select optimistic and speculative instructions with lower priority while BRANCH_FIRST selects branch instructions with a higher priority. These policies do not improve the instruction throughput significantly compared to the OLDEST policy. They are weak in term of fairness among threads and supporting real-time applications. For example, a thread can acquire the system resources for a long time and a thread with a very high requirement to functional units stays un-served. Implementing a thread-sensitive issue policy for an SMT processor with mixed IQ topology leads to a high hardware overhead. The reason is that the scheduler must receive the thread ID of every instruction in the IQ, partition the instructions according to the thread IDs and then perform the thread-sensitive selection. However, implementation of such policies is easily possible when using partitioned IQ topology since the required partitioning of the instructions is naturally provided in the IQ. Although the partitioned topologies are known to be better in general, no work has been specifically focused on the performance of their issue policies which is the target of this paper.

IV. PROPOSED POLICY FOR PARTITIONED SMT TOPOLOGIES

In this section we propose our thread-sensitive issue policy which is designed for the partitioned SMT topology. In this method, each of the partitioned schedulers shown in Fig.1 runs a local OLDEST policy on its assigned sub-queue (inter-thread scheduling). In the next stage, some of the selected OLDEST instructions in each sub-queue are granted to occupy ready functional units according to our thread sensitive policy (inter-thread scheduling).

Assume the number of active threads in the system is k which is also the partition factor. At time t (clock cycle), each thread i is associated with a value called thread metric or $M_{i,t}$. The maximum number of instructions granted for thread i is calculated according to (1). In each cycle, this value is updated for each thread and stored in its profile. Here $IssueRate_{total}$ is a constant equal to the maximum number of instructions which can be issued in each cycle by the processor. In fact this fixed value is the number of functional units which can be used in one cycle simultaneously and is always less than or equal to the total number of functional units.

$$G_{i,t} = \frac{M_{i,t}}{\sum_{i \in 1..k} M_{i,t}} * IssueRate \quad (1)$$

According to (1), a thread with higher metric value has higher chance to access functional units in each cycle. Each scheduler uses this limit ($G_{i,t}$) as the maximum issued instructions in its intra-thread scheduling phase (OLDEST algorithm on its sub-queue) for the next clock cycle. Assuming *OLDEST* (m) represents the OLDEST algorithm with the maximum issued instruction limit of m , a mixed SMT topology issue scheduler with OLDEST policy [7] can be represented by *OLDEST* ($IssueRate_{total}$). Our proposed scheduler is composed of k partitioned schedulers each running with *OLDEST* ($G_{i,t}$) at time t . Fig. 2 shows a schematic of one of k partitioned schedulers in our design.

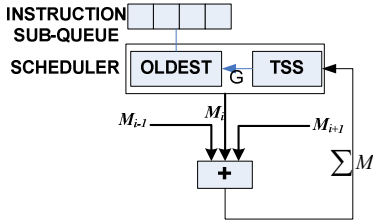


Fig. 2. Each partitioned issue scheduler in our design.

As shown in Fig.2, the scheduler is divided into two main sections:

1. An intra-thread section which runs a local OLDEST algorithm on its instruction sub-queue.
2. An inter-thread or thread-sensitive scheduler (TSS) which performs our issue policy (in a simple logic) and evaluates the result ($G_{i,t}$ values) which will be used in the next cycle with the distributed OLDEST algorithm.

The delay of the scheduler shown in Fig.2 is composed of three terms:

1. The delay of the partitioned intra-thread section (partitioned OLDEST scheduler) or T_{OLDEST} . It depends on the size of the sub-queue and is reciprocal of (k) as discussed before.
2. The delay of inter-thread section or T_{TSS} : According to (1), it is a function of bit-width of the used metric (M).
3. The delay of the adder or T_{ADDER} : It is proportional to both the width of the used metric (M) and the number of threads in the system (k). This direct dependency to the number of active threads in the system (k) is not desired. In other words, this central adder can be the bottleneck in the system and reduces its flexibility.

Considering T_{ADDER} and T_{TSS} terms, selecting a proper metric for the scheduler can significantly affect the system complexity and its required clock. We selected the ready instructions count (RIC) in the thread sub-queue as the scheduler. RIC values are limited to sub-queue size (n/k) which results in a small bit-width. For example in a CPU with an IQ of 256 (n) instructions, each sub-queue has 32 ($256/8$) entries for 8 active threads and so only 5 bits are required for storing the RIC of each thread. In addition, the maximum

value of $\sum M$ is equal to the size of IQ or n . So, we can also simplify (1) by replacing the sum in denominator with n . This will remove the central adder from the circuit of Fig. 2. On the other hand, n is usually a power of 2 or 2^i . Therefore, according to (1), the division required in TSS section will be converted to a simple right shift. It means that using RIC not only removes T_{ADDER} but also reduces T_{TSS} to the delay of a simple shift operation. In our implementation of *TSS*, we considered starvation condition. When there is any ready instruction for the thread but shifted G is equal to zero (which means $0 < 'G \text{ value according to (1)}' < 1$), G is replaced by 1 indicating one required functional unit. On the other hand, according to (1) we can deduce that $\sum G \leq IssueRate_{total}$. It means the system resources are fairly shared among active threads balanced on their real-time requirement and each thread with at least one active ready instruction in its IQ will have $G > 0$. This prevents starvation condition in the system.

In addition to delay considerations, using RIC metric can bring higher performance compared to other possible metrics. This metric reflects the run time requirements of each thread. It means that threads receive their required service according to their current requirements (ready instructions) for execution. Therefore, balancing the allocation of functional units in the system based on RIC leads to improved performance. This is checked in our simulation results which are presented in the next section.

V. EXPERIMENTAL RESULTS

We have evaluated our partitioned topology policy using RIC and IPC metrics and compared them with the most famous mixed topology issue policy, OLDEST [7]. We developed our SMT simulator on top of popular SimpleScalar Tool Set developed at the University of Washington-Madison [1]. This simulator supports a variation of MIPS instruction set [4]. That package contains several simulators for different purposes. In this package, *Simoutorder* simulator is a superscalar simulator. It simulates a superscalar processor with branch prediction, register renaming and out-of-order execution of instructions using two circular instruction queues: RUU (Register Update Unit) and LSQ (Load Store Queue). We modified this simulator such that the register files and thread context are replicated for each thread and system resources such as functional units, memory and TLBs are shared among them. Many features inherited from the original superscalar simulator, such as out-of-order and speculative execution, in-order commission, branch prediction and register renaming are still presented. Our fetch policy was *ICOUNT* [7] which provides higher performance [7]. We added the issue scheduler described in previous section to the simulator. In each cycle, our SMT scheduler selects threads for execution, receives their service requests and denies/accepts requests according to the scheduling metric of each thread. It accepts a request (ready instruction) from a thread if it does not violate the corresponding thread's maximum granted instructions limit (G) as defined in

previous section. We used programs in SPEC2000 and SPEC95 benchmarks [8] compiled for SimpleScalar version 2 as our testbenches. Processor parameters used in our simulations are shown in Table I.

TABLE I
PROCESSOR PARAMETERS USED IN SIMULATION

Parameter	Value
Instruction fetch queue size	4 instructions
Branch predictor	Bimodal (BTB size = 2048 bytes for each thread)
Miss prediction branch latency	3 cycles
Instruction decode bandwidth for each thread	4 inst/cycle
Instruction issue bandwidth (rate)	8 for 4 thread contexts, 12 for 8 thread contexts
RUU size (n/k)	16 each thread (separate)
LQS size	8 each thread (separate)
L1 cache	16K, 32-byte line, 4 way
L2 cache	64K, 32-byte line, 4 way
Cache latency	L1(1 cycle) L2 (3 cycles)
Functional units	8 integer alu, 2 integer multiplier, 4 memory ports, 8 floating point alu, 8 floating point multiplier

We run different combinations of 4-thread and 8-thread tests. Each combination includes various floating point, integer, I/O bound and Process bound test cases. Four combinations from them are shown in Table II.

TABLE II
THREAD COMBINATION SAMPLE IN OUR TESTS

4 thread test combinations	(compress, go, gcc, swim) (compress, go, mgrid, swim)
8 thread test combinations	(go, mgrid, swim, gcc, perl, applu, gzip, mcf, parser) (comp, go, gcc, perl, mcf, parser, swim, applue)

The tests are performed until each thread executes 40 million instructions. Results are shown in Fig. 3. The performance of the processor (overall instruction throughput or IPC) is evaluated for three schedulers:

1. MIX OLDEST: The famous mixed-topology scheduler [7].
2. OLDEST + TSS(IPC): Our distributed scheduler using famous IPC metric.
3. OLDEST + TSS(RIC): Our distributed scheduler using our newly defined metric (RIC).

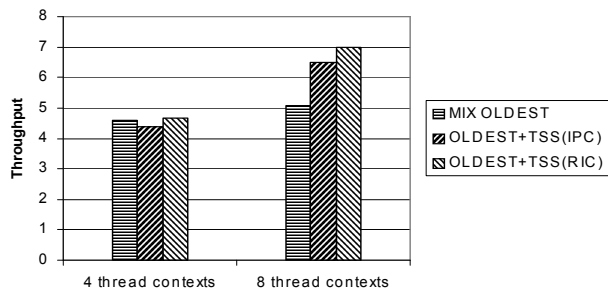


Fig. 3. Average performance results for 4 and 8-thread tests

All three schedulers behave similarly in low loads (with only 4) threads but partitioned topology with thread priority works much better (around 30% for 8 thread TSS(RIC)/OLDEST) in high loads because of higher inter-

thread parallelism. Adding new thread increases inter-thread parallelism and makes it dominant. This figure also shows that RIC brings much better performance compared to IPC when used as thread metric in our issue policy. As using RIC in 8-thread tests is more effective, we showed one of the 8-thread combinations in more details in Fig. 4. It is clear that RIC has positive effects on the throughput of all these programs which have different processing properties.

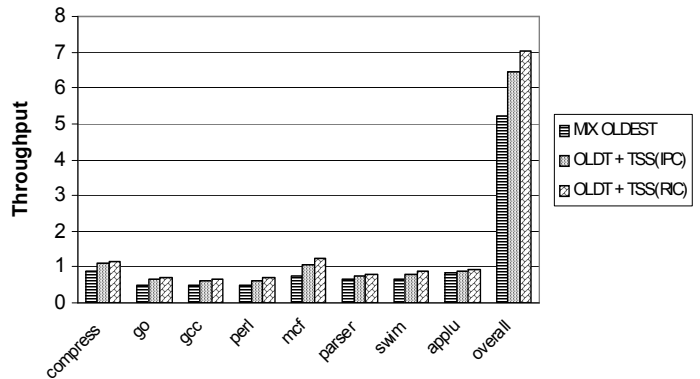


Fig. 4. Overall and individual Performance for one 8-thread test

VI. CONCLUSION

In this paper, we proposed a partitioned version of OLDEST SMT issue policy equipped with a thread sensitive scheduler which uses RIC scheduling metric. This policy uses both intra- and inter-thread parallelism in resource allocation. We showed that our method is scalable and more practical because of its distributed scheduling. In addition, as experimental results show, this policy achieves around 30% better performance for partitioned IQ SMT topology compared to the previous policies used for mixed IQ topologies.

REFERENCES

- [1] D. Burger and T. Austin, "The SimpleScalar Tool Set, v2.0," Technical Report UW-CS-97-1342, University of Wisconsin, Madison, 1997.
- [2] S. Egger, J. Emer, H. Levy, J.Lo, R. Stamm and D. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors," IEEE Micro, 1997.
- [3] R. Gonçalves, E. Ayguadé, M. Valero and P. Navaux, "A Simulator for SMT Architectures: Evaluating Instruction Cache Topologies," 12th Symposium on Computer Architecture and High Performance Computing, pp. 279-286, October 2000.
- [4] J. Hennessy and D. Patterson, "Computer Architecture--A Quantitative Approach," 3rd edition, Morgan Kaufmann Publishers, 2002.
- [5] S. Raasch and S. Reinhardt, "Applications of Thread Prioritization in SMT Processors", Proc. of Multithreaded Execution, Architecture, and Compilation Workshop (MTEAC), January 1999.
- [6] D. Tullsen, S. Eggers and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," Proc. of the 22nd Annual International Symposium on Computer Architecture, 1995.
- [7] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo and R. Stamm. "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," 23rd Annual International Symposium on Computer Architecture, 1996.
- [8] SPEC Website: <http://www.specbench.org>.