World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:3, No:7, 2009

QSI Dynamical Fetch Policy for SMT

Shu-Chiao Yang, Jong-Jiann Shieh

***Abstract*—**A Simultaneous Multithreading (SMT) Processor is capable of executing instructions from multiple threads in the same cycle. SMT in fact was introduced as a powerful architecture to superscalar to increase the throughput of the processor.

Simultaneous Multithreading is a technique that permits multiple instructions from multiple independent applications or threads to compete limited resources each cycle. While the fetch unit has been identified as one of the major bottlenecks of SMT architecture, several fetch schemes were proposed by prior works to enhance the fetching efficiency and overall performance.

In this paper, we propose a novel fetch policy called queue situation identifier (QSI) which counts some kind of long latency instructions of each thread each cycle then properly selects which threads to fetch next cycle. Simulation results show that in best case our fetch policy can achieve 30% on speedup and also can reduce the data cache level 1 miss rate.

***Keywords*—**SMT, QSI, DL1 miss rate.

I. INTRODUCTION

As the semiconductor technique improvement, the processor can have more transistors than before. So that computer designers use their wisdom to create a lot of architectures, such as branch predictor, memory hierarchy, trace cache, and superscalar, to improve computer performance.

In order to gain more performance, processor designers bring in two innovation concepts: instruction-level parallelism (ILP) and thread-level parallelism (TLP). Conventional superscalar processor can issue multiple instructions in single program each cycle. The purpose of conventional superscalar design is to exploit the ILP and improve the performance.

Simultaneous Multithreading (SMT) [1, 2, 3, 4] is a innovative technique combine the benefits of ILP and TLP. It can issue multiple instructions from multiple independent applications or threads each cycle. All treads in a SMT processor is active simultaneously, competing for all available resources each cycle.

The SMT architecture can be roughly divide into two parts: fetch engine and execution engine.

The fetch engine which is at the front-end of pipeline stages composes of fetch unit, instruction cache, branch predictor, decode unit, and register renaming unit. The major responsibility of the fetch engine is to fill the later pipeline stage with instructions. Each cycle, the fetch unit fetches multiple instructions from instruction cache. After decoding, the register renaming logic maps the logical register to the physical register to remove the data dependence that will cause data hazard. And
then the instructions will be fed into execution engine.

The execution engine which is at back-end of pipeline stages consists of reorder buffer, issue logic, functional units, data forwarding mechanism, and memory hierarchy. The responsibility of execution engine is issuing ready instructions to appropriate functional units, FUs, for execution. The renamed instructions will wait in the instruction queue for operands to become available. If the corresponding functional unit is free, the ready instruction is issued for execution.

II. The Bottlenecks of Simultaneous Multithreading

Although the SMT architecture dynamically sharing the processor resources to exploit both Instruction-Level Parallelism (ILP) and Thread-Level Parallelism (TLP) to enhance performance, but it does appear to have some potential drawbacks since the inter-thread contention. In SMT processor, multiple independent threads run concurrently to share resource of single processor and to increase resource utilization. However, the competing for resource between threads will degrade performance considerably. For example, sharing the cache with multiple threads, that is, partitioning the cache into pieces for threads will eventually reducing the cache space used by each thread, hence decrease the degree of locality and cause cache misses to arise.

Fetch unit is a prime bottleneck for SMT architecture. Branch frequency and PC alignment problems prevent SMT processor from fully utilizing the fetch bandwidth. Besides, since the instructions are from different threads now, the fetch unit needs to be smart enough to know which thread to fetch from. In fact, the fetch unit becomes one of the major bottlenecks of the SMT processor [3].

Issue logic is another candidate for bottleneck intuitively. The issue logic selects ready instructions from instruction queue for issuing in the pipeline stage. When the corresponding functional unit is free, the ready instruction is issued for an execution. As a matter of fact, the mechanism of selecting ready instructions from the instruction queue influences the throughput of processor significantly. A dynamically scheduled single thread processor may have enough ready instructions to be able to choose between them, but in SMT processor the options are more diverse. Because SMT processor have higher throughput than a single thread superscalar processor, the issue bandwidth is potentially a more critical resource, so avoiding issue slot waste seems beneficial [3].

III. Previously Investigated Fetch Schemes

Tullsen et al. proposed several fetch policies for SMT processors [3]. In order to improve the original fetch policy: round robin, they first bring in the new idea to choose the instructions that fetch in next cycle by using the feedback from the processor pipeline.

BRCOUNT fetch policy attempts to give highest priority to those threads which has fewest unresolved branches in the decode stage, the rename stage, and the instruction queues. Therefore, instructions fetch from the wrong path will be reduced by choosing the threads that have fewer branches or have more resolved branches.

MISSCOUNT fetch policy attempts to avoid the clog of IQ causing by long latency instructions. This fetch policy attempts to give highest priority to those threads which has fewest outstanding data cache miss instructions. Since dealing with the data cache miss instructions will spend more clock cycles. The subsequence instructions of the thread dependent on that data will wait in the IQs for a period of time that clogs the IQs.

ICOUNT fetch policy is a more general solution that use to deal with IQ clog problem. This police gives priority to threads with the fewest instructions in decode, rename, and the instruction queues.

Chen and Shieh [10] proposed a novel fetch policy called Instantaneous Commit Count (ICC) mechanism that selects which thread to fetch from according to the collection of each thread’s retired instructions each cycle.

IV. Our Fetch Policy

Let we review the two fetch policies that we mentioned above: ICOUNT and ICC.

ICOUNT proposed by Tullsen et al. that were designed to prevent the IQ clog. This fetch policy gives high priority to those threads which have fewest instruction in decode, rename, and the instruction queues. However, this priority tends to favor some threads, such as the thread was just flushed by mis-speculation. Moreover, the favor behavior results in the high utility rate of RUU.

ICC proposed by Shieh et al. that also attempted to prevent the IQ clog. This fetch policy gives high priority to those threads which have more committed instructions each cycle. This fetch policy has better performance than ICOUNT. It also has lower utility rate of RUU than ICOUNT. However, this priority only count the commit instruction each cycle but didn’t consider some instructions of high priority thread had stayed in queues more time than low priority threads.

In order to solve those problems we use several counters to record the states of the processor. Long latency loads is one of the well known factors to bring in IQ clog. When thread with load miss execute in SMT, the thread will eventually stall, potentially holding resources which other threads could be using to make forward progress. Hence, we use counters to count the load miss instructions of each thread each cycle. Floating point computation is another well known factor that incurs IQ clog. The average computation time of floating point operation is almost 6 times of that of integer operation. In all floating point computation, floating point multiplication and division spend more cycles than addition and subtraction. In situation that there are many threads with floating point computations run currently in SMT, threads that compete the limited functional units each other and wait the winner to release the resource will result in clog. So, we use counters to count the floating point multiplication and division instructions of each thread each cycle. Finally, we combine the three information, number of load miss instructions, number of floating point multiplication and number of division instructions, to a parameter which we called queue situation identifier (QSI).

In this paper, our fetch policy is dynamically switching according to the QSI. The fetch policy is described as follow:

The fetch unit fetches instructions from instruction cache according to fetch priority that decided in prior cycle. When an instruction is fetched and detected that instruction is one of the three kinds of instructions we mentioned above, the QSI will be add one. In commit stage, if the instruction is one of the three kinds instructions we mentioned above, the QSI will be decrease one. Then, we set a threshold to QSI and check the QSI at every commit cycle. The switch flag arise or not is according to which fetch policy used at this cycle and the relationship between QSI and the threshold. The state diagram of switching fetch policy is shown in Figure 1. The fetch unit sorts ICC and QSI counter and looks up which fetch policy is going to use in this cycle to decide the priority of thread to fetch. The circuit diagram of switch flag is shown in Figure 2.

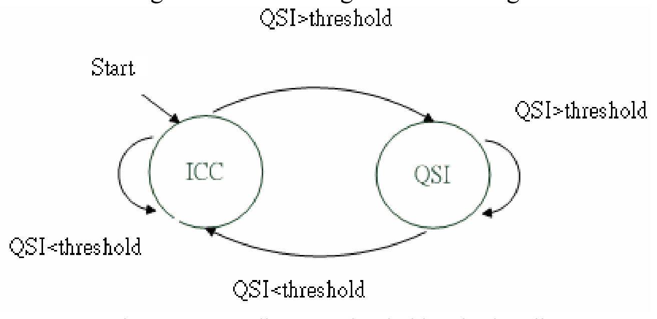
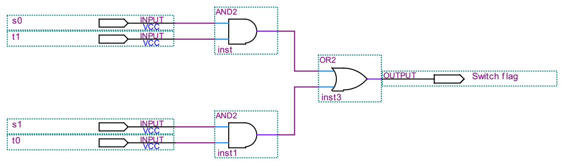


Figure 1. State diagram of switching fetch policy



s0: state 0 s1: state 1
t0: QSI > threshold t1: QSI < threshold

Figure 2. The circuit diagram of switch flag

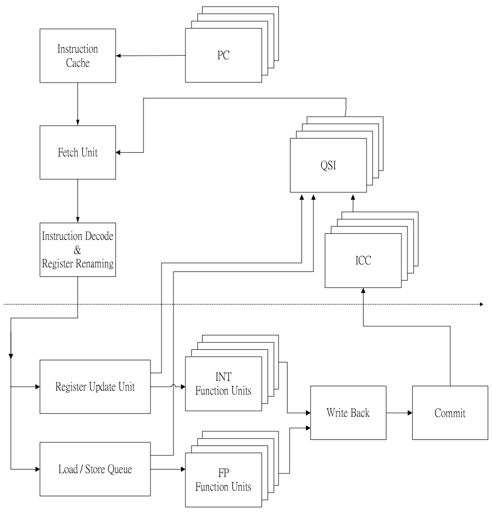


Figure 3.The block diagram of QSI architecture

V. Evaluation Methology

Our simulator is derived from the SimpleScalar Multithreading (SSMT) simulator which originally developed by Madon et al. [7] to implement our fetch schemes and gather detailed statistics. The simulator implements simultaneous multithreaded processor pipeline based on the out-of-order processor model from SimpleScalar tool set [8].

The major parameters of the simulator we used are shown in Table 1, and the configuration of functional units are shown in Table 2. The Instruction and memory access latency is shown in Table 3.

In this paper, we select 11 applications (alpha ISA) from the SPEC CPU2000 suite to construct our workloads. The workloads consist of eight integer based applications from CINT2000 benchmark suite and three floating-point based applications from CFP2000 benchmark suite.

TABLE 1.
Baseline parameter of simulator

Parameter	configure
Base Fetch Policy	ICOUNT
Fetch/Issue/Commit Width	8
Fetch Queue Size	32
Register Update Unit Size	128
Load/Store Queue Size	64
L1/L2 Cache Block Size	32Byte/64Byte
Instruction/ Data Cache	128KB,2 way
L2 Cache	2MB,4 way
Fast-Forward Instructions(Each	250,000,000
Commit Instructions(Each thread)	200,000,000

TABLE 2.
Functional Units Configuration

Parameter	Value
Integer ALU	8
Integer MULT/DIV	2
Floating ALU	8
Floating MULT/DIV	2

TABLE 3.
Instruction and memory access latency

Latency Type	cycles
Integer	1
FP Add	2
FP Multi	4
FP Div	12
L1 Hit	1
L2 Hit	10
Memory Access	122

TABLE 4. Workloads of 2, 4, 6 and 8 applications

Number of thread							
2		4		6		8	
Integer		Integer		Integer		Mix	
W21	mcf,	W41	mcf, gzip, crafty,	W61	mcf, gzip, crafty, twolf, vpr,	W81	mcf, gcc, gzip, crafty, twolf, bzip2, vpr, art
W22	bzip2,	W42	gcc, crafty, gzip,	W62	vpr, gcc, mcf, bzip2, twolf,		
W23	gap,	W43	mcf, gap, bzip2,	Mix		W82	mcf, gcc, gzip, gap, bzip2, vpr, art, mesa
Floating		W44	mcf, crafty, gcc,	W63	gcc, twolf, gzip, mesa, art,		
W24	equake,	Mix		W64	bzip2, crafty, gzip, twolf,	W83	mcf, gcc, gzip, twolf, bzip2, mesa, art, equake
W25	mesa,	W45	mcf, bzip2, mesa,	W65	mcf, gzip, twolf, equake,		
W26	equake,	W46	gcc, gzip, equake,	W66	mcf, gzip, crafty, twolf, gcc,	W84	mcf, vpr, gzip, twolf, bzip2, mesa, art, equake
Mix		W47	twolf , vpr, mesa,				
W27	gcc, art	W48	bzip2, mcf, vpr,	Floating: All Floating Point Based			
W28	vpr,			Integer: All Integer Based			
W29	bzip2,			MIX: Mix of Integer and Floating Point Based			

VI. Simulation Results

Figure 4 to 7 show the number of instructions fetch per cycle normalized to base fetch scheme. As Figure 4 shows our fetch scheme fetches almost the same number of instructions (the difference is within 1% on average) as ICOUNT and ICC in 2-thread workloads because the fetch unit can only fetch from two threads each cycle. In 4-thread workloads, our QSI fetch policy improves the fetch rate 11%, 10%, 13%, 5%, 18%, 11% and 3% in W41, W42,W43, W44, W45, W46 and W47 respectively, as shown in Figure 5. However, for workload W48 our fetch policy fetches fewer instructions because the counted number in QSI shows that for these benchmarks the number of the three specified instructions are distributed more looser than other workloads. This situation resulted in the switch to QSI state less than other workloads. In 6-thread workloads as we shown in Figure 6, QSI can achieves 36%, 16%, 7%, 24%, 31% and 23% in W61, W62, W63, W64, W65 and W66 respectively. In 8-thread workloads as we shown in Figure 7, our fetch policy can achieves 39%, 36%, 43% and 39% in W81, W82, W83 and W84 respectively.

Figure 8 shows the level 1 data cache miss rate in 2-thread workloads, our fetch policy has same DL1 miss rate (the reduction percentage is 0.7) as ICOUNT and ICC. In 4-thread workload as shows in Figure 9, QSI can reduce the DL1 miss rate 3.55% in average. In Figure 10 shows the 6-thread workloads, QSI can reduce the DL1 miss rate 7%. And in Figure 11, our QSI fetch policy use in 8-thread workloads can reduce the DL1 miss rate up to 8.2% in average.

Figure 12 to 15 show the IPCs of the combined workloads. As illustrating in Figure 12, 2-thread workloads get almost the same performance on each scheme (the difference is between 1%). In 4-thread workload as shown in Figure 13, our fetch policy improve IPC by 11.4%, 9.5%, 11.9%, 12.1%, 3.9%, 11.8% and 2.56% in W41, W42, W43, W44, W45, W46 and W47 respectively. In 6-thread workload as shown in Figure 14, QSI will gain 30.6%, 19.8%, 6.0%, 0.7%, 21.4% and 14% more IPC in W61, W62, W63, W64, W65 and W66 respectively. In 8-thread workload as shown in Figure 15, these numbers are 30.2%, 22.3%, 23.7% and 17.6% for W81, W82, W83 and W84 respectively.

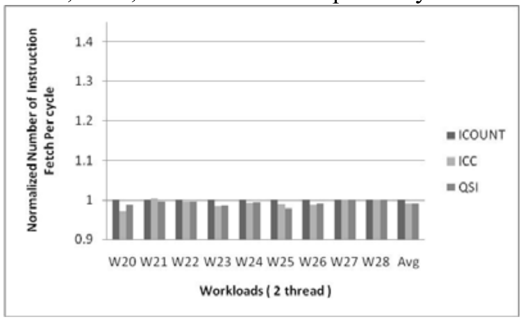


Figure 4. Normalized fetch rate of 2-thread workloads

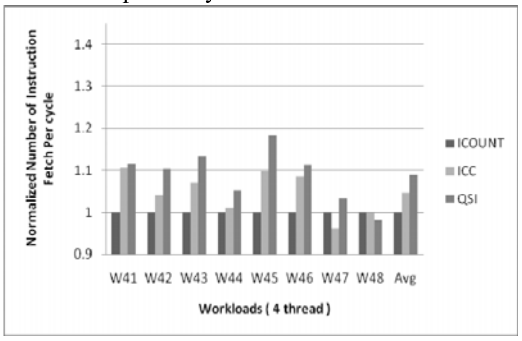


Figure 5. Normalized fetch rate of 4-thread workloads

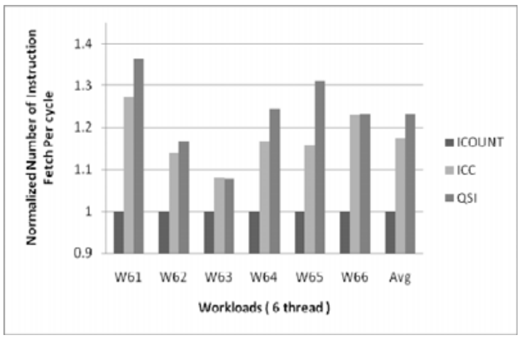


Figure 6. Normalized fetch rate of 6-thread workloads

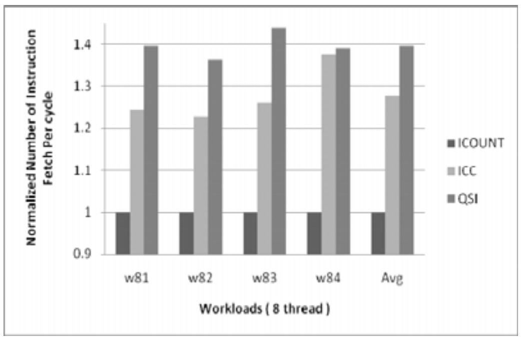


Figure 7. Normalized fetch rate of 8-thread workloads

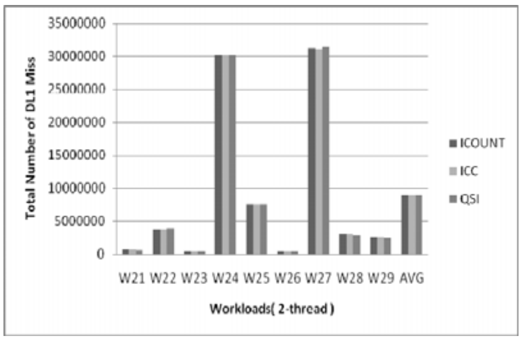


Figure 8. DL1 Miss Rate of 2-thread workloads

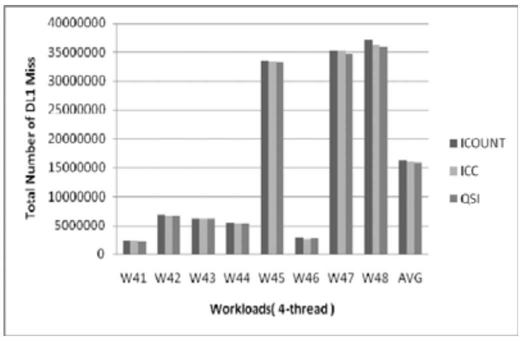


Figure 9. DL1 Miss Rate of 4-thread workloads

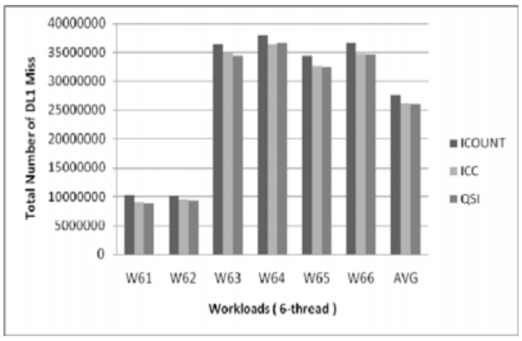


Figure 10. DL1 Miss Rate of 6-thread workloads

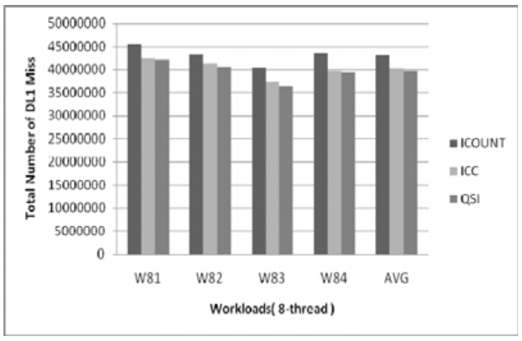


Figure 11. DL1 Miss Rate of 8-thread workloads

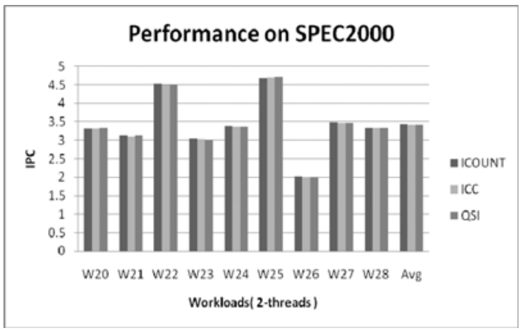


Figure 12. Performance of 2-thread workloads

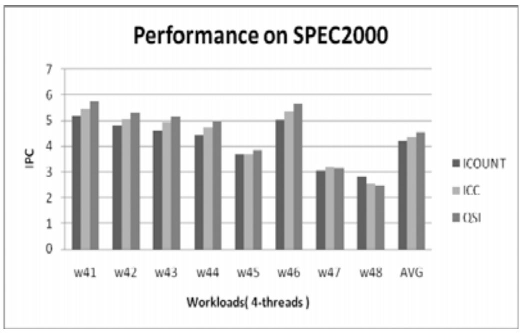


Figure 13. Performance of 4-thread workloads

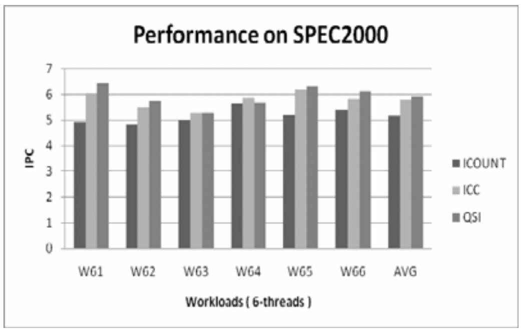


Figure 14. Performance of 6-thread workloads

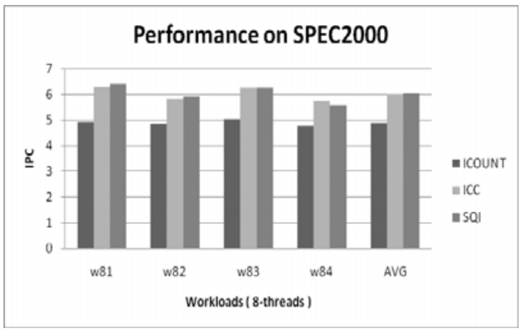


Figure 15. Performance of 8-thread workloads

VII. CONCLUSIONS

In this paper, we proposed a fetch policy which is based on ICC to solve the problem of clog that we find in ICC fetch policy. In our policy, we add QSI counters per-thread and dynamical monitor the QSI value. If the QSI is greater than the threshold then the fetching policy is switched to QSI mode, otherwise the used fetching scheme is ICC. Our motivation is to gain further performance by fast forwarding thread and using the resource effectively. Our results showed a significant improvement (achieve the highest performance up to 30.1% in some workloads) over the ICOUNT fetch policy and also can reduce the DL1 miss rate.

REFERENCES

[1] D. Tullsen, S. Eggers, and H. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” In 22nd Annul International Symposium on Computer Architecture, June 1995, Pages 392-403
[2] D. Madon, E. Sanchez, and S. Monnier, “A Study of a Simultaneous Multithreaded Architecture,” In Proceedings of EuroPar'99, Toulouse, Lectures Notes in Computer Science, Volume 1685, Springer-Verlag, Sep. 1999, Pages 716-726
[3] 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,” In 23rd Annul International Symposium on Computer Architecture, May 1996
[4] S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, and D. Tullsen, “Simultaneous multithreading: A platform for next-generation processors,” IEEE Micro, Sep. 1997, Pages 12-18
[5] D. Tullsen, and J. Brown, “Handling long-latency loads in a simultaneous multithreading processor” In 34th Annual International Symposium on Microarchitecture, December, 2001
[6] Y-H. Chen, and J.-J. Shieh, “ICC: A Simultaneous Multithreading Fetch Engine”2005 National Computer Symposium, 15-16 Dec. 2005, Pages 59-59
[7] D. Madon, E. Sanchez, and S. Monnier, “A Study of a Simultaneous Multithreaded Architecture,” In Proceedings of EuroPar'99, Toulouse, Lectures Notes in Computer Science, Volume 1685, Springer-Verlag, Sep. 1999, Pages 716-726
[8] T. Austin, E. Larson, D. Ernst, “SimpleScalar: an infrastructure for computer system modeling,” IEEE Computer Journal, Feb. 2002, Pages 59-67
[9] D.M. Tullsen, J.A. Brown.“Handling Long-latency Loads in a Simultaneous Multithreading Processr,”In 34th International Symposium on Microarchitecture, , Dec. 2001, Pages 318-327.
[10]Y-H. Chen, and J.-J. Shieh, “ICC: A Simultaneous Multithreading Fetch Engine”2005 National Computer Symposium, 15-16 Dec. 2005, Pages 59-59