Explore Be-Nice Instruction Scheduling in Open64 for an Embedded SMT Processor

Handong Ye Ge Gan Ziang Hu
Guang R. Gao
University of Delaware, Newark Delaware 19716, U.S.A
{handong,gan,hu,ggao}@capsl.udel.edu

Xiaomi An†
†SimpLight Nanoelectronics, Ltd., Beijing China 100088
xiaomi.an@simplnano.com

Abstract

A SMT processor can fetch and issue instructions from multiple independent hardware threads at every CPU cycle. Therefore, hardware resources are shared among the concurrently-running threads at a very fine grain level, which can increase the utilization of processor pipeline. However, the concurrently-running threads in a SMT processor may interfere with each other and stall the CPU pipeline. We call this kind of pipeline stall *inter-thread stall* (**ITS** for short) or *thread interlock*. In this paper, we present our study on the ITS problem on an embedded heterogeneous SMT processor. Our experiments demonstrate that, for some test cases, **50**% of the total pipeline stalls are caused by ITS. Therefore, we have developed a new instruction scheduling algorithm called **be-nice** instruction scheduling, based on Open64 Global Code Motion, to coordinate the conflicts between concurrent threads. The instruction scheduler uses the thread interference information (obtained by profiling) as heuristics to decrease the number of ITS without sacrificing the overall CPU performance. The experimental results show that, for our current test cases the be-nice instruction scheduler can reduce **15**% of the inter-thread stall cycles, and increase the IPC of the critical thread by **2**%-**3**%. The experiments are performed using the Open64 compiler infrastructure.

1. Introduction

SMT is a very successful architecture design that can effectively improve CPU utilization (1) (2) in face of the ever increasing long memory access latency (3) and the limits of instruction level parallelism available in a single thread (4). Many chipmakers support SMT in their high-end products. Some examples are, IBM Power5, Sun UltraSparc 3, Intel Xeon, and Alpha21464. Now, this trend has been also extended to the architecture design of embedded processors, which are widely used in hand-held devices.

CGO 2008 Open64 Workshop April 6th, Boston.
Copyright © 2008 ACM [Open64 Workshop]...$5.00

Simultaneous multithreading (1) (2) permits multiple independent hardware threads to share the CPU pipeline resources when they execute concurrently. In every CPU cycle, a SMT processor is able to fetch instructions from multiple hardware threads and issue these instructions into the processor pipeline (5). Therefore, hardware resources, i.e. functions units, are shared among the concurrently-running threads at a very fine grain level, which is an effective method to improve the utilization of the processor pipeline resources. However, concurrently-running threads in a SMT processor may interfere, or even conflict, with each other, and thus stall the CPU pipeline. We call this kind of pipeline stall as *inter-thread stall* (**ITS** for short) or *thread interlock*.

An ITS happens when:

1. A function unit, or all function units of the same type, are occupied. Therefore, the subsequent instructions (from different threads) in the issue queue (6) can not be dispatched into the pipeline; and
2. the occupied function unit is hold by an instruction that may execute for a long time, like load/store and unpipelined floating point operations.

ITS or thread interlock, essentially, is a kind of **resource hazard** that happens between two threads. In the single thread system, a crafty compiler with a smart instruction scheduler can decrease the probability that resource hazard would happen thus alleviate its effect on CPU performance. However, the conventional instruction scheduling algorithms (7) (8) (9) (10) (11) (12) (13) (14) (15) (16) only take into account the code in one thread when they do instruction scheduling. They assume that the code being compiled would run as a single thread program. Without considering the interference from another concurrent thread, these algorithms can do little to alleviate ITS. This motivates us to re-study the instruction scheduling problem in face of inter-thread stall in SMT processors.

Generally, ITS may happen in two sets of circumstances: *heterogeneous multithreading* and *homogeneous multithreading*. Heterogeneous multithreading means that, among all concurrent threads, a subset of threads (usually contains only one thread) perform mission critical jobs, so they have higher priorities than other concurrent threads. For this reason, the methods used to optimize ITS shall favor the threads with higher priorities and their performances are the major metric used to evaluate the effectiveness of the particular ITS optimization technique (intuitively, the ”overall performance” can not be deteriorated too much). On the other hand, homogeneous multithreading means that all the concurrent threads are equally important. In this condition, the overall throughput of the system shall be used as the metric to evaluate the ITS optimization.

ITS happens more often on embedded SMT processors than on general wide-issue superscalar processors. The reason is that an embedded processor usually has less function units than its superscalar counterpart, so resource contention on embedded processors is more intensive than on superscalar processors In addition, most embedded processors do not have cache memory. Examples are Motorola 68HC12 (17), Motorola MCore (18), and Texas Instruments TMS370Cx (19). Even some embedded chips have cache, the prefetch logic is usually not included in the chip because of its high power consumption and hardware complexity. This would cause more long-latency load operations that make ITS a serious problem.

In this paper, our study on the ITS problem is focused on an embedded SMT processor that supports only in-order execution. Besides, we restrict our discussion only to the heterogeneous multithreaded execution.

Our experiments demonstrate that, for some test cases, 50% of the total pipeline stalls are caused by ITS. To alleviate this problem, we have developed a new instruction scheduling algorithm called **be-nice** instruction scheduling to coordinate the conflicts between concurrent threads and therefore reduce the ratio of ITS. First, our be-nice instruction scheduler needs to obtain the thread interference information. So, in the first pass, we profile the multithreaded program to identify the pieces of code that cause most of the ITS, and record the runtime status of the processor pipelines. This information is collected, analyzed, and structured into well defined format. In the second pass, it is fed into the compiler, just right before instruction scheduling phase. The instruction scheduler uses the thread interference information as heuristics to try to decrease the number of ITS's without sacrificing too much the overall CPU performance. We have performed some micro-benchmark experiments on the Open64 (20) compiler infrastructure. The experimental results show that, our be-nice instruction scheduler can reduce 15% of ITS cycles, and increase the IPC of critical thread by 2%-3%.

The paper is organized as follows. Section 2 will introduce the ITS problem using a particular embedded SMT processor. We will also give some experimental data in this section to show how CPU performance is affected by ITS. Next, in section 3, we present our be-nice instruction scheduling algorithm, including profiling and scheduling. In section 4, we will use some micro-benchmarks to verify the effectiveness of the new algorithm. Related works will be briefly introduced in section 5 and the conclusions are made in section 6. Last, our future work is presented in section 7.

2. Inter-Thread Stall in an Embedded SMT Processor

In this section, we first introduce the architecture details of **JIAN**, a dual-threaded SMT embedded processor. Then, based on this particular SMT processor, we use a real example to introduce the ITS problem. At the end of this section, we provide some experimental data to show how ITS affects CPU performance.

2.1 JIAN: a Dual-Threaded Embedded SMT Processor

JIAN is an embedded processor targeted at wireless multimedia applications. The processor core adopts a simultaneous multithreading design that is capable of executing control, DSP, and multi-media applications in a single instruction set architecture. It provides substantial parallelism and high throughput for communication applications in hardware, and at the same time, it still maintains low power consumption, low cost and high-level language programmability.

JIAN is a multi-issue super-scalar machine that effectively utilizes and leverages expensive hardware resources and achieve performance goals without requiring too high a clock frequency. Figure 1 illustrates the structure of the processor core. JIAN has a unified processor core shared by domain-specific processing and general control-intensive processing. The two functional tasks, instead of having their specific execution resource, are represented by two hardware threads which run on the same execution pipeline with different storage space (register file). This allows better utilization of hardware resources and better load balancing.

Before going into details, we define three terms: *heterogeneous multithreading*, *α-thread*, and *β-thread* in the context of JIAN SMT processor. The definition of these terms would make it easy for us to describe the ITS problem. Thus they are used throughout this paper.

DEFINITION 2.1. *Heterogeneous multithreading is a non-symmetric thread execution model on JIAN processor. In this model, two hardware threads perform different types of tasks, and one thread has higher priority than the other because it performs mission critical tasks.*

DEFINITION 2.2. *α-thread is the hardware thread on JIAN processor that runs general control-intensive workloads, like runtime system or OS. Since these are not mission critical jobs, α-thread has lower priority in this heterogeneous multithreading system.*

DEFINITION 2.3. *β-thread is the hardware thread on JIAN processor that runs domain specific workloads, like DSP, mpeg4 encode&decode, etc. Since these are mission critical jobs, β-thread has higher priority than α-thread.*

On the JIAN processor, this *heterogeneity* is reinforced in hardware. See the next paragraph for details.

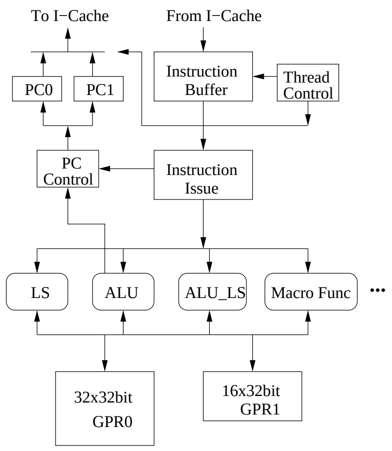

Figure 1. JIAN: Dual-Threaded SMT Processor Core Architecture

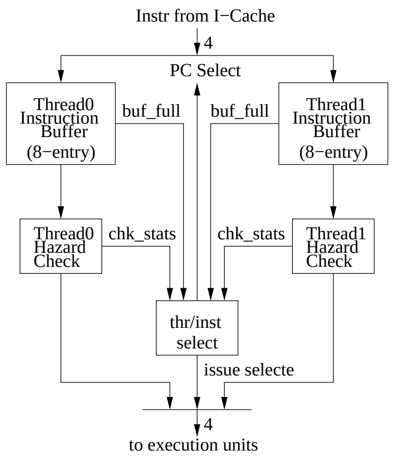

Figure 2. JIAN: Thread Control and Management

Every cycle, four instructions from one thread are fetched from the I-Cache into the thread's instruction buffer. The two threads are alternated during instruction fetch. In that case, only one thread's instructions will be fetched every cycle. Because the processor core is an in-order machine, only the earliest instructions will be checked to see if they can be issued or not. Currently, the maximum issue width from α-thread is two and the maximum issue width for β-thread is three. This is to save issuing logic and register ports. At every cycle, up to two earliest instructions from α-thread's instruction buffer and three earliest instructions from β-thread's instruction buffer will go through hazard checks. See Figure 2. Up to four cleared instructions will be selected to enter execution blocks at the next cycle. If everything is equal, i.e. all five instructions are cleared of hazard, β-thread has a higher priority than α-thread. That means three instructions from β-thread and one instruction from α-thread will be issued.

While the execution engine is unified and shared for two threads for better load balancing and hardware utilization, storage such as register file has to be separated. The register file of 32x32-bit is used for the α-thread to be like many typical RISCs. The smaller 16x32-bit is used for β-thread.

2.2 Inter-Thread Stall in Heterogeneous Multithreading

The current implementation of JIAN SMT architecture has one load/store unit (*LS*), one ALU unit (*ALU*), one hybrid function unit that can do both load/store and ALU (*ALU_LS*), and several macro function units which perform a specific kernel routine, like FFT or Viterbi. See Figure 1 for details. Currently, our study of the ITS problem is focused on the three simple functional units: LS, ALU, and ALU_LS. To make the problem easier to understand, we use the classical 5-stage pipeline to illustrate how ITS happens in the JIAN processor core. The five pipeline stages are denoted as *IF*, *ID*, *EX*, *MEM*, *WB*. These are very well known abbreviations, so we do not need to explain their meaning here.

The two pieces of code below is a runtime snapshot of the status of the instructions in the pipeline. These code are dumped out from the JIAN performance simulator. The first group of instructions are from α-thread, and the second group of instructions are from β-thread.

From α-thread

```
I1 FU:LS MEM ldw16 r31,r11 #miss
I2 FU:ALU_LS MEM add16.i r3,1
```

From β-thread

```
I3 FU:ALU_LS EX add16.i r10,4
I4 FU:LS ID ldw r24,r2,-8
I5 FU:ALU ID add16.i r3,1
I6 FU:ALU_LS ID ldw r25,r10,-4
```

Figure 3. An Inter-Thread Stall Example

The comment *#miss* at the end of I1 indicates that a cache miss happened when instruction I1 was executing on *LS*. Since JIAN is an in-order machine (in-order issue, in-order execution, and in-order commit), I2 was also stalled on *ALU_LS*. Therefore, all function units that can do load/store operations are occupied. This made the instrucitions from β-thread, i.e. I4&I6, could not be issued to the function units that were assigned to them. So, the cache miss in α-thread stalled β-thread, which is named *ITS*.

As we have mentioned, α-thread and β-thread are not equally important. β-thread always has higher priority than α-thread, thus we care for the performance of β-thread more than α-thread. So, this kind of ITS is what we intend to avoid at runtime.

ITS is caused by long-latency instructions executed on SMT processors. These instructions usually are cache-miss load and non-pipelined floating point operations. Theoretically, the amount of performance degradation caused by ITS is decided by three factors:

1. The number of long-latency instructions executed per time unit
2. The number of function units occupied by the same thread when it executes the long-latency instructions.
3. The number of idle cycles caused by the long-latency instruction.

On a general SMT processor, any thread may lock any other concurrent thread. In our case, a heterogeneous dual-threaded SMT processor, we care the performance of β-thread more than the performance of α-thread, because β-thread usually run mission critical jobs. In the next section, we will present some experiment data to show how ITS affects program performance.

2.3 How Inter-Thread Stall Affects the Performance of β-thread

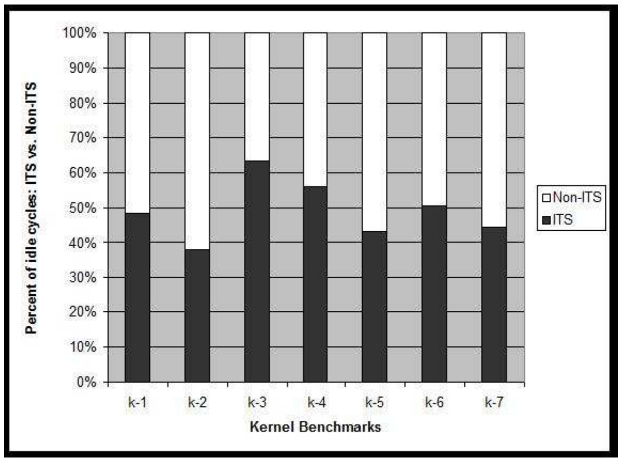

Figure 4. Stall Cycle Breakdown of β-thread: inter-thread stall vs. intra-thread stall

Before we start thinking of the solution for the inter-thread stall (or ITS for short) problem, it is very helpful to know whether ITS affects the performance of β-thread significantly, especially how it compares with other similar performance degradation factors. In an SMT processor like JIAN, there are roughly two types of pipeline stalls: intra-thread stall and ITS. Intra-thread stall is the pipeline hazard (control, data, or resource) caused by preceding instructions in the *same* thread. As we all know, intra-thread stall plus ITS account for all stall cycles in the execution of β-thread.

Figure 4 is the stall cycle breakdown for all β-threads of seven micro-benchmarks. In *k-2*, 37.75% (the lowest) stall cycles are ITS cycles, and in *k-3*, 63.40% (the highest) stall cycles are ITS cycles. In average, for all benchmarks, 50% of all stall cycles are ITS cycles. This means that ITS accounts for a significant proportion of performance degradation. This also indicates that β-thread is equally stalled by α-thread (ITS) and itself (intra-thread stall).

These data are obtained from the inside of JIAN performance simulator. At every CPU cycle, the simulator can check whether β-thread is stalled. If it detects that the β-thread is stalled, it can analyze the status of each pipeline stage to see whether this is an intra-thread stall or ITS. If it is an ITS, a finite state automata will be started to record all microarchitecture status changes at each following CPU cycle, until the stall disappears, thus obtains the length of this pipeline stall. Because in JIAN architecture, the instructions are issued, executed, and retired in order, only one finite state automata is needed.

3. A Be-Nice Instruction Scheduling Algorithm

In the last section, we introduced the ITS problem, and demonstrated how it degrades the performance of β-thread in JIAN - a heterogeneous multithreaded system. In this section, we will propose a new instruction scheduling algorithm that can reduce the number of ITS. Since this method boosts the performance of one thread (β-thread) through instruction scheduling in the other thread (α-thread), we call it *be-nice* instruction scheduling.

3.1 The Be-Nice Instruction Scheduling Framework

There are several hardware approaches that can either eliminate a certain number of ITS or alleviate their adverse impact to the performance of β-thread. First, the issue logic of the processor can be improved such that instructions from β-thread would not be issued into the functional unit that is already hold by a long-latency instruction in α-thread. Second, people can adopt an out-of-order issue & execution engine instead of an in-order one, so the successive independent instructions can be issued and executed even if the previous instructions are stalled. Third, hardware can use some kind of prefetching mechanism to reduce the number of cache miss, therefore reduce the number of long-latency load operations. All these hardware solutions require extra complexity in hardware design, and demand disproportionally high power consumption, which is not acceptable for an embedded processor. High-end SMT processors can use these hardware approaches to solve the ITS problem. For an embedded SMT processor like JIAN, we would prefer to using a power efficient method - static instructions scheduling at compile time.

The new instruction scheduling algorithm consists of four steps:

1. Use the SMT simulator to identify the code sequences in the two threads that would interfere with each other frequently. At the same time, the interference information will be collected and dumped into a file, which is called *interference record file*.
2. The interference records are analyzed offline. Not all interference records are useful, and only part of them will be analyzed and interpreted to make annotation possible and easy.
3. At the second round compilation for α-thread, the interference record file is read and used to annotate the instructions before scheduling.
4. When performing instruction scheduling, instructions that cause many ITS will be lazily scheduled according to the annotation.

So be-nice scheduling is only applied to α-thread, and β-thread isn't re-scheduled any more. In the next several sections, we will introduce how each step is performed in details.

3.2 Profiling and Analysis

The goal of profiling is to obtain the accurate thread interference information that can be used (as heuristics) by the instruction scheduler. Thread interference information tells where in α-thread and β-thread that ITS happened; the number of idle cycles caused by the ITS; the functional units that were blocked in the ITS; and the instructions (in both α-thread and β-thread) that were stalled in the ITS. In this paper, we only focus on the ITS problem in the context of the heterogeneous multithreaded SMT processor, i.e. the stalls happened in β-thread due to the long latency operations in α-thread.

Thread interference information is collected by JIAN processor, and we have implemented this feature in the performance simulator. When the simulator detects an ITS, it records the necessary machine states and program states. In order not to slowdown the simulation too much, we only record the address of the first instruction in β-thread that was blocked because of the ITS; the address of the instruction in α-thread that caused the ITS; and the number of idle cycles.

Formally, thread interference information consists of an array of *interference record*. An interference record is defined as triplet:

$$(I_i^\beta, I_j^\alpha, \tau_k) \tag{1}$$

The first element I_i^β is the address of the first instruction in β-thread that was blocked by α-thread. The second element I_j^α is the address of the instruction that caused ITS. The third element τ_k is the penalty caused by the ITS. So, the information contained in the *interference record file* is an array which looks like this:

$$[(I_{i_1}^\beta, I_{j_1}^\alpha, \tau_{k_1}), (I_{i_2}^\beta, I_{j_2}^\alpha, \tau_{k_2}), ..., (I_{i_l}^\beta, I_{j_l}^\alpha, \tau_{k_l}), ...] \tag{2}$$

Not all interference records will be used to direct instruction scheduling. We will filter out some records according to these criteria:

1. For an interference record $(I_{i_l}^\beta, I_{j_l}^\alpha, \tau_{k_l})$, if $I_{i_l}^\beta$ does not belong to the *hot* portion of code in β-thread;
2. For an interference record $(I_{i_l}^\beta, I_{j_l}^\alpha, \tau_{k_l})$, if τ_{k_l} is smaller than t - the penalty threshold, which is a configurable parameter;
3. For an interference record $(I_{i_l}^\beta, I_{j_l}^\alpha, \tau_{k_l})$, if the frequency that $I_{j_l}^\alpha$ appears in the interference array is not big enough.

Among all the above criteria, the last two is configurable. Currently, the *ITS penalty threshold* used in the second criteria is set to 5, and as to the third criteria, 25% of the least frequent $I_{j_l}^\alpha$ are removed. Later, we merge all interference records with the same $I_{j_l}^\alpha$ to one record. The ITS penalty of the new record is the arithmetic mean of the ITS penalties in the old records. Therefore, we get a new version of thread interference file which contains an array of records like this:

$$[(I_1^\alpha, \overline{\tau_1}), (I_2^\alpha, \overline{\tau_2}), ..., (I_j^\alpha, \overline{\tau_j}), ...] \tag{3}$$

For each record in this file, we map I_j^α back to the assembly code and get the information that can be used by compiler, i.e. a triplet (X^{pu}, Y^{bb}, Z^{op}), which are used to denote the *PU* number, the *BB* number and the *OP* number in Open64 code generator. So, the final version of interference file becomes an array of records:

$$[((X_1^{pu}, Y_1^{bb}, Z_1^{op}), \overline{\tau_1}), ..., ((X_i^{pu}, Y_i^{bb}, Z_i^{op}), \overline{\tau_i}), ...] \tag{4}$$

Finally, these records will be used to annotate the corresponding OP to direct the instruction scheduling.

3.3 Annotation and Instruction Scheduling

The annotation of ITS information and be-nice instruction scheduling are performed in the code generator of the Open64 compiler (20). The flow chart in Figure 5 outlines the framework of the Open64 code generator, *CG* for short. It shows the order of each important optimization phases performed in it. Open64 CG has its own intermediate representation, called *CGIR*. It is expanded from *WHIRL*, which is the major intermediate representation of Open64 used in *VHO*, *IPA/IPO*, *LNO*, and *WOPT*. CGIR is a language that is very close to the target machine language. Each CGIR operation can be mapped directly to a machine instruction. The operations in CG are partitioned to basic blocks (*BB* for short) and basic blocks are connected by direct arcs that denote correct control flow relationships.

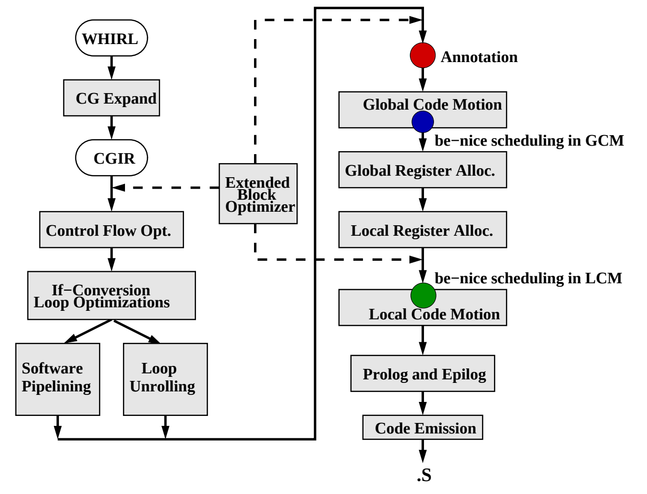

Figure 5. Annotation and Scheduling in Open64 Code Generator

```
input  : PU without being annotated
input  : An array of thread interference records
output: PU with OPs that are annotated
1.1 foreach Record in the thread interference record array do
1.2     BBnum ← Record.BBnum ;
1.3     OPnum ← Record.OPnum ;
1.4     slack ← CompSlack(Record.Penalty);
1.5     OP ← FindOP(BBnum, OPnum);
1.6     Mark OP as annotated;
1.7     OP.slack ← slack ;
1.8 end
```

Algorithm 1: Annotate Thread Interference Information

There are two passes of instruction scheduling in Open64 CG, the global code motion (GCM) phase and the local instruction scheduling (LIS) phase. Be-Nice algorithm applies to both phases, and Algorithm 2 demonstrates how to do it in GCM. We annotate *ITS information* on the corresponding instructions just before global code motion. Since we only perform be-nice instruction scheduling for α-thread, we only do annotation on the code of α-thread, not β-thread. The annotation algorithm is shown in Algorithm 1. The algorithm has two inputs, the current PU under processing and the associated thread interference information, which is an array of ITS records as shown in Equation (4). The algorithm traverse each record in the array. It easily locates the OP based on the PU number (*PUnum*) and BB number (*BBnum*) given by the ITS record, and then it needs to compute the *slack value* using a certain heuristic approach. The *slack value* decides how late that the operation would be scheduled.

```
input  : unscheduled PU
output: scheduled PU
2.1  foreach unscheduled loop body: loop_body in the current PU do
2.2      initialize candidate list cand_list of the current loop_body ;
2.3      while there are unscheduled OPs on cand_list do
2.4          cand_op ← select the best OP from cand_list ;
2.5          if (cand_op is annotated) && (cand_op.slack > 0) then
2.6              cand_op.slack −−;
2.7              temp_op ←cand_op ;
2.8              cand_op ← the next best OP from cand_list ;
2.9              put temp_op back to cand_list ;
2.10         end
2.11         if cand_op ≠ NULL then
2.12             if cand_op is not annotated then
2.13                 CircularSchedule(cand_op, loop_body);
2.14             end
2.15             EqUpCodeMotion(cand_op, loop_body);
2.16             EqDownCodeMotion(cand_op, loop_body);
2.17         end
2.18         update cand_list ;
2.19     end
2.20 end
```

Algorithm 2: be-nice instruction scheduling

In the Global Code Motion phase, the *slack* value in each annotated operation is used as heuristic to direct the instruction scheduling. Algorithm 2 shows the procedure. The algorithm treats each loop body in the PU as a scheduling unit, from inner most to outer most. For each loop body, it first create the initial candidate list, which consists of instructions that are ready to be issued into the pipeline (which means their control and data dependence relationships are satisfied). The algorithm tries to issue each instruction in the candidate list one by one, following an priority order determined by a particular heuristic. If an annotated instruction was selected from the candidate list, it checks that if the *slack* value of the instruction is zero. If zero, the annotated instruction is scheduled as a normal instruction. Otherwise, decrease its slack value and put the annotated instruction back to the candidate list, and another instruction is selected from the candidate list for scheduling. The annotated instruction only goes through equivalent upward code motion or equivalent downward code motion, but not circular scheduling (21), which is a light-weight software pipeline algorithm other than modulo scheduling (22). The candidate list is updated after the selected instruction has been scheduled.

The slack value of each annotated operation prevent the ITS instructions from being scheduled too aggressively. Therefore, the algorithm creates a certain amount of slacks in α-thread to make β-thread less likely to be stalled by α-thread. In the next section, we will use some experimental data to show how be-nice instruction scheduling can improve the throughput of β-thread.

4. Experiment Results

In order to verify the effectiveness of be-nice instruction scheduling, we performed a series of experiments on a JIAN performance simulator using seven micro-benchmarks. Currently, all micro-benchmarks are hand-write code. Each benchmark consists of two threads, running on JIAN simulator as α-thread and β-thread, respectively.

The simulator used in our experiments is a cycle accurate performance simulator that models JIAN architecture & micro-architecture described in section 2. In addition to the processor core, the simulator also simulates two caches, the instruction cache (I-Cache) and the data cache (D-Cache). The 32KB I-Cache is partitioned into two banks (16KB for each) for each thread. To reduce external memory transaction, the 16KB D-Cache is a write-back cache. Since all data used by β-thread are streaming data, β-thread is not designed to use D-cache. Instead, it use a piece of on-chip memory to help it to hide the memory access latency. This on-chip memory is dedicated to β-thread. α-thread use the D-Cache, and its miss penalty is 20 cycles.

In the simulator, we do not allow α-thread issue two memory operations or two ALU operations at the same cycle, which means that α-thread can issue a single load, or a single ALU, or an ALU plus a load. The simulator models an in order machine, and at each cycle it first updates the status of pipeline, from WB stage to ID stage, and then it take the instructions from the buffer to feed into the ID stage of relative FUs if the resource is available. ITS can be detected when the simulator scans and updates the status of pipeline stages. Therefore, the simulator can record all the information necessary for profiling.

We used Open64 compiler to compile each benchmark twice. In the first pass, we run the binary to collect the thread interference information; in the second pass, we applied the be-nice instruction scheduling with the knowledge of thread interference information collected in the first pass and runt it again. The details about how to do profiling and scheduling are presented in section 3.

We compared the performance data of two versions of binary code: the one with be-nice instruction scheduling and the one without be-nice instruction scheduling. Figure 4 shows the absolute number of ITS cycles that were decreased after we applied be-nice instruction scheduling. All

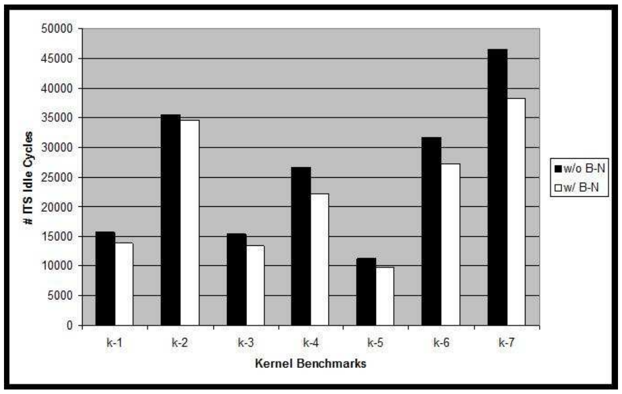

Figure 6. The Number of Inter-Thread Stall Cycles in β-thread: w/ Be-Nice Scheduling vs. w/o Be-Nice Scheduling

benchmarks except *k-2* got $> 10\%$ decrease in the number of ITS cycles. The biggest improvement was obtained on *k-7* whose ITS cycles reduced 17%. The smallest one is *k-2*, which only got 3% improvement. The average is 15%. This result indicates that our be-nice instruction scheduling method can effectively reduce ITS cycles in β-thread.

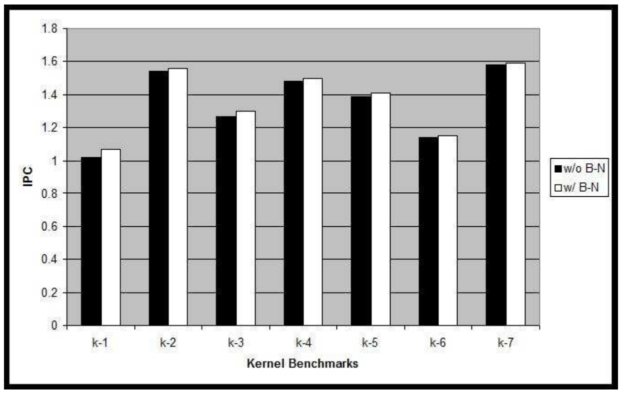

Figure 7. IPC Improvement with Be-Nice Instruction Scheduling

Figure 7 shows the instruction issue rate improvement of β-thread. The metric on y-axis is IPC: instruction per cycle. The bars with deep color represent the IPC of the code without be-nice instruction scheduling, and the bars with light color represent the IPC of the code with be-nice instruction scheduling. Since this is an in-order issue processor, none of the benchmarks have IPC bigger than 2. The improvements of IPC range from 0.63% to 4.90%. Average is 2.37%, not a significant improvement. We think the reason is that our benchmarks are still too small and can not run for a long time.

However, the average performance has a small decrease about **2%** on α-thread, because it has a lower issue rate in Be-Nice scheduling. Since this is an heterogeneous maltreating environment, we are OK with this trivial performance decrease.

5. Related Works

In order to increase the pipeline utilization and improve the overall throughput, many people have studied the thread sensitive scheduling problem on simultaneous multithreading processors (23) (24) (25) (26). However, they either adopt a hardware approach (23) (25), or try to explore the thread scheduling policies in operating system (24) (26), instead of instruction scheduling methods in compiler.

In (23), the SMT processor under consideration supports eight concurrent threads and can fetch up to eight instructions from one thread each cycle. The authors have studied many diverse instruction fetch schemes - either fetch four instructions from two different threads, or fetch two instructions from four different threads, etc. Meanwhile, threads are given priorities based on their characteristics of workload, the length of instruction queue, or the likelihood of branch mis-prediction and cache miss. (25) extended the work in (23) with a thread sensitive instruction scheduler that uses the ready-instruction-count (RIC) metric. The RIC metric is a dynamic and real-time metric that quantifies the urgency of a thread for CPU resources. Instruction issue is first performed at the intra-thread phase in the partitioned instruction queue (27), then at the inter-thread phase, in which the thread-sensitive scheduler perform thread-sensitive issue policy.

The above methods are hardware-based solutions. They try to implement the thread sensitive instruction scheduling in front-end of processor pipeline, i.e. fetch units and issue scheduler. These hardware costs are heavyweight which can not be afforded by a low-power embedded chip.

In (24), operating system uses the thread-behavior feedback information to choose the set of threads that can best utilize the CPU resources to try to maximize processor throughput. The thread-behaviors taken into consideration are cache (L1 or L2) miss rate, IPC, characteristic of workload (integer or floating point intensive), etc. Different thread-sensitive scheduling schemes are experimented and compared with those thread-oblivious scheduling schemes. (26) investigates the problem that how the contention for shared resources affect the overall system throughput. The author found that the contention for L2 cache has the greatest impact on system performance. Based on this finding, a balanced-set scheduling principle is adopted. It tries to schedule a group of threads whose combined working set has no problem to fit into L2 cache.

The above OS-based thread-sensitive scheduling and our compiler-based thread-sensitive instruction scheduling algorithm are perfectly complement to each other. The OS-based methods try to solve the resource contention problem at the thread level, while our method works at instruction level, which is more fine grain.

6. Conclusion

In the previous sections, we discussed the ITS problem in a heterogeneous multithreaded SMT embedded processor. We demonstrated that ITS caused many idle cycles in the thread that execute mission critical code. Instead of using the traditional hardware-based approach, we proposed a static compiler optimizing technique called *be-nice* instruction scheduling to solve this problem. The experimental results show that the *be-nice* instruction scheduling can effectively reduce the number of ITS happened at runtime. And therefore increase the throughput of the critical thread. The advantage of our approach is that it avoids adding more complicate hardware logics in the processor as the previous solutions. This makes a lot of sense for an embedded SMT processor that has very tight power budget.

7. Future Work

In this paper, our discussion of ITS is restricted to an in-order issue embedded SMT processor. We haven't yet studied this problem carefully on the general SMT processors that support out-of-order issue & execution (28). The reason is that, with more function units and the flexibility of speculative execution, the effect of ITS would be alleviated.

In the next step, we will study the impact of ITS to the CPU performance of general SMT processors that support out-of-order execution. At the same time, we will extend our be-nice instruction scheduling to do cross-thread instruction scheduling and solve the problem in a more general framework. We will also study the method that can profile a multithread program running on a SMT processor to identify the places in the code that would trigger most of the harmful ITS. In addition, we also want to extend our study to homogeneous multithread execution.

Acknowledgments

The author would like to acknowledge SimpLight Nanoelectronics Co. Ltd for providing us the simulator and toolchain on which we can conduct our research on this problem. First and foremost, we thank Sun Chan for his wisdom that direct us in the research. We also thank Xiaomi An and Lin Ma for giving us the benchmarks.

References

[1] Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: Maximizing on-chip parallelism. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, ACM SIGARCH and IEEE Computer Society (1995) 392–403 *Computer Architecture News,* 23(2), May 1995.

[2] Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L., Tullsen, D.M.: Simultaneous multithreading: A platform for next-generation processors. IEEE Micro **17**(5) (1997) 12–19

[3] Saulsbury, A., Pong, F., Nowatzyk, A.: Missing the memory wall: The case for processor/memory integration. In:

Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, ACM SIGARCH and IEEE Computer Society (1996) 90–101 *Computer Architecture News,* 24(2), May 1996.

[4] Wall, D.W.: Limits of instruction-level parallelism. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society (1991) 176–188 *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.

[5] Laudon, J., Gupta, A., Horowitz, M.: Interleaving: A multithreading technique targeting multiprocessors and workstations. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society (1994) 308–318 *Computer Architecture News,* 22, October 1994; *Operating Systems Review,* 28(5), December 1994; *SIGPLAN Notices,* 29(11), November 1994.

[6] Ponomarev, D.V., Kucuk, G., Ergin, O., Ghose, K., Kogge, P.M.: Energy-efficient issue queue design. IEEE Trans. Very Large Scale Integr. Syst. **11**(5) (2003) 789–800

[7] Gibbons, P.B., Muchnick, S.S.: Efficient instruction scheduling for a pipelined architecture. In: Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, California, ACM SIGPLAN (1986) 11–16 *SIGPLAN Notices,* 21(7), July 1986.

[8] Fisher, J.A.: Trace scheduling: A technique for global microcode compaction. IEEE Transactions on Computers **30**(7) (1981) 478–490

[9] Fisher, J.: Global code generation for instruction-level parallelism: Trace scheduling-2. Technical Report HPL-93-43, Hewlett-Packard Laboratories (1993)

[10] Hwu, W.M.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., Lavery, D.M.: The superblock: An effective technique for vliw and superscalar compilation. The Journal of Supercomputing **7**(1-2) (1993) 229–248

[11] Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, ACM SIGMICRO and IEEE-CS TC-MICRO (1992) 45–54 *SIG MICRO Newsletter* 23(1–2), December 1992.

[12] Bala, V., Rubin, N.: Efficient instruction scheduling using finite state automata. In: Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, Michigan, IEEE-CS TC-MICRO and ACM SIGMICRO (1995) 46–56

[13] Beaty, S.J.: Genetic algorithms and instruction scheduling. In: Proceedings of the 24th Annual International Symposium on Microarchitecture, Albuquerque, New Mexico, ACM SIGMICRO and IEEE-CS TC-MICRO (1991) 206–211

[14] Bernstein, D., Cohen, D., Krawczyk, H.: Code duplication: An assist for global instruction scheduling. In: Proceedings of the 24th Annual International Symposium on Microarchitecture, Albuquerque, New Mexico, ACM SIGMICRO and IEEE-CS TC-MICRO (1991) 103–113

[15] Bernstein, D., Rodeh, M.: Global instruction scheduling for superscalar machines. In: Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario (1991) 241–255 *SIGPLAN Notices*, 26(6), June 1991.

[16] Bradlee, D.G., Eggers, S.J., Henry, R.R.: Integrating register allocation and instruction scheduling for RISCs. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society (1991) 122–131 *Computer Architecture News*, 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices*, 26(4), April 1991.

[17] Motorola Corporation: CPU12 Reference Manual (2000) http://e-www.motorola.com/ brdata/ PDFDB/MICROCONTROLLERS/16BIT/68HC12FAMILY/ REFMAT/CPU12RM.pdf.

[18] Motorola Corporation: M-CORE - MMC2001 Reference Manual (1998) http://www.motorola. com/ SPS/MCORE/info documentation.htm.

[19] Texas Instruments: Tms370cx7x 8-bit microcontroller (1997) http://wwws.ti.com/ sc/psheets/ spns034c/spns034c.pdf.

[20] SGI: Open64 compiler and tools. URL http://www.open64.net (2008)

[21] Jain, S.: Circular scheduling: A new technique to perform software pipelining. In: Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario (1991) 219–228 *SIGPLAN Notices*, 26(6), June 1991.

[22] Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, Georgia (1988) 318–328 *SIGPLAN Notices*, 23(7), July 1988.

[23] Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L.: Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In: ISCA. (1996) 191–202

[24] Parekh, S., Eggers, S., Levy, H., Lo, J.: Thread-sensitive scheduling for smt processors. Technical Report 2000-04-02, University of Washington (2000)

[25] Robatmili, B., Yazdani, N., Sardashti, S., Nourani, M.: Thread-sensitive instruction issue for smt processors. IEEE Comput. Archit. Lett. **3**(1) (2004) 5

[26] Fedorova, A., Seltzer, M., Small, C., Nussbaum, D.: Throughput-oriented scheduling on chip multithreading systems. Technical Report TR-17-04, Computer Science Group, Harvard University, Cambridge, Massachusetts (2005)

[27] Goncalves, R., Ayguade, E., Valero, M., Navaux, P.: A simulator for smt architectures: Evaluating instruction cache topolo-

gies. In: 12th Symposium on Computer Architecture and High Performance Computing. (2000) 279–286

[28] Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. 2nd edn. Morgan Kaufmann Publishers, Inc., San Francisco (1996)