Master Thesis

A study of an in-order processor exploiting memory level parallelism with speculative execution
(投機実行によりメモリレベル並列性を抽出するインオーダプロセッサに関する研究)

Supervisor: Associate Prof. Kenji Kise

January, 2013

Submitter

Department of Computer Science
Graduate School of Information Science and Engineering

10M38384  Takayuki Matsumura
Abstract

In recent years, Chip Multi Processors (CMPs) have been widely used. CMPs execute multiple threads by employing multiple cores to extract thread level parallelism. The more the number of cores on chip, the more extracting high level of thread level parallelism.

Core Architectures for CMPs has two goals. One is maximizing the number of cores for using high level of thread level parallelism. The other is providing good per-thread performance. If we aim the former, cores that have small area and consume low power are needed. If we aim the latter, out-of-order cores exploiting instruction level parallelism are used. However, they need more complex structures than those of in-order cores. Therefore, out-of-order cores have disadvantages that increasing core area and power. If in-order cores are chosen for decreasing area and power, they can provide less performance than that of out-of-order cores. Therefore, it is hard to achieve these two goals.

Simultaneous Speculative Threading (SST), which employing in-order cores has been proposed for achieving both goals at the same time. SST exploits instruction parallelism and memory level parallelism by executing a single program thread simultaneously at two different points. An ahead thread speculatively executes under a cache miss and a behind thread executes instructions dependent on the cache miss. SST does not need expensive structures and has high performance. However, an ahead thread often continues to executes wrong path by mispredicted branches. Therefore, SST can provide more performance if branch mispredictions can be detected earlier.

This thesis presents the new threading method resolving the problems of SST described above and combining ALU cascading in order that the proposed method is more efficiently. In this method, two threads that are executed simultaneously in SST are executed exclusively. This enables the core can discover branch mispredictions earlier and decrease the number of instructions that executed speculatively.

ALU cascading is combined to tolerate instruction dependencies that exist in behind thread. This enables to use execution units more effectively.

An evaluation by software simulator shows that this method improves performance of in-order processors 21.9% on average and 45.5% at a maximum.
# Contents

Abstract

1. Introduction
   1.1 Background  
   1.2 Outline of this thesis

2. Instruction level parallelism and Memory level parallelism
   2.1 Instruction level parallelism
      2.1.1 Definition
      2.1.2 Parallel execution constraints
      2.1.3 Out-of-order execution
   2.2 Memory level parallelism
   2.3 Conclusion of this section

3. Related works
   3.1 Simultaneous Speculative Threading
      3.1.1 Architecture
      3.1.2 Example of SST
      3.1.3 Removing Register Hazard
      3.1.4 Problems on SST
   3.2 ALU cascading
      3.2.1 Concept

4. Proposed Method
   4.1 The new threading method
      4.1.1 Outline
      4.1.2 Example
   4.2 Combining ALU cascading

5. Evaluation
   5.1 Evaluation Environment
   5.2 Evaluation Results and discussion
      5.2.1 Execution Analysis
      5.2.2 Cascaded ratio
      5.2.3 IPC
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conclusion</td>
<td>26</td>
</tr>
<tr>
<td>Bibliography</td>
<td>28</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>In-order execution</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Out-of-order execution</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Example that extracting MLP is more effective than ILP[1]</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Pipeline augmented with SST logic [2]</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>An example of SST</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>SST mode breakdown[2]</td>
<td>12</td>
</tr>
<tr>
<td>3.4</td>
<td>ALU cascading</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of SST</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>An example of proposed threading method</td>
<td>16</td>
</tr>
<tr>
<td>4.3</td>
<td>An example of instruction sequence which can be executed efficiently by cascaded ALU</td>
<td>17</td>
</tr>
<tr>
<td>4.4</td>
<td>Two examples of applications. One is not memory intensive, and the other is.</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Execution Mode Breakdown</td>
<td>20</td>
</tr>
<tr>
<td>5.2</td>
<td>Cascaded ratio</td>
<td>21</td>
</tr>
<tr>
<td>5.3</td>
<td>Traditional pipeline</td>
<td>22</td>
</tr>
<tr>
<td>5.4</td>
<td>Pipeline that allows load instructions to access data cache on ID stage with ALU cascading</td>
<td>22</td>
</tr>
<tr>
<td>5.5</td>
<td>Cascaded rate(added ideal execution of load inst in behind threads)</td>
<td>23</td>
</tr>
<tr>
<td>5.6</td>
<td>IPC</td>
<td>24</td>
</tr>
<tr>
<td>5.7</td>
<td>Performance improvement by executing load instructions in behind threads ideally</td>
<td>25</td>
</tr>
</tbody>
</table>
List of Tables

5.1 The parameters for evaluation .............................................. 19
Chapter 1

Introduction

1.1 Background

In recent years, Chip Multi Processors (CMPs) have been widely used. CMPs execute multiple threads by employing multiple cores to extract thread level parallelism. The more the number of cores on chip, the more extracting high level of thread level parallelism.

Core Architectures for CMPs has two goals. One is maximizing the number of cores for using high level of thread level parallelism. The other is providing good per-thread performance. If we aim the former, cores that have small area and consume low power are needed. If we aim the latter, out-of-order cores exploiting instruction level parallelism are used. However, they need more complex structures than those of in-order cores. Therefore, out-of-order cores have disadvantages that increasing core area and power. If in-order cores are chosen for decreasing area and power, they can provide less performance than that of out-of-order cores. Therefore, it is hard to achieve these two goals.

Simultaneous Speculative Threading (SST)[2], which employing in-order cores has been proposed for achieving both goals at the same time. SST exploits instruction parallelism and memory level parallelism by executing a single program thread simultaneously at two different points. An ahead thread speculatively executes under a cache miss and a behind thread executes instructions dependent on the cache miss. SST does not need expensive structures and has high performance. However, an ahead thread often continues to executes wrong path by mispredicted branches. Therefore, SST can provide more performance if branch mispredictions can be detected earlier.

This thesis presents the new threading method resolving the problems of SST described above and combining ALU cascading in order that the proposed method is more efficiently. In this method, two threads that are executed simultaneously in SST are executed exclusively. This enables the core can discover branch mispredictions earlier and decrease the number of instructions that executed speculatively.

ALU cascading is combined to tolerate instruction dependencies that exist in behind thread. This enables to use execution units more effectively.
1.2 Outline of this thesis

This remaining thesis is organized as follows. The basic concepts in superscalar processor, instruction level parallelism and memory level parallelism are described in section 2. Simultaneous Speculative Threading and ALU cascading are described as related works in section 3. Proposed method is given in section 4. In section 5, it is evaluated by software simulator. Finally, Section 7 concludes the thesis.
Chapter 2

Instruction level parallelism and Memory level parallelism

In this section, the basic concepts of this study, instruction level parallelism (ILP) and memory level parallelism (MLP) are described.

2.1 Instruction level parallelism

2.1.1 Definition

Consider the following program:

\[
\begin{align*}
\text{prog0 :} & \quad \text{prog1 :} \\
i_0 : r1 &= r1 + 1 & i_0 : r1 &= r1 + 1 \\
i_1 : r2 &= r2 + 1 & i_1 : r2 &= r1 + 1 \\
i_2 : r3 &= r3 + 1 & i_2 : r3 &= r2 + 1 \\
i_3 : r4 &= r4 + 1 & i_3 : r4 &= r3 + 2
\end{align*}
\]

We call two instructions are parallel when one instruction and the other instruction are independent of each other and instruction level parallelism a measure of how many of the instructions are parallel. The instructions in prog0 are all independent, while the ones in prog1 are depends on last one. In this case, we call prog0 have higher level of ILP than prog1. If all instructions requires one clock cycle to execute and there are abundant execution units, prog0 requires cycle to be completed by contrast prog1 requires 4 cycles.

2.1.2 Parallel execution constraints

Parallel execution constraints are as follows.

- Resource dependence
- Data dependence
- Control dependence

These are explained below.
Resource dependence

Resource dependence is that one instruction requires the same hardware resource which is being used by a previous instruction.

Consider the following program:

\[
\begin{align*}
i_0 &: r_1 &= r_2 \times r_3 \\
i_1 &: r_4 &= r_5 \times r_6
\end{align*}
\]

Although i0 is independent on i1, they cannot be executed simultaneously unless there exits two multiplier.

Not only execution units such as multiplier but also register file and data cache are used by multiple instructions during one clock cycle. If \( N \) instructions are executed at the same time, \( N \) times resources are required. However, we need to take into consideration the frequency of each instruction and the cost of resources. Consider ALU instructions and load instructions. Employing multiple ALUs is cost-effective since the former come out more often than the latter. Although multiport cache is expensive, load instructions are less than ALU instructions. We must consider whether it is worth employing multiport cache or not.

Data dependence

Data dependencies are classified as follows.

- True data dependence
- Anti dependence
- Output dependence

True data dependence is that one instruction produces a result that is used by the other instruction. Consider the following program:

\[
\begin{align*}
i_0 &: r_1 &= r_2 + r_3 \\
i_1 &: r_4 &= r_1 + 1
\end{align*}
\]

i0 produces a result that used by i1. In other words i0 is data dependent on i1. If i0 is executed before i1, i1 cannot execute correctly because of wrong value compared to sequential execution. We call this RAW (Read after Write) hazard. This dependency cannot be removed because it originates from program mean.

Anti dependence is that one instruction writes register or memory location that the other instruction reads. Consider the following program:

\[
\begin{align*}
i_0 &: r_2 &= r_1 + 1 \\
i_1 &: r_1 &= r_3 + 1
\end{align*}
\]

The register r1, which is read by i0 is written by subsequent instruction i1. If i0 is executed before i1, i1 can not execute correctly because of wrong value which has been overwritten by i1 compared to sequential execution. We call this WAR (Write after Read) hazard.

Output dependence is that two instructions write the same register or memory location. Consider the following program:
2.1 Instruction level parallelism

\[
i_0 : r_2 = r_1 + 1 \\
i_1 : r_2 = r_3 + 1
\]

The register $r_1$, which is written by $i_0$ is also written by subsequent instruction $i_1$. If $i_0$ is executed before $i_1$, subsequent of $i_1$ cannot execute correctly because of wrong value which has been overwritten by $i_0$ compared to sequential execution. We call this **WAW (Write after Write) hazard**.

Anti dependence and Output dependence originate from reusing register not from program mean. These dependencies are called **False dependencies** False dependencies can be removed by allocating different register to each instruction (this called **Register renaming**).

**Controll dependence**

**Controll dependence** imposes execution of subsequent instructions of branch. Consider the following program:

\[
i_0 : \text{if}(r_1 == r_2) \text{ goto L1} \\
i_1 : r_2 = r_3 + 1 \\
L1: \\
i_2 : r_4 = r_5 - 1
\]

It depends on branch $i_0$ to execute $i_1$ or $i_2$. Therefore neither $i_1$ nor $i_2$ can execute until $i_0$ execution.

Controll dependence result in disturbing exploiting ILP in general, many kind of processors hold branch predictor to continue executing instructions speculatively unless it was misprediction.
2.1.3 Out-of-order execution

In-order execution is executing instructions in program order. Out-of-order execution is executing instructions in an order governed by the availability of operand rather than program order. The goal of out-of-order execution is exploiting instruction level parallelism aggressively.

Consider the following program executed by a processor which has two execution units:

\[
\begin{align*}
&i_0 : r_2 = r_1 + 1 \\
&i_1 : r_3 = r_2 + 2 \\
&i_2 : r_5 = r_4 + 1 \\
&i_3 : r_6 = r_5 + 2
\end{align*}
\]

$i_1$ is dependent on $i_0$ and $i_3$ dependent on $i_2$. If this program executed in-order, it is as Figure 2.1. It takes 4 cycles to complete all instructions.

$i_2$ is independent on $i_0$ and $i_1$. Thus, $i_2$ can be executed ahead of them. If this program executed out-of-order, it is as Figure 2.2. In case of in-order execution, only one execution unit is used in the first clock cycle. In this case, both two execution units can be in use and it takes 3 cycles to complete all instructions. One clock cycle can be saved by out-of-order execution.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_0$: $r_2 = r_1 + 1$</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i_1$: $r_3 = r_2 + 2$</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i_2$: $r_5 = r_4 + 1$</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i_3$: $r_6 = r_5 + 2$</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2.1 In-order execution

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_0$: $r_2 = r_1 + 1$</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>$i_1$: $r_3 = r_2 + 2$</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>$i_2$: $r_5 = r_4 + 1$</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>$i_3$: $r_6 = r_5 + 2$</td>
<td>EX</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2.2 Out-of-order execution
2.2 Memory level parallelism

Memory level parallelism (MLP) is ability to handling multiple cache misses. In other words, it is ILP that exists among instructions that cause cache misses. The latencies of instructions which caused cache miss is hundreds of cycles while those of arithmetic or logic instructions. Therefore exploiting MLP is more effective than exploiting instruction level parallelism.

Figure 2.3 shows an example of it. One horizontal line means one load instruction and the length of it represents the latency of the instruction. In case of execution of instructions that cause cache miss, it takes fewer clock cycles to complete all instructions if more cache miss can be handled rather than more instructions are executed simultaneously.

It is suggested that hardware should be spent for exploiting memory level parallelism rather than exploiting instruction level parallelism.
2.3 Conclusion of this section

In this section, ILP and MLP, basic concept of this study are described.

Many processors have achieved high performance exploiting ILP with wide issue width and large instruction windows.

However the area of RAM (Random access memory) increases with the square of the number of ports. Modules that out-of-order core needs are more complex than those in-order core needs. Hence out-of-order core has worse power- and area-efficiency. There are several researches about processors that do not need complex structure and have performance as good as those of out-of-order processors.

In the next section, Simultaneous Speculative Threading (SST) which exploits ILP and MLP, and ALU cascading which tolerate instruction dependencies are described from those researches.
Simultaneous Speculative Threading (SST) and ALU cascading are described in this section. For the details of SST such as implements, see [2].

### 3.1 Simultaneous Speculative Threading

Chaudhry et al. have proposed Simultaneous Speculative Threading (SST) [2]. SST achieves high performance by executing two threads simultaneously to extracting ILP and MLP: an ahead thread which continues to execute new instructions speculatively, a behind thread which executes instructions dependent on the cache miss.

#### 3.1.1 Architecture

In order to support SST, each core is extended with the hardware structures shaded in Figure 3.1. In an SST implementation with two checkpoints, there are two deferred queues (DQs) which hold decoded instructions and available operand values for instructions that could not be executed due to a cache miss. In addition to the architectural register file (the rightmost register files), two speculative register files for checkpoints (the shaded boxes behind the architectural register file) and register files for a behind thread (the shaded box behind working register files) are provided. Each of these registers has an NA bit which is set if the register’s value is “Not Available”.

#### 3.1.2 Example of SST

Figure 3.2 shows an instruction sequence executed by SST. One square represents one instruction. The white-colored squares are instructions executed in a non-speculative phase. The red-colored squares are load instructions which cause cache miss. The yellow-colored squares are instructions which is dependent on a cache miss. The green-colored squares are instructions which is independent on cache miss. The blue-colored squares are deferred instructions.

The core starts execution in a non-speculative phase. In this phase, all instructions write their results to a working register file and the architectural register file. The DQs and the speculative register files are not used (during executing i1 and i2 in Figure 3.2). When the
first instruction which has caused a cachemiss (i3 in Figure 3.2) is encountered, the core takes checkpoint of the architectural state (it is called “committed checkpoint”) The instruction marks its destination register as NA and it is placed in the first DQ, and the core starts a speculative phase. In this phase, all instructions are classified as “deferrable” or “retirable”. An instruction is deferrable if and only if it is a long-latency instruction (such as a load which misses in the L1 data cache) or if it can not be executed because at least one of its operands is NA. Deferrable instructions are placed in the first DQ and its destination register is marked as NA (i5 and i6 etc in Figure 3.2), while retirable instructions are executed and speculatively retired. The retirable instructions write their results to a working register file and a speculative register file and clear the NA bits for the destination registers because it has been overwritten by available data.

The core continues to execute instructions in this way until one of the deferred instructions can be retired (before i13 in Figure 3.2). At this point, one thread, called the “ahead thread”, continues to fetch and execute new instructions(i3 ~ i20 except for blue-colored instructions in Figure 3.2) while the other thread, called the “behind thread”, starts executing the instructions from the first DQ (from i3 to i16 instructions which is blue-colored in Figure 3.2).

The ahead thread can choose to take a speculative checkpoint and start using the next speculative register file and DQ at any time if hardware resources can be used. For example, the ahead thread could detect that DQ1 is nearly full or encountered branch and therefore choose to take a speculative checkpoint i + 1 and DQi+1(e.g., i9 is a branch).

At any time, the behind thread attempts to execute from the the oldest DQ. We assume that it is DQi. The behind thread waits until at least one of the instructions in DQi can be retired, at which point the behind thread starts to execute all of the instructions from DQi. Once all of the instructions in DQi have been speculatively retired, the committed checkpoint is discarded, speculative register filei becomes the new committed checkpoint, and speculative register filei is freed (In Figure 3.2, i = 1 and after blue-colored i8 retired, speculative register file1 becomes the new committed checkpoint). This operation is called as “commit”. Next, the behind thread attempts to execute instructions from DQi+1(blue-colored i9 is executed from DQ2). At this point, if the ahead thread is deferring instructions to DQi+1, it takes new
3.1 Simultaneous Speculative Threading

If there is no deferred instruction, a “thread sync” operation is performed (after blue-colored i16 retired in Figure 3.2). The thread sync operation ends the speculative phase. For each register, if the ahead thread’s value for the register was NA, it is overwritten by the behind thread’s value. The register’s NA bit is cleared, and starts a non-speculative phase.

It is possible that the ahead thread will run out of hardware resources (overflow of store queue or all DQs) when the core executes an ahead thread using speculative register file $i$ and DQ$i$. At this point, speculative checkpoint $i$ is marked as not commitable, the ahead thread stops placing instructions in DQ$i$, and it continues in order to prefetch data and instructions (this works as a hardware scout thread).
3.1.3 Removing Register Hazard

In speculative phase, two threads are executed simultaneously, and the core needs to remove register hazards.

**Read-After-Write hazard**
If an instruction which has any “Not Available” operand, it cannot be retirable. It can be retirable when all of its operands are not NA.

**Write-After-Read hazard**
Working register file for the ahead thread will be updated by subsequent instructions after one instruction was deferred. However, available operands of the instruction are stored with it in a DQ, and they cannot be overwritten by other instructions.

**Write-After-Write hazard**
If a register in architectural register file is NA, its correct value is generated by behind thread. The value of the register of behind thread is passed to ahead thread at thread sync. If a register in architectural register file is not NA, the value of the register of behind thread is discarded because architectural register has the latest value of the register.

3.1.4 Problems on SST

It is reported regarding speculation failure in [2].

If ahead thread run out of hardware resources, the ahead thread acts as a hardware scout thread in which the core can prefetch data and instructions as previously described. In contrast, when a deferred branch is determined to have been mispredicted, speculation fails immediately and the core starts a non-speculative phase from the committed checkpoint. In this case, hardware scout thread is not executed while speculation fails. Therefore, many clock cycles has been wasted by the deferred mispredicted branch. In figure 3.2, if i9 was misspredicted
branch, execution from i10 to i17 was useless.

To make matters worse, i14 which should not access data cache actually has done it. This results in cache pollution. In this case, speculative execution do harm rather than benefit. [2] said that if mispredictions of deferrable branches can be detected when they are first executed, it improves performance by nearly 20% on commercial benchmarks.

It is also said that applications spent almost all of the time in a speculative phase as shown in Figure [3.3]. It is highly likely that speculative execution do harm as mentioned above.

It is inferred from these results that executing a behind thread and an ahead thread simultaneously is a high-risk high-return method.
3.2 ALU cascading

3.2.1 Concept

ALU cascading is an execution units configuration that the first ALU can send its result to the second ALU if needed. The cascaded ALU executes multiple dependent operations in one cycle.

An example of ALU cascading is in Figure 3.4. Inst2 depends on Inst1. The cascaded ALU can execute them in one cycle although traditional superscalar processors cannot execute these instructions at the same time.

![Fig. 3.4 ALU cascading](image)

There are several research about ALU cascading [4], [5], [6].

Sasaki et al have proposed Adaptive Instruction Cascading (AIC) on Globally-Asynchronous Locally-Synchronous (GALS) processors [4]. It is said in the paper that the results showed average performance improvement when assuming that AIC is possible by lowering the clock frequency to 80%.

Ogata et al. have proposed instruction scheduler for ALU cascading in out-of-order processor [5]. They extends scheduler called DMT (Dependence Matrix Table) [7], which represents dependencies among instructions. This method can wake up producer instructions and consumer instructions at the same time.

Ichino et al. have proposed combining ALU chaining (the same method as ALU cascading) and value prediction [6]. ALU chaining tolerates dependencies that are among ALU instructions and value prediction tolerates dependencies that are among load instruction.
Chapter 4

Proposed Method

The problems of SST and ALU cascading are described in previous section. In this section, the new method to solve the problem will be given.

4.1 The new threading method

4.1.1 Outline

As mentioned in previous section, there is a gap between the actual performance and ideal performance of SST and almost all of execution time is spent in a speculative phase.

To solve these problems, we propose the new threading method that stops executing an ahead thread when starting a behind thread.

Deferred mispredicted branches are detected earlier owing to all hardware resources are used for executing behind thread.

In addition to this effect, deferred instructions are not generated during a behind thread thanks to less time spent in speculative phases. These result in less impact of speculation failure.

4.1.2 Example

Figure 4.1 shows an example of SST, and Figure 4.2 shows an example of the proposed threading method.

In this method, all of hardware resources of the processor are used for executing a behind thread. The core can detect branch mispredictions earlier and is less affected by speculation failure.

By not executing an ahead thread during executing an behind thread, the core does not generate new deferrable instructions. SST needs to execute i14 and i16 two times as shown in Figure 4.1 because the core is in a speculative phase when it executes them first. The proposed threading method shortens the time spent in a speculative phases as shown in Figure 4.2 and executes them once.

We assume that i9 is a mispredicted branch and i14 is load instruction that causes cache miss as described in previous chapter. The core can detect the misprediction before executing
i14, so the core does not pollute data cache. It cannot remove all of pollutions, but can reduce significantly.

Fig. 4.1 An example of SST

Fig. 4.2 An example of proposed threading method
4.2 Combining ALU cascading

SST extracts ILP, and can use execution units due to executing an ahead thread and a behind thread simultaneously. However, it is possible that the new speculative threading method is hard to extracts ILP because of executing one thread and the aspect of a behind thread.

An behind thread is composed of

1. instructions which has caused data cache miss.
2. instructions which depend on 1.

Thus, it contains several dependencies.

We combine ALU cascading with the proposed threading method for tolerate instruction dependencies to execute. If an instruction which depends on the latest one as shown in Figure 4.3, ALU cascading make the core to execute efficiently.

<table>
<thead>
<tr>
<th>Inst</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst1</td>
<td>add</td>
<td>$3$</td>
<td>$1$</td>
</tr>
<tr>
<td>Inst2</td>
<td>add</td>
<td>$5$</td>
<td>$4$</td>
</tr>
<tr>
<td>Inst3</td>
<td>add</td>
<td>$7$</td>
<td>$6$</td>
</tr>
<tr>
<td>Inst4</td>
<td>add</td>
<td>$9$</td>
<td>$8$</td>
</tr>
</tbody>
</table>

Fig. 4.3 An example of instruction sequence which can be executed efficiently by cascaded ALU

ALU cascading works well for not only executing an behind thread. Figure 4.4 shows the two examples of applications. If many instructions cause cache misses, the speculative threading works well, by contrast, ALU cascading does not because of pipeline stall. If few instructions cause cache misses, ALU cascading works well, by contrast, the speculative threading does not. These two methods, speculative threading and ALU cascading, work exclusively. In case that one is not effective, the other is.
Fig. 4.4 Two examples of applications. One is not memory intensive, and the other is.
Chapter 5

Evaluation

The proposed method is evaluated by software simulator in this section.

5.1 Evaluation Environment

A cycle-level simulator is used. This is based on SimMips\cite{SimMips}. We implemented four architectures, base line in-order superscalar without ALU cascading (cALU), in-order superscalar with cALU, proposed speculative threading without cALU, proposed speculative threading with cALU.

We used SPEC2006 benchmarks(401.bzip2, 429.mcf, 445.gobmk, 456.hmmer 436.cactu-sADM, 454.caculix, 459.GemsFDTD, 470.lbm) to evaluate and evaluate 100 M instructions after skipping 1 G instructions.

The core can cascade ALUs between integer instructions. The parameters used for evaluation are shown in Table \ref{tab:evaluation_parameters}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Pipeline & IF, ID, EX, MA, WB \\
Pipeline way & 2 \\
L1 I-cache & 32KB 4-way, 1 cycle, 64B line \\
L1 D-cache & 32KB 4-way, 1 cycle, 64B line \\
L2 cache & 4MB 8-way, 25cycle, 64B line \\
Main Memory & 300 cycle latency \\
Branch predictor & Bi-mode, 6bit GHR, 1k entries PHT \\
Branch target buffer & 1k entries, 2-way \\
Store queue & 64 entries \\
Checkpoints & 2 \\
\hline
\end{tabular}
\caption{The parameters for evaluation}
\end{table}
5.2 Evaluation Results and discussion

5.2.1 Execution Analysis

The core implemented proposed method can be in 4 phases as follows.

**Non-Speculative** There is no register that has unknown value. All instructions retire after writing back the results.

**Speculative** A phase in which the ahead thread has not detected a failure condition until the behind thread cannot start execution.

**Re-Execute** A phase in which the behind thread is executing from DQs.

**Hardware Scout (HWS)** A speculative phase in which the ahead thread acts a hardware scout thread.

Figure 5.1 shows the number of cycles spent in each of these phases for each benchmarks. The rate of each phase is different for every benchmark. Mcf spends about 40% of its time in a non speculative phase. It is affected by speculative threading rather than ALU cascading. Gobmk, hmmer and GemsFDTD spend almost all of their time in a non-speculative phase. They are affected by ALU cascading rather than speculative threading.
5.2 Evaluation Results and discussion

5.2.2 Cascaded ratio

The assumption mentioned in previous section that it may hard to extract ILP when only behind thread is executed will be verified. Cascaded ratio is defined as the ratio of the times of ALU cascading to the number of executed instructions.

Figure 5.2 shows the cascaded ratio in each application.

The total ratio in the baseline is 9.35%. That in behind threads is 9.5%.

The ratio in behind threads is lower than that in the baseline except for gobmk. If the assumption was valid, it would be higher than that in the baseline.

The following might cause the low ratio.

1. ALU cascading is impossible because a load instruction and successive instruction is dependent on it.
2. Forwarding between pipeline stages can is sufficient to provide data to instructions and the core does not need to cascade ALUs.
3. There are many instructions which has caused cache miss and few instructions that depends on them.

In case of 2 and 3, the core does not need to cascade ALUs and can use execution units effectively. In case of 1, however, the low ratio means that the core stalls the pipeline and can not use execution units sufficiently.

We evaluated the core in which load instructions get its result on MA stage. In this situation,
it can not have positive effects of ALU cascading when it encounters a producer instruction and a consumer instruction consecutively. Figure 5.3 shows an example of this.

Consider now the core has the mechanism such that [9], which enables load instructions to get data on ID stage.

If load instructions get data from data cache on ID stage, the core can save 2 clock cycles due to combining ALU cascading. Figure 5.4 shows an example of this.

<table>
<thead>
<tr>
<th>cycle</th>
<th>load R2, 0(R1)</th>
<th>add R4, R2, R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF ID</td>
<td>IF ID</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IF EX</td>
<td>EX</td>
</tr>
<tr>
<td>3</td>
<td>MA</td>
<td>EX</td>
</tr>
<tr>
<td>4</td>
<td>WB</td>
<td>MA</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5.3  Traditional pipeline

<table>
<thead>
<tr>
<th>cycle</th>
<th>load R2, 0(R1)</th>
<th>add R4, R2, R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IF ID</td>
<td>IF ID</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IF ID</td>
<td>IF ID</td>
</tr>
<tr>
<td>3</td>
<td>EX</td>
<td>EX</td>
</tr>
<tr>
<td>4</td>
<td>MA</td>
<td>MA</td>
</tr>
<tr>
<td>5</td>
<td>WB</td>
<td>WB</td>
</tr>
</tbody>
</table>

Fig. 5.4  Pipeline that allows load instructions to access data cache on ID stage with ALU cascading

Figure 5.5 shows the cascaded ratio under a situation where all of load instructions of behind threads get data on ID stage as "Behind thread (ideal)".

In this case, the total ratio in behind threads is 10.2%. Huge rise of cascaded rate in gobmk and hmmer means that cascaded rate was low with the first situation.

It is future work on the detail mechanism that enables load instructions to access data cache on ID stage.
5.2 Evaluation Results and discussion

Fig. 5.5 Cascaded rate (added ideal execution of load inst in behind threads)
5.2.3 IPC

Proposed method

Figure 5.6 shows IPC of each architecture. Proposed method achieves 21.9% higher performance on average and 45.5% at a maximum in cactusADM.

Speculative threading is not effective in some applications such as gobmk, hmmer and GemsFDTD. The core spent almost all the time in a nonspeculative phase in these. That means ALU cascading is effective, and Figure 5.6 shows that.

In contrast, ALU cascading is not effective in some application such as mcf and cactusADM, but Speculative threading improves performance.
Fig. 5.7  Performance improvement by executing load instructions in behind threads ideally

The effect that load instructions can get data earlier
As mentioned above, cascaded ratio improves if load instructions in behind thread get data on ID stage. Figure 5.7 shows performance improvement under this situation.

This condition improved 0.44% higher performance on average and 2.4% at a maximum in bzip2.

Although cascaded rate rise greatly in gobmk and hmmer, it improves performance slightly in these application. This is because these spent little time in a re-execute phase. Bzip2 spent a large percentage of all time in re-execute phase relatively compared to others and cascaded rate rise. That is why IPC increases significantly.
Chapter 6

Conclusion

We proposed the new speculative threading method, which exploiting MLP and ALU cascading, which tolerate dependencies among instructions based on area- and power efficient in-order processor.

An evaluation by software simulator shows that this method improves performance of in-order processors 21.9% on average and 45.5% at a maximum.

Future works are as follows

- the detail structure that enables load instructions access data cache on ID stage.
- compared to Simultaneous Speculative Threading
Acknowledgement

I would like to express my gratitude to my advisor, Associate Professor Kenji Kise for his advice and encouragement for three years.

I would like to thank all the members at Kise Laboratory, particularly Mr. Naoki Fujieda and Yuhta Wakasugi for simulator I used.

I would like to thank my parents during the years of my studies.
Bibliography


Publication

Main Author

Presentation

1. 松村貴之, 坂口嘉一, 吉瀨謙二：マイクロアーキテクチャの異なるプロセッサの性能比較, 情報処理学会第 73 回全国大会, 東京工業大学 大岡山キャンパス (2011 年 3 月 2 日発表), Vol.1, No. 1H-4, pp. 57-58 (March 2011).

Poster


Coauthor

Presentation

1. 坂口嘉一, 松村貴之, 吉瀨謙二：スーパースカラ版 MIPS®の実装と評価, 情報処理学会第 73 回全国大会, 東京工業大学 大岡山キャンパス (2011 年 3 月 2 日発表), Vol.1, No. 1H-2, pp. 53-54 (March 2011).
Conference