Evaluating the Renamed Trace Cache Architecture

(リネームド　トレース　キャッシュ
アーキテクチャの評価)

Supervisor: Professor Shuichi Sakai

Yen-chun Wang

DEPARTMENT OF INFORMATION SCIENCE AND TECHNOLOGY,
THE UNIVERSITY OF TOKYO

February 2010

Abstract

In order to realize out-of-order execution, register renaming is commonly used in superscalar processors now days. Within the register renaming process, RMT (register mapping table) is utilized to map logical (architecture) registers to physical registers. However, the RMT is said to be one of the most energy costly components in the processer due to its large area and frequent access. This makes register renaming costly and hard to widen. By using the Renamed Trace Cache Architecture[5] (RTCA in short, also known as Anti-dualflow architecture), RMT problems can be simplified and solved. Operands in this architecture are converted to the [-n] form which depicts the instruction distance to the value it references. After translation, these instructions are stored to the trace cache for further usage. As a result, data dependency is explicit upon instruction fetch and register renaming can be eliminated. Although capable of eliminating the register renaming stage gives RTCA the advantage of a shorter pipeline and miss penalty, degradation in trace cache hit rate is an unavoidable issue RTCA suffers from. Increase in trace numbers lowers cache hit rate and pulls down performance.

From the perspective that RTCA is a register renaming stage free architecture makes it a very appealing architecture. Nevertheless, very few researches regarding it have been conducted. In this thesis, we attempt to explore some of the design spaces that RTCA have and quantify the results. These results show the different characteristics RTCA have which can help develop techniques to further improve RTCA performance. Our results show that RTCA is a highly potential architecture and provide justification for future research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Maximizing ILP(instruction level parallelism) in modern processors has been the fundamentals in increasing execution throughput. The more instructions that can be processed in parallel, the more throughput can be gained. Different methods attempt to achieve this goal. Mainly, we can categorize these methods in two by their position in the pipeline, in other words the frontend or backend.

The processor pipeline is usually separated into the frontend and backend by issue buffers. Some examples of the issue buffer are reservation stations or issue queues. A rough impression is that instructions are fetched in the frontend and provide instructions into the buffer, while the backend consume or execute the instructions that come out of the buffer. The two sides are then linked by control dependencies, such as branches and jump instructions.

From the backend perspective, in order to process more instructions at the same time, issue bandwidths are made wider, followed by making the instruction window larger and increasing parallel functional units. Under such circumstances, the frontend side, which acts as the instruction provider, can become a bottleneck. One of the most typical proposals that increase the frontend bandwidth is the trace cache. The RTCA(Renamed Trace Cache Architecture) throughout this thesis uses the trace cache architecture as the base architecture.

The trace cache was proposed concerning the frontend instruction fetch bandwidth as a bottleneck. In an ordinary instruction cache, instructions are stored in their static or compile order. The main disadvantage of the ordinary cache is that it cannot capture the dynamic instruction stream in the program. Programs contain significant numbers of control and branch instructions, which are scattered around the ordinary cache. Basically, the trace cache aligns the scattered instruction blocks together. Once a trace inside the trace cache is accessed, a portion of the dynamic instruction stream can be provided consequently increasing the fetch bandwidth.

One of the most difficult issues in exploiting ILP is data dependency. Instructions cannot be considered at the same time if their data dependency problems are not solved. This dependency can cause three kinds of hazards in the pipeline, namely RAW(read after write) and WAR (write after read) and WAW(write after write). Modern superscalar processors use register renaming to solve these hazards. By renaming register names, WAR and WAW hazards are able to be removed. As a result, the true data dependency of the data flow is then revealed. Register renaming is what makes out-of-order execution possible and stands at an important place in modern superscalar processors.

Renaming between registers is supported by the RMT(register mapping table) which maps register numbers to their new names. However, the RMT is said to be one of the most energy consuming parts in the processor. This is considered a result of its large area and frequent access. A typical 4-way superscalar processor will need 16 ports for the RMT, not to mention wider superscalar processors. Researches show that area can be proportional to port number to the power of 2[11]. Area can rise substantially when fetch width grows larger.

The RTCA is an architecture proposed to solve problems that the RMT bring about. The main idea of the RTCA is that it translates instructions dynamically into a new form which depicts the displacement between dependent operands(We use the word ”translate” instead of ”renaming” to differentiate it from an normal register renaming stage). These translated instructions are further stored inside the trace cache for further usage. Translation only occurs when the trace cache misses. On the other hand, upon trace cache hit, translated instructions are provided to the instruction window. It has been shown that the bandwidth of this translation can be chose rather small[5], which means that the size of the mapping table used in RTCA can be constrained compared to an ordinary RMT used in register renaming. Another benefit the RTCA has is that since translated instructions already show the dependency, register renaming stages in the pipeline can be eliminated.

Little research has been conducted on such microarchitecture research. The fact that RTCA uses a small mapping table and the renaming stage is absence, makes it a highly potential architecture, yet only few evaluations have been made. In this thesis, we attempt to evaluate the impact that RTCA can have on performance and explore the possibilities of this architecture. Our research shows the different characteristics of RTCA which can help develop further improvement techniques.

1.2 Thesis Overview

The rest of this thesis is organized as the following. The basics of register renaming will be covered in chapter2. Chapter3 will describe the basic mechanisms of the trace cache architecture which play an important part in the RTCA. Chapter4 shows how the RTCA is built on the trace cache architecture. Details of its structure and the data flow in instruction translation will be elaborated. Different design spaces that this thesis considers are listed in chapter5. The results of some simulation will be presented in chapter6. Finally, summary and conclusion will be made in chapter7.

Chapter 2

Register Renaming and Data Dependence

To process in out-of-order, superscalar processors use register renaming to analyze true dependence between the instructions and eliminate WAW and WAR hazards. Nevertheless, register renaming can be a rather high cost stage.

High cost here refers to the RMT (register map table) used in register renaming, which mainly contains the mapping between the physical and logical registers. Take a typical 4-way superscalar processor as an example, if the RMT is implemented by RAM, ports may be as much as 16 ones. The circuit area of RAM is proportional to the number of ports with the power of 2[11]. This means that excess ports may create large area and consumption power. In addition, the frequent access to the RMT even consumes more energy. This leads to the proposal of renamed trace cache which tries to eliminate the register renaming stage, cutting off the RMT, and maintaining IPC while consuming less power.

2.1 Data Hazards

There are three main dependences that may cause hazards that can stall the pipeline. The first one is called data dependences (or true data dependences). This happens where an instruction produces a result that may be used by another instruction, in other words, when a register is written after read. If the read operation is preceded before the write, then the instruction reads the wrong value and thus causing a RAW (read after write) hazard in the pipeline. This is also called the true data dependence since program order must be preserved to ensure reading the right value.

The Other two dependences are antidependence and output dependence, which both are name dependences. Antidependence happens when an instruction tries to write to a register that another instruction reads while output dependence occurs when two instructions try to write to the same register. These two correspond to WAR (write after read) and WAW(write after write) hazards in the pipeline. Since there is no value being transmitted between the instructions, name dependences are not true data dependences and are able to eliminate using methods such as register renaming. Note that RAR (read after read) is not a hazard. Although WAR WAW hazards are unlikely to occur in the typical five stage pipeline, since writes are late and reads are early in the pipeline, they become possible if we want to execute in out-of-order sequence. This is why register renaming is important in now day out-of-order superscalar processors.

2.2 Renaming Scheme and the RMT

The basic idea of register renaming is to change one of the names of the same register so that values are written in different locations, hence eliminating the WAR and WAW hazards. Consider the following code as an example

```
Div     r0,r2,r4
Add     r6,r0,r8
Store   r6,0(r1)
Sub     r8,r10,r14
Mul     r6,r10,r8
```

There are three places that include the true data dependences : r0 between div and add, r8 between sub and mul, and r6 between add and store. R8 between add and sub have an antidependence and is potential in a WAR hazard. An output dependence is between add and mul and may cause a WAW hazard. With register renaming, we can change the register names into the following

```
Div     r0,r2,r4
Add     S,r0,r8
Store   S,0(r1)
Sub     T,r10,r14
Mul     r6,r10,T
```

In this case, r6 is substitute by S and r8 by T. The substitution of S makes the r6 in mul write to different locations and eliminating the WAW hazard. The same process happens in T when the sub and the add access to different locations. Thus register renaming is capable of getting rid of false dependence and leaving the true data dependence or data flow clear.

Register renaming utilizes a RMT (register map table). Like its name, the main purpose of the RMT is to map logical registers to corresponding physical register. However, this table is a big workload to the processor. When implemented by RAM, the RMT table can use up to 16 ports in a 4-way superscalar processor[11]. With these excess ports accompany large chip area and consumption power. In addition, as we can see from the above renaming example, frequently access to the RMT is necessary during the process. This results to even more energy consumption.

With register renaming comes the high energy consumption RMT component. If it is possible to somehow replace the register renaming stage, then several benefits may arise. Such as a shorter pipeline, wider front-end bandwidth, and lower power consumption.

Chapter 3

The Conventional Trace Cache Architecture

RTCA uses a trace cache architecture[2][3] as its base. Before further explanation is made on RTCA, we first introduce the basic concepts of the trace cache architecture for clearer understanding.

3.1 Trace Cache Composition

While the instruction cache stores instruction blocks according to their static or compile positions, the trace cache stores traces. A trace is a sequence of instructions which are a portion of the dynamic instruction stream. In the instruction cache, fetching stops when a branch is encountered inside the fetch block. Instructions after the branch must be fetched in the next cycle. This is not the case for the trace cache. The trace cache aligns the blocks after the branches which form a trace. From a point of view outside the cache, the trace cache simply makes noncontiguous instructions appear contiguous. Therefore, when fetching a trace, instructions after a branch are also available and fetch bandwidth can be increased. Traces inside the cache are specified by their starting address and the multiple outcomes of their branches inside the trace.

3.2 Trace Cache Fetch Mechanism

The trace cache works in cooperative with the instruction cache. It can be set that the access between the trace and instruction cache be parallel or sequential. In an

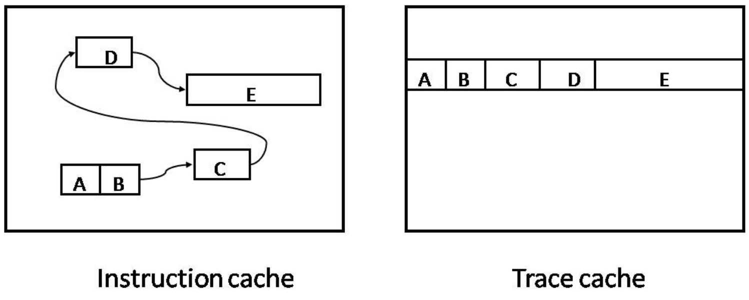


Figure 3.1: Comparing Caches

parallel structure, instructions inside both caches are fetched at the same time. The hit logic of the trace cache determines which set of instructions are chose. If a trace cache hit occurs, the trace is fetched, or else the other instructions from the cache are. On the other hand, in a sequential structure, the trace cache acts as the primary cache. Instruction caches are only accessed upon a trace cache miss. While miss penalty can be larger in sequential structures, accessing parallel may face problems such as larger energy consumption.

Hit logic of the trace cache determines when there is a trace cache hit. Like the instruction cache, this is usually done by comparing tags. Additional tags are required for traces because of the branches inside them. While determining hit or not, prediction bits from the multiple branch predictor are also necessary. The following are some tags chose in a typical trace cache.

- *Valid Bit* : Indicates if the traces is valid for fetch. Same case as the instruction cache.

- *Tag* : Address tag of the fetching address. Usually higher order bits of the trace starting address are used.

- *Branch Flags* : A single bit that represent the outcome of each branch. 1 as taken and 0 as untaken. The last branch of the trace does not need a bit because no instructions follow it.

- *Branch Mask:* States the number of branches inside the trace to insure the right number of bits inside the branch flag are used for comparison. Also used by the predictor when updating. A additional bit is used to indicate whether the last instruction is a branch or not.

- *Trace Target and Fall-through Address*: The corresponding addresses if the last instruction is a branch.

The trace starting address(sometimes along with the branch flags) is used to generate the index. Then the above tags are compared to determine a trace hit or not.

After instructions are retired, they are sent to the fill unit. The fill unit decides when the filling of a trace is completed and ready to be written inside the trace cache. Basically, the fill unit stops filling a trace when 1. the trace has meet its maximum size, or 2. the maximum branch number is reached, or 3. when encountering returns, indirect jumps, or traps. The first two reasons are rather obvious and the third one is because these control transfer instructions have more than one target address, whereas the predictor can only predict one.

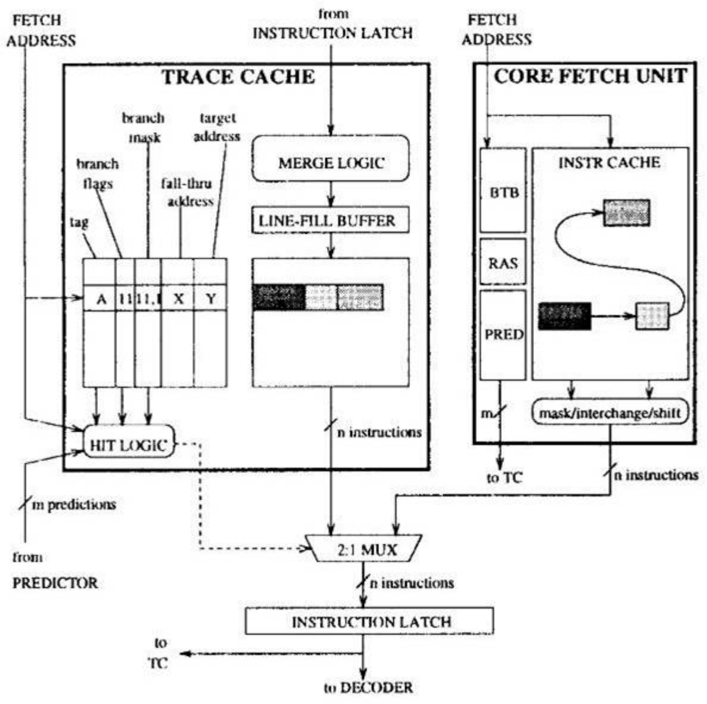


Figure 3.2: The Trace Cache

3.3 Trace Cache Design Space

The performance of the trace cache can depend on many other elements. The following are some issues that can influence the trace cache.

- *Indexing and Associativity* :The simplest way to index into the trace cache is just using the starting address. This implicitly means that the traces in the same set have to start in the same address. One way to decrease this conflict miss is to use additional bits in the branch flags to index. By doing so, path associativity is also provided.

- *Partial Matching* :Trace cache hit only indicates that the branch predictions match. It is still possible that some of the branches go off course. Partial matching supports fetching till the right prediction instead of flushing the whole trace.

- *Trace Selection* :One of the most important issues in trace cache is the method used to select stored traces. Trace cache can suffer from redundancy, in terms that many of the traces stored are only used for very few times. A possible solution for this is to add a smaller buffer to filter traces that have higher hit counts.

- *Filling Issue* :Trace filling can be done atomically or separately. Atomic filling means that the basic filling unit is an instruction block. Filling stops when the current filling block size is larger than the vacant space. Separate filling fills the trace as full as possible. Loops are unlooped by using this method. Although this method may seem to serve more instructions on a trace hit, the increment in total trace number can cause cache redundancy which can also lead to higher miss rate.

3.4 Multiple Branch Prediction

The performance of the trace cache is highly related to the predictor. Predicted results of the multiple predictor are sent to the trace cache to compare them with the branch flags. To obtain a trace hit, all predictions have to concur at the same time. If we use a similar predictor as the instruction cache, miss rate could be enlarged to branch number of times. For example, assume a trace can contain up to 3 branches and the predictor has a accuracy of 90% of a single branch, then the percentage that the branch flags match would be $0.9 \times 0.9 \times 0.9 = 72.9\%$. In other words, there is a $27.1 - 10 = 17.1\%$ increase in miss rate.

We use a Gshare correlated branch predictor[10] in this research. The global history register XORed with the trace starting address is used as an index to the PHT(pattern history table). Four set of counters consist the PHT. Prediction and history bits are then used to chose one set from the four. Fig3.3 illustrates the multiple branch predictor.

Many other researches on multiple prediction have been made. One for particular is the path-based next trace prediction. This predictor, designed especially for the trace cache, treats traces instead of instructions as the basic entity when predicting. The predictor uses past trace history and makes prediction based on them. More information can be found in [6].

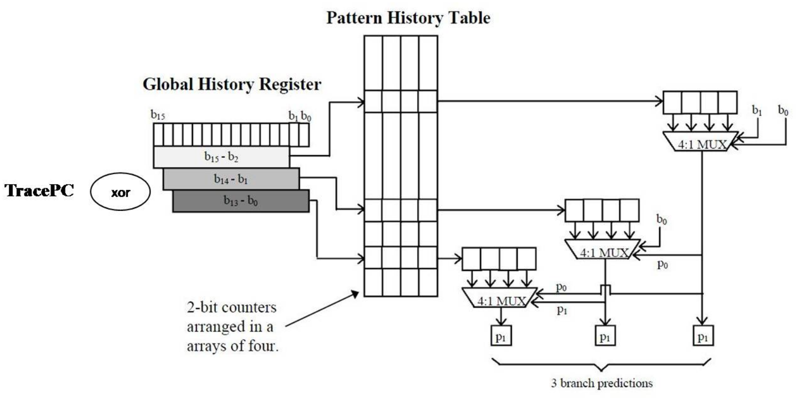


Figure 3.3: Correlated Muliple Predictor

Chapter 4

The Renamed Trace Cache Architecture

In the previous chapter, basic knowledge of the conventional trace cache architecture have been presented. The RTCA uses the trace cache to store information that are capable of making the register renaming stage unnecessary. By doing so, RMT problems can also be solved. Details of the process and schemes used in RTCA will be given in this chapter.

4.1 Architecture structure

The basic concept of the RTCA is to solve RMT problems through deletion of the register renaming stage. This is done by translation instructions into a new form after fetched. This new form, which we call the [-n] form, shows the displacement *n* between dependent instructions. As mentioned before, the purpose of register renaming is to detect the true data dependency. By renaming the logical registers into physical registers, results of write operations are written into different registers which eliminate possible WAR and WAW hazards. On the other hand, the displacement shown in the [-n] form are already dependence explicit. This means that by looking only at the instructions, we can know the true data dependency. As a result, the register renaming process can be skipped. This is one of the most remarkable characteristics RTCA architecture possess.

Producers are instructions or operands that produce and compute values needed for other instructions, whereas consumers are those instructions or operands that use those produced values. The displacement of each [-n] form shows that the current instruction, as a consumer, uses the value of its producer which was the [-n]th instructions before. Note that the RTCA is not an instruction set architecture, meaning that instructions that use their origin instruction set architecture are translated to the [-n] form dynamically in the pipeline. We will elaborate more on this translation process further on in this chapter.

First of all, we start our explanation from the backend side of the RTCA. Like any other modern superscalar processor, instructions are stored in the issue queue, or instruction window, before they can be sent to the execution unit. In the RTCA, instructions in the window are already in their [-n] form. These instructions are issued once the scheduler decides that they are ready.

In addition to the logical registers, RTCA uses a cyclic buffer that contains certain amount of physical registers. These physical registers act like the reorder buffer in other superscalar processors. Particularly, result values of instructions are stored in this buffer as the same sequence they are fetched. These stored value are then passed to the logical registers in-orderly after they retire. In other words, this buffer saves intermediate values of the logical registers between the time instructions are executed but not yet retired, just like the reorder buffer. The key point in all of this is that when an instruction is readied to be executed, the [-n] form tells us where to look for the source operand value. For example, a [-5] operand would mean to look 5 entries above in the buffer. If the displacement between dependent instructions is larger than the window size, it simply indicates that the producer instruction has already retired and the needed value should be referenced from the logical register instead of the physical registers inside the buffer.

Now we focus on the frontend side. Fetched instructions, that are in there original instruction set format, are translated to the [-n] form in the frontend. Once these instructions are translated, they are sent to the instruction window and further stored inside the trace cache. As a result, instructions in the trace cache are all in their [-n] form. The fetch mechanism starts with the trace cache. If there is a trace cache hit, since instructions are already in the [-n] form, translation is not needed and they are sent to the window directly. Otherwise, instructions are fetched from the instruction cache and then translated to the [-n] form before they go into the window. From the facts above we can clearly know that the trace cache hit rate plays an important role in performance.

4.2 Instruction Format

Instructions are translate from their original ISA(instruction set architecture) to their RTCA format, or their [-n] form, upon trace cache miss. The instruction format of the [-n] form that RTCA uses is shown in Fig4.1. As explained in section4.1, the operand value of an instruction can either reference from [-n] instructions above in the physical register buffer, or it can reference from the logical registers. This depends on the displacement which shows if the register value has retired yet. Displacements that are larger than the window size indicate this retirement. The role of each bit sections in the format is given as the following.

- *Opcode* : Stands for the instruction operation code
- *DSTReg* : Shows the destination logical register number.
- *RL/RR* : One bit which indicates if its corresponding section, SrcL or SrcR, shows the displacement or shows the register number used in value reference. 0 means displacement while 1 means register number.
- *SrcL/SrcR* : Stores either the register number or the displacement needed.
- *Imm* : Immediate values.

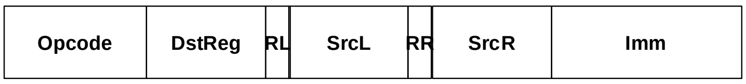


Figure 4.1: Instruction format

4.3 Translating Instructions

Instruction translation happens when the trace cache misses and instructions are fetched from the instruction cache. These translated instructions are then used in the window and stored in the trace cache. The fact that the trace cache only stores translated instruction makes translation irrelevant when the trace cache hits.

This instruction translation is done basically in the same way as register renaming. The basic procedure in this process is shown in Fig4.2 and below.

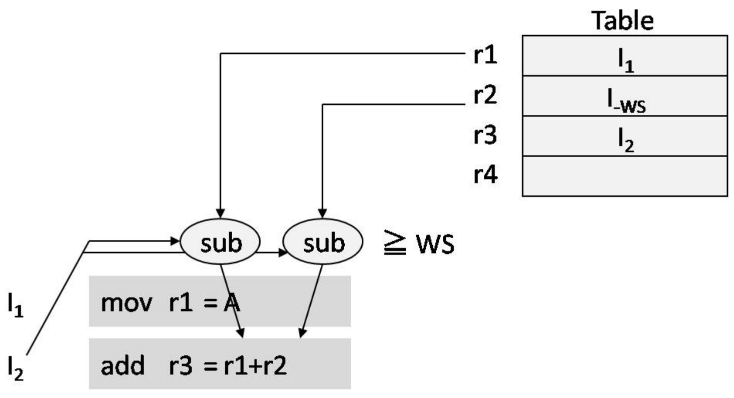


Figure 4.2: Instruction Translation

1. Each instruction has a index which shows its place in the dynamic instruction stream. This index is added by 1 and then assign to the instruction(s) during each fetch cycle.

2. When an instruction is encountered, its index and destination register number are stored in a mapping table. This table is updated every time a new instruction is examined. In other words, each register number in the table always maps to the index of the instruction that has last used it.

3. After examining the destination register number, source operands are examined. If the source uses a value from a register, the mapping table is accessed to get the corresponding index. The difference of the examined instruction index value and index value obtain from the table represents the displacement between the producer and consumer.

4. The calculated difference is compared with the window size. Differences larger than the window size indicate that the reference instruction has already retired and values should be referenced from logical registers. In this case, the logical register number will be written in the SrcL/SrcR section of the instruction format and RL/RR will be set to 1. For cases that the difference is smaller than window size, values should be referenced from the physical register buffer and the displacement will be written in the instruction format, making it become the [-n] form.

At first glance, this translation may seem not as beneficial since it uses a mapping table similar to the RMT. However, with the trace cache storing instructions already translated and assuming that the trace cache has a hit rate high enough, translation frequency can drop dramatically compared to the RMT. Less access means less consumption energy. Furthermore, with the same assumptions made, the translation bandwidth can be set rather small compared to conventional register renaming stages. By doing so, access ports and cycles can be constrained and the chip area of the mapping table can be controlled. Yet again we see the important role the trace cache plays in the RTCA. The reason RTCA can solve RMT problems is highly based on the fact that it uses the [-n] form along with a high trace cache performance.

4.4 Execution Model

In the last section, details of the translation that takes place in the frontend side were given. For further clarification, we give a concrete example of how things work in the backend side of the RTCA. Fig4.3 illustrates how this is done. In this example, we assume the window size is 4 and instruction I3 to I6 in Fig4.3 are currently in the window. We start from examining the instruction I4.

1. During I4, the result value A will be written in its corresponding place in the physical register buffer.
2. The left source operand of I5 is in the [-n] form. [-1] indicates that this operand value is stored one entry above in the physical register buffer.
3. The left source operand of I6 in not in the [-n] form. r3 indicates that this operand value is stored in the logical register r3. Actually, when looking back in the instruction stream, r3 is dependent to the result value of I2. However the displacement between them is larger (or equal) than the window size, 4 in this case. This indicates that the value in I2 has already retired from the buffer and is also the reason why I6 has not been translated to the [-n] form.

In summary, from the backend side point of view, reference of operand values is easily done. The physical register buffer is simply accessed when operands are in the [-n] form and logical register are referenced when they show register numbers.

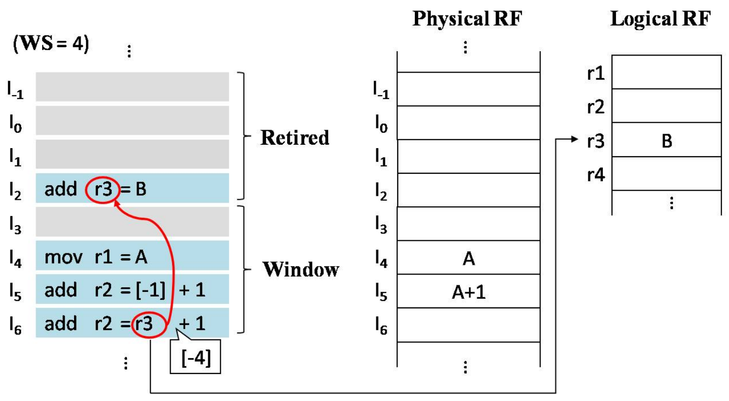


Figure 4.3: Execution Model

4.5 Instruction Scheduling

Elaboration of instruction translation in the frontend side and execution in backend side have been made. The instruction window is what links these two parts together. The main role of the scheduler is that it needs to decide which instructions are ready for issue by looking that if their source operand values are all ready. Thus, the dependences between instructions also need to be examined in this phase. Basically, when a producer produces its value, it needs to tell all the consumers that depend on it that the request value is ready for use. This is called the associativity of the wakeup logic. However, association memory can cause time delay issues that are possibly critical.

The dependence matrices that was proposed in the dualflow architecture and other researches done by Goshima et al.[12] moves heavy parts of the dependence detection to the frontend. The matrices itself shows the dependence between instructions and is updated as this dependence is detected in the frontend side. As a result, the wakeup logic can be realized by just reading the matrices. Since the dualflow architecture and the RTCA share similarities in resolving dependence, the dependence matrix can also be easily implement in the RTCA. Fig4.4 shows how the dependence matrices works.

The dependence matrix shows the dependence between instructions by displaying the relation of consumer and producer. The instruction indices are first distributed to the row and column numbers. Rows instruction indices show the current examined instructions in the window while column instruction indices show the instructions that produce the needed value(s) for each row. A bit of 1 in the left matrix shows that the column index instruction is one of its producer. For example, [-2] and [-1] in row I3 means that I3 uses values from 2 and 1 instructions above. By adding -2 and -1 to the row instruction index, which is three, we get the producer index and set column I1 and I2 to one. Once the ready bit of I1 and I2 are 1,which indicate their values are computed, their consumers are set to 0. The ready bit is set to one when the bit values in the row are all 0. This can be done by NORing all the bits. The procedure of Fig4.4 is summarized below.

1. In row I3: 3+ [-2] = 1 ; 3+ [-1]=2 which sets column I1, I2 to one

   In row I4: 4 + [-1] = 3 ; which sets column I3 to one

2. In row I1,I2: Ready bits are set to one right after their values are ready

3. In row I3: Bits in column I1, I2 are set to 0 which indicate producers are ready

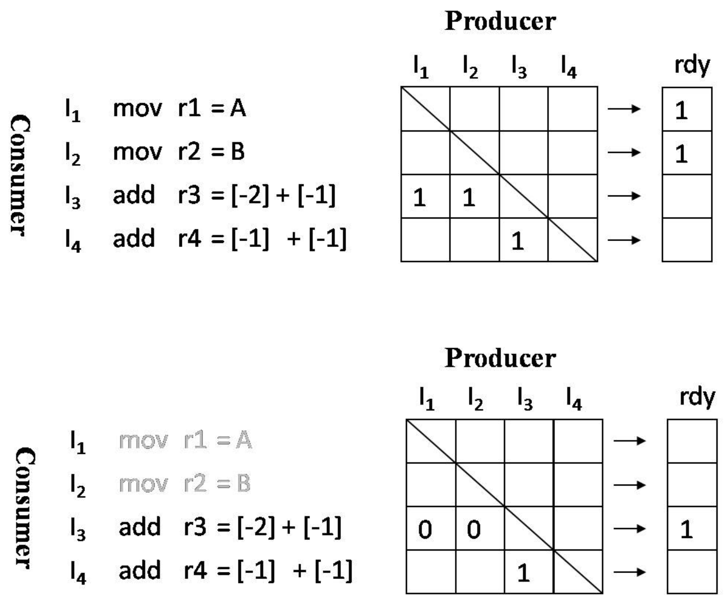


Figure 4.4: The Scheduling Matrix

4. In row I3: NORing all the bits sets ready bit to one.

5. Next instruction I4 is examined.

RTCA has the advantage that it can easily find a producer by simply adding the [-n] form to the instruction index. Just a reminder that the [-n] form, which is translated in the frontend, decreases the complexity of examining dependence in the wakeup logic. The dependence matrix further removes association issues.

```
          変換前                              変換後
                                    (untaken)              (taken)

     mov   r1  = ...            mov   r1  = ...          mov   r1  = ...
     bgt   r1  > 0 then L1      bgt  [-1] > 0 then L1    bgt  [-1] > 0 then L1
     neg   r1  = -r1            neg   r1  = -[-2]    L1: add   r3  = r2 + [-2]
L1:  add   r3  = r2 + r1    L1: add   r3  = r2 + [-1]
```

Figure 4.5: Translation Dependent on Path

4.6 Expressing Paths

The dependent displacement value of [-n] in instructions may change according to the control instructions executed. This means that the translation outcome is also dependent to the control flow, which can complex things in the RTCA. We illustrate this by explaining the example in Fig4.5.

4.6.1 Path Concerns

Fig4.5 considers the situation when a branch is encountered in RTCA. First we assume that the second branch instruction is an untaken branch. This way the third neg instruction will be executed and its source operand r1, because it depends on the mov instruction 2 above, will be translated to [-2] in the [-n] form. Consequently, the r1 in the next add instruction will be translated to [-1].

Secondly, this time we assume the branch is a taken branch. The taken branch would lead to the add instruction and skip neg. In this case, the r1 in the add instruction depends on the mov(2 instructions above) instead of the neg(one above) and is turned into [-2]. This example shows that the results of the translation can defer depending on branch outcomes. In other words, the [-n] form of an instruction depends on the path the instruction has followed, or is path dependent.

4.6.2 Path Expression

Path dependence of the [-n] form is solved in the trace cache used. The trace cache stores instructions only in their [-n] form. In order to determine the correct dependent instruction the displacement points to, the path that each instruction followed also needs to be known. This is done by adding path tags in the trace cache that indicate the past path. While instructions contained in a trace show the desired fetch instructions in the future dynamic instruction stream, the path shows instructions that are before the fetch address. Notice that both the trace and path are a portion of the dynamic instruction stream. This is also the main reason why the trace cache was chose to realize the RTCA.

The usage of path tags are just as same as the trace tags. Tags that are chose must exclusively represent each path the correspond trace fetch address had followed. In the RTCA, the following are some of the tags that were chosen.

- Starting Address *H*: The starting address of the path, which is the farthest instruction that the trace needs. In short, the maximum [-n] address of the trace.

- Length *L*: The length of the path. It can be calculated by looking at the instruction numbers between the trace starting address and path starting address.

- Past Branch Flag *pbflag*: Bits that show the control flow in the path. The bit is set to one if an instruction's next instruction is not its fall through address. 0 stands for a untaken branch or not a branch at all. Since the path cannot exceed the size of the window, (window size -1) bits are the most bits needed to be compared

- Indirect branch number *jcount*: The *pbflag* is insufficient to represent control flows when indirect branches are encountered. This is because one bit in the flag can only depict two outcomes, taken or untaken, while these indirect branches can have indefinite target addresses. Hence, indirect branch information should also be given. *jcount* represents the numbers of indirect branches in the path.

- Indirect branch targets *jtarget*: In addition to *jcount,* the target address(es) of the indirect branches in the path.

Fig4.6 shows an example of path tags. The starting address B is computed by finding the maximum [-n] in the trace. Starting from that address, we can construct the *pbflag*. In Fig4.6, 00100 indicates that the third instruction is a taken branch. If indirect branches are encountered, *jcount* and *jtarget* are set to their values.

In addition to the trace tags that depict branch outcomes of the trace, the above path tags are also added inside to determined the desired trace. Since different paths can lead to the same trace and the tag numbers needed to be compared upon a trace cache fetch is increased, decrease in trace cache hit rate is expected.

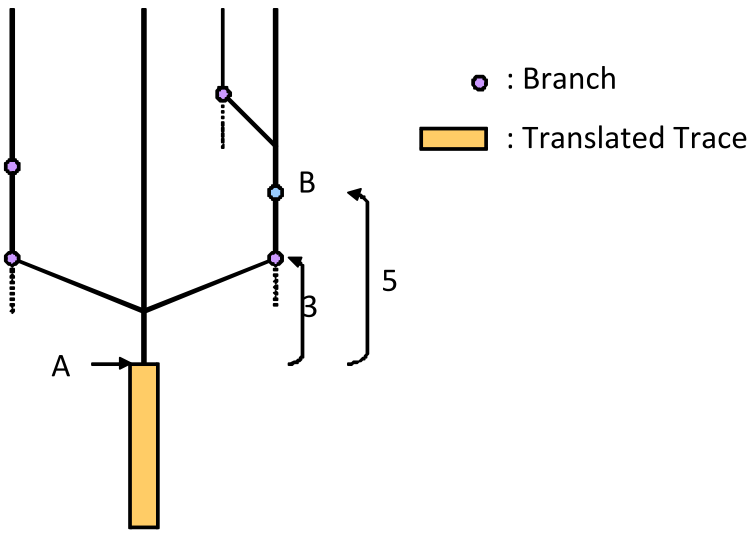


Figure 4.6: Path Description

4.7 Adding the Trace Cache

After an instruction is translated, it is sent to the fill unit where traces are constructed. If a trace filling is complete, instructions inside the trace are used to construct trace tags while path address histories and path branch histories are used to construct path tags. These tags are used to decided the trace cache hit.

The expected decrease in trace cache hit is cause by conflict misses due to the path tags. One of the methods to decrease conflict misses is by changing the index used in the cache. In related research, an addition parameter $L_{\min}$ is chose to decrease conflict miss[5]. We describe a little about this index generation using $L_{\min}$.

4.7.1 Index Construction

Althogh the easiest way to index into the trace cache is by using the starting address of the trace, conflict issues may arise. Many researches on trace cache indexing have been made, some of which involve hashing. These methods are also available in the RTCA. One particular methods used to decrease the conflict miss cause by past tags was proposed in [5]. It is done by using a parameter $L_{\min}$ with the past branch history and past address history.

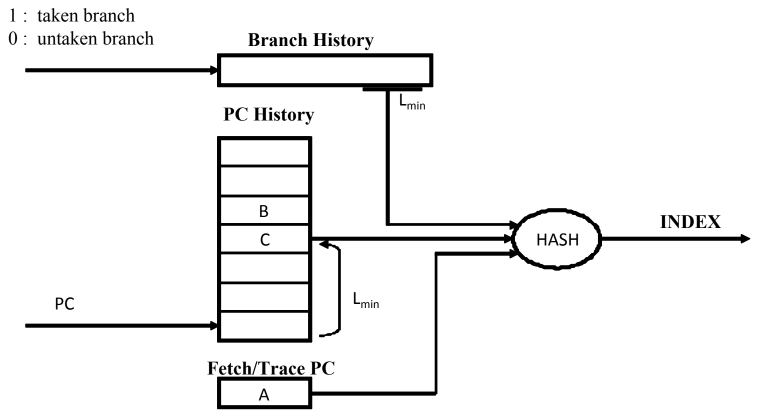


Figure 4.7: $L_{\min}$ Indexing

Besides the global branch history register, the past address histories are also stored. Once an instruction is retired, it is sent to the fill logic and past address history at the same time. While the fill unit shows the instructions currently inside a trace, the past address history shows the past path the trace has followed. The H tag also uses this history for comparison.

The addition parameter $L_{\min}$ can be chose according to the program or just randomly. $L_{\min}$ limits the minimum required length of the past path. In other words, if a past path of a trace is shorter than the value $L_{\min}$, additional path length confirmation upon fetch is needed. $L_{\min}$ is also used in index generation. An example is using the corresponding $L_{\min}$th past address and past branch history in addition to the starting address of the trace. A hash function on the $L_{\min}$ past address, $L_{\min}$ past branch histories, and the starting trace address generates the index. The simplest form of this function is by using a XOR function. Fig4.7 shows how the $L_{\min}$ is used in index generation.

$L_{\min}$ can be changed on a program basis. However, choosing the value of $L_{\min}$ can be tricking. Although $L_{\min}$ reduces the numbers that can be stored in a single set in the cache, it can also scatter possible useful traces that have shorter lengths. The effeteness of $L_{\min}$ is highly based on the assumption that the traces frequently used have longer lengths than $L_{\min}$. However, this may not always be the case. In this thesis, we discuss the effect of $L_{\min}$ on performance.

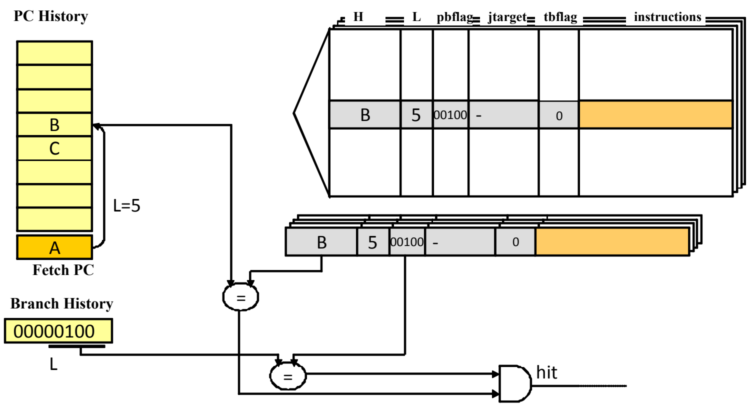


Figure 4.8: Tag Comparison

4.7.2 Tag Comparison

By observing the past path tags mentioned in the previous section, we can see that the value of the starting address H is dependent of the length value L. Note that at the point of time when fetching inside the trace cache occurs , the length value of L is unpredictable. These two facts make tag comparison in RTCA slightly more complicated than the conventional trace cache.

The first step when fetching inside the trace cache is index generation. Several methods used for generating index have been mentioned in the previous section. The generated index is further used to lookup a specific set inside the cache. The tags of the traces inside the set are then compared for trace hit. Trace tags are compared with the results of the multi-correlated branch predictor that produces it in advance. On the other hand, path tag comparison involves using the past branch history and past address history. Fig4.8 shows how this is done.

The length value L of the compared trace is first used to lookup inside the past branch history and past address history. Secondly, the Lth past instruction is then compared to the starting past address H inside the tag. The latest L bits of the past branch history is also compared in the same way with the *pbflag*. As for indirect branches, the nearest *jcount* numbers of indirect branch target addresses are compared with the *jtarget* area. Concurrence in all these section insure the required right past path.

The unpredictability of L is the main reason that complicates the tag comparison. Hence the set inside the cache needs to be first accessed in order to achieve the value of L, and then comparing tags is done by using it. This may seem a bit slower in time compared with normal tag comparison, but since the tag size is rather small compared to trace size, the effect on trace fetch latency can be neglected.

4.8 Renamed Trace Cache Architecture Issues

The RTCA has its own advantages and disadvantages. We list the main issues of RTCA below.

- **RMT and Translation Bandwidth**

  The main purpose of the RTCA is getting rid of the RMT, and the problems it arises, used in conventional register renaming process. Nevertheless, the RTCA uses its own mapping table when [-n] form translation occurs. This mapping table in the RTCA is considered small in area, and has less access ports with less access frequency. However, these benefits can only be achieved under the constraint of small translation bandwidth. Effects of this constraint on fetch bandwidth is overcame by getting the most pre-translated instruction from the trace cache possible. While the suitable translation bandwidth is highly related to the trace cache, trace cache performance stands in an important position in the RTCA.

- **Maintaining Performance**

  Overall performance can depend on two main factors. The first one is pipeline stage numbers. By using the [-n] form, the register renaming stage can be skipped. Less number in stages imply less penalty on miss prediction. This factor contributes a positive side on performance.

  Trace cache performance can also have a big impact on overall performance. RTCA trace cache suffers from the additional tags needed to express past path. These tags cause degradation in hit rate which can lower the overall performance. If the tradeoff between pipeline stage and trace cache performance can be controlled, performance can be maintained while eliminating RMT problems at the same time. In this thesis, we consider the trace cache performance as the main factor of altering performance. Further observations will be made in the next chapter.

4.9 Comparing with Pentium4

The Netburst Architecture is one of Intel ’s microarchitecture used on the Pentium4 product line [4]. This Netburst architecture uses various schemes and concepts similar to those of the RTCA. Simple introduction and elaboration will be made in the section.

Before issuing the instructions to the renaming logic, they are decoded first. This decoding step is somewhat similar to the translation step of the RTCA mentioned in previous sections. IA- 32 instructions are decoded into smaller instructions called micro-operations, or uops, before they are sent for execution. At uop level, more analysis can be made for reordering code sequences in order to dynamically optimize mapping and scheduling of uops onto machine resources. In the newer Intel core architecture, the opposite is also done for better performance. This process is called macrofusion which enables common instruction pairs, like a compare followed by a conditional jump, be combined into one internal instruction during decoding. Two instructions can be executed as one which reduces the overall workload.

However, the decoding of IA-32 instructions can be troublesome. The decoding logic needs to sort out which option of uop the complex instruction should be transformed in and this becomes even harder when trying to decode several instructions simultaneously. Thus the Netburst uses a trace cache (Intel calls it the execution trace cache) to solve this problem. Instuctions encountered for the first time are sent to the decoder and decoded to uops. These uops are then stored in the trace cache for further use. If complex IA-32 instructions, such as a string move, are encountered, they reference from the microcode ROM. The uops that come from the trace cache and microcode ROM are then buffered to a in-order queue which flows to the out-of-order execution engine.

Like the RTCA, the trace cache in Netburst also stores decoded traces of instructions. If the trace cache hit rate is high enough, the throughput of the decoding process, or translation, can be chosen as small as possible. According to Intel, the P4 decoder only needs to decode one IA-32 instruction per clock cycle. This is similar to the case of the RTCA. In the Netburst, trace cache miss happens as often as previous processors miss their L1 instruction cache. This is a rather high hit rate compared to the RTCA. The main difference between the two is that the Netburst still uses register renaming and the uops fetched from the trace cache may still contain false data dependences.

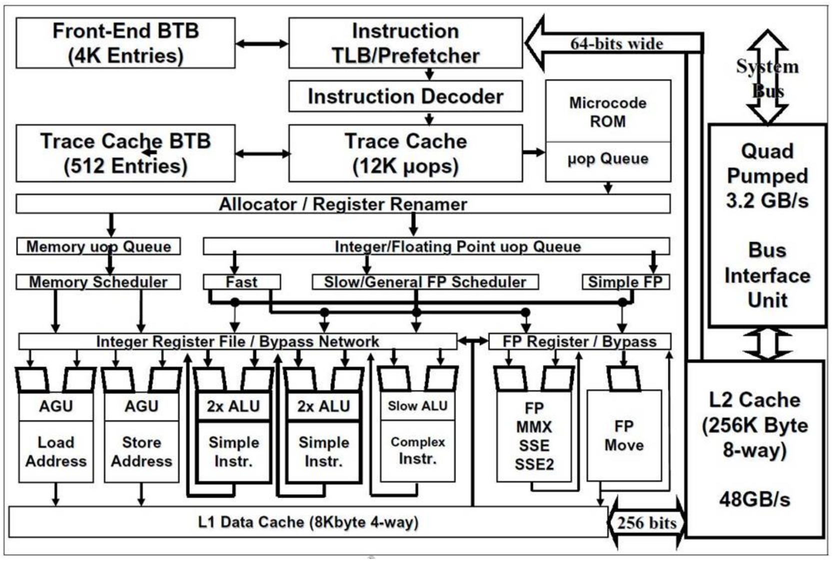


Figure 4.9: P4 Architecture

Chapter 5

Thesis Contribution

The RTCA is a rather novel architecture. Although it benefits by putting dependence displacement in instructions, past path expressions can become a large issue. With the limited related researches conducted on the RTCA, lots of part of its characteristics and issues have not been made clear yet. This thesis will attempt to quantify some of RTCA's features which may stand as useful references when further future works are conducted.

We consider the trace cache as the critical component of the RTCA. Since the main problem of RTCA is the increment of tags due to past path problems, fetch bandwidth and hit rate can have a direct impact consequently. The focus of the results lies on the relation between trace cache performance and different RTCA design space.

In the few related researches done on RTCA, models of the trace cache use traces of the size of 4 with a max branch of 1. The largest advantage of the trace cache is to widen the frontend fetch bandwidth by linking basic blocks after branches. With such simulation models chose, the impact effect of RTCA on the trace cache can be hard to observe. Simulation model used in this thesis enlarges trace size to 16 instructions and 3 max branches per trace. We discuss the impact that RTCA has on high fetch bandwidth frontends. Some of the main design spaces of the RTCA this thesis include are listed below.

5.1 Trace Filling Logic

The filling logic decides when a trace filling should stop and be stored inside the trace cache. Just as the conventional trace cache, the filling logic in this thesis stops

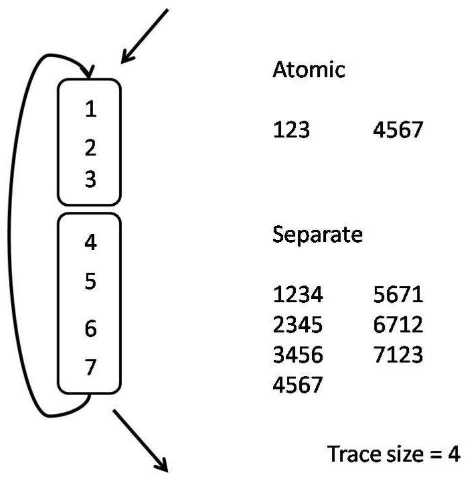


Figure 5.1: Traces Constructed by Using Different Filling Logic

a filling when the trace is full or encounters an indirect branch. Stopping on indirect branches simplifies tag comparison since these instructions have infinite possible target addresses. In addition, the predictor can only predict one target address at a time.

Another important issue in the filling logic is to fill in atomic or separate[7]. Filling in atomic means that the basic unit of filling is a basic block. In other words, if the empty space inside the current filling trace is smaller than the filling block, the trace will be stored first. Fig5.1 shows this difference when assuming a trace size of four. When filling in atomic, after the first basic block of 3 instructions are written in the trace fill buffer, the next block cannot be filled continuously since there is no more space for a block of 4. In the case of a separate fill logic, the fill buffer is stored to the fourth instruction and then written in the trace cache. Both have their own merits and demerits which will be discussed later on.

5.2 Index Selection

The most straightforward way to construct an index is by using the fetch address alone. This is the most common case in caches. However, in the trace cache, a single fetch address can lead to different traces. By adding branch outcomes of the trace into the index, path associativity can be provided. Path associativity allows the traces that start from the same fetch address be spread to multiple sets in the cache which lowers the conflict miss.

In the related researches, an additional $L_{\min}$ parameter was used to further decrease the conflict miss by storing paths only when their lengths are larger than $L_{\min}$. Surprisingly, there were no results regarding to normal indexing when $L_{\min}$ is not involved. Although $L_{\min}$ decreases conflict miss because less paths compete the same cache block, it also needs additional path confirmation for those that have smaller path lengths than $L_{\min}$. We consider the effects that $L_{\min}$ can cause and whether or not this indexing method is suitable.

5.3 Logical Register Reference

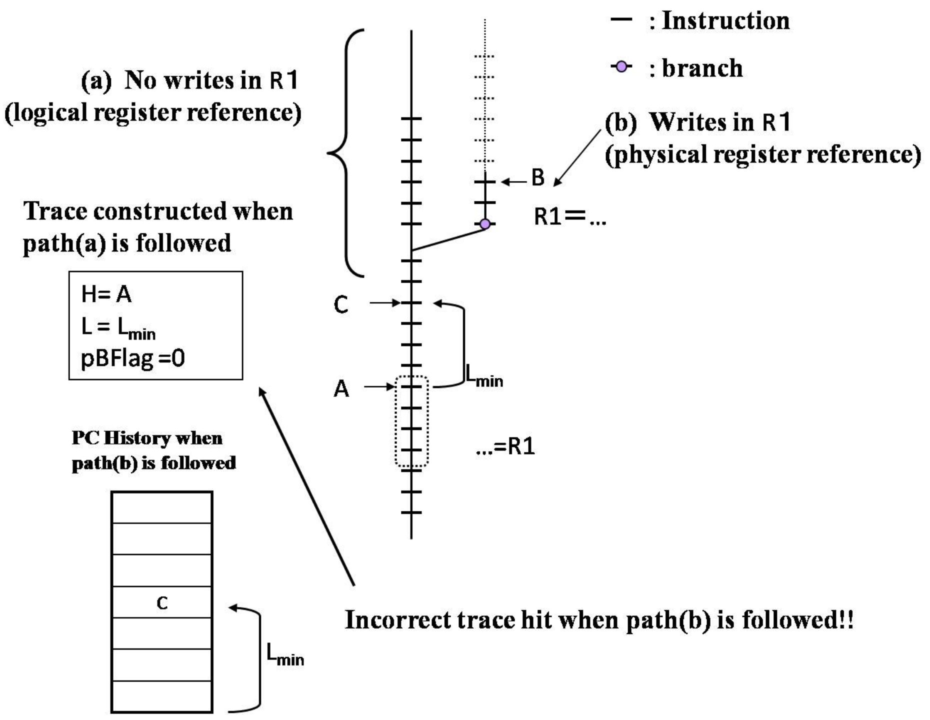
Previously, explanation on when to reference to logical registers have been made. In short, operands that are not in the [-n] form reference from logical registers while operands in [-n] form look inside the physical register buffer. A new issue may arise according to this reference method. Consider Fig5.2 which will illustrate this.

Assume that for the first time the left path in Fig5.2 has been taken. After this path is taken, the corresponding trace will be constructed and stored inside the trace cache. The path length L will be set to $L_{\min}$ since it is the longest reference distance inside the window. On the other hand, since the distance to R1 is larger than the window size, The fourth instruction that uses it as an operand does not turn it to the [-n] form.

Next we consider the situation that the right path is taken and the trace is already stored inside the trace cache. After a trace cache hit is determined and the fourth instruction is fetched, it will attempt to grab the value of R1 from the logical register. However, the distance to R1 on this taken path can possibly be shorter than the window size. Hence, path length adjustments are required in order to maintain correctness.

The RTCA prevents cases like this to happen by adjusting path length to the window size whenever an instruction inside the trace references to a logical register while construction. As a result, when instructions inside the trace are not in the [-n] form, the whole path beneath the window size needs to be checked so the above case does not occur.

This thesis observes this impact due to the path length adjustment by adding additional tags to the trace. These tags should show which registers are referenced as logical registers by the trace. Furthermore, the position of that instruction inside the trace which references it should also be known. Fig5.3 shows how the simulation



— : Instruction
: branch
(a) No writes in R1
(logical register reference)
(b) Writes in R1
(physical register reference)
B
R1=...
Trace constructed when path(a) is followed
H= A
L = L_{min}
pBFlag =0
C
A
L_{min}
...=R1
PC History when path(b) is followed
C
L_{min}
Incorrect trace hit when path(b) is followed!!

Figure 5.2: Issues Caused by Logical Register Reference

model in this thesis is performed.

Two sets of counter arrays of 32 entries are prepared in advance. These counters represent the distance of the fetch address to a register when it acted as a destination operand for the last time. Each entry represents a logical register and one set is for integer registers while the other is for floating point registers. When a register is a destination operand inside the fetch instruction, its corresponding entry is set to 0. Every time an instruction is fetched, the counters are added by 1 until they reach the window size.

When tag comparison is conducted, the register tags are used to determine which registers are referenced as logical registers in the trace. The corresponding entry inside the counter arrays is then access to calculate the distance from the instruction to the register. Trace cache hit occurs only when this distance is larger than the window size.

Many researches on various design spaces in the conventional trace cache architecture have been done. Since the RTCA uses the trace cache as its base, efforts on improving the conventional trace cache can also impact the RTCA. The simulation in this thesis mainly concerns only the above design spaces listed.

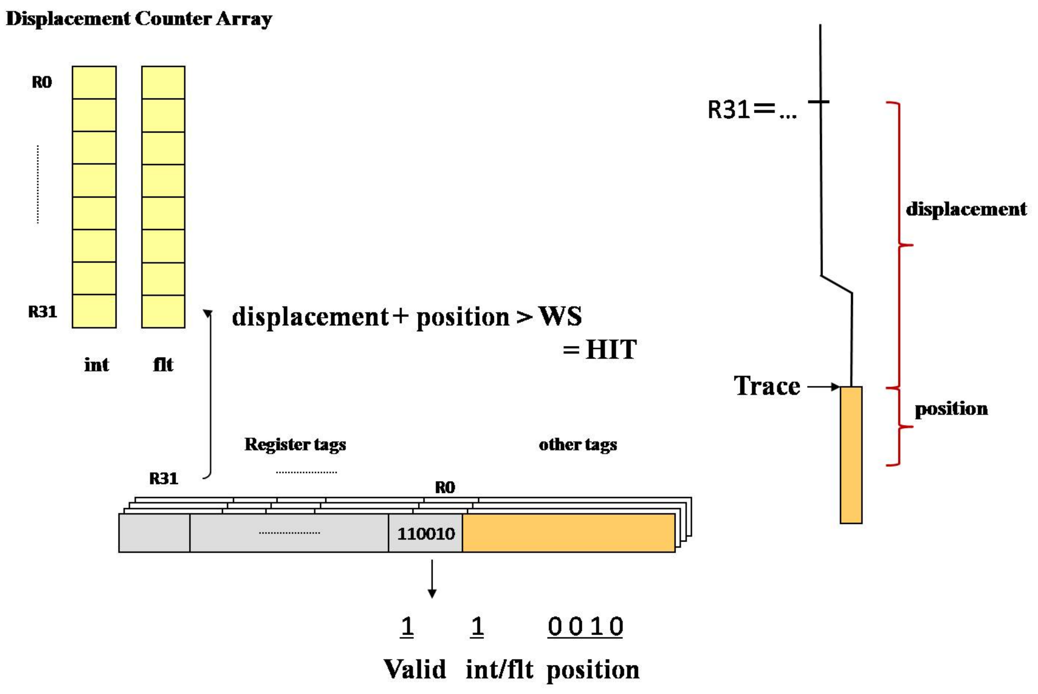


Figure 5.3: Simulation Model using Register Tags

Chapter 6

Simulation and Results

6.1 Simulation Environment

The simulator used in this thesis is the *Onikiri* simulator developed by our lab[14]. This simulator has various advantages compared to the most widely used SimpleScalar simulator[1]. SPEC CINT2000 benchmarks with ref input sets were used. INT benchmarks are chose over FP is because that FP have more simple control flows and properties of the trace cache can become harder to observe. 100M instructions were executed after 1G instructions were skipped in the beginning. Table6.1 shows the parameters used in the base model simulation. Traces can consist up to 16 instructions and 3 branches. The correlated multiple predictor used is described in section3.4 . The following show other assumed characteristic of the trace cache used in our simulation.

- **Partial Fetching**

  The correlated multiple predictor predicts three branch outcomes when fetching inside the trace cache. In some occasions, only a portion of the predicted outcomes is accurate. In conventional methods, the whole trace is flushed if there are any mispredictions in the branch outcomes. With partial fetching, fetching ends only when the mispredicted outcomes is encountered. For instance, suppose a case where only the first branch prediction is right. The first two blocks will be fetched and flushing starts from the third. Fig6.1 shows this case. Partial fetching is assumed in all the simulations in the simulations covered in this thesis.

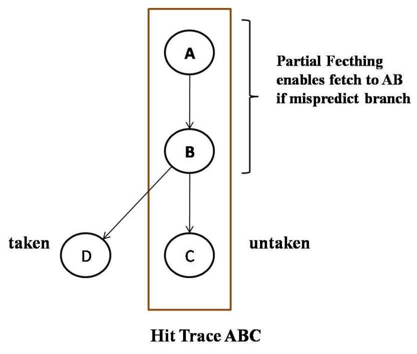


Figure 6.1: Partial Fetching

- **Indirect Branches**

  Indirect branches can make path determination more complex when comparing path tags. To fully express the past path, additional tags are needed every time an indirect branch shows up. This thesis simplifies the process by fully expressing every target address of indirect branches inside the tags. Related research[5] show that a number of 2 target addresses inside the tag can cover most of the cases. Note that while too many tags used in represent target addresses can cause unnecessary overhead, insufficient target address tags can lead to pipeline stalls that occur in the frontend when tag comparison is done.

ISA Benchmark Fetch width Window size L1 Instruction Cache	Alfa SPEC CINT2000 16 32 entry 32KB 4way
Branch Prediction	BTB: 2K entry Pht: 32 entry 2bit counter Multi pred: 4 Pht Global history: 10 bit RAS: 8 entry
Trace Cache	1K entry, 4way Entry size: 16 instructions Max branch in trace: 3

Table 6.1: Simulation Parameters

6.2 Results and Analysis

6.2.1 Selection of Filling Logic

Before the RTCA simulations, we first observe the conventional trace cache and decide which filling logic is better to use. In particular, we analyze the differences between atomic and separate filling logic.

In atomic filling, filling only occurs when there is room for the next fetch block. This has the advantage of creating less traces and the fetch addresses of traces are aligned well. However, the usage of trace cache capacity is not as efficient as separate filling. Separate filling unrolls loops and maximizes the cache space usage. This can also lead to higher fetch bandwidth. Fig6.2 shows the simulation result of the two methods.

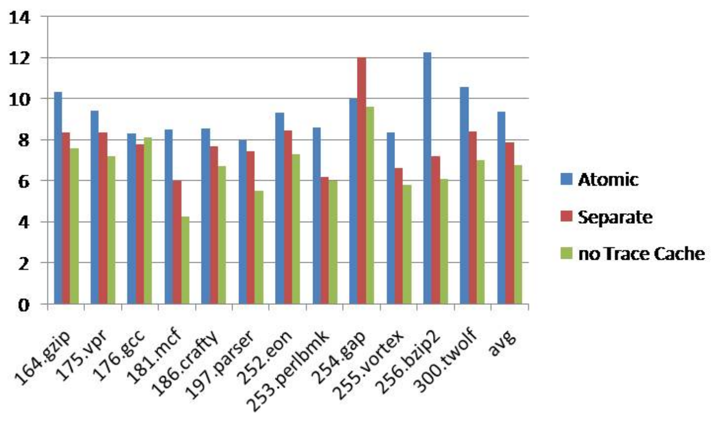


Figure 6.2: Fetch Bandwidth from Atomic/Separate Filling Logic

Results show that atomic does better than separate. This implies that trace fetch address alignment is a more dominant factor. From Fig5.1 we can see that although separate filling can provided the maximum instructions on a trace hit, it needs to construct 7 different traces in advance. In other words, atomic filling can hit on the second time execution of the loop while separate starts hitting on the fifth time.

Separate filling can cause redundancy caused by unrolling loops. One way to improve this is to stop the trace filling when the loop ends. We call this method trace stopping. An end of a loop is detected by a backward branch in the program. From the example in Fig5.1, we can see that by using trace stopping and stop the filling when instruction 7 is encountered, the total trace numbers decrease from 7 to 2 (trace1234 and 567). This may also effect the fetch bandwidth. However, hit rate can possibly be improved. Fetch bandwidth of separate filling using trace stopping and not are shown in Fig6.3.

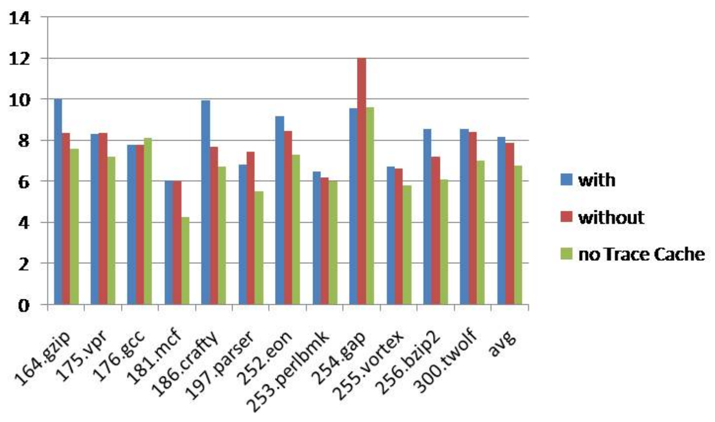


Figure 6.3: Fetch Bandwidth with/without Trace Stopping

From the above results we can see that using trace stopping can slightly improve the separate filling logic. On the other hand, an atomic filling logic overall does better than the separate filling logic whether or not trace stopping is used. Results in the rest of this thesis all assume an atomic logic.

6.2.2 Effects of $L_{\min}$ Index Method

The simplest way to construct an index is by using partial bits of the fetching address. In order to reduce conflict miss and provide path associativity, branch flags are added in the index. This enables different traces that start with the same address to be stored in different sets in the cache.

Expressing indirect branches in past path can be complex since the numbers included may vary by traces. To simplify this complexity in our simulation, we assume that the target addresses inside the tags can fully express the path. Relate research show that a number of 2 target address can be sufficient to obtain similar results[5]. Insufficient target address tags can cause addition stalls in the pipeline when referencing values over the indirect branch. Fig6.4 shows the results of RTCA when using the fetch address XOR branch flags as the index.

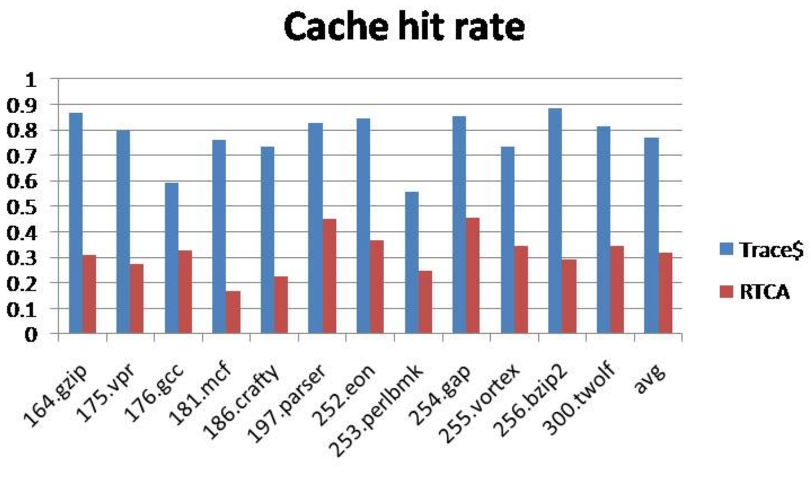


Figure 6.4: Conventional Trace Cache and RTCA Trace Cache Hit rate

Results show that there is a 45.5% degradation on trace cache hit rate. This degradation is mainly caused because in RTCA traces not only have to match their fetch address and branch outcomes, but also have to share the same past path followed.

In related research[5], the $L_{\min}$ method was proposed to reduce conflict miss caused by past path expression. However, less observation was done on this method. We discuss whether or not the $L_{\min}$ method is suitable and its effects. Fig6.5 show results of the $L_{\min}$ methods.

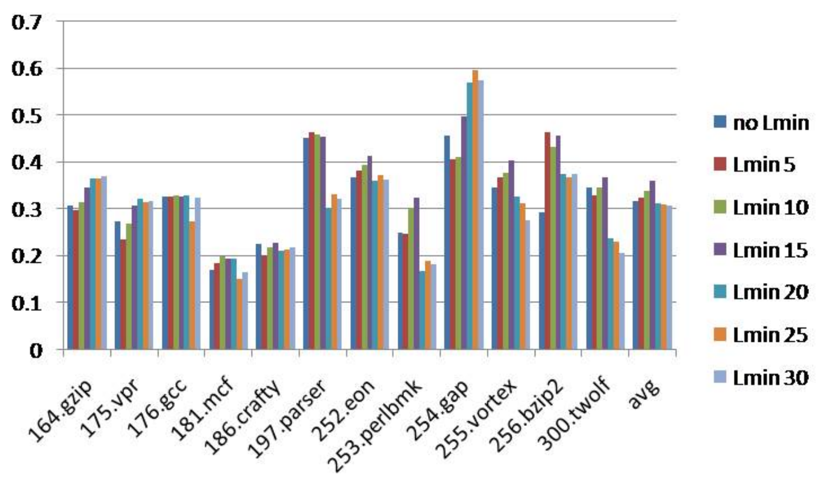


Figure 6.5: RTCA Trace Cache Hit Rate from Using Different $L_{\min}$ Index Method

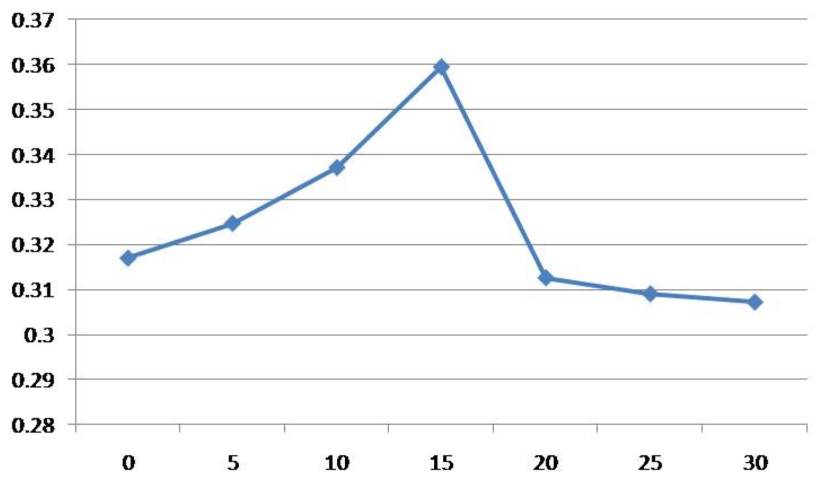


Figure 6.6: Average Hit Rate Change due to Different Values of $L_{\min}$ Indexing

The $L_{\min}$ method tends to reduce conflict miss by storing a single trace that followed different paths into different sets inside the trace cache. The ideal case is that all path lengths are larger than the $L_{\min}$ value so that these traces can be stored properly. However, if path lengths are smaller than the $L_{\min}$ value, the whole path to $L_{\min}$ has to be confirmed. This factor can also decrease the trace cache hit rate. Fig6.6 summarizes the relation between $L_{\min}$ and the average hit rate of the benchmarks.

Results show that the $L_{\min}$ index methods performs well around a value of 15, which is around half the size of the window. Path Lengths that are too short can result in similar problems with the none $L_{\min}$ method. Fig6.7 shows that if $L_{\min}$ is chose too short, the corresponding $L_{\min}$ index can frequently become the same value and still cause conflicts in the cache. On the other hand, if $L_{\min}$ values are chose too large, most path lengths become shorter than $L_{\min}$ and the confirmation of the additional path can also lower the hit rate. This is why the cache hit rate tends to fall down on higher $L_{\min}$ values. Picking the right value for $L_{\min}$ can be tricky since different portions of the program may have different characteristics. Results show that a length around the middle of the window size can overall do better.

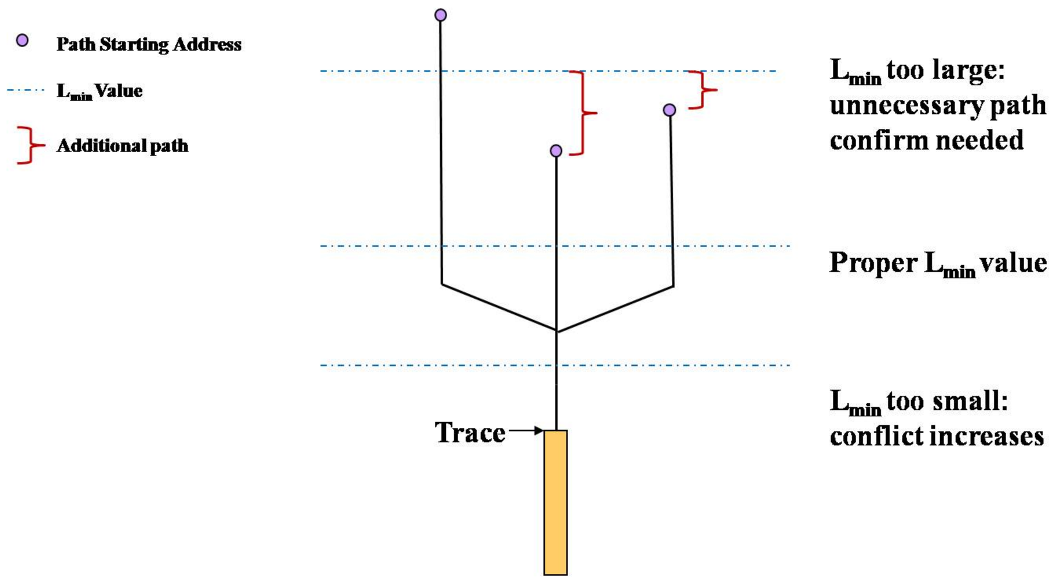

Path Starting Address
L_min Value
Additional path
L_min too large: unnecessary path confirm needed
Proper L_min value
L_min too small: conflict increases
Trace

Figure 6.7: Relation between $L_{\min}$ values and Path Lengths

6.2.3 Effects of Path Length Adjustment

In the last chapter, explanation on path adjustment due to logical register reference has been made. In specific, the path length has to adjust to the window size if instructions inside the trace have referenced to a logical register before. This section attempts to observe the effects that this adjustment can bring about.

The simulation model was covered in section5.3. Register tags and counters are added to check if the referenced logical register falls in the window upon fetching. The hardware overhead of the counter array is $2 \times 32entry \times 5bits = 40bytes$. Register tags each contain 6 bits which $\log_2 16 = 4bits$ are used to represent the instruction position in the trace, 1 bit to show if it is an integer or floating instruction, and 1 valid bit to depict if the register is used as an logical register in the trace. Hence $32registers \times 6 = 192bits$ are additional added to each entry of the trace cache.

Fig6.8 compares the hit rate of using path adjustment to register tag checking when encountering logical registers. $L_{\min}$ index method is not used.

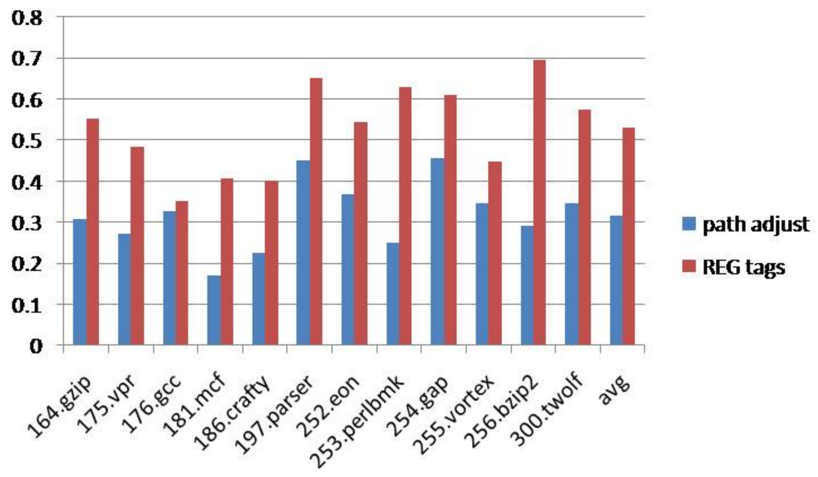


Figure 6.8: Trace Cache Hit Rate When using Path Adjustment and Register Tags

Fig6.8 shows a 21.2% increase in trace cache hit rate. As one can imagine, most traces contain instructions that reference to logical registers. This makes path length adjustment happen frequently. Since the extended path lengths are all set to the size of the window, the constant confirmation of unnecessary additional paths lead to the fall of trace cache hit rate.

Further more, we observe the effects that the $L_{\min}$ index method can have on this method that uses register tags to avoid path adjustment. Fig6.9 shows the average hit rate change of the benchmarks according to the $L_{\min}$ values.

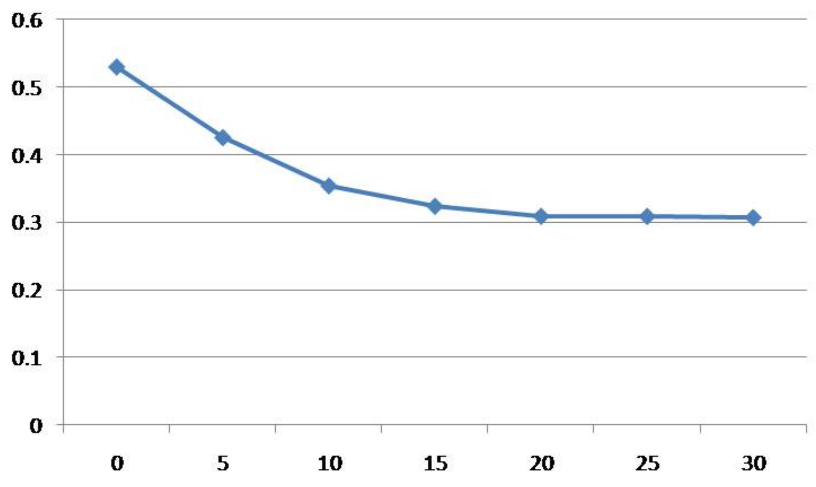


Figure 6.9: Average Hit Rate Change due to Different Values of $L_{\min}$ Indexing with Register Tags

From the above graph we can see that the $L_{\min}$ method does not work as well as normal indexing methods. In the RTCA that uses path adjustment, large portions of path lengths are extended. Path lengths shorter than $L_{\min}$ are all extended longer than $L_{\min}$ and have a better probability to obtain the proper $L_{\min}$ index. However this is not the case when using register tags. When path adjustment is not necessary, the path lengths can vary according to the executed program. Chances that the value of $L_{\min}$ to lie in the proper zone become harder. In addition, the value of $L_{\min}$ is set constant while different parts of the program may require different proper $L_{\min}$ values.

The trend in Fig6.9 show that additional path confirmation can have a larger impact on trace cache hit rate compared to the benefits Lmin index may have. From another point of view, different way associativity of the cache and the path associativity provide enough margin for conflict miss.

Chapter 7

Conclusion

Out-of-order superscalar processors use register renaming to analyze dependency between instructions in order to exploit the most ILP. The register renaming stage is done at a rather high cost in terms of RMT issus. Multi-port structure increases RMT area exponentially and frequent access lead to large energy consumption. This makes register renaming costly and hard to widen.

The RTCA was proposed in order to solve such RMT problems. In RTCA, instructions are translated to their unique [-n] form which explicitly shows the displacement between dependent instructions. These instructions are stored inside the trace cache for further usage. Since the dependency is explicit, register renaming can also be eliminated and obtain a shorter pipeline depth. Nevertheless, RTCA can have its own problems. The main issue it suffers from is degradation in trace cache hit rate. This is caused mainly because the translation is path dependent and additional path tags are required to be attached. The criterion for a hit not only relies on fetch address and branch prediction, but also the followed path.

Although the RTCA is an original architecture, less related researches have been conducted. In this thesis, we attempt to explore the characteristics of the RTCA in different design space. First we show that the path dependence of the RTCA can cause a 45.5% drop down in trace cache hit rate compared to the conventional trace cache architecture. Then the adequacy of the $L_{\min}$ index method proposed was verified. Results show that with the proper $L_{\min}$ value chose, an increase of 13.3% in trace cache hit rate can be achieved. This proper value of $L_{\min}$ should be around the middle value of the window size.

In addition to different indexing methods, we observe the impact that path adjustment can bring about. RTCA adjusts path lengths to the window size in order to solve problems that logical register referencing may cause. The simulation model in this thesis uses additional register tags to avoid this path adjustment. Absence of path adjustment can bring a 21.2% increase in trace cache hit rate. Results also show that the $L_{\min}$ indexing method does not work as well when register tags are applied.

The results in this thesis quantify some of the characteristic RTCA have in different design spaces. We address the issues that the path adjustment in RTCA may not be an optimal way to solve problems brought up by logical register reference. In additional, the effects of $L_{\min}$ indexing method on performance may vary and determining the right $L_{\min}$ value may be a difficult task. The results also show that improvement in different design space can lead the performance of RTCA to approach those of the conventional trace cache. The focus behind this thesis attempts to further improve the RTCA by exploring deeper in various design spaces.

Bibliography

[1] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin-Madison Computer Sciences Department, 1997.

[2] J.E.Smith E.Rotenberg, S.Bennet. Trace cache: A low latency approach to high bandwidth instruction fetching. In *29th International Symposium on Microarchitecture (MICRO)*, pp. 24–35, 1996.

[3] J.E.Smith E.Rotenberg, S.Bennett. A trace cache microarchitecture and evaluation. In *Computers, IEEE Transactions on*, pp. 111–120, 1999.

[4] G. Hinton, D.Sager, M.Upton, D.Boggs, D.Carmean, A.Kyker, and P.Roussel. The microarchitecture of the pentium4 processor. In *Intel Technology Journal*, Vol. 5, 2001.

[5] Hironori Ichibayashi, Ryota Shioya, Hidetsugu Irie, Masahiro Goshima, and Shuichi Sakai. Anit-dualflow architecture. In *SACSIS*, 2008.

[6] James E. Smith Quinn Jacobson, Eric Rotenberg. Path-based next trace prediction. In *30th International Symposium on Microarchitecture (MICRO)*, p. 14, 1997.

[7] Yale N. Patt Sanjay Jeram Patel, Marius Evers. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th annual international symposium on Computer architecture (ISCA'98)*, pp. 262–271, June 1998.

[8] The Standard Performance Evaluation Corporation. *SPEC CPU2000 suite http://www.spec.org/cpu2000/.*

[9] The Standard Performance Evaluation Corporation. *SPEC CPU2006 suite http://www.spec.org/cpu2006/.*

[10] Michael Sung. Design of trace caches for high bandwidth instruction fetching, May 1998.

[11] Y. Tatsumi and H. Mattausch. Fast quadratic increase of multiport-storage-cell area with port number. In *Electronics Letters*, Vol. 35, pp. 2185–2187, December 1999.

[12] 五島正裕. Out-of-order ILP プロセッサにおける命令スケジューリングの高速化の研究. PhD thesis, Kyoto Univeristy, 2004.

[13] 五島正裕, グェンハイハー, 縣亮慶, 森眞一郎, 富田眞治. Dualflow アーキテクチャの提案. 並列処理シンポジウム JSPP 2000, pp. 197–204, 2000.

[14] 渡辺憲一. The design and implementation of processor simulator ”onikiri2”, 2008.

Acknowledgement

I would like to express my gratitude to numerous people. First of all, my supervisor Professor Sakai Shyuichi, who has helped me since before enrollment in Tokyo University. The efforts he has made to maintain such wonderful research environment is substantial. His advice and encouragement has always been help to me for the past two years. Regardless of his profound achievement in science research, his polite and modest attitude has always been a role model to me and the whole lab.

I truly appreciate Associate Professor Masahiro Goshima, for his help and opinion on my research. The unique thoughts he has stimulated different minds and brought much innovation to me. His philosophy of teaching students and further guiding them to become independent individuals has truly helped me not just as a student but also as a person.

Special thanks to Shoya Ryota and Horio Kazuo, who have constantly helped my on my research and taught me. Many thanks to Kunbo Li, who gave me lots of support when I first arrived in the lab. Also appreciations to the secretaries, Ms.Harumi Yagihara, who has helped me a lot on my application and to Ms.Tomoyo Ise, Ms.Tamaki Hasebe for their contribution to the lab.

Appreciations to the Panasonic Scholarship Co., which made all this possible. The financial support not only colored my life in Japan, but enabled me to experience the wonders of the Japanese Culture.

Last but not least, I would like to give my greatest gratitude to my parents, Jay Wang and Chiu-lan Chen. If not for your support, faith, and love, I would not have become who I am. Also thanks to my brother, Jesse, who has been a best friend and companion throughout the years.