Low-overhead security-tagged architecture

低オーバヘッド セキュリティ・タグ・アーキテクチャ

Supervisor: Professor Shuichi Sakai

Daewung Kim

DEPARTMENT OF INFORMATION SCIENCE AND TECHNOLOGY,
THE UNIVERSITY OF TOKYO

February 2009

abstract

A tagged architecture is a system that applies tags on data, recently used in the field of information security.

The previous studies using tagged architecture mostly focused on how to utilize tags, not how the tags are implemented. A naive implementation of tags simply adds a tag field to every byte of the cache and the memory. Such technique, however, results in a huge hardware overhead and performance degradation, as well as is unable to support variable-length tags.

This thesis proposes a low-overhead security-tagged architecture that supports variable-length tags. We achieve our goal by separating tag and data completely. The proposal technique is composed of two parts, separation in storage and separation in execution.

First, in the separation of storage, we exploiting some properties of tag, the non-uniformity and the locality of reference. Our design includes a use of uniquely designed multi-level table and various cache-like structures, all contributing to exploit these properties.

Second, in the separation of execution, we suggest to propagate tags after the completion of execution of data. This allows to have dedicated tag register file and L1 tag cache, so that prevent to increase in the access latency for register and cache.

Under simulation, our method was able to reduce the memory overhead to 3.48% of the naive implementation, and 4.96% of IPC degradation compared with conventional computer system, in addition to supporting variable-length tag.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
  - 1.1.1 Early tagged architecture . . . 1
  - 1.1.2 Recent tagged architecture . . . 2
- 1.2 Research Contributions . . . 2
- 1.3 Thesis Organization . . . 3

2 Related Work **4**

3 Low-overhead security-tagged architecture **6**
- 3.1 Overview . . . 6
- 3.2 Separation in storage of tag . . . 8
  - 3.2.1 Tag Table . . . 10
  - 3.2.2 Tag Management Unit . . . 16
- 3.3 Separation in execution of tag . . . 21
  - 3.3.1 Pipeline structure . . . 22
  - 3.3.2 Tag Propagation Unit(TPU) . . . 24
  - 3.3.3 Tag register file . . . 25
  - 3.3.4 Tag cache . . . 25

4 Evaluation **28**
- 4.1 Evaluation Environment . . . 28
- 4.2 Evaluation item . . . 29
- 4.3 Separation in storage . . . 30
  - 4.3.1 Memory overhead . . . 30
  - 4.3.2 Time overhead . . . 32
- 4.4 Seperation in execution . . . 33
  - 4.4.1 Time overhead . . . 33

5 Conclusion **36**

Bibliography **38**

Publications **40**

List of Figures

3.1 Block diagram of proposal technique 7
3.2 Block diagram of *the separation in storage of tag* 9
3.3 Tag table and virtual address translation 10
3.4 Composition of Tag Table Entry 11
3.5 Composition of Extended Tag Table Entry 13
3.6 Composition of Tag Line Vector 13
3.7 Composition of Tag Line 14
3.8 Contraction Algorithm 16
3.9 Contraction of fifth-level table 17
3.10 Composition of pointer cache 17
3.11 Read algorithm 18
3.12 Write algorithm 19
3.13 Block diagram of *the separation in execution of tag* 22
3.14 Pipeline structure of the proposal technique 23
3.15 Pipeline structure including the memory instruction 24
3.16 Block diagram of the TPU 24
3.17 Composition of the L1 tag cache 26

4.1 Memory overhead for various length-tag (with contraction) 30
4.2 Memory overhead for various length-tag (without contraction) . . . 31
4.3 Memory overhead for various length-tag (contraction vs. without contraction) 32
4.4 IPC degradaion by length of tag 33
4.5 IPC degradaion by contraction 34
4.6 IPC evaluation of separation in execution 35

List of Tables

3.1 Linear address space covered by Tag field 12
3.2 Initial value of each table entry 15
3.3 Cache access time in ns for various cache configurations in a 50nm technology[1](extracted part from original table) 26

4.1 Architectural parameters 28
4.2 Tag propagation rules 29

Chapter 1

Introduction

Computer security is a critical problem in our recent society, because many implements for maintaining our life such as financial system and communication system, depend on the computer system. Over the years, significant development efforts for the computer security including the information security were done.

A tagged architecture[5][6] is a system that applies tags on data, recently used to research for the information security, which includes preventing information leakage, detecting malicious attacks and so on. The previous studies using tagged architecture mostly focused on how to utilize tags, not how the tags are implemented. Such technique, however, results in a huge hardware overhead and performance degradation. In this thesis, we present the low-overhead security-tagged architecture. We believe the low-overhead security-tagged architecture will improve the research for information security.

The rest of this chapter is organized as follows. First, we describe the tagged architecture which is the background of this thesis. Then, we describe the research contributions. Finally, we describe the organization of this thesis.

1.1 Background

1.1.1 Early tagged architecture

In the 70's and 80's, tagged architectures were mainly used for identifying the data types. By checking tags, the processors can automatically identify and convert the data types at run-time. For example, Burroughs B6500[7] which is famous as stack machine, incorporated 3 tag bits on each memory word. In this machine, the first 3

bits of the 51 bit words as tag bits, which serve to identify the arithmetic types of the data. Another example, Rice Computer R-2[4] employed 3 bit tags for every 62 bit word. They used tag for identifying types of numeric operands and information used by the operating system.

In this era, tagged architectures are claimed to simplify hardware design and facilitate software development. For example, processor could manipulate matrices made up of variable-length rows at once. Thus additional time for decoding multiple instructions were not needed. Also, pre-defined data types by tagging, could simplify the generation of code for a compiler, because the operands have their own semantics.

1.1.2 Recent tagged architecture

In the recent years, tagged architectures are used in the field of information security, which includes preventing information leakage, detecting malicious attacks and so on. The basic concept of tagging that is *self-identifying data* is the same as early tagged architecture, but the meaning of tag is changed from the data types to the security informations. The most significant change of recent tagged architectures is to propagate tags. Recently established method called, *dynamic information flow tracking(DIFT)* is used for detecting a broad range of malicious attacks such as code injection attack. The main idea of DIFT is to tag input data and track its propagation based on the information flow.

For exmple, *RIFLE* uses tags to identify between the data that must be protected and not, and propagate tags based on the information flow. Some data that must be protected are personal data and copyrighted works, by avoiding leakages and illegal copies, respectively.

This thesis focuses on recent tagged architecture, not early tagged architecture. For preventing confusion, we use the term *security-tagged architecture* as recent tagged architecture, from now on.

1.2 Research Contributions

This thesis presents the low-overhead security-tagged architecture. Despite many previous studies, the technique to compactly store tags have rarely been the target of research. Most previous studies assume a naive implementation in which the tags are always stored with the data in pairs, all the way from register file to main

memory. This means if a 1-bit tag is added to 4-byte words, each word then becomes virtually 33 bits in size.

The previous studies also leave the use of variable-length tag totally out of scope. The previous techniques can only use fixed-size tags and can not change their size from one to another.

The goal of this thesis is to support variable-length tag with low-overhead. We achieve our goals by separating tag and data completly. This is composed of following two parts :

- *Separation in storage of tag* : We exploiting some properties of tag, the non-uniformity and the locality of reference. Our design includes a use of uniquely designed multi-level table and various cache-like structures, all contributing to exploit these properties.

- *Separation in execution of tag* : We suggest to propagate tags after the completion of execution of data. This allows to have dedicated tag register file and L1 tag cache, so that prevent to increase in the access latency for register and cache.

1.3 Thesis Organization

The rest of the paper is organized as follows. In Chapter 2, we review related work on security-tagged architecture. Chapter 3 will give details of our proposal, the separation in storage and the separation in execution. In Chapter 4, we evaluate the performace overhead. In Chapter 5, we state the conclusion.

Chapter 2

Related Work

Previous studies adopting tagged architecture focus on how to utilze tags, not how the tags are implemented. This Chapter describes some previous studies adopting tagged architecture and their approaches to the tag implementation.

RIFLE[11] is an architectural framework to prevent information leaks by tracking the flow of data which must be protected. RIFLE uses the tagged architecture as a means of information-flow tracking. RIFLE mainly focuses on the method of tracking information flow using tags, and the implementation of the tags used is not touched in the paper.

Minos[3] is a microarchitecture that implements Biba's low-water-mark integrity policy on individual words of data. Minos applies a 1-bit tag which represents the data integrity, to every words of the memory. The Minos implements tags in a naive way, that is, the tags are coupled with the data all the way from register file to main memory.

Dynamic Infomation Flow Tracking by Suh et al[10] is an architecture to detect both control and non-control attacks. Their method applies a 1-bit tag to every words of the memory for identifying spurious information flows. The OS marks the tags spurious for potentially malicious data, and the processor tracks its flow by propagating the tags along with their operation. Their method decouples the storage of tags from instructions and data throughout the memory hierarchy. It uses L1 and L2 tag caches, which are the dedicated caches for tags, and on the main memory, tags are stored in their dedicated area seperate from the data. Finding a tag stored separately from the data requires a special address translation, and their method uses a tag TLB for this purpose. Their method however, doesn't describe the detail of tag implementation, such as the structure of the tag storage and the tag TLB, and so on.

Mondrian Memory Protection(MMP)[12] is an architecture that allows multiple protection domains to control access permissions on individual words of data. The MMP applies a 2-bit permission field to every words of the memory. The permission field acts somewhat similarly to tags in the tagged architectures, though the term is different. As opposed to tags, the permission does not propagate, because it is not for tracking information flow. Unlike the naive implementation of tags, the MMP separates the permission and the data in the memory space by introducing a multi-level permissions table (MLTP) as a storage of the permission bits. The MMP also caches the MLTP entries the same way as a TLB to improve the table look-up speed. The basic concept of the MLTP is similar to the tag table of our system, but it is not suitable for a system that changes the its contents frequently. This is due to that the MMP design assumes the permission modification occurs more frequently than page table modifications, but less frequently then the tag propagation.

Overall, the previous studies do not pay great attention to the implementation of tags. RIFLE does not touch it in the paper, and Minos simply adopts a naive implentation. Dynamic information flow tracking by Suh et al uses a technique that exploits non-uniformity to an extent. Their method achieves this by supporting the multi-granularity of tag mapping, but it can not dynamically adjust the tag storage to its minimal size, which is possible by our technique of contraction, explained in Section 3.2.1. Moreover, their implementation is not described in details. The MMP takes a close approach to us for reducing overhead, but their technique is for a different purpose, and is not suitable for a system that changes the its contents frequently.

Chapter 3

Low-overhead security-tagged architecture

This chapter presents the composition and the operation of low-overhead security-tagged architecture. The goal of this thesis is to support variable-length tag with low-overhead. We achieve our goals by separating tag and data completly. This is composed of two parts, *separation in storage* and *separation in execution*. We first give an overview of the proposal technique. Then, we describe the separation in storage of tag in detail. Finally, we present the separation in execution of tag in detail.

3.1 Overview

The disadvantages of the conventional naive tagged architecture what we discussed in chapter 2 are caused by the tags combined with data. In this thesis, we suggest separating tag and data completly. Figure 3.1 shows the block diagram of entire proposal technique.

This figure consists of upper and lower two parts, and calls of each *the data system* and *the tag system*. The data system can be considered the same as existing non-tagged architecture. In this figure, the processing unit of the processor is distributed to the right, and the memory hierarchy that consists of register file, Level 1 cache(L1$), Level 2 cache(L2$), and the main memory is depicted toward the left.

The tag system is divided into two parts. One is storing tags corresponding to memory hierarchy in data system. This technique is called *Separation in storage of tag*(lower left side of the figure). Another is propagating tags corresponding

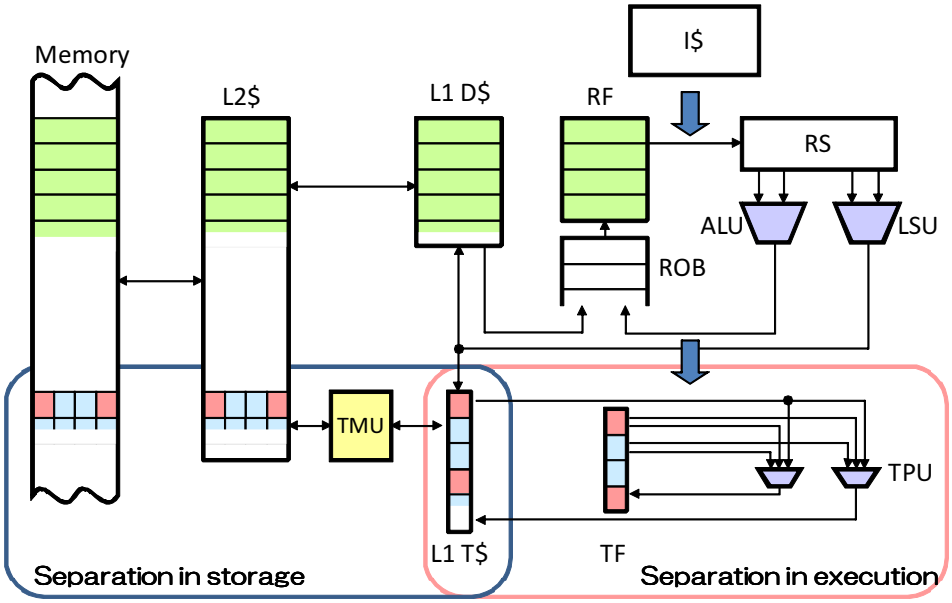

Figure 3.1: Block diagram of proposal technique

to executing instructions of data system. This technique is called *Separation in execution of tag*(lower right side of the figure). Note that the tag system is completly separated from the data system by these two techniques.

In the following two sections, we describe these two techinques, separation in storage of tag and separation in execution of tag in more detail.

3.2 Separation in storage of tag

This section describes the separation in storage of tag. The proposal technique exploits some properties of tag, the non-uniformity and the locality of reference, in our design. Reviewing, these characters are summarized as follows:

- Non-uniformity : The memory is divided into tag-assigned blocks and non-tag-assigned blocks. Within a tag-assigned block, the tags are likely a same value. Across the blocks, the values may be different. This is because a file, which is a block of data, is likely to have a single tag on it.
- Locality of reference : Tags show the locality of reference just as much as data does. Moreover, since tags are applied to only selected amount of data, they are cached more effectively than data.

Figure 3.2 is a block diagram of our system. The figure shows the placement and the interactions of important components of our design, the tag table and the cache hierarchy.

Tag Table Built on the main memory and also partly cached on the L2 cache is a virtual storage for tags, the tag table. The tag table is characterized by a multi-level structure. It posesses mechanisms for dynamically adjusting to its minimal size and obtaining tags with least number of accesses. These mechanisms reduce the memory and latency overhead of the system.

The different-level table entries are differently designed, and they do more than just point to the next-level tables. The *Tag Line Vector* field of an entry serves to reduce both the memory and latency overhead.

The table dynamically adjusts to its minimal size by the operation of *expansion* and *contraction*. The contraction operation expoits the non-uniformity of tags, and ensures the memory allocated for tags by a process do not just keep getting larger.

The entry of the bottom-level table, the one that contains the tags themselves, are called a *Tag Line*. It is important that this size matches the cache line size for optimal performance. In the same way, the free list chunk size is matched to the table size, and so on for other parameters.

Cache Hierarchy The cache hierarchy of our system includes a dedicated L1 tag cache and a unified L2 cache for instruction, data and tag, and a *Tag Management Unit(TMU)* placed between them. The tag cache is hardware implemented at level

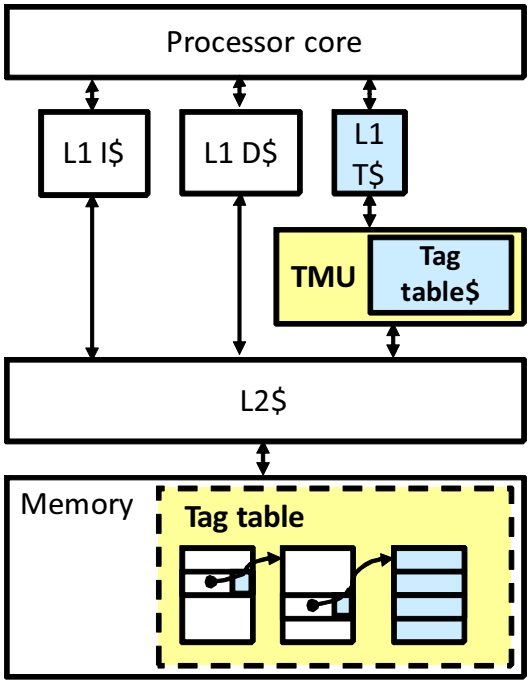

Figure 3.2: Block diagram of *the separation in storage of tag*

1 but not at level 2, for the hardware implementation of large L2 tag cache results in a significant overhead.

The TMU is placed between the L1 tag cache and the virtual tag table on the L2 cache, and serves as a mediator. The virtual tag table on the L2 cache and beyond is accessed by a 64-bit virtual address, and the TMU handles its translation.

Also noticeable in the figure is the *pointer cache* as a part of the TMU. The pointer cache caches the results of the address translation, and reduces the number of access neccessary to obtain a tag.

Our system requires neither an extension of tag field to the memory nor a separate storage for caching tags beyond L2 cache. The tag table and the TMU virtualizes the storage. Therefore, our system significantly reduces the memory overhead from conventional implementations.

The rest of this section gives details of the implementation of the tag table and TMU.

3.2.1 Tag Table

This subsection gives details of the implementation of the tag table.

The tag table is a multi-level table like a page table.The tables at each level has entries containing pointers to the next-level tables, and the tables altogether form a tree, each table as a node and typically pointing to multiple child nodes. The tag table is composed of five levels of tables. Figure 3.3 shows a diagram of the tag table and its virtual address translation.

Our system supports variable-length tag. The tag table shown in Figure 3.3 is for tags that are 1/2 the data size. The length of tag only affects the size of the bottom-level tables, the fifth level. Also note that the tag table in the Figure 3.3 is designed under the assumption that the pages are 8 KB and always aligned on 8 KB boundaries. The same is assumed throughout the papaer.

The tag table is accessed by a 64-bit virtual address. The virtual address is divided into fields as shown in Figure 3.3. The index fields of different levels provide an offset from the base address of the corresponding table.

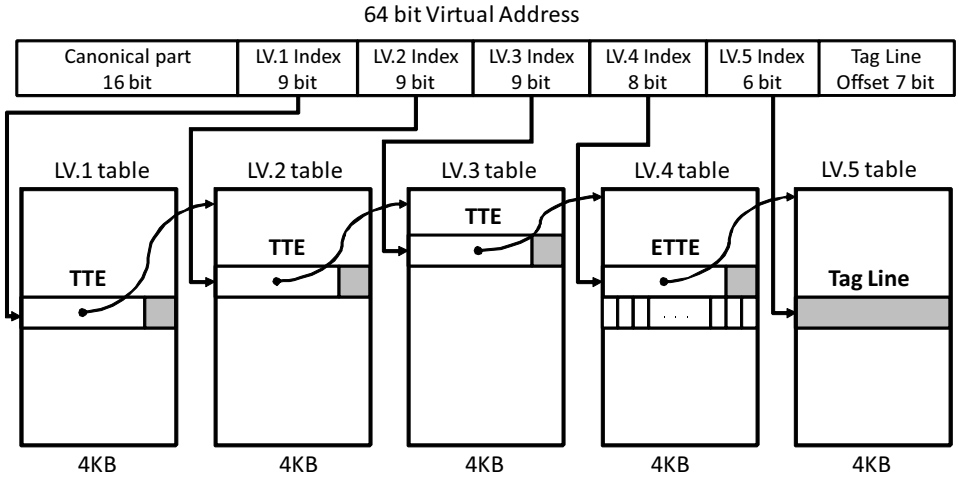

Figure 3.3: Tag table and virtual address translation

Whereas the fifth-level table entries are for storing tags, the first to fourth level table entries are mostly for pointing to the next level tables. These entries, however, contain more than just pointers, specifically, the tags themselves. The entries of the diffrent level tables vary in their structures and are given different names:

- Tag Table Entry(**TTE**): An entry of the first to the third level tables

- Extended Tag Table Entry(**ETTE**): An entry of the fourth level table
- Tag Line(**TL**): An entry of the fifth level table

The tag field on the upper-level table entries allows a tag to be obtained without tracking the pointers down to the bottom-level. This is possible when a same tag is mapped to a significant-size block of memory. In such case, the use of lower-level tables are superfluous, because a same tag will be stored in all of their entries. The tag can instead be represented on a upper-level table entry, on its tag field. This reduces the number of access to the tag table. The existing lower-level tables may actually be freed, and this is done for a specific table level. This operation is called contraction, and reduces the total table size.

The opposite happens when such uniformly tag-mapped block is disturbed. In this case, a new table is allocated. This operation is called expansion.

The following sections will give details of the structures of the different table entries. We also describe the specifics of the free list, and the operation of expansion and contraction.

Structures of Table Entries

Tag Table Entry A Tag Table Entry(**TTE**) is the format of the table entry for the first to the third level tables. It simply provides a pointer to the next level table in most cases. The composition of TTE is given in Figure 3.4.

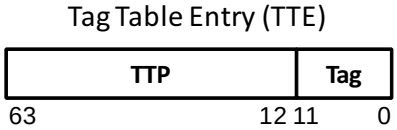

Figure 3.4: Composition of Tag Table Entry

- **Tag Table Pointer(TTP)** The upper 52 bits of the virtual address of the next-level table. The 64-bit virtual address is derived by adding the lower 12 bits, all zeros, to the TTP.
- **Tag** Since each table consisiting the tag table is 4KB and always aligned on 4KB boundaries, the lower 12 bits of the 64-bit address are always 0. Thus the lower bits may be used to store a tag.

Our system may support tags of up to 12 bits in length, for this field is 12 bits. However, we currently choose to support only up to 4 bits, because supporting longer tags calls for multiple table free lists. We describe the reason in Section 3.2.1. When storing tags smaller than 12 bits, the remaining upper bits are filled with 0.

As stated earlier, when a significant-size block of memory is mapped a same tag, the upper-level table entries provide the tag. In such case, the upper-level table entry, or the TTE, does not provide a pointer to the next-level table. Specifically, its TTP field is set to NULL. When TMU at address translation encounters a null TTP, it returns the tag on the Tag field instead of further tacking down pointers. Thus the number of access to the tag table is reduced. When TTP is not NULL, the Tag field is ignored.

The block of tags represented by a single TTE Tag field differs by the table level. A first-level TTE providing tag means that the tags obtained via the whole table hierarchy beyond level 2 is a same value, and thus mapped to a single Tag field. The sizes of linear address space possibly covered by a Tag field are shown in Table 3.1.

Table 3.1: Linear address space covered by Tag field

Table Level	Block Size
4	8 KB
3	2 MB
2	1 GB
1	512 GB

Extended Tag Table Entry The Extended Tag Table Entry(**ETTE**) is the format of the fourth level entries of tag table. The ETTE is composed as in Figure 3.5.

The ETTE is composed of a TTE, which we explained above, and a Tag Line Vector(**TLV**). The TLV is a bitmap 64 bits in length. The TTE and the TLV the upper and the lower 64 bits of the ETTE, respectively.

- **Tag Line Vector(TLV)** A 64-bit bitmap. These bits correspond one-by-one to the fifth level table entry(Tag Line). The TLV gives a hint on the content of the fifth-level table. An example of this is shown in Figure 3.6.

  A bit in the TLV is set to 1 if and only if:

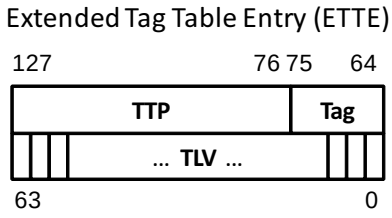

Figure 3.5: Composition of Extended Tag Table Entry

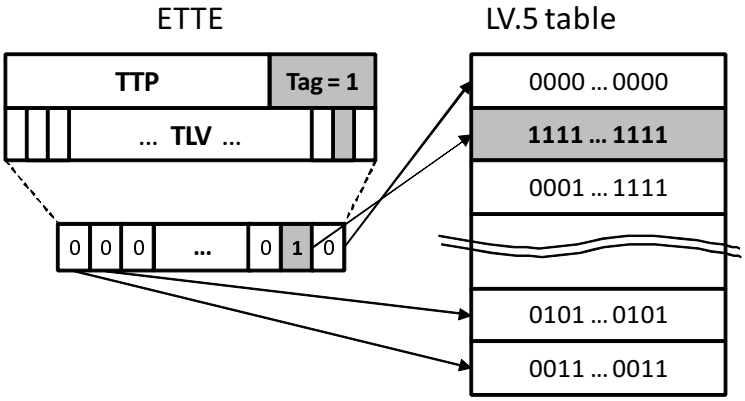

Figure 3.6: Composition of Tag Line Vector

– All tags on the corresponding tag line are equal
– The tags on the tag line match the TTE tag field

Hence, if a TLV bit is set, the tag may be obtained without accessing the fifth-level table. When all bits of the TLV are set to one, all the tags that exist on the fifth-level table are the same value. In such cases, if the TTP field is not NULL, or in other words, the fifth-level table still exists, we may free the fifth level table from tag table. This operation is called *contraction*. In Section 3.2.1, we describe contraction in more detail.

Tag Line The Tag Line(**TL**) is the format of the fifth-level entries of tag table. It is composed as in Figure 3.7.

The size of TL is 64 bytes, and it is composed entirely of tags. Note that it

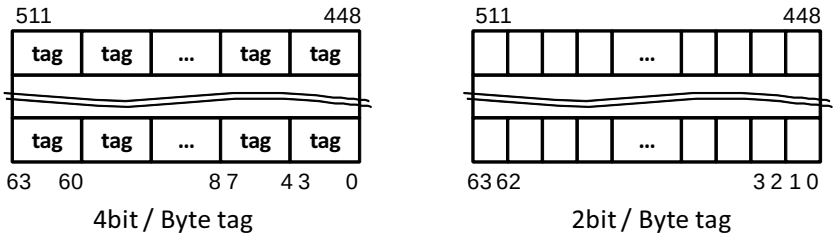

Figure 3.7: Composition of Tag Line

is important this size matches the line size of the L1 tag cache. The tag table is modified when a line of L1 tag cache, to which some changes were made, is written back. Since a line of L1 tag cache corresponds directly to a TL, the TMU is able to determine the value of the corresponding TLV bit at the time of write-back. If, for instance, the size of TL was larger than the cache line size, comparing the cache line bits during the write-back is not enough to determine the TLV bit. In this case, an extra access to the tag table will be neccessary to obtain the remaining bits for comparison. Such extra overhead is avoided in our design.

Free List

Because our system supports variable-length tag, the sizes of the fifth-level tables, containing only tags, differ by the length of tag. For example, if the length of tag is 1/2 of the data, the fifth-level table size will be 4 KB (because a page of data consumes 8 KB). If the length of tag is 1/4 of the data, the fifth level table size will be 2 KB, and so on.

We introduce a free list to our system for managing the tag table. The free list allocates and frees memory for tables in 4 KB chunks. When the length of tag is smaller than 1/2 of the data,two or more fifth-level tables are allocated at once, so that their total size matches the chunk size. We adopt this policy for not wasting memory.

Expansion and Contraction

Through *expansion* and *contraction*, the tag table dynamically adjusts itself to the minimal size. These operations exploit the non-uniformity of tag.

Expansion The tag table initially has no tables allocated beyond level 1. As tags are applied to data, the table expands, or new lower-level tables are allocated just the necessary amount.

The expansion allocates memory for new table from the free list, and initializes it. The initial value of each table entry is summarized in Table 3.2.

Table 3.2: Initial value of each table entry

Entry Type	Field	Value
TTE	TTP	0 (null)
	Tag	Tag of upper level entry
ETTE	TLV	All bits are one
TL	Tag	Tag of upper level entry

When the initialization process ends, the TTP of the entry for which the new table is built is set to the base address of the new table.

Contraction At anytime a large block of data is uniformly mapped a same tag, the whole block of tags will be reduced to a single tag table entry, or the tags will be contracted. In contrast to the expansion, the contraction removes a table from the tag table. The contraction attempt is triggered on the writing of tags to the tag table.

Figure 3.8 summarizes the contraction algorithm. First, all tags on the table are compared. This calls for accesses to all entries of the table, except in the case of a fifth-level table. For the fifth-level table, comparing all entries may be performed at once by simply inspecting the TLV bits. This mechanism is displayed in Figure 3.9.

A tag on the TTE Tag field is ineffective if the TTP field is not NULL. At the presence of such table entry, the contraction attempt is immediately abandoned. Otherwise, if all tags on the table are equal, or for the fifth-level table, if all bits of the TLV are set to one, the contraction occurs. The contraction frees a table on which all tags were a same value. The tag then is represented on the Tag field of the upper-level TTE.

The contraction of a low-level table further triggers a contraction attempt at its upper-level, for it modifies the upper-level table entry. The contraction is repeated recursively as long as it is successful, until reaching the first-level table.

Since table entries provide no means of moving up a level in the hierarchy, the TMU stacks the table pointers when first tracking them down and re-uses them during the recursive contraction.

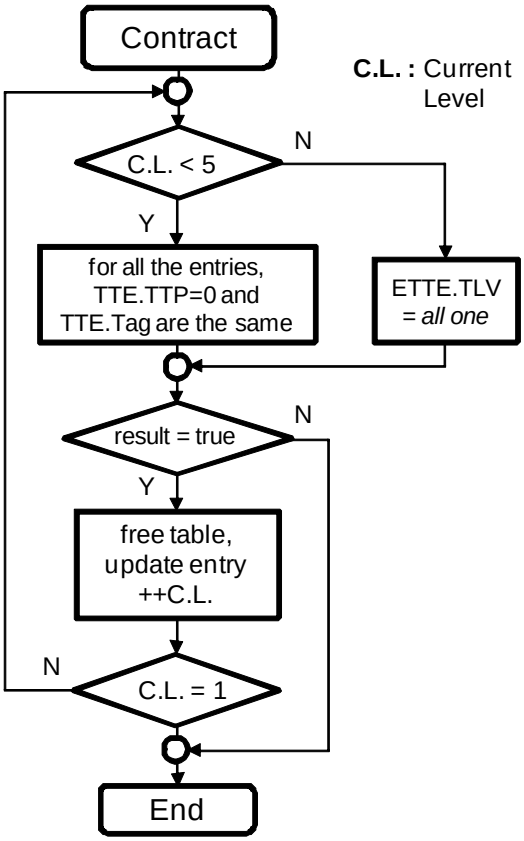

Figure 3.8: Contraction Algorithm

3.2.2 Tag Management Unit

This section gives details of the Tag Management Unit (TMU).

The TMU is placed between the L1 tag cache and the tag table on the L2 cache, and handles reading/writing of the table. The reading/writing of the tag table is a complicated operation, and thus requires a dedicated hardware unit. The reading/writing of the tag table incorporates the following procedures:

- The virtual address translation
- The expansion/contraction of the tag table

If TMU needs to access L2 cache on every reference of each different levels of tables, typically the case when tracking down the pointers for an address trans-

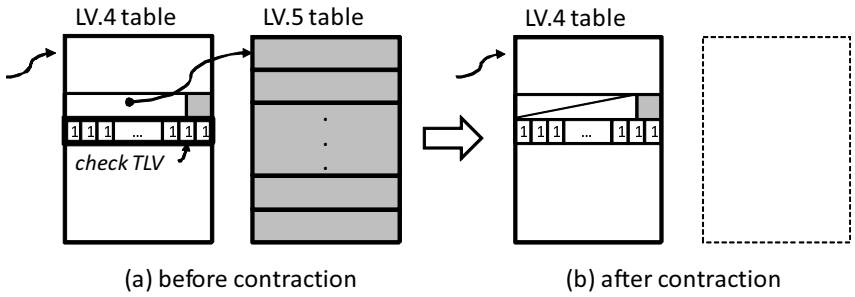

Figure 3.9: Contraction of fifth-level table

lation, the number of L2 cache accesses causes a large overhead. Thus we introduce a *pointer cache* for caching the intermediate product of address translation, the addresses of middle-level tables. The pointer cache usually provides all table addresses required for address translation. This significantly reduces the number of accesses to L2 cache.

In the following sections, we describe first the pointer cache and then the specifics of the read/write operations.

Pointer Cache

The pointer cache caches entries of the tag table in the same way a TLB caches page table entries. The composition of pointer cache is shown in Figure 3.10.

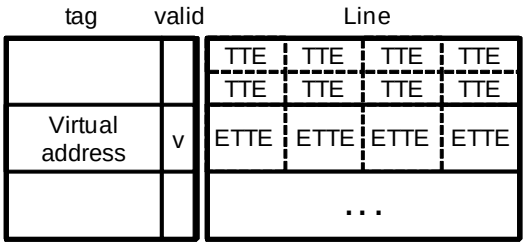

Figure 3.10: Composition of pointer cache

The basic structure of the pointer cache is no different from a conventional cache memory such as L1 data cache. The pointer cache caches entries of the level 1 to 4 tables (the TTEs and ETTEs). Whenever TMU needs to access the tag table, it first

accesses the pointer cache. If the pointer cache misses, the TMU accesses the L2 cache and refills the missed entry to the pointer cache.

Read/Write of Tags

Reading/writing of the tag table is a complicated operation. Both read/write requires a virtual address translation. The write further requires an expansion/contraction. The TMU handles their algorithms.

Read The algorithm of the read operation is shown by the flow chart in Figure 3.11.

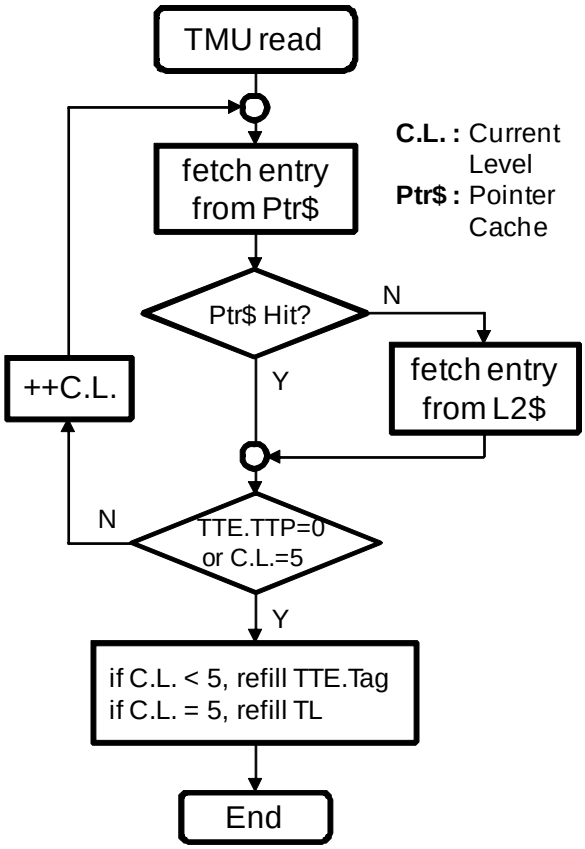

Figure 3.11: Read algorithm

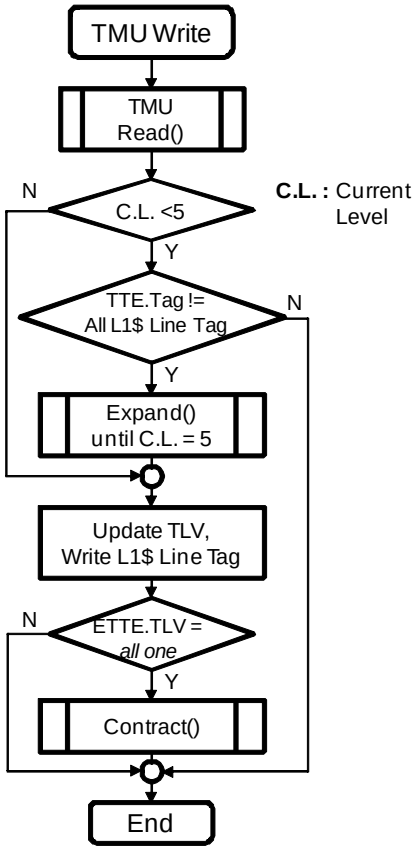

Figure 3.12: Write algorithm

For the read address translation, the TMU repeatedly accesses the tag table. On every reference, the TMU first reads the pointer cache, then on a miss, accesses the L2 cache.

The accesses to tables return the addresses of the next-level tables, and TMU caches them on the pointer cache. When the target Tag Line is obtained, the TMU returns it to the L1 tag cache.

Write The algorithm of the write operation is shown by the flow chart in Figure 3.12.

The virtual address translation is performed much the same way as the read operation, except for the added expansion operation. The expansion occurs in the

middle of the address translation. It is neccessary if any of the tags on the cache line written back differs from the tag on the currently refered TTE, and the TTE does not have a pointer to the next-level table. In this case, the TMU expands the tag table until the fifth-level table to hold the line is allocated.

After writing the line, the TLV bit is also set according to the rule described in Section 3.2.1. If the TLV bits then are all set to one, the TMU performs the contraction operation.

3.3 Separation in execution of tag

This section describes the separation in excution of tag which is mentioned in the beginning of this chapter.

In the conventional naive tagged architecture, the tags are combined with data from register to memory, and executed simultaneously with data. In this case, when the length of tag is much smaller than data, latencies for register file and caches are not affectted compared with non-tagged architecture. However, when the length of tag is large enough compared with data(e.g. 4bit tag for each byte of data), latencies for register file and caches are increased. Because the access latency increases by the size of the register file and caches typically, run-time overhead increases compared with non-tagged architecture. Moreover, to support variable-length tag in the conventional naive tagged architecture, all the tags must be fixed to the maximum length supported by the system. Access latencies for register file and caches are also fixed to the maximum length of tag, even if the security method use smaller-length tag than the maximum.

This thesis presents the separation in execution of tag. The key mechanism of the proposal techinque is that the tags are propagated after the completion of execution of data. This allows to separate tag from all of the data path in the processor completely. In the proposal technique, latencies for register file and caches are not affected compared with non-tagged architecture, beacause register file and caches are independent for tag and data.

Figure 3.13 is the block diagram of the proposal technique. The figure shows following two features :

- *Tags are propagated after the end of the execution of data*: Additional pipeline stage for propagating tag is added before the instruction retires. Tag propagation is executed by *Tag Propagation Unit(TPU)* in the program order. The big arrow from upper part of the figure(called data system) to lower part(called tag system) means *information of instruction flow*. TPU get this information of instruction from reorder buffer(ROB) for propagating tag.

- *Tags are separated from all of the data path in the processor*: The proposal technique includes dedicated tag register file(TF) and L1 tag cache(L1 T$). Thus, the data system and the tag system are completely separated.

The rest of this section gives details of the pipeline structure and the implementation of TPU and tag cache.

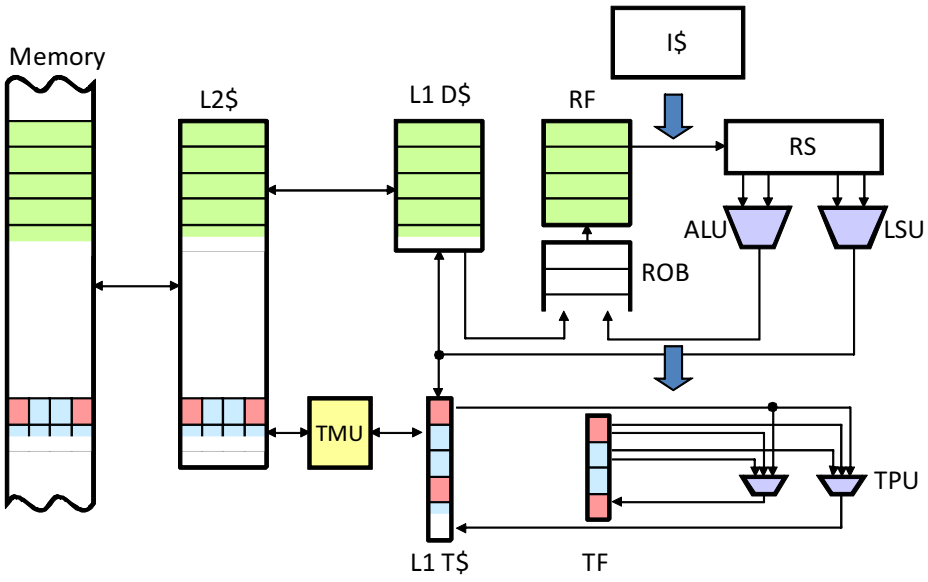

Figure 3.13: Block diagram of *the separation in execution of tag*

3.3.1 Pipeline structure

This thesis assume Out-of-Order superscalar processor with reservation station and reorder buffer. In the Out-of-Order superscalar processor, the processing of the instruction can be summarized as follows.

1. Instruction fetch : In-Order
2. Instruction execute : Out-of-Order
3. Instruction complete : In-Order

Typically, Out-of-Order superscalar processor fetches and decodes several instructions at a time in program order. Instructions are executed in parallel rather than original program order. When completion of instructions, the state of processor is updated in program order, so that correct result is obtained.

As mentioned before, tags are propagated after the end of the execution of data in the proposal technique. In this case, the processing of the instruction can be summarized as follows.

1. Instruction fetch : In-Order

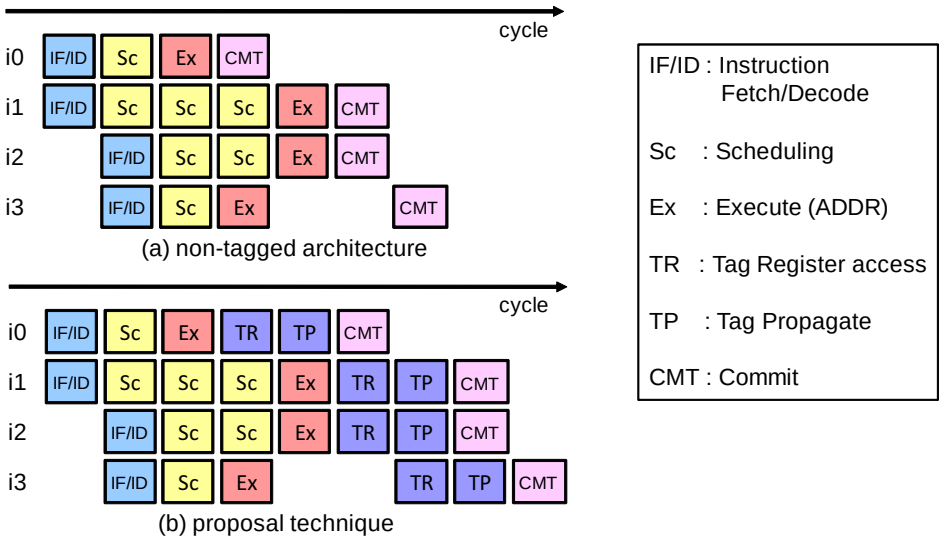

Figure 3.14: Pipeline structure of the proposal technique

2. Instruction execute : Out-of-Order
3. *Tag propagate : In-Order*
4. Instruction complete : In-Order

When execution of the instruction ends, several informations for propagating tag such as instruction type and dependency, are passed to the Tag Propagation Unit(TPU) and tags are propagated. The proposal technique obtaines these informations from reorder buffer. Because instructions wating for retirement in the reorder buffer are resequenced in program order, so that TPU can propagate tags in program order. This will be discussed in the next section for details. After tag propagation ends, TPU sends signal to the reorder buffer for retiring instructions.

Figure 3.14-b is the pipeline structure of the proposal technique. For the comparison, the pipeline structure of the non-tagged architecture is shown in Figure 3.14-a. The figure shows that tag propagation stage is added after execution stage. Tag propagation stage takes two constant cycles for propagation : one for the tag register file access*(TR)* and another for the tag propagation*(TP)*. Thus, it can be considered that the retire stage is two cycles longer than the non-tagged architecture.

When a memory access occurs, tag propagation must wait until the tag correspond to memory address returns. Figure 3.15 shows the pipeline structure includ-

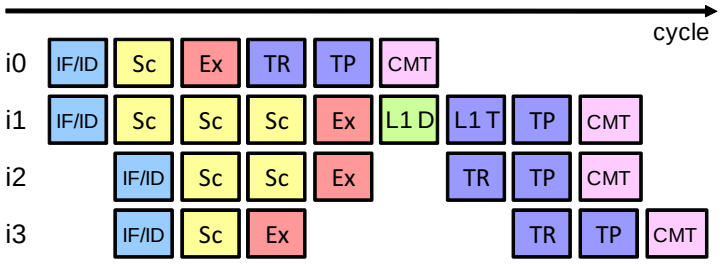

Figure 3.15: Pipeline structure including the memory instruction

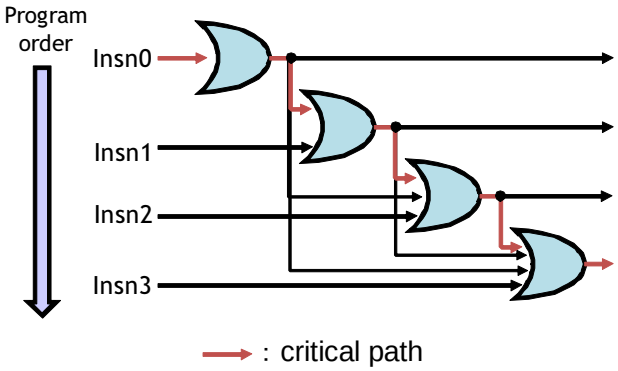

Figure 3.16: Block diagram of the TPU

ing the memory instruction. Access of the L1 tag cache*(L1 T)* is occurred at the same cycle as accessing the tag register file. Note that the memory address is decided when the execution stage(ADDR) is ends. Tag propagation is occured after the access of memory is ends.

In this section, we discussed the pipeline structure of the proposal technique and how it works. Following sections gives details of the implementation of TPU, tag register file and L1 tag cache.

3.3.2 Tag Propagation Unit(TPU)

This section gives details of the Tag Propagation Unit(TPU). As discussed before, TPU propagates tags in program order, after the end of the execution of data.

Figure 3.16 is the block diagram of the TPU. In this figure, the propagation of tag

is defined by OR operation of tag with the dependency. OR units of TPU are connected in cascade. The result of the upstream instruction is selectively propagated according to the instruction dependency. Thus, the time necessary for propagating tag becomes $number\ of\ instructions \times OR\ operation\ time$. Typically, a processor cycle time is decided by the execution time of ALU. Because OR operation is a bitwise operation, TPU propagates tags of multiple instructions at each cycle. It can be considered that the TPU is a small in-order processor for the tag system.

Tag propagation stage takes two cycles for propagation. TPU performs following operations at each cycle.

1. Read tag from tag register file / Access to L1 tag cache
2. Propagate tag / Write tag / Check tag

Note that these operations are fully pipelined.

TPU obtains PC and op code from reorder buffer. Informations for the instruction dependency are obtained from the frontend of processor. This is because TPU propagates tag in program order. TPU refers from the head of the ROB to the retire width, gets information of instruction in which execution ends. Therefore, retire width of the processor will not be affected by propagating tag.

Next two sections describe the tag resigter file and the L1 tag cache more detail.

3.3.3 Tag register file

Tag register file is a dedicated register file for tag. The hardware composition of the tag register will not be touched in this thesis because it is similar to a usual register file. The size of tag register file is enough to the number of logical registers, because TPU propagates tags sequentially. In general, the number of logical registers is less than the number of physical registers, thus it can be considered that tag register file is smaller and faster than register file of data.

Although, the proposal technique supports variable-length tag in many ways, the length of tag in the tag register file is fixed to the maximum length supported by the system. The main purpose of this is to minimize access time by making the tag register file a simple composition as possible, and decrease the influence by the tag propagation.

3.3.4 Tag cache

As we discussed before, when the length of tag is large enough compared with data, access latency for caches are increased in the conventional naive tagged ar-

chitecture. Table 3.3 shows the cache access time for various cache size in a 50nm technology[1]. When we assume that 8bit tags are assigned for each byte of data, it

Table 3.3: Cache access time in ns for various cache configurations in a 50nm technology[1](extracted part from original table)

Size (KB)	Ports	4-way set associative Block size (bytes) 32	64	128	256
16	1	0.39	0.50	0.78	1.50
	2	0.47	0.67	1.16	2.52
32	1	0.44	0.57	0.86	1.65
	2	0.53	0.78	1.32	2.81
64	1	0.50	0.63	1.00	1.86
	2	0.65	0.89	1.58	3.24

can be treated that each line of data cache increased to twice, compared with non-tagged architecture. In this case, the cache access time increases by the twice or more according to table 3.3.

Because L1 cache greatly influences on the performance, an increase in the cache access latency is not preferable. To avoid this, the proposal technique separate tag from L1 data cache. The hardware composition of dedicated L1 tag cache is shown in figure 3.17. Basically, hardware composition of the tag cache is the same

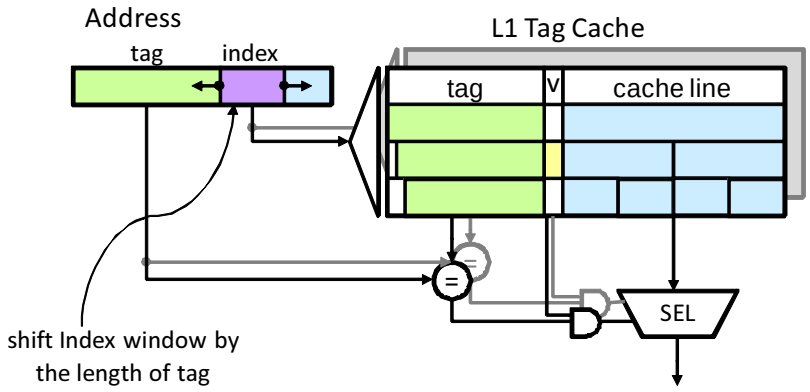

Figure 3.17: Composition of the L1 tag cache

as usual cache except address indexing. To support variable-length tag effectively, tag cache changes indexing depending on the length of tag.

Read and write operation of the tag cache is also the same as usual cache. When a cache miss occurred, tag cache requests line correspond to the memory address to TMU. Note that TMU is cache like structure which provides to accessing tag table transparently (see the section 3.2.2).

So far, we have discussed the composition and the operation of low-overhead tagged architecture for variable-length tag. The main idea of proposal technique is to separate tag and data completely. This is composed of two parts, separation in storage and separation in execution.

Section 3.2 described the separation in storage of tag. We presented the tag table and TMU for providing variable-length tag with low memory overhead. The tag table is a virtual storage for tags and characterized by a multi-level structure. TMU is cache like structure which provides to accessing tag table transparently.

Section 3.3 described separation in execution of tag. We suggested that propagating tags after the completion of execution of data. TPU propagates tags in program order, after the end of the execution of data. Dedicated tag register file and L1 tag cache prevent to increase in the access latency for register and cache. This allows to implement variable-length tag effectively.

From the next section, we evaluate and discuss various aspects of the proposal technique.

Chapter 4

Evaluation

In this section, we evaluate proposal technique through simulation. We first describe the evaluation environment, and then discuss the performance overheads.

4.1 Evaluation Environment

Our simulation uses a cycle-accurate processor simulator *Onikiri2*, developed in our laboratory. Unlike SimpleScalar Tool set [2], which is used widely for researches on processor architecture, Onikiri2 emulates an execution of instruction on the exact cycle it is supposed to be on the execution stage. The parameters used for the simulation are shown in Table 4.1.

Table 4.1: Architectural parameters

Architectural parameters	Specifications
ISA	Alpha
fetch width	4 inst.
execution unit	int: 2, fp: 2, mem: 2.
instruction window	int: 32, fp: 32, mem: 16
register file	int: 96, fp: 64
L1 I/D cache	32 KB, 4 way, 64 B/line, 3 cycles
L1 Tag cache	4 KB, 4 way, 64 B/line, 1 cycles
L2 cache	4 MB, 8 way, 64 B/line, 15 cycles
main memory	200 cycles

We used 27 programs of the SPEC CPU2000[8] and SPEC CPU2006[9] benchmark with *train* data sets. The programs were compiled using gcc 4.2.2 with "$-$O3"

options. We skipped the first 1G instructions and collected statistics from the next 300M instructions. Although we skipped the first 1G instructions, tags are propagated correctly from the begining of the programs. This is because the tag propagation is rely on the initial state of tag.

To evaluate proposal technique, we applied tags to every input data, and propagated them by rules shown in Table 4.2. This propagation rule is for tracking explicit flow of data. The length of tag applied was 1 bit for a byte of data.

Table 4.2: Tag propagation rules

Inst. type	Example	Tag propagation
Arithmetic or Logical	addl R1, R2, R3	T1 ← T2 OR T3
	addli R1, R2, #Imm	T1 ← T2
Load	ldl R1, Imm(R2)	T1 ← T[R2+Imm] OR T2
Store	stl Imm(R1), R2	T[R1+Imm] ← T1 OR T2
Branch/Jump	jmp R1	do not propagate tag

In this table, the notation 'Rn' is used to indicate the data of register number n. The notation 'Tn' indicates the tag applied to the data 'Rn'. Also, the notation T[x] indicates the tag stored on the address x.

4.2 Evaluation item

Compared to non-tagged architecture, the following performance overheads are added to tagged architecture.

- Memory overhead : Extra memory consumed by tags.
- Time overhead : Additional time caused by propagating and accessing tags.

In this thesis, we suggested separating tag and data completly to support variable-length tag with low-overhead. The proposal technique is composed of two parts, separation in storage and separation in execution. We evaluate these techniques individually.

The following sections describe how proposal technique reduces these overheads and shows the result of the evaluation. We first evaluate the separation in storage of tag. Then, we evaluate the separation in storage of tag next.

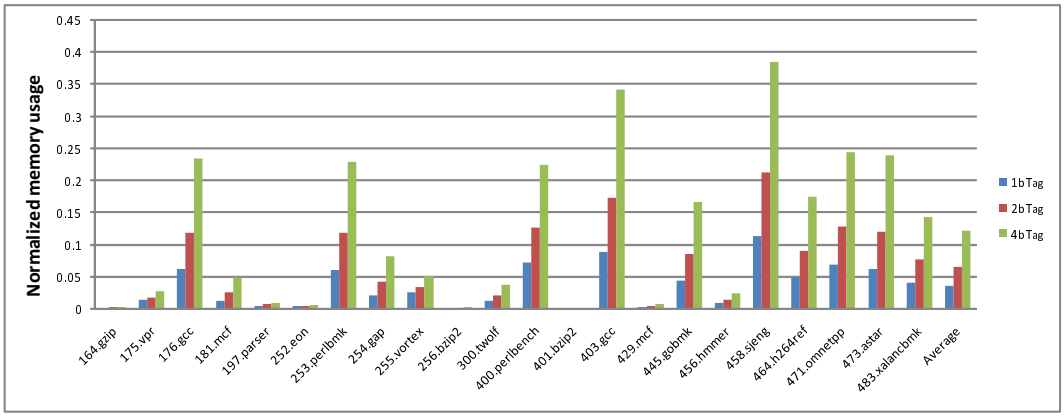

Figure 4.1: Memory overhead for various length-tag (with contraction)

4.3 Separation in storage

This section evaluates the separation in storage of tag.

4.3.1 Memory overhead

To evaluate memory overhead, we measured the total amount of memory consumed by the tag table. The measurement was taken from the snapshot of the memory after running 300M instructions. Figure 4.1 presents the memory used by the tag table with contraction for all benchmark programs. The values on the graph are normalized by the amount of memory dynamically allocated by the programs. Each labels indicate the length of tag, 1 bit, 2 bit, 4 bit of each data byte. The overhead varies among the benchmarks, because it strongly depends on the character of each program. Some factors that affect the overhead are the input data size, which determines the initial amount of tags, and the behavior of the program, which determines how the tags are propagated. The result shows that the memory overhead is 3.48% at the 1 bit tag, 6.48% at the 2 bit tag, and 12.2% at the 4 bit tag in average. This result is about 1/4 smaller than the naive implementation of tagged architecture, which simply adds a tag field on ecah data byte. This naive implementation does not, however, add tag fields to every bytes of the memory. It selectively adds tag fields to the bytes of the memory allocated by the program only. Thus its memory overhead, when normalized by the amount of memory allocated by the program, is a fixed value. Since we added 1 bit of tag per a byte of data, the overhead is

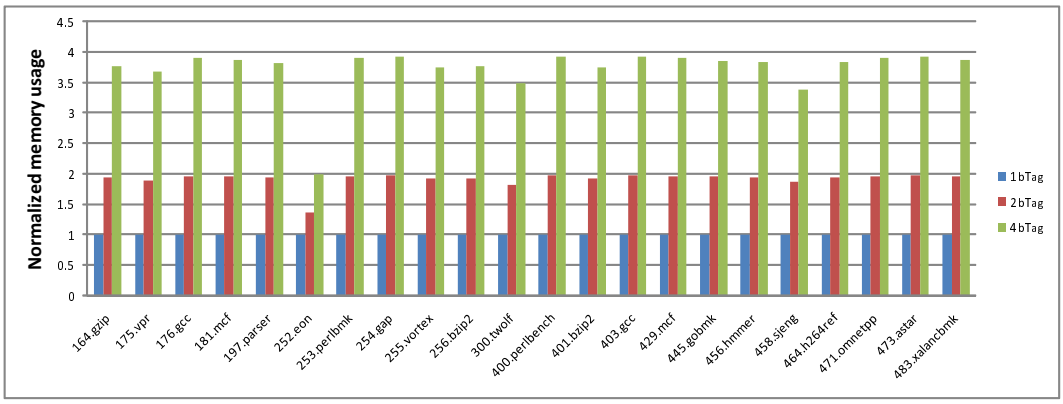

Figure 4.2: Memory overhead for various length-tag (without contraction)

12.5%(1/8).

Figure 4.2 presents memory used by the tag table without contraction for all benchmark programs. The values on the graph are normalized by the amount of memory used by the tag table at 1 bit tag. This result shows that the memory used by the tag table grows nearly linear by the length of tag. This means that the overehead by intermediate tables of multi-level table structure is negligible except 252.eon. The reason of this result is that total amount of tags assigned to 252.eon is small so that the last(leaf) level tables are rare compared with intermediate tables.

Figure 4.2 compares the memory usages of the tag table when using contraction and not. The values on the graph are normalized by the amount of memory used by the tag table without contraction. The result shows that the contraction operation reduces the memory overhead to 69.2Thus we can see that the use of contraction significantly contributes to the reduction of memory overhead. The result also shows that the use of contraction has more impact on the overhead when longer tags are used.

This result can be analyzed as follows. Without contraction, no allocated tables are freed until the end of the process. The contraction enables that these tables are freed if a significant-size block of memory is applied a same tag by the program. The tag mapping as a result of the program behavior is not affected by the length of tag. This means that the number of contractions triggered in a program does not change by the length of tag. As stated in Section 3.2.1, the size of the bottom-level table is proportional to the length of tag. Hence, thememory consumption reduced by the contraction is proportional to the length of tag.

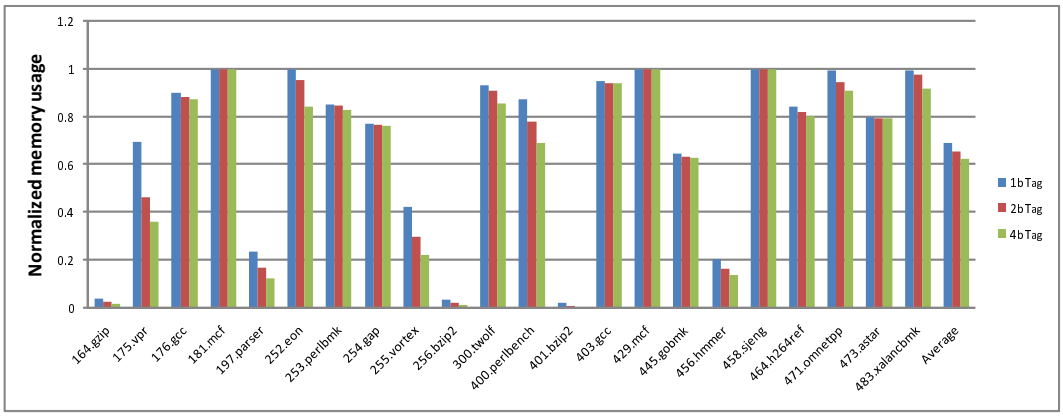

Figure 4.3: Memory overhead for various length-tag (contraction vs. without contraction)

4.3.2 Time overhead

We evaluated the IPC degradation of the proposal technique from a non-tagged architecture.

Figure 4.4 presents the IPC degradation for various-length tags. The size of the tag table increases by the length of tag large. On the same hardware conditions, the longer the length of tag, the larger size of caches are needed to put out the same performance. The figure shows that the performance degradation is modest when the length of tag is changed. The average IPC degradation of proposal technique is 4.96% at 1 bit tag.

Contraction also affects performance. We presents the IPC degradation by contraction in Figure 4.5. We can observe from this figure that the IPC degradation by contraction is 1.66% smaller than the model which does not use contraction, although it needs to compare whole entries in the specified table for every write operation. This is due to the use of TLV for contraction. As we discussed before, TLV reduces the overhead for comparing sequence to O(1). The reason why the contraction model shows the better performance than the model which does not use contraction is that the contraction improve the utilization of L2 cache and also pointer cache. By contraction, the tag table maintains its size as compact as possible. Thus, memory for caching the tag table is less needed. As result, it can increase performance.

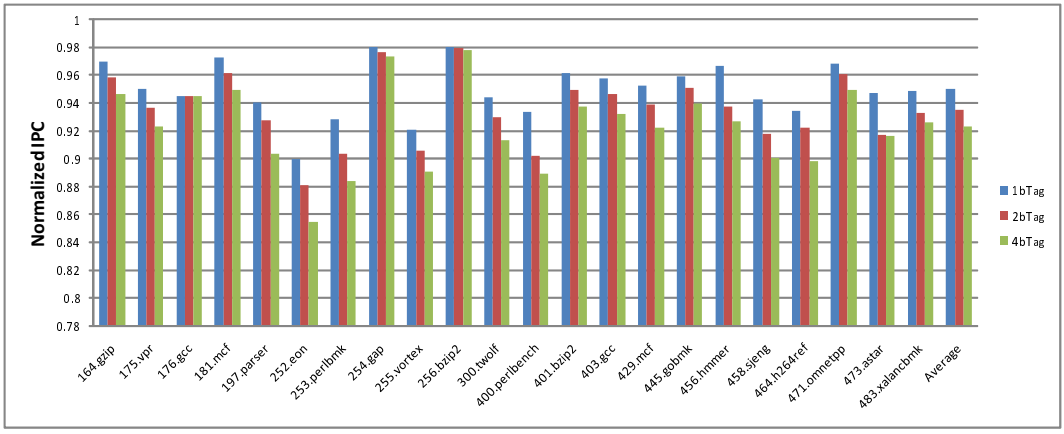

Figure 4.4: IPC degradaion by length of tag

4.4 Seperation in execution

This section evaluates the separation in execution of tag. The proposal technique of the separation in execution has hardware memory overhead, but we will not touch in this thesis. Because quantitative analysis of the hardware amount was not assumed to our simulation. This will be one of our future works. The rest of this section gives the evaluation of time overhead.

4.4.1 Time overhead

We evaluated the IPC degradation of the proposal technique from a non-tagged architecture. In the proposal technique, the factor to cause the performance degradation is as follows.

- An increase in pipeline stage : By adding tag propagation stage, instructions need more cycles to retire. While instructions just finished execution wait for the tag propagation result, the resources of the processor are not liberated. Thus, the resources of the processor becomes more insufficient than non-tagged architecture. This decreases the number of executable instructions next, IPC degradation occur.

- Tag cache miss : When the tag cache access is occurs, the tag might arrive later than data. In this case, memory access latency of the proposal technique is bigger than non-tagged architecture. Because, TMU walks the tag

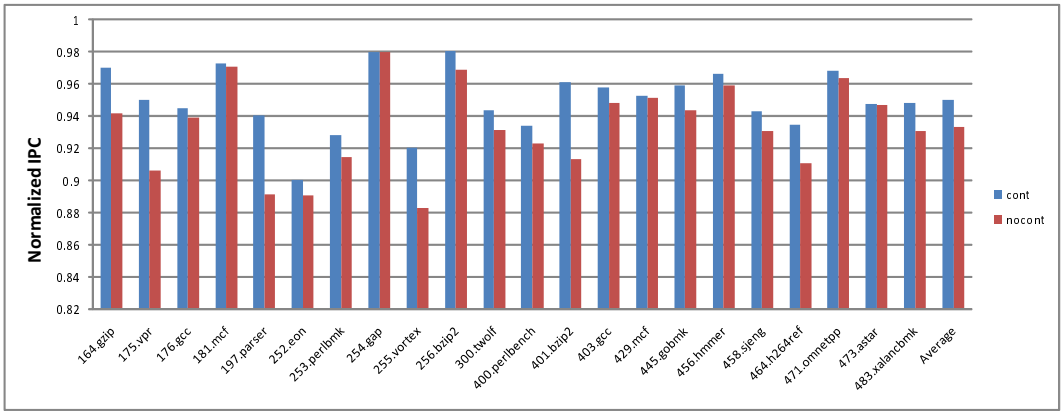

Figure 4.5: IPC degradaion by contraction

table in the virtual memory, and on every reference, the TMU first reads the pointer cache, then on a miss, accesses the L2 cache. This can cause the IPC degradation.

Figure 4.6 shows the IPC of proposal technique, normalized by the IPC of the base model, a non-tagged architecture. The *pseudo* labels indicate a non-tagged architecture that increased the retire stage by two cycles. The *all* labels indicate the proposal technique applied both separation of storage and separation of execution.

The average IPC degradation of pseudo is 2.64%. And the average IPC degradation of proposal technique is 4.96%.

The result of all can be calculated as $time\ overhead\ by\ tag\ propagation + tag\ cache\ miss\ penalty$. Thus, it can be considered that the diffrence between pseudo and all is the affection of the tag cache miss penalty. This result shows that the major factor of time overhead is the tag cache miss penalty.

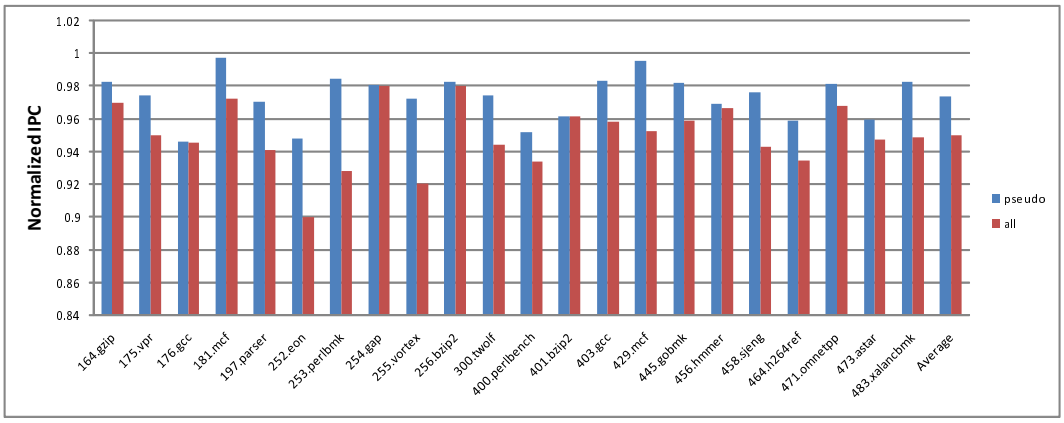

Figure 4.6: IPC evaluation of separation in execution

Chapter 5

Conclusion

In this paper, we presented low-overhead security-tagged architecture. We focused on separating tag and data completly. And this is composed of two parts, *separation in storage* and *separation in execution*.

In Section 3.2, we described the separation in storage of tag. For supporting variable-length tag with low overhead, our system exploits two characters of tag, which are the non-uniformity and the locality of reference of tags. We presented uniquely designed multi-level table, the tag table and various cache-like structures including TMU. The tag table is a virtual storage for tags and characterized by a multi-level structure. And TMU provides to accessing tag table transparently.

In Section 3.3, we described separation in execution of tag. We suggested that propagating tags after the completion of execution of data. TPU propagates tags in program order, after the end of the execution of data. Dedicated tag register file and L1 tag cache prevent to increase in the access latency for register and cache. This allows to implement variable-length tag effectively.

By simulation, we showed that our system can significantly reduce the memory overhead compared to a naive implementation in which a tag field is added to every data bytes. We also evaluated the latency overhead. This implies that some mechanisms to reduce the L1 cache-miss penalty, such as a pointer cache, is effective to our system. Overall, our system is able to reduce the memory overhead to 3.48% of the naive implementation, and showed an IPC degradation of 4.96%.

Our plan for the future study is actually applying some techiniques of tracking information flow on our system. The feature of our system that it supports variable-length tag allows us to apply multiple techniques at once, even if they apply different-length tags on different-length data.

We recongnize the importance of the information flow tracking techiniques, and

believe a method of their low-overhead implementation will contribute to information security.

Bibliography

[1] Vikas Agarwal, Stephen W. Keckler, and Doug Burger. The effect of technology scaling on microarchitectural structures. Technical report, University of Texas at Austin Department of Computer Sciences, 2001.

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

[3] J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th International Symposium on Microarchitecture (MICRO)*, 2004.

[4] E. A. Feustel. The rice research computer: a tagged architecture. In *AFIPS '71 (Fall): Proceedings of the November 16-18, 1971, fall joint computer conference*, pp. 369–377, New York, NY, USA, 1971. ACM.

[5] E.A. Feustel. On the advantages of tagged architecture. In *IEEE Transactions on Computers*, Vol. c-22, pp. 664–656, July 1973.

[6] Jeffrey Hagelberg, Weimin Lu, and Shaozhi Ye. Tagged architecture: A quantitative analysis. In *ECS 201A Project Report*, Fall 2005.

[7] E. A. Hauck and B. A. Dent. Burroughs' b6500/b7500 stack mechanism. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 245–251, New York, NY, USA, 1968. ACM.

[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] G.Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pp. 85–96, 2004.

[11] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, M. Vachharajani G. A. Reis, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *37th International Symposium on Microarchitecture (MICRO)*, 2004.

[12] E. Witchel, J. Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of ASPLOS-X*, Oct 2002.

Publications

1. 可変長タグをサポートする低オーバヘッド・タグ・アーキテクチャ
   金 大雄，塩谷 亮太，入江 英嗣，五島 正裕，坂井 修一
   先進的計算基盤システムシンポジウム SACSIS2008，於 つくば国際会議場，pp. 177-185, June, 2008

2. (pending) Low-overhead tagged architecture for variable-length tag
   Daewung Kim, Kazuo Horio, Ryota Shioya, Masahiro Goshima and Shuichi Sakai
   The 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)

Acknowledgement

First, I would like to record my gratitude to my advisor, Professor Shuichi Sakai, for his supervision, advice, and guidance through the three years I have spent in his laboratory. He provided me with continuous support, encourage and unusual environment for research as well as gave me the highest opportunity to conduct my own research. I also thank to him for being not only a good research advisor, but an excellent mentor.

I gratefully acknowledge Associate Professor Masahiro Goshima for his polite guidence and invaluable technical comment on my research. I learned many things from his enthusiasm and his great effort to explain things clearly and simply. He also gave me helpful advice on how to improve presentation skill.

I am gratitude to Ryota Shioya for supporting and teaching me devotedly. He gave me various suggestions for my research and technical supports. I always admire his creative thinking. He seems like a constant fountain of ideas.

Many thanks go in particular to Kazuo Horio. He helped me in writing this thesis in various ways. This paper could not have been written without the invaluable help of him.

I am indepted to the rest of laboratory members for providing invaluable discussions and fun environment to learn. I am especially grateful to Ken Sugimoto, Sho Tarui, Toru Ando, Takahiro Yamada and Takanobu Kita. They dedicated their time to maintain the network and computers in the laboratory, and manage the whole of the laboratory. To Young-Kwang Moon and Kunbo Li, it is pleasure to collaborate with you. Special thanks go to Mrs. Harumi Yagihara, Ms. Tomoyo Ise, and Ms. Tamaki Hasebe for their dedications to make administration affairs run smoothly. I wish to warmly thank the following former members, Dr. Hidetsugu Irie, Dr. Luong Dinh Hung, Hiroyuki Kurita and Kazuto Shimizu for assisting me a lot when I started my research at the laboratory.

I also express my thanks to goverment of Korea and Japan, and The Korean Scholarship Foundation. Without their generous financial support, I would have not

been able to fully dedicate myself to the research.