PAPER

Low-Overhead Architecture for Security Tag

Ryota SHIOYA[†a)], **Daewung KIM**[†*], **Kazuo HORIO**[†**], **Masahiro GOSHIMA**[†], ***Nonmembers***, ***and*** **Shuichi SAKAI**[†], ***Member***

SUMMARY A security-tagged architecture is one that applies tags on data to detect attack or information leakage, tracking data flow. The previous studies using security-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 a technique, however, results in a huge hardware overhead. This paper proposes a low-overhead tagged architecture. We achieve our goal by exploiting some properties of tag, the non-uniformity and the locality of reference. Our design includes the use of uniquely designed multi-level table and various cache-like structures, all contributing to exploit these properties. Under simulation, our method was able to limit the memory overhead to 0.685%, where a naive implementation suffered 12.5% overhead.

key words: *processor architecture, tagged architecture, information security, information flow tracking*

1. Introduction

A tagged architecture [1] is an architecture in which each piece of data has a tag applied to describe its property. 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 runtime.

In recent years, tagged architectures have been used in the field of information security, which includes preventing information leakage, detecting malicious attacks and so on. For example, *dynamic information flow tracking (DIFT)* [2] applies a 1-bit tag to every word of the memory, for detecting a broad range of malicious attacks such as code injection attack. Another example, *RIFLE* [3], is a technique that uses tags to distinguish between the data that must be protected and not. Some data that must be protected are personal data and copyrighted works, by avoiding leakages and illegal copies, respectively.

Despite many previous studies, the techniques 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 each byte, it becomes 9 bits in size.

Manuscript received June 1, 2010.
Manuscript revised September 13, 2010.

[†]The authors are with the Graduate School of Information Science and Technology, The University of Tokyo, Tokyo, 113–8656 Japan.
[*]Presently, with LG Electronics.
[**]Presently, with Fujitsu Laboratories.
a) E-mail: shioya@mtl.t.u-tokyo.ac.jp

DOI: 10.1587/transinf.E94.D.69

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 cannot change their size from one to another. For example, Raksha [4] applies 1- or 2-bit tag to each word for capacity, but they also point out that per-byte tags is preferable against attacks using strings.

There are two sides to the overhead of tagged architecture, the memory overhead and the latency overhead.

Memory overhead The extra capacity required to store tags. For example, the memory overhead of the naive implementation which applies 1-bit tag per byte is $1/8 = 12.5\%$.

Latency overhead The latency overhead can be represented by the extra time for loading and storing tags.

The goal of our research is to minimize both of these overheads. We achieve our goal by exploiting the non-uniformity and the locality of reference of tags. The non-uniformity and the locality of reference are the characters of tag summarized as follows:

Non-uniformity The memory can be divided into areas of data that have tag and ones that don't. Within a tag-assigned area, the tags are likely a same value. Across the areas, the values may be different.

Locality of reference Tags have the locality of reference just as much as data does. Moreover, since tags are not applied to all data, they are cached more effectively than data.

Our system exploits these characters using a uniquely designed multi-level table and various cache-like structures. We adopt the following:

Data structure Multi-level *Tag Table* which has similar structure to the page table to store security tags.

Physical structures A dedicated level-1 and a unified level-2 cache for security tags.

The rest of the paper is organized as follows. In Sect. 2, we review related work on security-tagged architecture. Sections 3, 4 and 5 will give details of our proposal, focusing on the design of the Tag Table and the tag caches, each of which is a key component of our system. In Sect. 6, we evaluate the efficiency of the system. In Sect. 7, we state the conclusion.

2. Related Work

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

RIFLE [3] 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 [5] 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 word 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. [2] is an architecture to detect both control and non-control attacks. Their method applies a 1-bit tag to every word 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 separate 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) [6] 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 word 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 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 than 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 implementation. 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 cannot dynamically adjust the tag storage to its minimal size, which is possible by our technique of *contraction*, explained in Sect. 4.5. 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 its contents frequently.

3. Proposal

We propose a low-overhead tagged architecture for security tag. We exploit some properties of the tags in our design, the non-uniformity and the locality of reference.

3.1 Properties of Security Tags

Non-uniformity

The memory can be divided into areas of data that have tag and ones that don't. Within a tag-assigned area, the tags are likely a same value. Across the areas, the values may be different.

When files or data from a network are read into a consecutive area of the memory, the same tag will initially be applied throughout the data. This leads to the situation described above. Figure 1 shows an image of the memory in this situation. In this figure, there are areas of data with tags applied and areas without. The different tag-applied areas have different tags on them, for the tags are derived from different origins, which may be files or IO.

Locality of Reference

Tags have the locality of reference just as much as data does. Moreover, since tags are not applied to all data, they are cached more effectively than data.

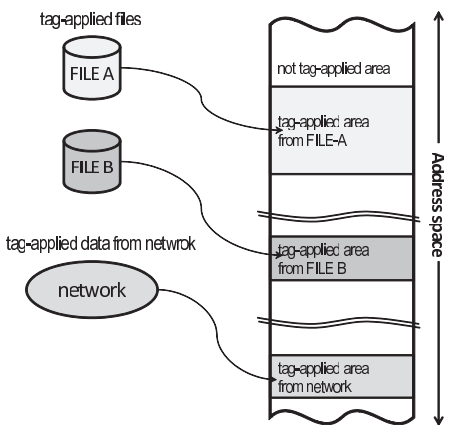

Fig. 1 Non-uniformity of tag-applied area. Non-uniformity is summarized: 1. Tag applied areas are partial and contiguous. 2. Tags are likely same value within these areas.

3.2 Overview

Figure 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

The Tag Table is a data structure for storing tags, built on the main memory. The Tag Table is characterized by a multi-level structure like the page table. The index of the page table is virtual address and the value of it is the corresponding physical address. In the case of the Tag Table, the index is virtual address but the value is the tag of the corresponding data.

The page table usually has a multi-level tree structure. Figure 3 shows the structure of the Tag Table. The Tag Table consists of 5 levels of subtables. The leaf subtables have the values of tags, while the non-leaf subtables have the pointers to the next-level subtables. The value of the tag is obtained by walking on the path indicated by the pointers.

Just like the page table, subtrees for the areas that do not have tags assigned are not allocated on the memory, saving large amount of memory.

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.

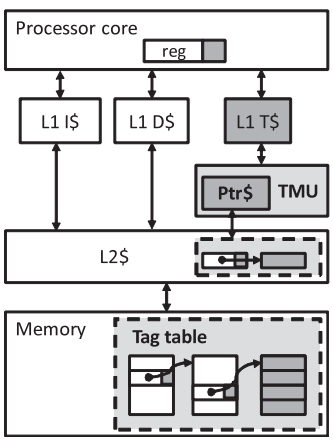

Fig. 2 Block diagram of system.

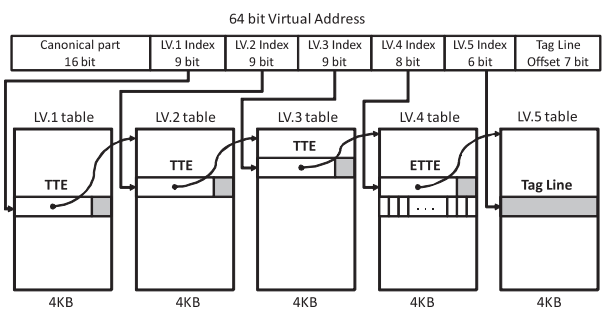

Fig. 3 Tag table and virtual address translation.

On the L1 cache miss, the TMU walks on the Tag Table, retrieves the tag, and refills it to the L1 tag cache. Meanwhile, L2 cache is just the same as conventional one, and does not have any special functionality for tags. Therefore, the image of the Tag Table is cached on the L2 cache as it is.

The Tag Table is protected by usual memory protection system, thus user programs cannot access the Tag Table on the L2 cache directly. The user programs can access tag information only through the L1 tag cache. This can avoid malicious attacks by accessing to the Tag Table in the case that our proposal is used in the field of information security.

We do provide a dedicated L1 cache for tags, but not for L2 cache for the following reasons:

- If L1 cache is unified, the Tag Table Walk is needed for each execution of load/store instructions, resulting in an intolerable overhead. In our configuration, the Tag Table Walk is only needed on L1 cache miss, making the overhead tolerable.
- The dedicated L2 cache means the TMU is placed between L2 cache and the main memory. The TMU then must perform Tag Table Walk on the main memory, also resulting in intolerable overhead.

The combination of Tag Table and the cache hierarchy described above minimizes the memory overhead and latency overhead.

The following sections give details of the implementation of the Tag Table and the cache hierarchy.

4. Tag Table

This section gives details of the Tag Table.

As described before, the Tag Table is a multi-level table like a page table. Figure 3 shows the structure of the Tag Table. The Tag Table consists of 5 levels of subtables. The leaf subtables have the values of tags, while the non-leaf subtables have the pointers to the next-level subtables. The Tag Table is accessed by a 64-bit virtual address. The virtual address is divided into fields as shown in Fig. 3. The index field for a non-leaf level subtable provides offset from the base address of the corresponding table. This structure of the Tag Table is designed after x86-64 page table [7].

Note that the Tag Table in Fig. 3 is designed under the assumption that the pages are 4 KB and always aligned to 4 KB-boundaries. The same assumption is applied throughout the paper.

4.1 Tree Structure

The conventional page table can largely reduce its size by creating subtrees only for allocated virtual pages. The Tag Table also adopts this mechanism. It creates subtrees only for data which have tags applied on them. In addition, the Tag Table is able to delete subtrees under certain conditions,

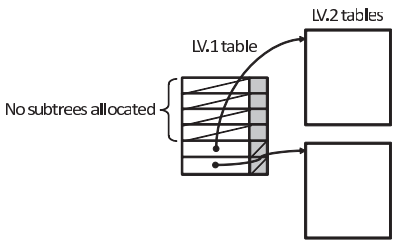

Fig. 4 Subtrees allocated only if tags are applied.

achieving even more efficient storage than page tables. In Fig. 4, most entries of the first-level table are not allocated, reducing the area significantly.

The non-leaf level subtables are mostly for pointing to the next level subtables. These entries, however, contain more than just pointers, specifically, the tags themselves. The entries of the different level tables vary in their structures as follows:

Tag Table Entry (TTE) An entry of the first to the third level subtables

Extended Tag Table Entry (ETTE) An entry of the fourth level subtables

Tag Line (TL) An entry of the leaf-level subtables

The TTE and the ETTE have TAG fields, which allows a tag to be obtained without tracking the pointers down to the leaf-level. This field can be used when a same tag is mapped to a significant-size block of memory. In such case, the use of lower-level subtables 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 subtables 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 subtable 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.

4.2 Data 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 subtables. It simply provides a pointer to the next level subtable in most cases. The composition of TTE is given in Fig. 5.

Tag Table Pointer (TTP)

The upper 52 bits of the physical address of the next-level subtable. The 64-bit physical address is derived by adding the lower 12 bits, all zeros, to the TTP.

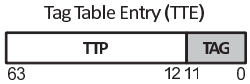

Fig. 5 Composition of Tag Table Entry.

Table 1 Linear address space covered by TAG field.

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

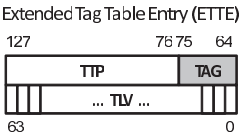

Fig. 6 Composition of Extended Tag Table Entry.

TAG

Since each table consisting the Tag Table is 4 KB and always aligned on 4 KB 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 Sect. 4.3. 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 subtable. Specifically, its TTP field is set to NULL. When the TMU at Tag Table Walk encounters a null TTP, it returns the tag on the Tag field instead of further tacking down the 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 are 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 1.

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 Fig. 6.

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 are the upper and the lower 64 bits of the ETTE, respectively.

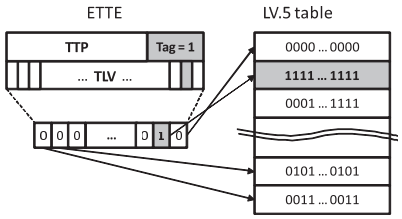

Fig. 7 Composition of Tag Line Vector.

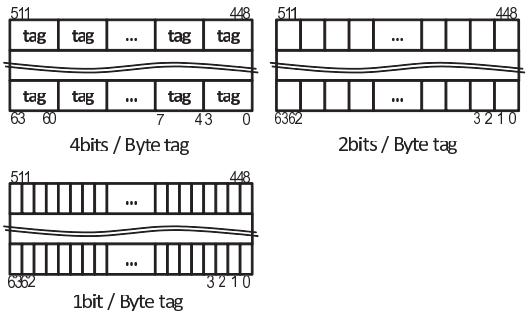

Fig. 8 Composition of Tag Line.

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 subtable. An example of this is shown in Fig. 7.

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

- 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 subtable. When all bits of the TLV are set to one, all the tags that exist on the fifth-level subtable are the same value. In such cases, if the TTP field is not NULL, or in other words, the fifth-level subtable still exists, we may free it from the Tag Table. This operation is called *contraction*. In Sect. 4.5, we describe contraction in more detail.

Tag Line

The Tag Line (**TL**) is the format of the fifth-level entries of the Tag Table. It is composed as in Fig. 8. This figure shows formats of TL in each case that the length of tag applied was 1, 2, and 4 bits for a byte of data.

The size of TL is 64 bytes, and it is composed entirely of tags. The size of TL is equal to that of the line of the L1 tag cache. The reason is described in Sect. 5.1.

4.3 Free List

Our system supports variable-length tag. The sizes of the fifth-level subtables, 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 subtable size will be 2 KB (because a page of data consumes 4 KB). If the length of tag is 1/4 of the data, the fifth level subtable size will be 1 KB, and so on.

The free list allocates and frees memory for tables in 4 KB chunks. When the length of tag is equal to or smaller than 1/2 of the data, two or more fifth-level subtables are allocated in a single chunk.

4.4 Consideration about Address Space

We use a virtual address to access the Tag Table rather than a physical address for the following reasons:

- A physical address assigned to a virtual address may be changed on page swapping. Therefore, the use of the physical address requires additional processes that include updating the Tag Table to the changed physical address. The use of the virtual address can avoid these processes.
- We reduce the size of the Tag Table by exploiting non-uniformity of tags, Tag applied areas are partial and contiguous. The use of the physical address can weaken this non-uniformity because contiguous areas on a virtual address space are not always contiguous on a physical address space.

The Tag Table itself is placed on the physical address space and is managed to avoid being swapped out by an OS as conventional page tables. Therefore, the pointer of the subtable is a physical address and this can remove address translation on Tag Table Walk.

4.5 Expansion and Contraction

Through *expansion* and *contraction* operations by the TMU, the Tag Table dynamically adjusts itself to the minimal size. These operations exploit the Non-uniformity of tags. In this section, we first describe expansion and contraction operations, and then discuss the overheads of them.

4.5.1 Expansion

The Tag Table initially consists only of a level-1 subtable. As tags are applied to data, the table is *expanded*, or new lower-level subtables are allocated as necessary. The Tag Table obtains new chunks from the free list, allocates them to new subtables, and then link the subtable with pointers.

4.5.2 Contraction

Whenever a block of data is applied the same tag, the table can be *contracted*, or the block of tags will be reduced to the single TTE (or ETTE) of the upper-level subtable. In contrast to the expansion, the contraction removes subtables from the Tag Table.

The contraction is triggered on the write of tags to the Tag Table. The contraction starts when the write changes a tag and all the tags on the subtable has the same value as a result. This requires accesses to all the entries of the subtable, except in the case of a leaf-level subtable.

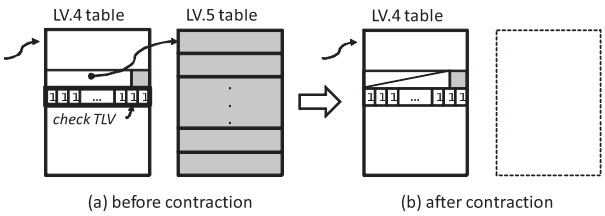

Fig. 9 Contraction of fifth-level subtable.

Leaf-level Subtable

For the leaf-level subtable, checking all entries can be performed by simply inspecting the TLV bits. This mechanism is displayed in Fig. 9. As described in the previous subsection, a bit of TLV is set when all the tags of the Tag Line are the same value as the TAG field of the TTE. Therefore, if all the bits in the TLV are set, all the tags of the subtable has the same value.

If the condition is met, the subtable is freed.

Non-leaf-level Subtable

The contraction process of a non-leaf-level subtable is triggered by the contraction of a lower-level subtable, for it modifies the table entry.

Firstly in the contraction process of non-leaf-level subtable, the TAG fields of all the TTEs have to be checked if they are of the same value.

The TTE TAG field is invalid when the TTP field is not NULL (it is a pointer to the next-level subtable). At the presence of such table entry, the contraction process immediately stops.

The contraction is repeated recursively as long as it is successful, until reaching the first-level subtable.

4.5.3 Overhead of Expansion and Contraction

As described before, the operations required for expansion are allocating subtables from the free list and linking the subtables with pointers. This operations cause several write accesses to the L2 cache for the linking. The evaluation result in Sect. 6 shows that the frequency of expansion is small enough, thus the write accesses do not decrease performance greatly.

The overhead caused by contraction varies according to the level of a subtable. For the leaf-level subtable, checking all entries can be performed by simply inspecting the TLV bits. On the other hand, for the non-leaf-level subtable, the TAG fields of all TTEs in a subtable have to be checked, therefore up to 4 KB, which is the size of the subtable, read accesses to the L2 cache are performed. As described before, this checking process for the non-leaf-level contraction is triggered only by the leaf-level contraction. The evaluation result in Sect. 6 shows that the frequency of the leaf-level contraction is small enough, thus the frequency of read accesses caused by checking the TAG fields for the non-leaf-level contraction is small and it does not decrease performance greatly.

5. Cache Hierarchy

The cache hierarchy of our system includes the dedicated L1 tag cache and the unified L2 cache, and the TMU placed between them as described in Sect. 3.2. This section gives details of the cache hierarchy of our system.

5.1 Level-1 Tag Cache

The cache hierarchy of our system has a dedicated L1 tag cache. This L1 tag cache uses virtual address for indexing and physical address for tag matching. This is similar to L1 cache of some microprocessors [8], [9] and it does not require flushing operations on context switching.

As described before, the Tag Line (TL) of the Tag Table has the same size as the level-1 tag cache (L1 tag cache). In other words, a TL is mapped directly to a line of L1 tag cache on a refill.

It is important this size matches the line size of the L1 tag cache. The modification of Tag Table is made when a line of L1 tag cache, to which some changes have been made, is written back to the L2 cache or the memory. 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 necessary to obtain the remaining bits for comparison. Such extra overhead is avoided in our design.

5.2 Tag Management Unit

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

The TMU handles the reading/writing of the Tag Table. Since TMU is placed between the L1 tag cache and the L2 cache, all its operations are triggered by the L1 tag cache miss.

The operations to the Tag Table includes the following:

- Tag Table Walk
- Expansion/contraction

5.2.1 Overhead of Tag Table Walk

The TMU does not have additional address translation mechanism such as a TLB, though the Tag Table is accessed by virtual address as described in Sect. 4. This is because the each part of virtual address is used as the index of an entry on each subtable. On Tag Table Walk, a pointer to the next level subtable does not require address translation, because the pointer obtained from the TTP field is physical address.

The overhead of the Tag Table Walk is mainly caused by the L2 cache access. In the case of Tag Table Walk, TMU needs to access the Tag Table on the L2 cache five times,

and this will cause a large overhead. Thus, we introduce a *Pointer Cache* to cache TTEs on the path of Tag Table Walks. The Pointer Cache provides the TTEs required for the table walk on hit.

Pointer Cache

Some MMUs (Memory Management Unit) of microprocessors have the pointer cache to cache the pointers included in the page tables to speed up the page table walks. The Pointer Cache of TMU caches TTEs just for the same purpose of conventional MMUs.

Pointer Cache is a small set-associative cache, whose value is TTEs and ETTEs. The Pointer Cache uses physical address because TTEs and ETTEs have pointers as physical address and the TMU accesses the Pointer Cache using these pointers on Tag Table Walk. When TMU needs to access the Tag Table, it first accesses the Pointer Cache. If the Pointer Cache misses, the TMU accesses the Tag Table on the L2 cache and refills the missed TTEs or ETTEs to the Pointer Cache.

6. Evaluation

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

6.1 Evaluation Environment

Our simulation uses a cycle-accurate processor simulator *Onikiri2* [10], developed in our laboratory. Unlike SimpleScalar Tool set [11], which is used widely for researches on processor architecture, Onikiri2 replays execution of instruction in the exact cycle when it is on the execution stage. Thus, Onikiri2 can simulate more precisely than SimpleScalar Tool set, when adopting data predictions such as address match/mismatch prediction. The parameters used for the simulation are shown in Table 2.

We used all 29 programs of the SPEC CPU2006 [12] benchmark with *ref* data sets. The programs were compiled using gcc 4.2.2 with "−O3" options. We skipped the first 3G instructions and evaluated the next 500M instructions.

To evaluate our system, we applied tags to every input data, and propagated them by rules shown in Table 3. This propagation rule is for tracking explicit flow of data and is used by most researches [2], [4]. The length of tag applied was 1 bit for a byte of data. The formats of TL and fifth-level subtable in this case are described in Sect. 4.2.

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.

Table 2 Architectural parameters.

Architectural parameters	Specifications
ISA	Alpha
fetch width	4 inst.
execution unit	int: 2, fp: 2, mem: 2.
instruction window	int: 32, fp: 16, mem: 16
register file	int: 128, fp: 128
L1 I/D cache	32 KB, 4 way, 64 B/line, 3 cycles
L1 Tag cache	4 ∼ 32 KB, 4 way, 64 B/line, 1 cycle
Pointer Cache	128 B ∼ 2 KB, 4 way, 16 B/line, 1 cycle
L2 cache	4 MB, 8 way, 64 B/line, 10 cycles
main memory	100 cycles

6.2 Result

Compared to architectures without tags, the following performance overheads are added to tagged architecture.

Memory overhead Extra memory consumed by tags.

Latency overhead Extra time to read/write tags. In our system, this is represented by a increase of memory access latency caused by tag access.

The following sections describe how our system reduces these overheads and shows the result of the evaluation.

6.2.1 Memory Overhead

We measured the total amount of memory consumed by the Tag Table. The measurement was taken from the average memory consumption for the execution. Figure 10 shows the result for all the benchmark programs.

The *naive* is a naive model of tagged architecture, which simply adds a tag field on each byte. For example, if 1-bit tag is added per byte, the overhead is statically $1/8 = 12.5\%$, independent of the amount of the memory consumed by programs.

The *nocont* is a model of our system without contraction, while the *cont* is one with contraction. The values on

Table 3 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

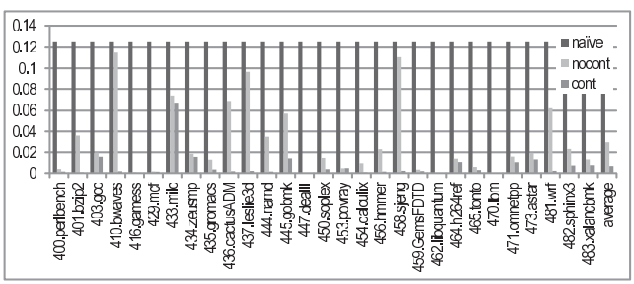

Fig. 10 Memory overhead with and without contraction.

the graph are the ratio of the amount of the memory allocated for the tags to that NOT for the tags.

Note that the value on the graph of the *naive* model (12.5%) is the ratio to the amount of total physical memory installed to the system, while the value of the *nocont* and *cont* is the percentage to the amount of memory used by the program.

The result shows that the memory overhead is significantly reduced from the *naive* model in both of our models. The overhead varies among the programs, because it strongly depends on the characteristics of each program. One factor that affects the overhead is the input data size, which determines the initial amount of tags. The behavior of the program, which determines how the tags are propagated, also affects the overhead.

The *nocont* and *cont* models show memory overhead of 2.97% and 0.685% in average, respectively. Thus we can see that the use of contraction significantly contributes to the reduction of memory overhead.

6.2.2 Latency Overhead

We evaluated the IPC degradation of our system from a baseline model which does not have tags.

Figure 11 shows the IPCs of our system with 1 KB Pointer Cache for various L1 tag cache sizes. The IPCs in this figure are normalized by the IPC of the base model and average of the programs in the benchmark. The average degradation is 6.01%, 3.55%, 1.13%, and 0.491% for an 4-, 8-, 16- and 32 KB L1 tag cache, respectively. This graph shows that 8- or 16 KB L1 tag cache is sufficient to maintain the performance.

Figure 12 shows the IPCs of our system with 8 KB L1 tag cache for various pointer cache sizes. The IPCs in this figure are normalized and averaged just like Fig. 11. The "none" label show the IPC degradation for a model without the Pointer Cache. The model without the Pointer Cache shows significant performance degradations, and the average performance degradation of the model is 22.6%. On the other hand, the performance degradations of the models with the Pointer Cache is considerably small. The average degradation is 8.20%, 5.55%, 4.24%, 3.76%, and 3.57% for 128 B, 256 B, 512 B, 1 KB, and 2 KB Pointer Cache, respectively. This result shows that a small capacity of the Pointer Cache significantly improves IPCs of benchmarks with large IPC degradation by the L1 tag cache miss.

Figure 13 shows the IPCs of our system with 8 KB L1 tag cache and 1 KB Pointer Cache for each benchmark programs. The IPCs in this figure are normalized just like Fig. 11. The average IPC degradation of our system is 3.55%. This graph shows that each IPC degradation is not much different between benchmark programs.

Figure 14 shows the L1 tag cache hit rate of our system for each benchmark program. Some programs show low cache hit rates. The cache hit rates of 410.bwaves and 447.dealII are 83.1% and 66.5%, respectively. This is because the line size of L1 tag cache is much larger than that of L1 data cache. The length of tag applied was 1 bit for a byte of data, thus one line of the L1 tag cache is corresponding to 8 lines of the L1 data cache. In this case, cache hit rates of programs with low spatial locality become very low.

These low cache hit rates do not always degrade the performance of the programs. This is because the hit rate of

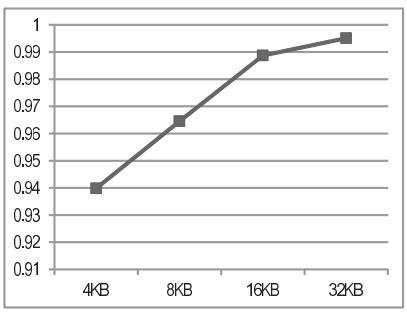

Fig. 11 Avg. relative IPC for various L1 tag cache size.

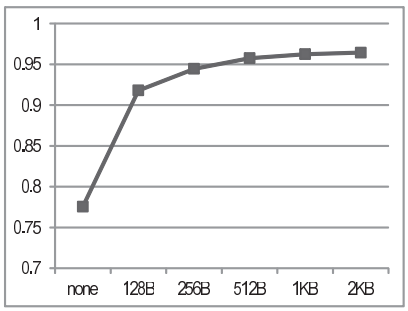

Fig. 12 Avg. relative IPC for various pointer cache size.

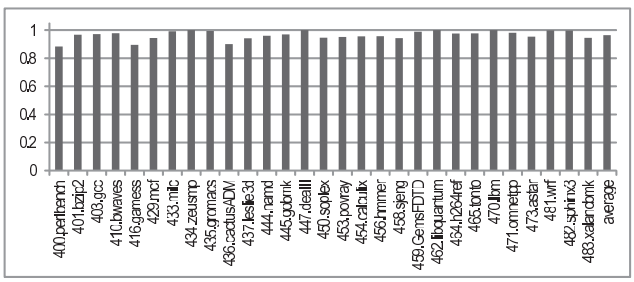

Fig. 13 IPC degradation for each benchmark program.

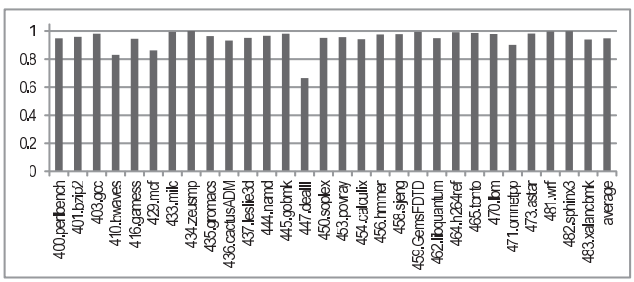

Fig. 14 Hit rates of L1 tag cache (8 KB).

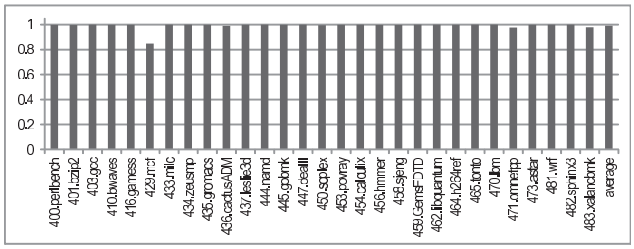

Fig. 15 Hit rates of Pointer Cache (1 KB).

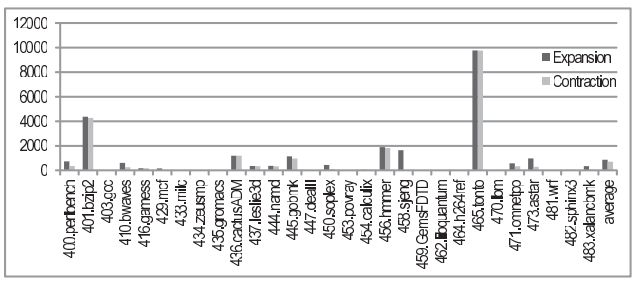

Fig. 16 Number of occurrences of expansion and contraction (leaf-level).

the Pointer Cache, which is accessed on the tag table walk caused by L1 tag cache miss, is usually quite high. Figure 15 shows the Pointer Cache hit rate of our system with 1 KB Pointer Cache. The average Pointer Cache hit rate over all the programs is 99.2%. The cache hit rates of 410.bwaves and 447.dealII, which show low L1 tag cache hit rate, are 99.9% and 100.0%, respectively.

Figures 14 and 15 show that the L1 tag cache and Pointer Cache hit rates of 429.mcf are 86.2% and 83.4%, respectively. These hit rates are relatively low, but Fig. 13 shows that the IPC degradation of 429.mcf is 5.56%, which is relatively small. This is because the L1 cache and L2 cache hit rates of 429.mcf for normal data access are very low, they are no more than 85% and 40%, respectively. These hit rates are much lower than those of the L1 tag cache and Pointer Cache, thus relatively low hit rates of the L1 tag cache and Pointer Cache do not have significant impact to performance in 429.mcf. Moreover, as shown in Fig. 10, the size of the tag table is much smaller than that of memory allocated by the program, thus the hit rate of L2 cache on Tag Table Walk is much higher than that of normal data access.

The above observation shows frequent accesses to the Tag Table affect the performance, and implies importance of the mechanisms to reduce the L1 tag cache miss penalty. The Pointer Cache is introduced to reduce accesses to the L2 cache, and consequently average access latency of the Tag Table.

Figure 16 shows the numbers of occurrences of expansion and contraction for leaf-level, respectively. We only show the results for leaf-level in the graph, because the numbers for non-leaf-level are very small, they are no more than 20 times among all benchmark programs. These numbers are much smaller than those of instructions executed, which is 500M instructions, thus additional cycles for expansion and contraction do not degrade the performance of the programs.

7. Conclusion

In this paper, we presented low-overhead security-tagged architecture. To achieve low overhead, our system exploits two characteristics of the tags, which are the Non-uniformity and the Locality of reference of tags. Our design uses a uniquely designed multi-level table and various cache-like structures to exploit these characteristics.

Simulation result shows that our system can significantly reduce the memory overhead compared to a naive implementation. The ratio of the memory used for the tags to that not for the tags is no more than 0.685% in average.

We also evaluated the latency overhead, and the IPC degradation to the base model which does not support tags is no more than 3.55% in average. We also observed that a small capacity of the Pointer Cache can significantly reduce L1 tag cache miss penalty.

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

We recognize the importance of the information flow tracking techniques, and believe our low-overhead implementation of them will contribute to information security.

References

[1] E.A. Feustel, “On the advantages of tagged architecture,” IEEE Trans. Comput., vol.C-22, no.7, pp.644–656, 1973.
[2] G. Suh, J. Lee, and S. Devadas, “Secure program execution via dynamic information flow tracking,” Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.85–96, 2004.
[3] N. Vachharajani, M.J. Bridges, J. Chang, R. Rangan, G. Ottoni, J.A. Blome, M.V.G.A. Reis, and D.I. August, “Rifle: An architectural framework for user-centric information-flow security,” Proc. 37th International Symposium on Microarchitecture (MICRO), pp.243–254, 2004.
[4] M. Dalton, H. Kannan, and C. Kozyrakis, “Raksha: A flexible information flow architecture for software security,” Proc. 34th International Symposium on Computer Architecture (ISCA), pp.482–493, June 2007.
[5] J. Crandall and F. Chong, “Minos: Control data attack prevention orthogonal to memory model,” Proc. 37th International Symposium on Microarchitecture (MICRO), pp.221–232, 2004.
[6] E. Witchel, J. Cates, and K. Asanović, “Mondrian memory protection,” Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.304–316, Oct. 2002.
[7] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer’s Manual, http://www.intel.com/products/processor/manuals/, March 2009.
[8] Sun microsystems, UltraSPARC User’s Manual, 1997.
[9] MIPS Technologies, MIPS R10000 Microprocessor User’s Manual, 1995.
[10] R. Shioya, M. Goshima, and S. Sakai, “Design and implementation of a processor simulator onikiri2,” Proc. Symposium on Advanced Computing Systems and Infrastructures (SACSIS), pp.120–

121, 2009.
[11] D. Burger and T.M. Austin, "The SimpleScalar tool set, version 2.0," Tech. Rep., University of Wisconsin-Madison Computer Sciences Department, 1997.
[12] The Standard Performance Evaluation Corporation, SPEC CPU2006 suite, http://www.spec.org/cpu2006/

Ryota Shioya was born in 1981. He is currently a Ph.D. student in Information and Communication Engineering in The University of Tokyo. He received his M.E. degree in Information and Communication Engineering from The University of Tokyo in 2008. He was a research fellow of the Japan Society for the Promotion of Science from 2009. He is a member of IPSJ.

Daewung Kim currently works for LG Electronics Inc. He received M.E. at The University of Tokyo in 2009. His field of research is processor architecture.

Kazuo Horio currently works for Fujitsu Laboratories Ltd. He received M.E. at The University of Tokyo in 2010. His field of research is processor architecture.

Masahiro Goshima was born in 1968. He received his M.E. in engineering and Ph.D. in informatics from Kyoto University in 1994 and 2004, respectively. He was a research fellow of the Japan Society for the Promotion of Science from 1994. From 1996, he was an assistant professor in the Graduate School of Informatics, Kyoto University. Since 2005, he has been an associate professor in the Graduate School of Information Science and Technology, the University of Tokyo. He has been engaging in the research area of computer architecture. He received IPSJ Yamashita SIG research award and IPSJ best paper award in 2001 and 2002, respectively. He wrote a book titled "Digital Circuits". He is a member of IPSJ and IEEE.

Shuichi Sakai was born in 1958, received B.S., M.S. and D.E. from the University of Tokyo in 1981, 1983 and 1986, respectively. He had been working at Electrotechnical Laboratory (1986–1998), Massachusetts Institute of Technology (MIT, 1991–1992), Real World Computing (RWC, 1993–1996), University of Tsukuba (1996–1998). In 1998, he became an Associate Professor of The University of Tokyo where he has continuously been a Full Professor since 2001. His major concerns are computer systems and their applications, especially computer architectures, interconnection networks, optimizing compilers, low power architectures and dependable systems. He received several awards, including IPSJ Best Paper Award (1991), IBM Science Award (1991), Ichimura Academic Award (1995), IEEE Outstanding Paper Award (1995), Sun Distinguished Speaker Award (1997). He is a member of IPSJ (IPSJ fellow since 2010), IEICE (chair of CPSY), JSAI, ACM and IEEE.