NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ANALYSIS OF INTEL IA-64 PROCESSOR SUPPORT FOR SECURE SYSTEMS

by

Bugra UNALMIS

March 2001

Thesis Advisor: Cynthia Irvine
Second Reader: Frederick W. Terman

Approved for public release, distribution is unlimited

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2001	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Analysis of Intel IA-64 Processor Support for Secure Systems			**5. FUNDING NUMBERS**
6. AUTHOR(S) Unalmis, Bugra			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT

Current architectures typically focus on the software-based protection mechanisms rather than hardware for providing protection. In fact, hardware security mechanisms can be critical for the construction of a secure system. If hardware security mechanisms are properly utilized in a system, security policy enforcement can be simplified. Systems could be constructed for which serious security threats would be eliminated.

This thesis explores the Intel IA-64 processor's hardware support and its relationship to software for building a secure system. To analyze the support provided by the architecture, hardware protection mechanisms were examined. This analysis focused on the following mechanisms: privilege levels, access rights, region identifiers and protection key registers. Since protection checks are made through the translation lookaside buffer (TLB) during the virtual-to-physical translations, the TLB structure was an area of focus throughout the research for this thesis.

Proper use of the TLB-based hardware protection features permits protection in the IA-64 architecture. It enables the processor hardware and the operating system to collaborate to enforce security policies efficiently.

14. SUBJECT TERMS Protection, Intel IA-64 architecture, Secure Systems.			**15. NUMBER OF PAGES** 100
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release, distribution is unlimited

ANALYSIS OF INTEL IA-64 PROCESSOR SUPPORT FOR SECURE SYSTEMS

Bugra Unalmis
Ltjg, Turkish Navy
B.S., Turkish Naval Academy, 1995

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN
ELECTRICAL ENGINEERING**

from the

**NAVAL POSTGRADUATE SCHOOL
March 2001**

Author: _____
Bugra Unalmis

Approved by: _____
Cynthia E. Irvine, Thesis Advisor

_____
Frederick W. Terman, Second Reader

_____
Jeffrey B. Knorr, Chairman
Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Current architectures typically focus on the software-based protection mechanisms rather than hardware for providing protection. In fact, hardware security mechanisms can be critical for the construction of a secure system. If hardware security mechanisms are properly utilized in a system, security policy enforcement can be simplified. Systems could be constructed for which serious security threats would be eliminated.

This thesis explores the Intel IA-64 processor's hardware support and its relationship to software for building a secure system. To analyze the support provided by the architecture, hardware protection mechanisms were examined. This analysis focused on the following mechanisms: privilege levels, access rights, region identifiers and protection key registers. Since protection checks are made through the translation lookaside buffer (TLB) during the virtual-to-physical translations, the TLB structure was an area of focus throughout the research for this thesis.

Proper use of the TLB-based hardware protection features permits protection in the IA-64 architecture. It enables the processor hardware and the operating system to collaborate to enforce security policies efficiently.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I very much appreciate this opportunity to express my gratitude to the people who made this research possible. I would especially like to thank my advisor Prof. Cynthia Irvine, for the technical guidance she provided throughout the course of this research. I would also like to thank my second reader, Frederick W. Terman, for his comments and suggestions regarding this research.

Finally, I also wish to thank my friends, Tolga, Kadir, Seval, Samuel, Ilker, and Arzu for their support and encouragement.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The purpose of this thesis is to explore the IA-64 architecture's support for building a secure and reliable system. The IA-64 supports the paged virtual memory model, where protection is applied to each page on a memory access. Actually, there are two different types of protection mechanisms provided by the IA-64 processor prior to accessing a page. The first one is the page access right bits in the TLB associated with each translation. This protection mechanism provides privilege level-granular access to a page. The second mechanism is the protection keys that allow domain-granular access to a page. A domain is defined by protection key registers and offers an efficient method for the operating system to control access rights to groups of pages.

A secure system must allow the processes to share data in a safe and efficient manner to be able to enforce a security policy reliably. To achieve this, the IA-64 architecture implements hybrid SAS/MAS models that combine unique RIDs for per process regions and shared RIDs with protection keys for per-page protection.

This analysis found that the IA-64 architecture protection features enable the processor hardware and the operating system to collaborate to enforce the security mechanisms efficiently among the processes.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

The architectural design goal of the IA-64 is to address an efficient hardware/software approach in which the processor hardware, the compiler, and the operating system cooperate with each other to deliver higher-performance systems [Ref. 18].

To support security, processor architectures must provide security mechanisms without sacrificing efficiency and performance. Currently, there are a limited number of computer systems available to process information with different levels of sensitivity. Also, there are various methods of providing protection associated with these systems. The main purpose of these methods is to permit the operating system to protect itself from misuse by applications. Misuse could involve the unauthorized modification of the operating system and its databases. It could also involve jumps into the operating system without the validation and code vectoring provided by carefully constructed entry points. To achieve protection for the operating system, a privilege bit is used by processors to distinguish between operating system and user modes. This is required since the operating system must protect itself from users and must have a means of distinguishing privileged from non-privileged execution sequences. There should also be a mechanism to protect processes from each other and the operating system from the processes. This is accomplished through the use of both the privilege bit and an address space isolation mechanism.

Today, virtual memory schemes are considered to be the building block of address space separation mechanisms in processor architectures. There are two different types of virtual memory models: segmented and paged virtual memories. Segmented virtual memory divides the memory into different size segments and supports the use of descriptor registers to control access to memory address spaces. Paged virtual memory generally provides protection through translation lookaside buffer (TLB) structures during virtual-to-physical address translations.

B. THESIS SCOPE

The purpose of this thesis is to explore the IA-64 architecture's support for building a secure and reliable system. The IA-64 supports the paged virtual memory model, where protection is applied to each page on a memory access. On the other hand, unlike traditional architectures, where one of two address space models is supported, the IA-64 processor hardware is designed to support both multiple address space (MAS) and single address space (SAS) models. In the IA-64 architecture, there is actually no "SAS operating system" or "MAS operating system" mode. The difference between the two operating system models is the encoded policy enforced by the operating system. In other words, the difference comes through the management of the region identifiers (RIDs) and protection keys by the operating system [Ref. 2].

In this thesis, support for SAS/MAS operating systems and the security features associated with these models have been surveyed. Moreover, this thesis provides an analysis of the protection mechanisms provided by the processor architecture that could be used to enforce security policies. In addition, hardware features provided by the architecture that may make security policy enforcement difficult are discussed.

C. ORGANIZATION OF THESIS

The remainder of this thesis is organized as follows:

Chapter II describes the security requirements (privilege bit, memory management, and restricted privilege) needed to build a robust and secure system. In addition, the protection mechanisms supported by the Intel x86 architecture are reviewed.

Chapter III gives an overview of the innovative features of IA-64 architecture such as speculation, predication, register rotation and the register stack engine. Additionally, important concepts regarding memory management and protection implemented by the processor architecture are explained (i.e. TLB organization and virtual addressing).

Chapter IV gives a detailed description of the protection mechanisms supported by the IA-64 architecture. Four protection mechanisms provided by the architecture are: privilege levels, address space isolation, access rights and protection keys.

Chapter V gives a summary of functional hardware protection features for protecting processes from each other and makes a comparison between MAS and SAS operating system modes. In addition, some challenges related to these architectural features are discussed.

Chapter VI summarizes the previous chapters and provides a conclusion for the thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

II. SECURITY REQUIREMENTS OF A SYSTEM

A. CONCEPTS FOR BUILDING A SECURE SYSTEM

As computer architectures become more sophisticated, new applications and features that support the manipulation and storage of information emerge almost daily. Control of access to stored information is a critical issue, as multiple system users should not necessarily be able to access all stored data, (e.g., a manager may need to access confidential information which is not made available to company employees). Thus, the computer itself, in order to protect sensitive information, should be able to support mechanisms that limit access to a company's confidential files.

Information protection within a computer system can be defined as the security techniques that control access to stored information by each currently executing process. A process can be defined as a program's living space; that is a running program plus any state needed to continue running the program. The goal of a secure system is to prevent any unauthorized use of information. Unauthorized access to resources of a system is not possible if the secure operating system works, as it should. Users should not be able to penetrate the operating system in order to circumvent security policy. To achieve protection in operating systems, three major concepts are introduced: status information, memory management and restricted privilege [Ref. 17, 10].

To achieve the first concept, a status flag or privilege bits is used to allow the system to work in different privilege modes. These bits permit the system to distinguish between operating system and user modes. In a system that provides protection, there should be at least two modes: system mode and a user mode. This is required since the

operating system should protect itself from users and must have a means of distinguishing privileged from non-privileged execution sequences. The following architectures are the examples of such systems supporting different modes of operation;

Intel x86: two-status bits (four modes)

Motorola 68000 family: one status bit (two modes: user and supervisor)

The second concept requires that a user process be able to use a portion of the CPU state but it should not be allowed to write or modify it [Ref. 9]. This is achieved using base/bound registers and user/supervisor mode bit(s). User processes need to be prevented from writing this state because if they could, the operating system would not have complete control over users' processes. Without this mechanism, a user process might change the address range checks and modify another process' address space.

The simplest hardware protection mechanism that controls exactly which part of the memory is accessible is called a descriptor register. This register contains two components: a base value and a bound value. Here, the base value represents the lowest numbered address that the program may use, and the bound value represents the number of locations beyond the base that may be used [Ref. 10]. User programs are not permitted to load or change descriptor registers. However, the operating system must be able to change these values so that it can switch between processes. This is achieved by implementing a privilege bit in the processor as described before. This prevents user programs from loading the descriptor register and isolation among processes is accomplished. A supervisor program runs with the privilege state bit 'ON', and controls the use of some special instructions called *privileged instructions*. Privileged instructions include operations such as input-output and memory management. Since these

instructions might have an effect on certain system registers, user programs must not execute them. Whenever a user program tries to execute a privileged instruction, program control is transferred to the supervisor program via traps or interrupts and the execution of such instructions is performed by the operating system. Supervisor programs are always allowed to turn the privileged state bit 'OFF' and transfer control to user programs.

Virtual memory provides a different approach to memory management. The most crucial use of virtual memory is to support address spaces that are larger than the physical memory. Virtual memory describes methods that give processes more memory than is physically available, or that make the computer appear to have more memory than is physically available. Since there are multiple processes running in the system, each with its own address space, it would be unreasonable to allocate a full physical memory address space for each process. "Here, there should be a way of sharing a smaller amount of physical memory between processes. One way to do this, virtual memory, divides physical memory into blocks and allocates them to different processes" [Ref. 9]. With virtual memory, the CPU produces virtual addresses that are later translated to physical addresses at runtime, which may be used to access main memory. In this approach, there should be a protection mechanism that restricts a process to its own address space. This will be explained in Section B.

The third concept, which protects the state of the privileged mode, requires that the privileged instructions cannot be executed by user programs. Attempts to execute such instructions transfer control to the supervisor mode at a pre-defined entry point,

generally referred to as the "trap," or "gate" [Ref. 10]. The return to user mode is like a subroutine return that restores the previous user/supervisor mode.

In addition to all these protection mechanisms, the system should also be able to verify the claimed identity of a user. This mechanism is called *authentication.* Authentication mechanisms require that there be a match between something the system knows and something the user knows and/or possesses. The solution comes through the use of passwords. During the login procedure to a system, the password entered by the user is compared against the entries stored in a password file. Login will succeed if a valid password is entered. Otherwise, it will fail. In this scheme, the password file is a piece of information that must be strictly protected by the system. A common problem with passwords is that they authenticate the user to the computer system but not vice versa. To overcome this problem, a mechanism called *trusted path* is implemented in operating systems. A trusted path establishes a direct communication between two entities (user, program, or hardware) on a host. This communication provides the required properties so that it cannot be intercepted by another entity and so that the two entities can mutually authenticate. In the absence of a trusted-path mechanism, malicious software can masquerade as "trusted" software to the user or can masquerade as the user to the trusted function [Ref. 13].

B. ESSENTIALS OF COMPUTER SECURITY

Active entities operating on information on behalf of the system's users are called *subjects.* Subjects can also be considered to be the executing processes in a particular domain. The domain may be defined, in part, by the mode of the system. An information repository in a computer system that the subjects may access is called an *object.* "A

domain of a process is defined to be the set of objects which the process currently has the right to access according to each access mode" [Ref. 8]. Here, access modes represent the permissions such as read and write for describing the access control to a particular object.

At the simplest level, a computer system consists of a processor, memory and input-output devices. Objects can be considered to be well-defined portions of memory within the system, such as segments. To be able to control the operations on each object, an access control mechanism is required. The abstract mechanism for this is provided by the Reference Monitor Concept. "The Reference Monitor is an abstraction that allows subjects to make references to objects, based on a set of current access authorizations" [Ref. 8]. The Reference Monitor makes references to an authorization database to ensure that policy is enforced on all subject attempts to access objects. In an audit trail, the Reference Monitor can record both granted and unauthorized access attempts. In this scheme, an authorization database identifies which subjects may access each stored object. The Reference Monitor implementation in a computer system must meet three requirements: completeness, isolation and verifiability [Ref. 8].

Completeness: The Reference Monitor must be invoked on every reference by a subject to an object.

Isolation: The Reference Monitor and its database must be protected from unauthorized alteration.

Verifiability: The Reference Monitor must be simple and understandable so that it can be completely analyzed to guarantee it performs its functions properly.

To meet the reference monitor requirements to some extent, different types of hardware and software mechanisms are implemented in the systems. One such mechanism to isolate processes' address spaces from each other is the virtual memory model. Virtual memory systems can be categorized into two classes: those with fixed size blocks, called *pages*, and those with variable size blocks, called *segments*. As mentioned before, with virtual memory, the CPU addresses must go through a translation structure from virtual to physical addresses. This is very useful since it provides the opportunity for the hardware to protect processes from each other. The simplest way of accomplishing this is to add access permission flags to each page or segment.

Today's architectures support page tables and fast address translation structures such as translation-lookaside buffers (TLBs) not only to store virtual to physical address translations but also to provide protection. The information for each page is called a page table entry (PTE). This entry is like a cache entry where a tag holds portions of the virtual address and the data portion of the page table entry holds a physical page number, and a protection field. When the page table entry is changed during context switches by the kernel for a virtual-to-physical address translation, that entry should be updated to change the protection information and to flush any hardware (i.e. TLB) that caches this information. With respect to context switches, there are two performance problems: first, the cost of translation lookaside buffer (TLB) misses caused by the address space change and second, the cost of saving and restoring state during TLB flushes.

To avoid TLB flushes during context switches and to enable more efficient isolation between processes, the IA-64 architecture provides new mechanisms such as

tagging TLB entries with region identifiers (RIDs) and protection keys. This will be explained in detail in Chapter IV.

C. PROTECTION IN X86

The Intel x86 architecture is of interest because the IA-64 can execute the full x86 instruction set. The Intel Pentium (II, III, IV) is a 32-bit microprocessor that supports two different modes: real mode and protected mode. In protected mode, in addition to paging, there are segment registers that are used to index into special tables. These tables are the Global Descriptor Table (GDT), which is global to all tasks; the Local Descriptor Table (LDT), which is one per task with only one LDT active at a time; and the Interrupt Descriptor Table (IDT), which is used to direct, interrupts [Ref. 14]. Each of the tables contains descriptors and an 8-byte structure that maps a memory region using the following attributes:

- Base address in memory
- Limit (If the effective address (base plus offset) is greater than this limit, the processor aborts the instruction and generates a protection violation exception).
- Control bits (present bit, granularity bit and 2-bit field privilege bits)
- Descriptor type (code or data, read-only or read/write, executable, or a special system segment)

A segment register contains a 13-bit index field, a 1-bit table description field (GDT/LDT), and a 2-bit protection level. Whenever a segment is addressed, the CPU checks the related descriptor and the control bits relating to that descriptor. If the required

control bits do not permit adequate access authorization for that particular operation, an exception is raised and control is transferred to the operating system for exception handling.

As previously mentioned, segment register descriptors are placed in either the GDT or the LDT. If only the GDT were implemented in the system, any task would be able to load another task's descriptor from the GDT since a task must be given the right to access any descriptors in a GDT. Hence, the need for The Local Descriptor Tables (LDTs) and privilege levels is obvious [Ref. 14].

An LDT is built by the operating system whenever a task is loaded in memory. A system might have several LDTs at the same time, but only one of them (indicated by the LDT register) can be active at a time. Writing the LDT register is, of course, a privileged operation.

Although LDTs are useful for isolating processes from each other, the use of LDTs by themselves still does not prevent a task from accessing arbitrary descriptors in the GDT. To prevent such an access, the Intel Pentium supports both ring mechanisms (0, 1, 2, 3) that can be used collectively with LDT structures and a multiple descriptor aliasing feature. "Multiple Descriptor Aliasing" will be explained later in this chapter. Level 0, the most privileged, is used for kernel code and the system databases by the operating system, whereas Level 3, the least privileged, is used for applications. Each descriptor provides a descriptor privilege level (DPL) to specify the segment's privilege and it is the operating system's job to appropriately set the DPL of descriptors to prevent unauthorized accesses to segments.

A problem may arise if the operating system wants to write into a particular segment whose descriptor indicates that it is not writable. The solution comes through the use of multiple descriptors, which are used to provide different views of the same segment. Hence, the same segment may be viewed as a code segment by the applications and as a data segment by the operating system. This feature is called *multiple descriptor aliasing* and it is used to accomplish sharing between separate processes that are supposed to access a segment with different access rights.

Another type of protection provided by the processor is a check of the offset field, which is used to prevent addressing beyond the segment's limit. If an attempt to address beyond the segment's limit is made, an exception is raised. This protection feature is useful for isolating processes from each other, since each process is limited to work on the segments in its address space.

In addition to LDTs and DPLs, task state segments (TSSs) are implemented by the architecture to hold all the registers of a task during context switches. In a fashion similar to the local descriptor tables, only one TSS can be active at a time. The TSS is described by a descriptor that indicates the base address, size, protection and the type, which, in this case, is a TSS. The selector of the currently running TSS is held in a special register called the Task Switch Segment Register (TR).

In spite of all the protection features explained above, without the use of a special gate mechanism, processor's behavior would still be ambiguous if a task with privilege level 3 seeks to execute code with a higher privilege level. In such cases, the task should make a call to the more privileged domain by using a special type of descriptor known as a *call gate*. The x86 architecture uses the DPL in the call gate to insure the invoker is

allowed to use the gate: the caller must be at least as privileged as the gate. It then switches to the privilege level indicated in the descriptor pointed to by the gate. Thus, an executing program must have enough privilege to use a call gate. For instance, if the call gate's privilege level is 1, only programs with privilege levels 0 and 1 can use the call gate [Ref. 14].

Call gates increase the task's privilege level while executing the program called through gate. However, it is possible to make a call to a less privileged segment if and only if the conforming bit field in the descriptor register is set. For instance, if a task executing in privilege level 1 calls a function in a segment with the conforming bit set and DPL 2, that function will execute at privilege level 1 during the call.

D. SUMMARY

Thus far, three protection mechanisms needed to build a robust and secure system have been described. First, the system should support supervisor and user modes by employing ring mechanisms or privilege levels. An operating system should run with the most privilege to control system software and hardware, whereas the applications should run with the least privilege. In addition, only a supervisor program should be able to use privileged instructions that might affect the processor's state. Thus, if an attempt is made to execute a privileged instruction in a non-privileged state, program control is transferred to the supervisor mode and the execution of such instructions is dealt with by the operating system.

The second mechanism requires that there be a controlled call, i.e. a well-defined entry point, to the more privileged function. This entry point is called a gate, and is initialized and controlled by the operating system, but at the same time used by the less

privileged elements within a task. Gates provide a secure method of privilege level transfer within a task by raising software interrupts to operating system functions.

The third mechanism is to support protection by restricting tasks from accessing, modifying or deleting anything exclusive to another address space. To achieve this restriction, each task must be allocated a separate region and must confine itself to its own address space. Segmentation offers effective protection in terms of allocating a separate address space for each task.

Another method to allocate a separate address space is called paging. The Intel x86 architecture provides a memory model that supports both segmentation and paging. Paging both supports segmentation's main advantages and offers a very large address space. In this scheme, each process has a very large address space that is composed of potentially thousands of pages, but only the pages in current use by the application must be in the actual memory.

In the next chapter, I will explain how the IA-64 supports these three protection concepts by expanding the current architecture with new features and approaches.

THIS PAGE INTENTIONALLY LEFT BLANK

III. INTRODUCTION TO IA-64 ARCHITECTURE

Today's microprocessors have certain features that limit their performance, such as mispredicted branches and memory latency. As the processor clock rates go up faster than cache and memory access speeds, a performance limitation is presented by the memory loads since there will be a long latency unless the data is already in the highest level cache.

In order to overcome such performance limitations, the IA-64 architecture integrates many new features and techniques, including predication, speculation and explicit parallelism. Predication and speculation address problems relating to branches and memory latency and explicit parallelism enables the compiler to execute more instructions per clock cycle.

In addition to these techniques and features, the IA-64's large register set (128 integer registers, 128 floating point registers), helps to minimize the negative effects of hardware complexity by transferring some of this complexity to the software. In the IA-64 architecture, general registers, floating point registers and predicate registers can be rotated by the compiler loop optimization.

The objective of this thesis is to present an analysis of the IA-64 processor's support for secure systems, and this chapter will give a brief overview of the innovations of IA-64 architecture.

A. INNOVATIONS OF IA-64 ARCHITECTURE

1. Predication

Traditional architectures typically suffer from mispredicted branches and the penalties incurred as a result of these misdirected branches. Conditional branches are difficult to execute since the processor must decide what instructions to fetch beyond a branch before it computes the condition for that branch. To mitigate such performance penalties, traditional processors use branch prediction and execute instructions while the branch direction and target are resolved. The processor speculates as to the outcome of the conditional and executes one sequence of the branch in advance. If the branch prediction is correct, the processor capitalizes on these speculatively executed instructions and there is a performance benefit. When the prediction is wrong, the processor must throw away all the speculatively executed instructions and must execute the correct set of instructions. To avoid such misdirected branches, the IA-64 provides a technique called predication that removes branching by using a predicate register. This feature transforms control flow dependency associated with conditional branching into a data dependency by making comparisons between the contents of the registers and writing the predicate registers based on the result of the comparisons.

There are several types of comparisons in the IA-64 architecture. A "normal compare" is a Boolean instruction that makes a comparison, and sets the first predicate register (p1) to the result, and the second predicate register (p2) to the complement of that result. For example;

cmp.eq p1, p2 = r1, r2

(p1) sub r5 = r9, r10

(p2) add r8 = r6, r7

The compare instruction generates two predicates (p1, p2). If GR[r1] is equal to GR[r2], then the instruction will set p1 to 1(true) and p2 to 0 (false). This means that the sub instruction will execute whereas the add instruction does nothing in this case.

The unconditional compare instruction differs from the conditional instruction as it writes 0 to both of its predicate registers if the qualifying predicate value is 0. If the qualifying predicate value is 1, then execution is exactly the same as for normal conditions. This compare instruction is used to remove "nested-if" conversions and enhances optimization [Ref. 3]. For example, consider the C code and the corresponding compiled code,

The C code is:

```
If (a>b)  {        \\ block 1
   c++;            \\ block 2
} else {
   d += c;         \\ block 3
   if (k==m)  {  \\ block 3
       n++;       \\ block 4
   } else  {
}
```

```
        p += n;   \\ block 5
    }
```

The compiled code is:

```
cmp.gt    p1, p2 = ra, rb
(p1)  add rc = rc +1
(p2)  add rd = rd, rc
(p2)  cmp.eq.unc  p3, p4 = rm, rk
(p3)  add rn = rn, 1
(p4)  add rp = rp, rn
```

If a is greater than b, the first compare instruction will set p1 to 1, and p2 to 0 and the unconditional compare will set both p3 and p4 to 0 since the qualifying predicate, p2, is 0. In this schema, there will be no branch mispredictions since there are no branches.

Three other types of comparisons, AND, OR and ANDOR, reduce execution cycles due to computations of complex branches, thereby helping to increase instruction level parallelism.

2. Speculation

Memory latency is one of the biggest performance limitations in traditional architectures. Since memory speed is much slower than processor speed, the processor must load data from memory as early as possible to guarantee that the data is available when needed. To overcome memory latency problems, the IA-64 processor uses a technique called *speculation* to allow the compiler to schedule loads much earlier, even

before it is known that the data contained in these loads will be needed in the flow of the program.

However, there are two problems in scheduling loads in advance. One of them is the unpredictability of address aliasing with prior store instructions. This means that a load instruction cannot be executed prior to a store instruction, because the store instruction might address the same memory location as the load instruction. Second, if the memory access fails, the load instruction should raise an exception, but the program does not actually know whether the flow of the program will need that load instruction [Ref. 3]. Thus, an exception should not be raised until it is clear that the program requires that instruction. For example, when a speculative load instruction is executed, if the memory access is successful then the value is put into the specified register as if it had been a regular load instruction. If the memory access fails, instead of raising an exception as would have occurred when a regular load instruction is executed, the speculative load instruction sets a deferral token in the target register. It is the compiler's responsibility to construct code that will examine the deferral token later and, if necessary, execute recovery code. This is useful because if a regular load instruction is used, the operating system has to be involved each time the memory access fails without knowing whether the load is actually needed. In contrast, in the case of a speculative load, the operating system is interrupted only if the load instruction is actually used.

To bypass these two potential difficulties, the IA-64 employs two different types of speculation: control speculation and data speculation.

a. Control Speculation

IA-64 employs speculative load instructions (ld.s, ldf.s, ldfp.s) and speculative check instructions (chk.s) to address the execution of exceptions at the time control flow of the program requires them. In cases where a regular load instruction raises an exception when the memory access fails, the speculative load instruction sets a deferral token called the NaT (Not a Thing) bit in the target register [Ref. 3]. When the chk.s instruction is executed, the NaT bit in the target register is checked. If it is '1', the control flow of the program is directed to the recovery code where a regular load instruction is executed. A NaT bit of '0' indicates that the speculative load instruction succeeded, and no additional operation is performed.

Load instructions can have as their targets either general purpose registers or floating point registers. For general registers, the deferral token is called a NaT bit. When floating point registers are addressed, the deferral token is called a NaTVal (Not a Thing Value) bit. Because of the addition of the deferral token, the actual length of general registers is considered to be 65 bits [Ref. 3]. Floating point registers are 82-bits long. In this case, NaTVal is the special encoding of these 82-bits.

If more than one speculative instruction is used in the program, NAT and NaTVal bits corresponding to these loads propagate during arithmetic instructions, and are checked only once with a speculative check instruction as shown below [Ref 3].

ld8.s r1 = [r5]

ld8.s r2 = [r6]

sub r4 = r1, r2

(p1) br.cond label

chk.s r3, recover

b. Data Speculation

Data speculation is another type of speculation that eliminates the obstacle encountered when a load instruction is moved ahead of a store instruction (i.e., that the instructions might address the same space in the memory). To address this problem, the IA-64 employs advance load instructions (ld.a, ldf.a, ldfp.a), and a check load instruction (ld.c) at the original location in the code of the load to test to see if a store conflicted with the associated load instruction. Whenever an advance load (ld.a) instruction is executed, an entry is made on a special address table called an *Advance Load Address Table* (ALAT). "When a ld.a inserts the entry into the ALAT, it first checks for any existing entry with the same register number, and overwrites it, if found. Thus there can only be one entry in the ALAT for any given register. This entry is used to detect collisions with stores" [Ref. 3]

ALAT consists of three fields: register number, physical address and the size of the load instruction. This mechanism is useful for finding possible collisions between stores and loads. When a store instruction is executed, a search with the physical address of the store instruction is started through the ALAT table, which makes a comparison with every load entry in the table. If a matching entry is found, the corresponding load instruction will be discarded from the ALAT table. Accordingly, when a check load (ld.c) instruction is executed, it will search for an entry for the load

instruction. If it finds the entry, then data speculation has succeeded. If a store instruction between ld.a and ld.c had modified the same memory address space, which the ld.a instruction loaded, that store would have removed the associated ALAT entry. In this case, ld.c instruction does a normal load and execution continues [Ref 3].

Another instruction called "Speculative Advanced Load," combines speculative load and advance load. This instruction is useful for detecting both load-store conflicts and deferred exceptions. Its application is the same as that of normal data speculation, however, the ALAT entry for the target register is discarded from the table if an exception is raised associated with the load instruction.

3. Register Rotation

Loop optimization is one of the crucial performance criteria in today's architectures because of its ability to reduce code expansion and overcome overhead problems. "Branching takes time and since the compiler knows ahead of time what the branches are, many branches can be avoided by simply duplicating the code in the loop two, four, or even more times to reduce the number of branches and allow greater parallelism. This is called "unrolling" the loop and is supported in both IA-64 and RISC architectures" [Ref. 5]. The problem with this scheme is that the set of virtual registers that are duplicated must use a different set of physical registers to avoid possible collisions. Even though the IA-64 supports a large set of registers, these duplicate instructions may still cause a code expansion.

To enhance loop optimization, the IA-64 implements a software register rotation mechanism which is applicable to general registers, floating point registers, and predicate registers. A register rotation base (RRB) field in the CFM (Current Frame Marker)

register holds the base value for renaming of registers, which provides every iteration with its own set of registers. This type of register renaming is called register rotation. In each rotation, register addresses will increase by one, (e.g., for register 55, the next rotation would use register 56). A special branch instruction (br.ctop) decrements the RRB value in the CFM register at the end of each iteration.

Register Rotation is the base of modulo-scheduling (i.e. new iterations can be started before the last one has finished). Modulo scheduled loop execution, as shown in Figure 1, increases the parallelism by allowing multiple iterations to be overlapped and contains 3 code sections: Prolog, Kernel and Epilog. The prolog instructions fill the software pipeline. Software pipelining is analogous to hardware pipelining but is created in code by the compiler. The epilog instructions that terminate the loop drain the software pipeline and the kernel code starts and finishes one iteration as shown in Figure 2. "A predicate is assigned to each stage to control the activation of instructions in that stage. To support the pipelining effect of predicates and registers, a fixed sized area of the predicate and floating point register files (PR16-PR63 and FR32-FR127), and a programmable area of the general register file, are defined to rotate" [Ref. 1]

Cycle	Load	Store	Branch
1	ld1		
2	ld2	st1	br.ctop1
3	ld3	st2	br.ctop2
4	ld4	st3	br.ctop3
5	ld5	st4	br.ctop4
6		st5	br.ctop5

Figure 1. Modulo-Scheduled Loop Execution Sequence

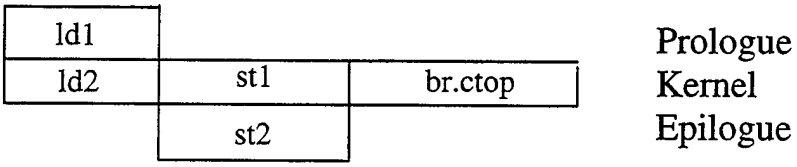

Figure 2. Schedule of Modulo-Scheduled Loop from [Ref. 3]

4. Register Stack Engine (RSE)

In the IA-64 architecture, general registers are divided into two subsets. A static subset (GR-0 through GR-31) is visible to all processes. A stacked subset (GR-32 through GR-127) is local to each process. The static subset must be saved and restored on context switches. The stacked subset is automatically saved and restored by the Register Stack Engine (RSE). "The Register Stack is implemented by renaming the register addresses as a side effect of procedure calls and returns" [Ref. 1]. By renaming the registers, there is no need to spill (store a register to memory) and fill (load a register from memory) the register contents to memory on subroutine calls and returns. The callee can use available registers without spilling and restoring the caller's registers.

The registers in a stacked subset are called a register stack frame. The frame is further divided into two parts: a local area and an output area. After a call, the size of the local area of the new frame is set to zero and the size of the output area is set to the size of the caller's output area. The local and output areas can be resized by using the *alloc* instruction [Ref. 2]. Figure 3. shows the register stack behavior on procedure call and return.

"The IA-64 register stack engine moves registers between register stack and the backing store in memory without explicit program intervention. The backing store is

designed as a stack in memory and a special register called Backing Store Pointer (BSP) contains the address of the first memory location reserved for the current frame (i.e. the location at which GR32 of the current frame will be spilled)" [Ref. 2]. Another register associated with the RSE, called the BSPSTORE, contains the address at which the next RSE spill will occur.

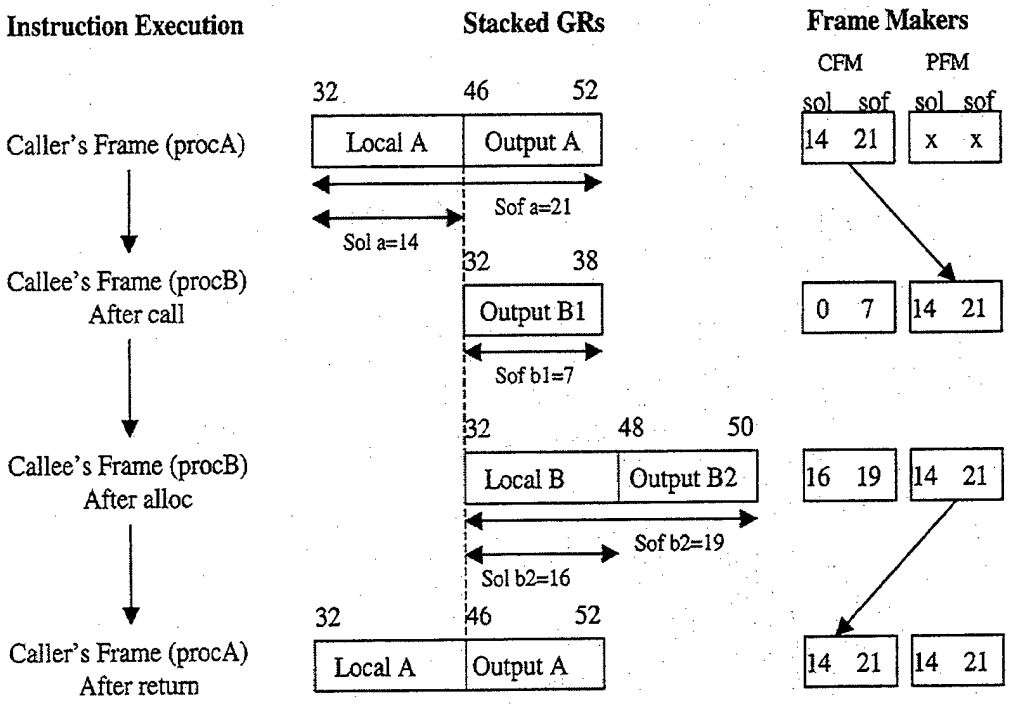

Figure 3. Register Stack Behavior on Procedure Call and Return from [Ref. 2]

B. IA-64 SYSTEM ARCHITECTURE FEATURES

1. Support For Multiple Address Space (MAS) Operating Systems

In multiple address space operating systems, process isolation is enforced among processes by allocating a separate address space for each process. The IA-64 employs region identifiers (RIDs) to distinguish between different address spaces.

In the IA-64 architecture, a 64-bit virtual address space consists of 3 fields: Virtual Region Number (VRN), Virtual Page Number (VPN), and Page Offset [Ref. 2].

The upper three bits of the 64 bit virtual address space selects one of the eight region registers. Each register contains a 24 bit RID. Using these identifiers, an operating system can map up to 2^{24} virtual regions. Regions are used to share data between processes by loading the same RID into the region registers of multiple processes. Each process can have up to eight region identifiers.

2. Support for Single Address Space (SAS) Operating Systems

Single Address Space (SAS) operating systems allocate only one large virtual address space for all processes. They are useful for large multiprocessor systems, since data sharing between processes is more efficient than between multiple address space systems.

"In a single address space system, protection keys permit domain granular access to a page. This is useful for mapping shared code and data segments in a globally shared region, and for implementing domains in a single address space (SAS) operating system" [Ref. 2]. Unlike multiple address space systems, the single address space systems extends

the virtual address space to 2^{85} bytes since the 24 bit region identifiers are considered to be the upper bits of the global address space.

C. IA-64 EXECUTION ENVIROMENT

1. General Registers (GRs)

IA-64 employs 128 (gr0-gr127) general purpose registers for integer computations. Although each of these registers is 64 bits long, the additional NaT bit that is used to raise exceptions can be considered as the 65^{th} bit.

General registers 0 through 31 are called static general registers. These registers are used during procedure calls and are preserved by callers and callees at all privilege levels. GR0 always reads zero if addressed as a source operand, and writing to this register causes an illegal operation fault.

General registers 32 through 127 are called stacked general registers, and these registers are local to each process. The stacked registers allocated to a particular process are called a register stack frame, and they can be renamed for loop optimization purposes.

2. Floating Point Registers (FRs)

In addition to general registers, the IA-64 also employs 128 (fr0-fr127) 82 bit floating point registers. Floating point registers 0 through 31 are called static floating point registers. Of these registers, FR0 always reads +0.0 and FR1 always reads +1.0. Neither of these registers can be used as a destination register, otherwise a fault is raised [Ref. 1].

Registers from 32 to 127 are called rotating floating point registers, and can be renamed for "Modulo-scheduled loop support" [Ref. 1].

3. Predicate Registers (PRs)

The IA-64 provides 64 (pr0-pr63) 1 bit predicate registers to store the results of compare instructions. Predicate registers 0 through 15 are called static predicate registers, and are used for conditional execution (predication).

Registers 16 through 63 are called the rotating predicate registers, and can be renamed for "Modulo-scheduled loop support" [Ref. 1].

4. Branch Formats (BRs)

IA-64 employs 8 (br0-br7) 64 bit branch registers for indirect branches. There are two types branch formats used in the IA-64 architecture. One is called *indirect branch format* and uses branch registers to assign a value to the Instruction Pointer. The other, *the IP relative branch format*, does not use branch registers. In instances where the latter format is used, the instruction pointer is incremented by a 21-bit offset field [Ref. 1].

5. Instruction Pointer (IP)

In the IA-64 architecture, three instructions are grouped together into 128-bit containers called *bundles* [Ref. 1]. A bundle consists of three instructions of 41 bits each and a template field of 5 bits. Each instruction occupies the first, second, or third syllable of a bundle. The IA-64 provides six types of instructions and four types of excution units (instruction, memory, floating point, and branch). The template field is used by the processor to quickly decode the bundle, and issue the instructions to the execution units. This field also restricts the instruction combinations in a bundle. The Instruction Pointer (IP) holds the address of the currently executing bundle. The contents of the IP change as

instructions are executed. This register cannot be written directly, and can only be read by the privileged *mov ip* instruction.

6. Current Frame Marker (CFM)

When a function call (br.call) is made, the 38 bit Current Frame Marker (CFM) is copied to the previous frame marker field of the Previous Function State Register (PFS). The current frame marker is then assigned to a new value with a set of output registers known as the *caller's output registers*, and all register rename base registers (RRBs) are subsequently assigned to '0'. The Frame Marker Format is shown in Figure 4 and the fields are described in Table 1.

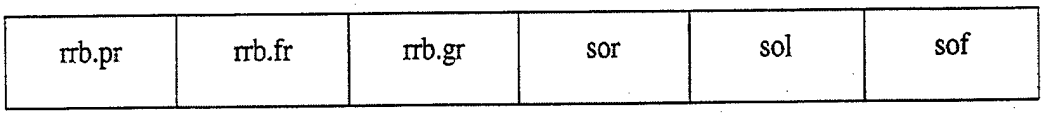

Figure 4. Frame Marker Format from [Ref. 1]

Field	Bit Range	Description
sof	6:0	Size of stack frame
sol	7:13	Size of local portion of stack frame
sor	14:17	Size of rotating portion of stack frame
rrb.gr	18:24	Register Rename Base for general registers
rrb.fr	25:31	Register Rename Base for floating point registers
rrb.fr	32:37	Register Rename Base for predicate registers

Table 1. Frame Marker Description Fields from [Ref. 1]

7. Application Registers (ARs)

Application Registers consist of special data registers and control registers for both the IA-32 and IA-64 instruction sets are shown in figure 5.

a. Kernel Registers (KR0-7)

IA-64 provides eight 64-bit kernel registers that direct information from the operating system to the application. When the privilege bit is not zero, writes to these registers cause a Privilege Register Fault; however, they can be read at any privilege level.

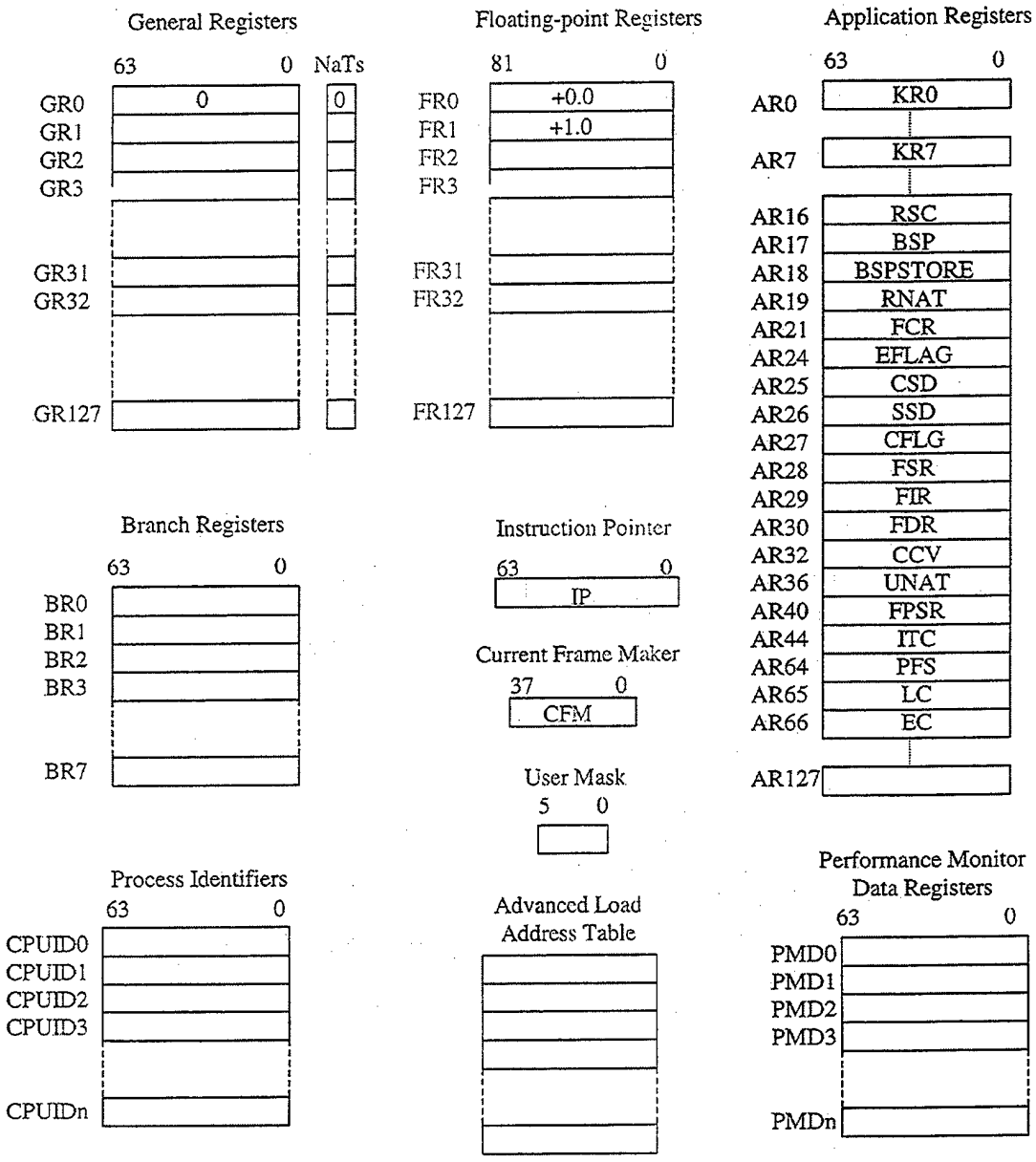

Figure 5. Application Register Set from [Ref. 2]

b. Register Stack Configuration Register (RSC)

The Register Stack Configuration Register controls the IA-64 Register Stack Engine (RSE). "Two mode bits field in the RSC register determine when the RSE

generates register spill or fill operations" [Ref. 1]. Instructions that change the contents of this register can not set the current process's privileged level to a more privileged level.

c. RSE Backing Store Pointer (BSP)

The IA-64 employs a 64-bit RSE Backing Store Pointer to designate the address space in memory for the current frame [Ref. 1]. See " Register Stack Engine (RSE)" in Section A.

d. RSE Backing Store Pointer for Memory Stores (BSPSTORE)

The BSPSTORE application register provides the address space where the next RSE spill will occur. See " Register Stack Engine (RSE)" in Section A.

e. RSE NaT Collection Register (RNAT)

A 64-bit RSE NaT Collection Register is used to collect NaT bits whenever RSE spills a register to the Backing Store in memory [Ref. 1].

f. Compare and Exchange Value Register (CCV)

This application register is used to hold the compare value for compare and exchange (cmpxchg) instruction [Ref. 1].

g. User NaT Collection Register (UNAT)

A 64-bit UNAT register is used by ld8.fill and st8.spill instructions to hold NaT bits [Ref. 1].

h. Floating Point Status Register (FPSR)

FPSR provides information and control for floating point operations such as traps, flags, overflows and rounding control [Ref. 1].

i. Interval Time Counter (ITC)

64-bit ITC register is used for timing. It is incremented at a submultiple of the processor clock frequency. ITC can be secured by software from non-privileged accesses and can only be written at the highest privilege level 0 [Ref. 1].

j. Previous Function State (PFS)

The Previous Function State Register consists of five fields: Previous Privilege Level (ppl); Previous Epilog Count (pec); Previous Frame Marker (pfm); and two reserved fields. On a subroutine call (br.call), CFM, EC and PSR.cpl (current privilege level in the Processor Status Register) are directly copied to the PFS Register, and the old contents are discarded. " When an IA-64 br.ret instruction is executed, the PFS is copied to the CFM and PFS.ppl is copied to PSR.cpl, unless this action would increase the privilege level" [Ref. 1]. The Previous Function State Register is saved and restored during context switches.

k. Loop Count Register (LC)

A 64-bit Loop Count Register is used in IA-64 counted loops. Traditional architectures use branch history to predict the next outcome. If the branch was taken more than once or twice, it will always be predicted taken next time. In this scheme, the last iteration of a loop will always be mispredicted. This creates a problem where loops are to execute a small number of iterations, since the branch prediction mispredicts the last iteration everytime [Ref. 3]. To overcome this limitation, the IA-64 provides a special register (LC) and an instruction (cloop) for loop closing branches. The compiler generates an instruction that assigns the number of iterations to the LC register. The loop

The loop closing branch instruction (cloop) checks the value in the LC register. If the value is zero, then the next instruction in the sequence is executed. If the value is not zero, then the LC register is decremented by one and a new iteration starts from the beginning of the loop [Ref. 3].

l. Epilog Count Register (EC)

A 6-bit Epilog Count Register is used to count the termination stages of Modulo-Scheduled Loops [Ref. 1].

m. Performance Monitor Data Register (PMD)

Performance Monitor Registers provide values from the monitors and are useful for operating systems and system performance. Writes to these registers are allowed only at the most privileged level whereas reads are allowed at all privilege levels [Ref. 1].

D. IA-64 VIRTUAL ADDRESSING

IA-64 architecture provides a 64 bit virtual address space that is divided into 8 virtual regions. Each virtual region can be selected by the upper three bits of the virtual address. Each of the eight virtual regions contains a 24 bit region identifier (RID) which can be mapped by the operating system onto the system's 2^{24} virtual regions. RIDs help translation lookaside buffers (TLB) hold translations from many address spaces at the same time [Ref. 2]. See "Translation Lookaside Buffer (TLB)"

A single virtual address consists of three fields: Virtual Region Number (VRN), Virtual Page Number (VPN) and Page Offset as shown in Figure 6. When memory is accessed, the VRN selects the region identifier from one of the eight region registers

under operating system control and then a search is initiated through the TLB for the matching RID and VPN. If there is a matching entry in the TLB, the associated physical page number is concatenated with the page offset to form the physical address. If the entry is not found in the TLB, an optional search can be initiated through the memory resident virtual hash page table (VHPT). If the entry is not found either in the TLB or the VHPT, an exception is raised and the operating system provides the necessary translation [Ref. 2]. The general order of searching the TLB and VHPT is shown in Figure 7.

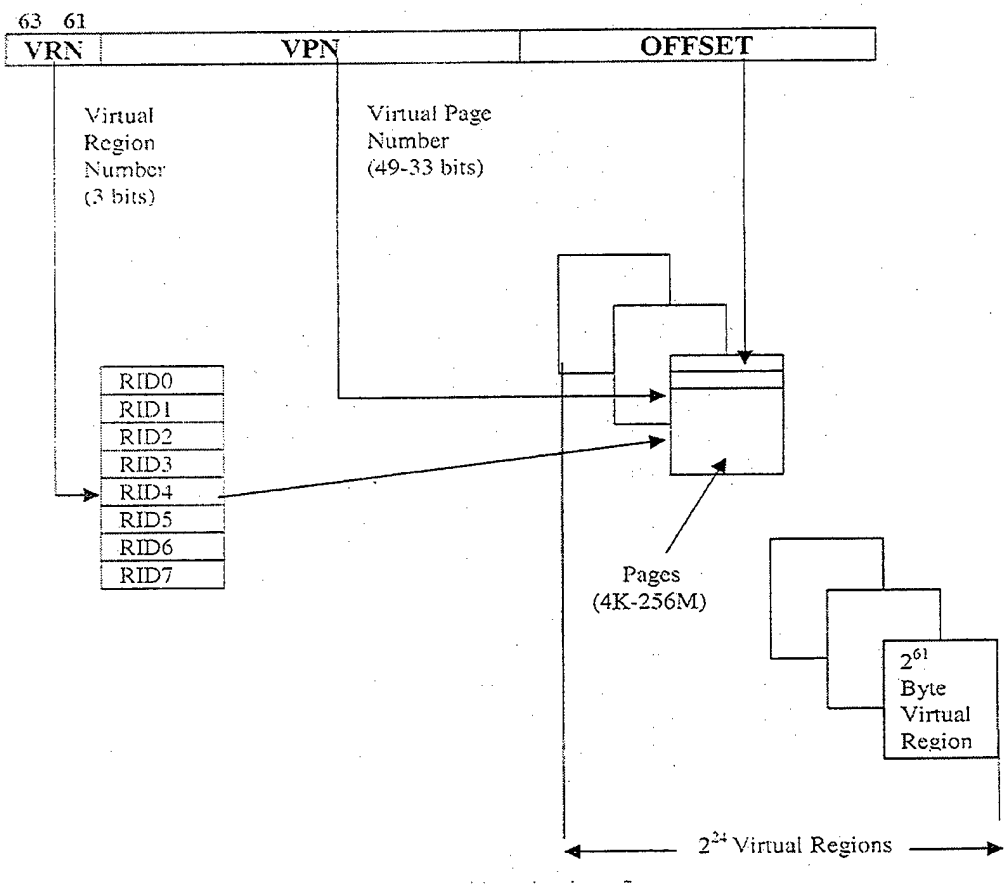

Figure 6. Virtual Address Spaces from [Ref. 6].

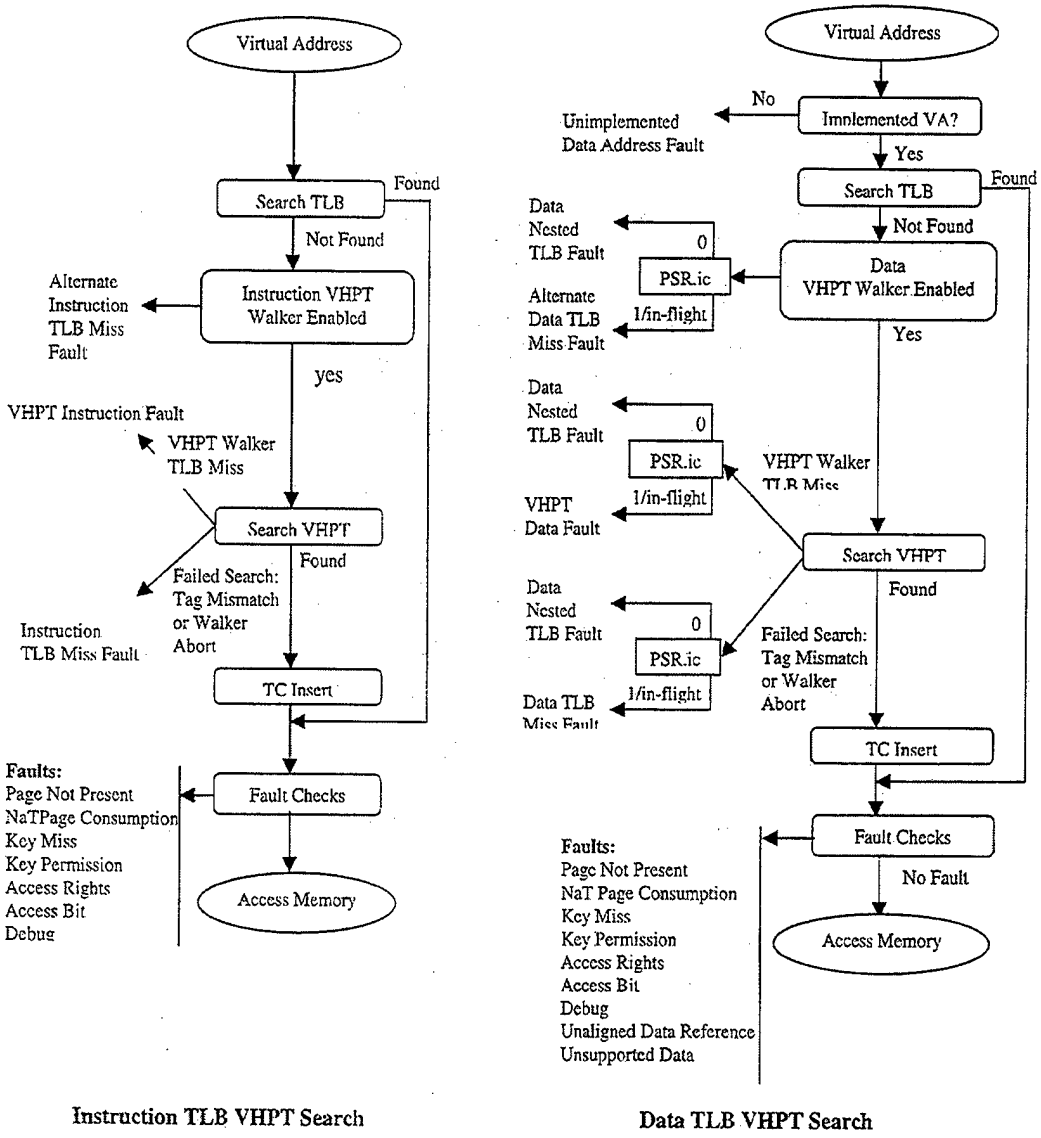

Figure 7. TLB/VHPT Search from [Ref. 2]

1. Translation Lookaside Buffer (TLB)

Similar to most architectures, the IA-64 provides translation lookaside buffers (TLBs) that are local to each processor. These buffers hold recent virtual to physical page translations. There are two architectural TLBs maintained by the processor: the instruction (ITLB) and data (DTLB). Further, each TLB is divided into two subsets called translation registers (TRs) and translation caches (TCs) that are used to hold different types of translations. Figure 8 shows the TLB organization.

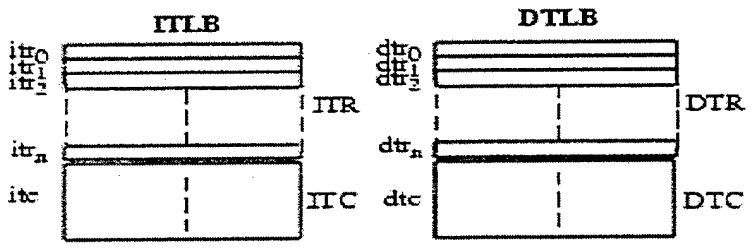

Figure 8. TLB Organization from [Ref. 2]

The 24 bit region identifier and virtual page number (33-49 bits) of the virtual address are used to look up TLB entries. At the end of the TLB lookup, the physical page number and access privileges are generated for a TLB hit. Memory protection on the basis of access rights, privilege levels and protection keys are supported by the processor as shown in Figure 9. See Chapter IV for details.

Translation registers managed by software are basically used to map large and static virtual memory translations such as kernel memory spaces, page tables and interrupt vector tables. Translation register slots are filled according to register numbers by using the insert translation register (itr.i, itr.d) instructions at the highest privilege

level. Once a translation entry is inserted into a TR slot, it will remain there unless it is overwritten by software or it is purged.

Two address translations are said to overlap when one or more of the same virtual addresses are mapped by both translations. In this context, "overlap" refers to two translations with the same region identifier. Software must check all TR entries to make sure that there are no overlapping translation before filling a TR slot. Otherwise, either undefined behavior or an exception occurs.

Translations are removed from the TRs by specifying a virtual address, page size and a region identifier that are used as parameters by the purge translation register (ptr.i/ptr.d) instructions.

There should be a minimum of 8 instruction and 8 data TR slots implemented in software for a particular process. Translation registers support all implemented page sizes and are not directly readable.

Translation caches (TCs) use the same format as TRs, but can be managed by both software and hardware. The hardware management of translation caches is achieved when the VHPT walker is enabled for a translation after a failed TLB search. In this case, if the required translation is found in the VHPT, it is installed into the translation cache that resides in the TLB used by the VHPT walker. In addition, TCs are used to hold smaller and more dynamic translations. TC slots are also filled by using insert translation cache (itc.i, itc.d) instructions. Insertions always purge overlapping entries.

There are three types of purge instructions used for translation caches. The most common type is the local TC purge instruction (ptc.l). It is used to remove translation

cache entries in the local processor TLBs that match a specified virtual address range and region identifier [Ref 2].

In order to purge all entries from the local processor's instruction translation cache (ITC) and data translation cache (DTC), software uses a series of purge translation cache entry (ptc.e) instructions. The IA-64 uses purge translation cache global (ptc.g, ptc.ga) instructions to purge translation cache (TC) entries matching a specified virtual address and a region identifier from all processors in a TLB coherence domain. A TLB coherence domain represents a group of processors that is on the same bus in a multiprocessor system. These instructions obviate the need for performing inter-processor interrupts to maintain TLB coherence in a multi-processor system [Ref. 2].

The Ptc.g and ptc.ga instructions present a release operation; all memory references prior to the ptc.g instruction are made visible to all procedures before the ptc.g is executed. Even though use of the ptc.ga instruction is almost identical to a ptc.g instruction, the ptc.g instruction does not modify the page tables nor any other memory location. However, the ptc.ga instruction removes any *Advance Load Address Table* (ALAT) entries that are in the address range specified for the ptc.ga instruction from all processors' ALATs. This task is performed to ensure that old ALAT entries are discarded whenever a translation is remapped to a different physical address.

All processor models should support a minimum number of 1 instruction and 1 data translation cache register. In some cases, unused TR slots can be used for TC entries by the processor.

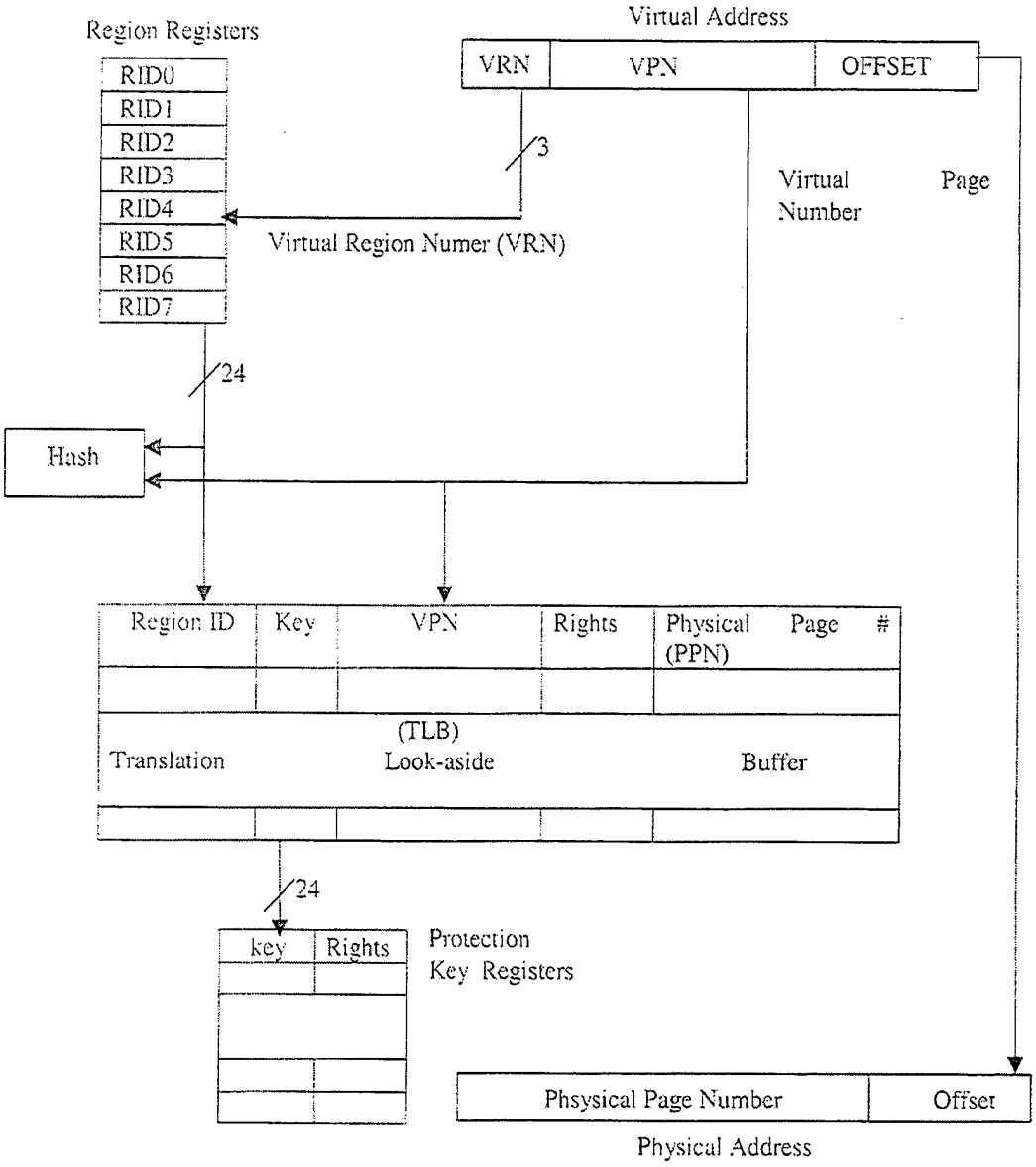

Figure 9. Virtual Address Translation

2. Virtual Hash Page Table (VHPT)

IA-64 provides a per processor base structure called a *Virtual Hash Page Table* to enhance the performance of the TLB. VHPT resides in virtual memory and can be searched by the processor. Figure 10 shows Virtual Hash Page Table configuration. If a

TLB miss occurs, the processor searches the VHPT for a matching entry by enabling the VHPT walker. If the matching entry is found in the table, the hardware tablewalker inserts the entry into the translation cache (TC). If additional TLB misses occur during the VHPT search, a VHPT translation fault is raised. If the entry is not found in both the TLB and VHPT, a VHPT Fault or a TLB Miss Fault is raised. A more detailed view of TLB/VHPT search is shown in Figure 7. "Operating system must regard the VHPT walker as a performance optimization and must be prepared to handle TLB misses if the walker fails" [Ref. 2].

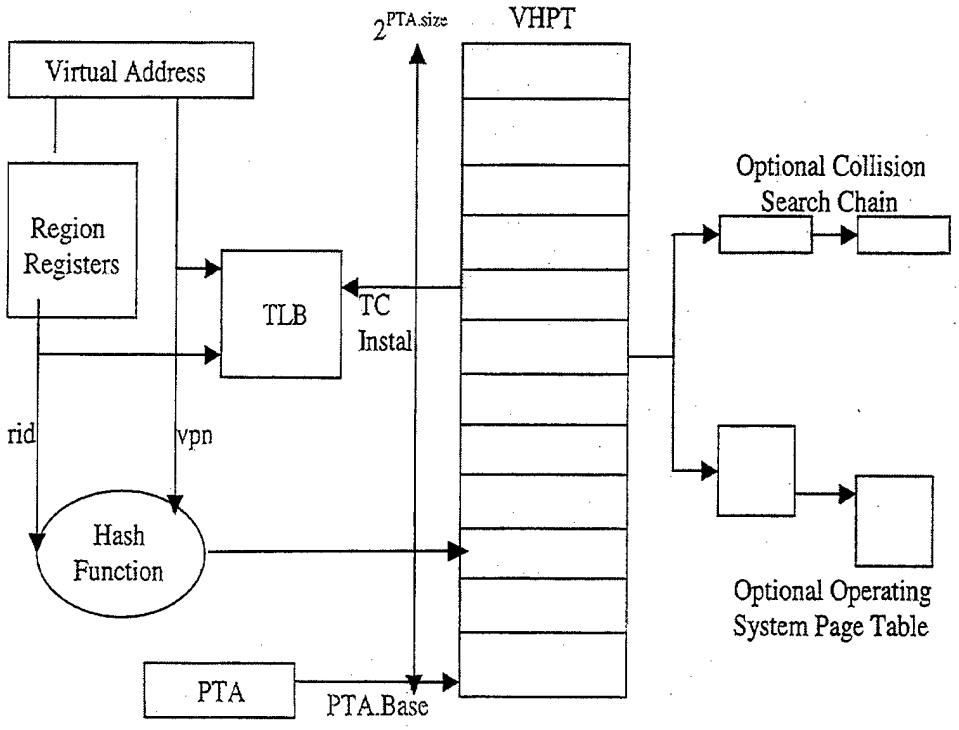

Figure 10. Virtual Hash Page Table (VHPT) from [Ref. 2]

The VHPT can be designed in two different ways, as a linear page table or as a hashed page table. The linear page table uses 8-byte page table entries (PTEs), and is indexed only by the virtual page number (VPN). This format is used with applications running on Multiple Address Space (MAS) operating systems, and can not use protection keys due to the insufficient number of page table entry (PTE) bits for the keys. In this scheme, the per region page table is in the same address space as the actual virtual address being translated.

An alternative page table format is called a hashed page and uses 32-byte PTEs. This format employs protection keys for data sharing and access control purposes, and is used with applications running on Single Address Space (SAS) operating systems. In this format, the hash function is embedded in hardware. The long format VHPT is useful for sparsely populated address spaces since the short format VHPT has a linear layout and would consume a large amount of memory [Ref. 2].

E. CONTEXT MANAGEMENT IN THE IA-64

Intel's IA-64 manual [Ref. 2] states context management and state preservation rules as follows;

1. User Level Thread Switch

The following steps need to be taken to execute a user level thread switch [Ref. 2].

- The caller saves GRs 2-3, GRs 8-11, GRs 14-15, Grs 16-31, FRs 6-15 and FRs 32-127.
- The callee saves GRs 4-7, FRs 2-5 and FRs 16-31

- GRs 32-127 are preserved by the Register Stack Engine (RSE)
- GR 0 is always zero. FR 0 is always +0.0 and FR1 is always +1.0.
- GR 1, GR 12 and GR13 have special uses.
- Preserve predicate, branch, and application registers.
- Flush outgoing register stack to backing store, and switch to incoming thread's backing store.
- Switch thread memory stack pointers.
- Restore incoming thread's predicate, branch, and application registers.
- Restore incoming thread's preserved register state.

2. Thread Switches Within The Same Address Space

When the operating system switches between different threads in the same address space, the following steps should be followed [ref. 2]:

- Application architecture state (GRs, FRs, PRs, BRs, ARs) associated with each thread are saved and restored. This is described in "User Level Thread Switch" in Section E.1.
- Memory ordering operations are performed. The IA-64 requires a memory synchronization (sync.i) and a memory fence (mf) during a context switch to ensure that all memory operations prior to the context switch are made visible before the context changes.

3. Address Space Switching

When the operating system switches the address space, it is necessary that the same steps as in the thread switch be preformed [Ref. 2]. Additionally, operating system is required to take following steps:

- Save the contents of the protection key registers (PKRs) corresponding to the outgoing process, and then invalidate the protection key registers.
- Default Control Register (DCR) of the outgoing address space is saved. The DCR specifies default parameters for Processor Status Register values on interruption.
- Region Registers (RRs) of the outgoing address space are saved.
- Region Registers (RRs) of the incoming address space are restored.
- DCR of the incoming address space is restored.
- PKRs of the incoming address space are restored

F. SUMMARY

As described in this chapter, the IA-64 processor provides many new architectural features. The innovative use of predication and speculation allows the processor to overcome the limitations of mispredicted branches and memory latency problems whereas register rotation provides increased loop performance over traditional architectures. In addition, this chapter has focused on some architecture features such as virtual addressing schemes, execution environments, TLB based protection mechanisms

and context management. Chapter IV provides a detailed description of protection mechanisms supported by the architecture. In Chapter V, the security challenges related to these new features will be examined.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. PROTECTION IN THE IA-64

The IA-64 architecture provides many protection mechanisms to enforce security mechanisms among processes. These features are: privilege levels, access rights, isolation of processes into separate address spaces and protection keys. Privilege levels, access rights, and protection key-based memory protection mechanisms are embedded into the translation lookaside buffer (TLB) mechanism supported by the processor architecture. This chapter will describe each of these mechanisms and then outline the translation lookaside buffer support provided by the IA-64.

A. PRIVILEGE LEVELS

The IA-64 processor supports four different privilege levels (0 to 3) to control access to memory address spaces, system instructions and system registers. Level 0 represents the most privileged level whereas level 3 is the least privileged one. In this hierarchy, privileged instructions and registers can only be accessed in the most privileged level. When attempts to access privileged elements of the processor are made by unauthorized privilege levels, a Privilege Operation Fault is raised. Each memory access compares the current privilege level of the process with that of the target resource. The current privilege level (CPL) field of the Processor Status Register (PSR) provides the current privilege level of the executing process to the processor. The CPL field can only be modified by the operating system due to interrupts, the break instruction (break/rfi) and the enter privilege code instruction (epc/br.ret). See "Privilege Level Transfer" in section B.

On each memory access, the current privilege level of the executing process is obtained from the PSR and compared with the privilege level of the associated page given by the TLB entry (TLB.pl) for that page. If the current privilege level is less than or equal to the privilege level defined in the TLB, access is permitted. Further, the type of the access (i.e read, write, execute) is obtained from the "access rights" field of the TLB entry (TLB.ar). See "Access Rights" in section C.

B. PRIVILEGE LEVEL TRANSFER

As mentioned in Chapter II, privilege level transfer between processes must only be made through controlled entry and exit points that are strictly managed by the operating system. Otherwise, processes in different privilege levels can modify each other's address spaces and intra-process leakage becomes inevitable. The IA-64 provides two instructions that may cause a privilege level transfer: the break instruction (break/rfi) and the enter privilege code instruction (epc/br.ret).

The break instruction, usually paired with return from interruption (rfi), causes a Break Instruction Fault that is used to change privilege levels. "When the break instruction is executed, it branches to the Break Fault Handler, which should be a valid address mapping in the privileged mode for each user application, and raises the privileged mode to the most privileged level" [Ref. 2]. The break instruction also contains a 21-bit immediate value that is passed to the corresponding fault handler to differentiate system calls from some other usage of the break instruction (such as debugging). The return from interruption (rfi) instruction is used to return from the fault handler to the previous mode.

On the other hand, the enter privilege code instruction (epc) increases the current privilege level without an interrupt or a control flow transfer. Some execute-only kernel pages, also known as promotion pages, have a higher privilege level specified in their TLB entries. To make a system call with the epc instruction, a user system branches to an execute-only kernel page (TLB.ar7) controlled by the operating system, using a branch call instruction (br.call). At this point, the epc instruction is executed in the kernel page to raise the current privilege level to a higher privilege level. After the epc instruction is executed, the current privilege level (PSR.cpl) is raised to the privilege level of the promotion page that is specified by the TLB entry and the next instructions are executed at the target privilege level. To return back to user mode, the kernel executes a branch return (br.ret) instruction. " The br.ret instruction demotes the privilege level, by restoring the privilege level contained within the Previous Function State (PFS) application register. To ensure operating system integrity, the epc instruction checks that the previous privilege level field in the Previous Function State register (PFS.ppl) is no greater than the PSR.cpl at the time the epc is executed" [Ref.2].

As described earlier, execute-only pages (TLB.ar7) are used to promote the privilege level on entry into the operating system. Although, these promotion pages are controlled by the operating system, any application that has a lower privilege level than the promotion page can branch into these pages by executing a branch call (br.call) instruction. After the execution of the epc instruction, an application executing in privilege level 3 can be directly promoted to privilege level 0 bypassing the intermediate privilege levels 1 and 2. In the IA-64 architecture, there is no architectural hardware feature to prevent such promotions from bypassing these intermediate privilege levels. To

force the thread of execution to follow a privilege hierarchy restricts a malicious user from gaining unauthorized privilege by writing malicious code to circumvent system security mechanisms and jump right into the highest privilege level. Because the epc must be executed as the first instruction in the kernel page, it presents a well-defined control point on entry into the kernel page and prevents such exploitations. If the epc is not executed as the first instruction of a promotion page, the processor raises a privileged operation fault.

C. SEPARATE ADDRESS SPACES

In the IA-64 architecture, virtual memory is a crucial feature of the operating system's protection mechanisms and multitasking support. The IA-64 can map a 2^{64} byte virtual address space whereas traditional 32-bit architectures can only map up to a 2^{32} byte address space. Further, the memory space required by the page table entries is much larger than those in 32-bit processors and, hence, is conveniently placed in virtual memory. All of these combine to put more pressure on the processor's translation lookaside buffer (TLB). Thus, the IA-64 supports some new features not only to lessen the pressure on the TLB but also to provide efficient resource sharing while supporting required protection mechanisms. These features are: regions and region identifiers and protection key registers.

1. Regions

As mentioned in Chapter I, the IA-64 provides a 64-bit virtual address space. The upper three bits of the virtual address select one of the eight region registers to which a particular address belongs. Each region register (RR), which is managed by the operating system, includes a region identifier (RID) associated with the region. RIDs may vary

from 18 to 24-bits in size depending on the implementation. This feature allows the operating system to be able to map from 2^{18} up to 2^{24} address spaces as shown in Figure 5, Chapter III. To make an address space visible to software, the operating system loads its region identifier into one of eight region registers.

In Multiple Address Space (MAS) operating systems, where a separate address space is provided for each process, the RIDs are used as address identifiers. It is the operating system that assigns a unique region identifier to a process for a process-private region. In cases where a process requires more than one private region, the operating system can assign multiple unique RIDs to a process.

Each translation entry in the TLB contains a RID associated with it. This feature eliminates the need to invalidate the entire TLB on a context switch as in traditional architectures. Historically, when a context switch is performed in such architectures, each page table entry and the protection information in the TLB related to the switched-process must be invalidated by the processor. In the IA-64 architecture, when a context switch is performed from process A to process B, the operating system removes only process A's private region identifiers from the CPU's region registers and swaps them with process B's region identifiers [Ref. 2]. If a region is to be shared by two processes, the region identifiers (RIDs) of the shared regions stay in place in the processor's region registers. In this case, data sharing and protection are achieved on the basis of privilege level (TLB.pl) and corresponding access rights (TLB.ar) fields in the TLB. For instance, if two processes with different privilege levels (process A=pl2 and process B=pl3) are supposed to share the same address space, first, the privilege level of each process is compared with the privilege level of the page in the TLB (TLB.pl). If the process's

privilege level is equal to or greater than the TLB.pl, access is granted to the page in physical memory. Further, the type of the access, such as read, write or execute, is attained from the access rights field in the TLB (TLB.ar). Table 2 shows "Access Rights" to a page. To be precise, assume that TLB.pl = 3 and TLB.ar = 4. Both processes are permitted to access the page and they can also read, write and execute on the page. See "Access Rights" in section C. In this scenario, it is not possible to enforce appropriate data sharing if the processes are to execute in different classification levels. (i.e. process A executes in Top Secret, process B executes in Unclassified) since they do not reach the page with different access rights. Instead, they are either denied access to the page according to their privilege levels or allowed to share the page with the same access rights if and only if they have the right privilege levels. In this scenario, regardless of the privilege levels that the processes are currently executing, it is not possible to assign different security levels (secret, unclassified) to these processes. The problem of providing different security levels is solved through the use of protection keys and protection key registers that are described in section D.

In the IA-64 architecture, there is actually no "SAS operating system" or "MAS operating system" mode. The difference between the two operating system models is the encoded policy enforced by the operating system. In other words, the difference comes through the management of the region identifiers (RIDs) and protection keys by the operating system. In a Single Address Space operating system, multiple processes can share data by using the same RID and the protection is enforced by the operating system using protection keys. In a MAS operating system, per-process RIDs enforce per process protection and isolation [Ref. 2].

On the other hand, Hybrid SAS/MAS models that combine unique RIDs for per process regions and shared RIDs with protection keys for per-page memory protection in shared regions are also possible to enforce an efficient security policy [Ref. 2].

TLB.ar	TLB.pl	Privilege Level				Description
		3	2	1	0	
0	3	R	R	R	R	read only
	2	–	R	R	R	
	1	–	–	R	R	
	0	–	–	–	R	
1	3	RX	RX	RX	RX	read,execute
	2	–	RX	RX	RX	
	1	–	–	RX	RX	
	0	–	–	–	RX	
2	3	RW	RW	RW	RW	read,write
	2	–	RW	RW	RW	
	1	–	–	RW	RW	
	0	–	–	–	RW	
3	3	RWX	RWX	RWX	RWX	read,write,execute
	2	–	RWX	RWX	RWX	
	1	–	–	RWX	RWX	
	0	–	–	–	RWX	
4	3	R	RW	RW	RW	read only / read,write
	2	–	R	RW	RW	
	1	–	–	R	RW	
	0	–	–	–	RW	
5	3	RX	RX	RX	RWX	read,execute / read,write,exec
	2	–	RX	RX	RWX	
	1	–	–	RX	RWX	
	0	–	–	–	RWX	
6	3	RWX	RW	RW	RW	read,write,execute / read,write
	2	–	RWX	RW	RW	
	1	–	–	RWX	RW	
	0	–	–	–	RW	
7	3	X	X	X	RX	exec,promote / read,execute
	2	XP2	X	X	RX	
	1	XP1	XP1	X	RX	
	0	XP0	XP0	XP0	RX	

Table 2. Access Rights from [Ref. 2]

D. ACCESS RIGHTS

As mentioned before, the IA-64 supports four privilege levels to control access to pages. Page access is determined by the TLB.ar, and TLB.pl fields and by the privilege level of the access. On memory accesses, the privilege level of the currently executing process is obtained from the Processor Status Register (PSR.cpl), and is compared with the privilege level of the related page (TLB.pl). Along with the result of the comparison and the access right field of the TLB (TLB.ar), the type of the access is obtained. Table 2 shows "Access Rights" to a page. Within the table, "—" means no access, "R" means read access, "W" means write access, "X" means execute access, and "Pn" means promote PSR.cpl to "n" whenever the enter privilege code (epc) instruction is executed" [Ref. 2]. Here, "n" represents target privilege level specified by the promotion page.

Software can verify page level permissions by executing the *probe* instruction. This instruction is used to check accessibility to a specific virtual page by verifying both privilege levels and page level and protection key permissions.

E. PROTECTION KEYS

There are two different types of protection mechanisms provided by the IA-64 that may be used to mediate access to a page. As described earlier, the first one is the page access rights bits in the TLB associated with each translation. This protection mechanism provides privilege level-granular access to a page. The second mechanism is the protection keys that allow domain-granular access to a page. Domains are defined by protection key registers and offer an efficient method for the operating system to control access rights to groups of pages. "These keys are especially useful for mapping shared

code and data segments in a globally shared region, and for implementing domains in a single address space (SAS) operating system" [Ref. 2].

The Protection Key Registers (PKR) represent a register cache of all protection keys required by a process. Management and replacement policies for this cache are determined by the operating system. All IA-64 processors employ at least 16 protection key registers (PKRs) that contain 18 to 24-bit protection keys depending on the implementation. To enable the protection key check, the protection key bit in the Processor Status Register (PSR.pk) must be set to '1'. This way, all the memory references are directed to go through protection key checks during the virtual-to-physical address translations. Figure 11 and Table 3 describe the protection key register format and protection key register fields respectively.

When protection key checking is enabled, the protection key tagged to a translation is checked against all protection keys in the protection key register cache before a memory access is permitted [Ref. 2]. Here, tagged translation defines a protection key attached to an entry in the TLB. If a matching key is found, the access rights (i.e wd, rd, xd) determined by the key are applied to the translation. If the access being performed is allowed by the matching key and the TLB.ar field, access is granted to the page. Otherwise, a Protection Key Permission Fault is raised by the processor and the operating system may either terminate the program or grant it the requested access [Ref. 2]. The protection key register's access denials always override TLB access rights. For instance when 'wd' is set by the in protection key register, 'write' permission is denied to translations even if the TLB access rights permit the write. If the matching key

is not found, a Protection Key Miss Fault is raised by the processor, and the operating system must insert the correct protection key into PKRs and retry the access.

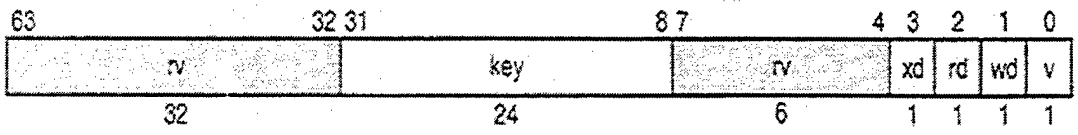

Figure 11. Protection Key Register Format from [Ref. 2]

Field	Bits	Description
v	0	Valid - When 1, the Protection Register entry is valid and is checked by the processor when performing protection checks. When 0, the entry is ignored.
wd	1	Write Disable - When 1, write permission is denied to translations in the protection domain.
rd	2	Read Disable - When 1, read permission is denied to translations in the protection domain.
xd	3	Execute Disable - When 1, execute permission is denied translations in the protection domain.
key	31:8	Protection Key - uniquely tags translation to a given protection domain.
rv	7:4, 63:32	reserved

Table 3. Protection Registers Fields from [Ref. 2]

Protection key insertion into the protection key registers is performed with the *move to PKR* instruction. During the execution of this instruction, the processor checks the new valid protection key against all protection key registers to guarantee the

uniqueness of the key. If a valid matching key is found in any of the PKR registers, the processor invalidates the matching PKR that exists in the cache by setting PKR.v bit to zero. The other fields in the matching PKR remain unchanged when it is invalidated.

Protection keys can be read from the processor's data TLBs by executing the *tak* instruction. However, instruction TLB key values can not be read directly. These values must be held in software-maintained data structures [Ref. 2].

F. SUMMARY

Thus far, protection requirements to build a secure system have been explained in Chapter II. This chapter has focused on the protection mechanisms in the IA-64 architecture to meet these requirements.

The first concept, that requires the implementation of privilege levels or ring mechanisms, is evidently supported by the processor. As mentioned before, the IA-64 supports four different privilege levels (0 to 3) to control access to memory address spaces, system instructions and system registers. Further, when attempts to access privileged elements of the system are made by unauthorized privilege levels, a Privilege Operation Fault is raised by the processor.

As explained in Chapter II, the second concept indicates that there should be a non-bypassable controlled call or a well defined entry point to a more privileged function. By implementing two special instructions; the break and the enter privilege code (epc), the IA-64 architecture increases the current privilege level to a higher privilege level via controlled entry points. Of these instructions, the epc increases the current privilege level without an interrupt or a control flow transfer. Alternatively, the break instruction

branches to the Break Fault Handler as in traditional architectures and then can raise the privileged mode to the most privileged level.

The third concept is to support protection by restricting tasks from accessing, modifying or deleting anything exclusive to another address space. To achieve this restriction, each task must be allocated a separate region, managed by the more privileged operating system that defines the process address space. The IA-64 provides a paging structure to apply protection to pages. In a multiple address space operating system (MAS), each process is placed within a unique address space. Region identifiers tag virtual address mappings to a given process. Associated with each translation, there is an access rights field in the TLB that provides privilege-level granular access to the page. In a single address space system (SAS), processes exist within a global virtual address space and protection among processes is enforced through protection key registers. In both operating system models, the translation lookaside buffer (TLB) structure implemented by the processor establishes the base for protection algorithms.

V. SECURITY ANALYSIS OF THE IA-64 ARCHITECTURE

Thus far, three major requirements for a secure system have been discussed. Further, the ways in which the IA-64 architecture supports these requirements have been explained. This chapter will compare the support provided by the hardware and contrast it with that provided by software to meet these requirements. In addition, security challenges related to some architectural features will be examined.

A secure computing system must be able to enforce security policies reliably, even in worst-case scenarios. Protection mechanisms that support these policies are implemented in a computer's operating system and hardware. Although software is the primary focus of the attention, hardware components are obviously critical to a system's security. Indeed, hardware security features are an indispensable part of the implementation. Hardware components are often designed with an incomplete understanding of security issues and, as a result, may be poorly matched to a secure system [Ref.11].

In hardware, the Central Processing Unit (CPU) which includes a Memory Management Unit, I/O processor and some other function units (i.e. arithmetic logical unit (ALU), stack pointer) is responsible for supporting operating system mechanisms that isolate the system state from untrusted subjects, and subjects from each other. As mentioned in Chapter II, such isolation mechanisms are user/supervisor state, address space separation, segment or page protection (address protection).

A. ANALYSIS OF THE IA-64 SECURITY FEATURES

As described in Chapter IV, the IA-64 architecture supports several mechanisms for building secure systems:

- Multiple operating system modes: Multiple Address Space (MAS) operating systems and Single Address Space (SAS) operating systems
- Address Space Management: Regions and Region Identifiers (RIDs)
- Multitasking: Context Switching
- Privileged levels and privilege level transfer
- TLB based per page protection: TLB access rights and TLB privilege level
- Domain-level granular protection: Protection Key Registers (PKRs)
- Miscellaneous: Application Registers (ARs), System Registers, Register Stack Engine (RSE).

1. Previous Function State Application Register (PFS)

As described in Chapter III, the Previous Function State Application Register (PFS) contains multiple fields: pfm, pec and ppl. These values are copied automatically on a call from the Current Frame Marker (CFM) Register, Epilog Count Register (EC), and PSR.cpl (current privilege level in the processor status register) to accelerate procedure calling.

When a br.call instruction is executed, the CFM, EC and PSR.cpl are copied to the PFS application register, which can be directly read or written by the application software. Here, the CFM holds the state of the current stack frame and register rename

bases (RRBs) for GRs, FRs and PRs. RRB is used to rename the registers within the rotating subset of each register type. The PSR.cpl is a piece of information that must not be modified by the user programs. Otherwise, a process might gain unauthorized privilege and execute a privileged instruction that affects the processor state. However, the PSR.cpl field in the PFS can be modified to escalate the privilege level on execution of a branch return (br.ret) instruction since this register is writable. To prevent such modification, a check exists in the IA-64 architecture to make sure that copying PFS.ppl to PSR.cpl will not increase the privilege level before the copying is actually performed.

2. Privileged Level Transfer

As mentioned in Chapter IV, the IA-64 architecture provides two instructions, the epc and break to provide privilege level transfers.

The break instruction, as in traditional architectures, jumps to a fault handler, which should be a valid address mapping for each user application, and raises the privilege level to the most privileged level via a system call. Here, a valid address mapping determines the page address to handle an interrupt by using processor's interruption vector tables. These tables include interrupt types associated with interrupt vector names (in this case, Break Instruction vector). From the type of the interrupt given in the table, the operating system determines the handling routine.

The epc instruction implements calls to higher-privileged routines without interruption to reduce system calls overhead. Using the br.call instruction, a user application branches to an execute only kernel page (TLB.ar = 7). Further the epc instruction is executed as the first instruction in the kernel page to raise the privilege level. The new privilege level is given by the TLB entry of the page containing the epc

instruction. After completing the system call, the kernel returns to the user system with a branch return (br.ret) instruction.

When the branch call (br.call) instruction is executed, the following actions occur: {Ref. 15]

- The current values of the Current Frame Marker (CFM), the Epilog Count (EC) application register and the current privilege level in the Processor Status Register (PSR.cpl) are saved in the Previous Function State Register.
- The caller's stack frame is saved and the callee is provided with a frame containing only the caller's output region. See "Register Stack Engine" in Chapter III.
- The Rotation Rename Base (RRB) registers in the CFM are reset to '0'.

When the br.ret instruction is executed, following actions occur: [Ref.15]

- CFM, EC, and the current privilege level are restored from PFS (The privilege level is restored only if this does not increase privilege level).
- The caller's stack frame is restored.
- If the return lowers the privilege (numerically increases) and lower privilege level transfer trap field in the PSR is '1', then a lower-privilege Transfer trap is taken.

In general, the system call mechanisms give the user process the potential to call malicious code rather than the code intended by its caller. This can be accomplished

using a stack-smashing attack, which represents the majority of current overflow attacks. The attack takes advantage of the call function, which inserts the return address in the stack. The stack-smashing attack corrupts the return address of the function being called. To prevent such attacks, the compiler can be designed to generate code that checks for stack smashing every time a function returns [Ref. 16]. In the IA-64 architecture, although return addresses are stored into branch registers, they might be spilled to the register stack eventually. In cases where the return address resides in a branch register, a malicious user can modify the return address by making indirect branch calls since these registers are not privileged registers in the IA-64.

With respect to stack-smashing attacks, another protection mechanism can be offered by making the data segment of the program's address space non-executable. This makes it impossible for attackers to execute the malicious code they injected into the program's input buffers. However, this form of protection does not provide protection against other forms of attacks such as corrupting the code pointers (program state that points at code) by changing the pointer arguments. Thus, the kernel page code in the IA-64 architecture should be written carefully by the operating system programmers to guard against the cases where a buffer overflow is used to sneak code into the system [Ref. 16].

3. Hardware Based Protection Mechanisms

As mentioned earlier in the third chapter, the IA-64 architecture defines many functional features in hardware as well as in software for protecting processes from each other. These protection mechanisms are address space separation via region identifiers, privilege levels, access rights, and protection keys. Of the mechanisms supported by the architecture, address space separation is defined on a region basis and is enforced by the

handling of region identifiers (RIDs) by trusted software (in this case, the operating system). Region identifiers, placed in one of eight region registers by software, can be considered unique IDs in a Multiple Address Space (MAS) operating systems and assign a private region to each process. In the IA-64 architecture, because each translation in the TLB is tagged with its region identifier, the TLB can manage translations associated with different address spaces. Accordingly, on an address space switch, the operating system need not purge all the TLB entries which belong to a process. Historically, with respect to the context switches, there are two components: the cost of saving and restoring state and the cost of translation lookaside buffer (TLB) misses caused by the address space change [Ref. 7]. By implementing process ID tags in its TLB entries, the IA-64 architecture has decreased the rate of TLB misses and the overhead of context switching.

"Appropriately configured, processes have no way to generate memory addresses that reference regions owned by some other processes, unless the operating system grants shared access to a particular region" [Ref. 6]. The major advantages of private address spaces are: (1) they increase the amount of address available to all programs; (2) they provide hard memory protection limits; and (3) they allow a cleanup when a process exits from the region [Ref. 12]. The disadvantage of this type of approach is that process isolation within private virtual address space might generate problems such as effective cooperation between protected processes. Specifically, pointers, whose function is to make references to various data structures during the execution of a program, have no meaning beyond the boundary or lifetime of the process that creates them. Moreover, the use of pointers usually requires prior coordination for the use of the address space for

shared regions. Thus, sharing patterns must be known statically. As a result, pointer-based information is not easily shared, stored or transmitted [Ref. 12].

As described in Chapter IV, in the IA-64 architecture, sharing between processes in a Multiple Address Space Operating Systems (MAS) is achieved by mapping region identifiers into the region register set of multiple processes. The use of regions and region identifiers provides a mechanism for efficient address space-to-address space copies (cross-address space copies). Cross address spaces allow a user process to link a program in another address space without compromising the security of the address space. In this scheme, if two processes working on the same processor want to communicate with each other, process A's address space is mapped to process B's address space statically by the kernel. If process B has the access right read-only, then it does not need its own copy of address space. When process B wants to write this address space, the operating system makes a copy of the region, and assigns it to process B. From then on, process B can have both read and write access rights to the new region. In the IA-64 architecture, "because address spaces are tied to RIDs and not to a particular static region, a MAS operating system can simply copy a memory range from one address space to another by temporarily remapping the target location to another region" [Ref. 2]. Furthermore, since each page can have its own access rights associated with the RID, the operating system can assign different access rights to the mapped page by using protection key registers. Allocation of a common region to multiple processes is strictly managed by the operating system. However, without the use of protection key registers, this type of sharing will still be somewhat limited, since there is no way to provide different views of the same page. If multiple processes are to share the same page, they cannot access the page with

different access rights because two page table entries cannot reside in the TLB with the same region identifier and having different access rights. Consequently, this limitation will result in enforcing a poor security mechanism among processes since they cannot be assigned to different hierarchical classifications and security clearances (e.g. secret, top secret). Thus, it may be difficult to enforce security policy in the system efficiently and coherent views of information may be lost.

Unlike in the Multiple Address Space model, in the Single Address Space model the system rather than the applications coordinates the address bindings to hold dynamic sharing patterns in a uniform way. In the IA-64 architecture, the Single Address Space (SAS) model requires the use of protection key registers. This removes sharing problems between processes by treating all virtual address space as a global resource controlled by the operating system. The implementation of a Single Address Space mode IA-64 architecture is achieved by enabling the protection key field in the Processor Status Register (PSR.pk) by the operating system.

As described in Chapter IV, protection keys provide a method to restrict permission by tagging each virtual page with a protection domain identifier. The Protection Key Registers (PKRs) define a set of registers that caches all protection keys required by a process. In addition, each TLB entry contains a protection key field, which is placed in the TLB when a virtual-to-physical translation is created. When a memory reference hits in the TLB, the processor searches for the matching entry's key in the PKR cache. A key match results in additional access rights (write, read, execute) being checked before the memory reference is granted. Otherwise, the hardware generates a Key Miss fault and the operating system must abort the access or insert the correct

protection key into the PKRs and retry the access. This is explained thoroughly in Section D, Chapter IV. In this scheme, "the processor has no notion of which protection keys belong to which process. The only check the hardware performs is to compare the protection key from the translation to any valid protection keys in the PKR cache" [Ref 2]. On a context switch, the operating system must purge any valid protection keys from the PKRs that might provide unauthorized access rights to a switched-process.

Protection keys are used to provide different access rights to shared TLB entries for each process. For instance, consider a page tagged with a protection key number 0xF. Two processes are supposed to share this page: one is the owner of the page and the other is a user. When the owner process is running, the operating system will insert a valid key into the PKR cache with the protection key 0xF and the 'wd', 'rd', and 'xd' bits cleared. This will permit the running process to access the page with read, write and execute access rights. However, when the user process is running, the operating system will insert a valid key with the protection key 0xF, but this time, only the 'rd' bit is cleared to permit this process to read the page, but denying write or execute access [Ref. 2].

To make use of the single address space model without sacrificing the major advantages of the multiple address space model, the IA-64 implements hybrid SAS/MAS models that combine unique RIDs for per process region and shared RIDs with protection keys for per-page protection. Similar to the MAS model, the current privilege level field in the Processor Status Register (PSR.cpl) is compared with the page's TLB entry (TLB.pl) on a memory reference in the hybrid SAS/MAS model. If PSR.cpl is less than or equal to TLB.pl, access is permitted. Additionally, the type of the access is checked against the access-rights field, TLB.ar, in the TLB entry. But this time, the processor also

searches for the matching entry's key in the PKR cache. As previously explained, a key match results in additional access rights (write, read, execute) being checked before granting or denying the memory reference. In the hybrid SAS/MAS model, access denials in the protection key registers always override the access rights field in the TLB entry.

4. Conclusion

This chapter has focused on how IA-64 architecture protection features enable the processor hardware and the operating system to collaborate to enforce the security features among the processes. Since the hardware protection mechanisms are crucial to meeting the requirements of Reference Monitor Concept abstraction, it is required that hardware security features be supported with complementary software.

In the IA-64 architecture, translation lookaside buffer (TLB) structure is central to the platform's protection mechanisms. Implementing privilege levels, access rights and protection keys in its structure, the TLB provides protection and sharing in a well-defined manner. Protection keys are especially useful in the sense that they provide different access rights for each process to shared TLB entries. This feature enables memory locations to be assigned different security attributes for different processes for efficient enforcement of security policy.

In addition, privilege level transfer in the architecture has been reviewed. Since system call implementations are usually not designed to prevent exploitations, operating system programmers should take into account possible attacks commonly associated with these calls and returns when writing code for TLB-resident execute-only pages.

VI. SUMMARY AND CONCLUSION

This thesis presents an analysis of the IA-64 processor's support for secure systems. A secure system must provide security features in hardware as well as in software without sacrificing performance and efficiency. Hardware security features are very critical to a system's security. Current architectures mostly suffer when trying to implement hardware security features efficiently, and therefore have trouble enforcing security policies reliably and correctly.

In this thesis, the hardware requirements of a secure system are described. And further, these requirements are mapped to the IA-64 processor's hardware security mechanisms. Hardware protection mechanisms supported by the architecture are privilege levels, access rights, isolation of processes into separate address spaces (regions, region identifiers) and protection keys. Privilege levels, access rights, and protection key-based memory protection mechanisms are embedded in the translation lookaside buffer (TLB) structure implemented by the processor architecture.

Four different privilege levels (0 to 3) are supported by the architecture to control access to memory address spaces, system instructions and system registers. Level 0, the most privileged level, is reserved for the operating system use and it is the only level in which a privileged instruction can be executed. Each memory access compares the current privilege level of the process with the privilege level that resides in the TLB entry (TLB.pl) for the associated page during virtual-to-physical address translations. According to the result of the comparison, access is granted or denied. If granted, the type of the access (i.e. read, write, execute) is obtained from the "access rights" field of the TLB entries (TLB.ar).

Region identifiers (RIDs) are implemented in the architecture to provide a distinct address space separation among processes. These identifiers, placed in one of eight region registers by the operating system, are used to map a process's address space onto the system's virtual regions. Furthermore, each translation entry in the TLB has a RID associated with it. This feature enables the TLB to hold translations from many different address spaces. As a result, the processor does not have to flush the entire TLB upon address space switches.

To support the Single Address Space (SAS) operating system model, the protection key registers (PKRs) are provided by the architecture. These restrict permission by tagging each virtual page with a protection domain identifier. The Protection Key Registers (PKRs) define a set of registers that caches all protection keys required by a process. Although the hardware performs the check to compare the protection key from the translation to any valid protection keys in the PKR cache, the operating system must be responsible for management and replacement policies of these registers at all times. Protection keys enable the processor to provide different access rights to each process that shares a TLB entry. This permits security attributes to be assigned to memory resources on a per process basis and supports efficient enforcement of security policy.

In conclusion, the virtual memory model and translation lookaside buffer (TLB) structure are crucial features of the hardware protection mechanisms and multitasking support in the IA-64 architecture. The TLB structure supports memory protection on the basis of access rights, privilege levels and protection keys. Hardware privilege levels (0 to 3) are used to distinguish between supervisor and user modes, and also ensure that

privileged instructions are executed only by the operating system. The access rights field in the TLB is used to restrict access to memory locations when privilege level of the currently executing process is less than the privilege level of the TLB-resident page. An alternative approach to controlling access to memory is accomplished by implementing protection key registers. Protection key checking can be enabled by the operating system. When enabled, "all the memory references go through protection key access checks during the virtual-to-physical address translations" [Ref. 2]. Protection keys are useful for providing different access rights to shared TLB entries. With appropriate configuration of these mechanisms, processes may be constrained from generating unauthorized memory references.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual*, Volume 1: IA-64 Application Architecture, July 2000.

[2] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual*, Volume 2: IA-64 System Architecture, July 2000.

[3] Geva, R., Morris D., *IA-64 Architecture Disclosures White Paper,*1999. [http://www.ia-64.hp.com/infolibrary/whitepapers/ia64_arch_wp.pdf].

[4] Gwennap, L., *Intel Discloses New IA-64 Features,* Rotating Registers Reduce Code Expansion; Merced Touted for Big Servers, March, 1999. [http:// www.chipanalyst.com/mpr_public].

[5] Joint whitepaper between Intel & HP, *IA-64 Architecture Innovations*, http://www.ia-4.hp.com/infolibrary/whitepapers/ia64_overview_wp.html, February, 1999.

[6] Diefendorff, K., HP, Intel Complete IA-64 Rollout, *Virtual Memory, Interrupts More Conventional Than ISA*, April, 2000. [www.MPRonline.com].

[7] Anderson, T. E., Levy, H. M., Bershad, B. N., Lazowska, E. D., *The Interaction of Architecture and Operating System Design*, ACM Digital Library, pp. 108-115, 1991, [http://www.acm.org/dl/].

[8] Brinkley, D. L., Schell, R. R., *Concepts and Terminology for Computer Security*, IEEE, May 1993, [http://ieeexplore.ieee.org/lpdocs/epic03/].

[9] Hennessy, J. L., Patterson, D. A., *Computer Architecture A Quantitative Approach*, Morgan Kaufman Publishers, 1990.

[10] Saltzer, J. H., Schroeder, M. D., "The Protection of Information in Computer Systems", *Proceedings of the IEEE*, Vol. 63, No. 9, September 1975.

[11] Sibert, O., Porras, P. A., Lindell, R., "The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems", *IEEE Symposium on Security and Privacy*, pp. 212-214, 1995.

[12] Chase, J. S., Levy, H. M., Feeley, M. J., Lazowska, E. D, "Sharing and Protection in a Single-Address-Space Operating System", *ACM Transactions on Computer Systems*, Vol. 12, No. 9, November 1994.

[13] Ferraiolo, D., Mell, P., "Operating System Security: Adding To The Arsenal of Security Techniques." [http:// crsc.nist.gov/publications].

[14] Gareau, J., *"Advanced Embedded X86 Programming: Protection and Segmentation."* [http://www.embedded.com].

[15] Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual*, Volume 3: Instruction Reference Manual, July 2000.

[16] Cowan, C., Wagle P., Pu, C., Beattie, S., Walpole, C., 'Buffer Overflows: *Attacks and Defenses for the Vulnerability of the Decade*', IEEE Digital Library, pp. 119-122, 1999, [http://ieeexplore.ieee.org/lpdocs/epic03/].

[17] Gollmann, D., *Computer Security*, Gray Publishing, 1999.

[18] Zahir, R., Ross, J., Morris, D., Hess, D., *OS and Compiler Considerations in the Design of the IA-64 Architecture,* ACM Digital Library, p.212, 2000 [http://www.acm.org/dl/].

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Suite 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Chairman, Code EC 1
   Electrical and Computer Engineering Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

4. Prof. Cynthia Irvine, Code CS/Ic 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Senior Lecturer Frederick W. Terman, Code EC/Tz 1
   Electrical and Computer Engineering Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Deniz Kuvvetleri Komutanligi 2
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY

7. Deniz Kuvvetleri Komutanligi 1
   Kutuphane
   Bakanliklar
   Ankara, TURKEY

8. Deniz Harp Okulu Komutanligi 1
   Kutuphane
   Tuzla
   Istanbul, TURKEY

9. Bugra Unalmis 2
   Dr. Ergin Cad. Tumer Apt. No:35/10
   Goztepe-Istanbul, TURKEY

10. Carl Siel 1
Space and Naval Warfare Systems Command
PMW 161
Building OT-1, Room 1024
4301 Pacific Highway
San Diego, CA 92110-3127

11. Commander, Naval Security Group Command 1
Naval Security Group Headquarters
9800 Savage Road
Suite 6585
Fort Meade, MD 20755-6585
San Diego, CA 92110-3127

12. Ms. Deborah M. Cooper 1
Deborah M. Cooper Company
P.O. Box 17753
Arlington, VA 22216

13. Ms. Louise Davidson 1
N643
Presidential Tower 1
2511 South Jefferson Davis Highway
Arlington, VA 22202

14. Mr. William Dawson 1
Community CIO Office
Washington DC 20505

15. Capt. James Newman 1
N64
Presidential Tower 1
2511 South Jefferson Davis Highway
Arlington, VA 22202

16. Mr. Richard Hale 1
Defense Information Systems Agency, Suite 400
5600 Columbia Pike
Falls Church, VA 22041-3230

17. Ms. Barbara Flemming 1
Defense Information Systems Agency, Suite 400
5600 Columbia Pike
Falls Church, VA 22041-3230

18. CDR Keith Fuller 1
Defense Information Systems Agency, Suite 400
5600 Columbia Pike
Falls Church, VA 22041-3230