ABSTRACT

JIANG, XIAOWEI. Architecture Support for Operating System Survivability and Efficient Bulk Memory Copying and Initialization. (Under the direction of Professor Yan Solihin).

Operating System (OS) is the fundamental layer that provides and mediates accesses to a computer system’s resources for user application programs. The ever increasing size and complexity of the OS code bring the inevitable increase in the number of security vulnerabilities that can be exploited by attackers. A successful security attack on the OS has a profound impact because the OS runs at the highest processor privilege level. An OS kernel crash can freeze the entire system, terminate all running processes, and cause a long period of system unavailability. Given the increasing trend of OS security faults and the dire consequences of successful OS kernel attacks, we strive to make the OS kernel survivable, i.e. able to keep normal system operation despite security faults.

This works makes several contributions. First, we propose an OS survivability scheme that consists of three inseparable components: (1) Security attack detection mechanism, (2) security fault isolation mechanism, and (3) recovery mechanism that resumes normal system operation. We analyze the underlying performance requirement for each of the components and propose simple but carefully-designed architecture support to reduce the performance overhead. When testing with real world security attacks, our survivability scheme automatically isolates the security faults from corrupting the kernel state or affecting other executing processes, recovers the
kernel state and resumes execution.

Second, in order to overcome the performance overhead incurred by the checkpointing-based recovery mechanism that extensively uses bulk memory copying and initialization operations, we propose efficient architecture support for improving bulk memory copying and initialization performance. While many of the current systems rely on a loop of loads and stores, or use a single copying instruction to perform memory copying, in this work we demonstrate that the key to significantly improving the performance is removing pipeline and cache bottlenecks of the code that follows the copying instructions. We show that the bottlenecks arise due to (1) the pipeline clogged by the copying instruction, (2) lengthened critical path due to dependent instructions stalling while waiting for the copying to complete, and (3) the inability to specify (separately) the cacheability of the source and destination regions. We propose FastBCI, an architecture support that achieves the granularity efficiency of a bulk copying/initialization instruction, but without its pipeline and cache bottlenecks. When applied to OS kernel buffer management, we show that on average FastBCI achieves anywhere between 23% to 32% speedup ratios, which is roughly 3×–4× of an alternative scheme, and 1.5×–2× of a highly optimistic DMA; When applied to our OS survivability scheme, we show that an average of 1.0% performance overhead can be achieved by our survivability scheme.
Architecture Support for Operating System Survivability and Efficient Bulk Memory Copying and Initialization

by

Xiaowei Jiang

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2010

APPROVED BY:

Dr. Gregory Byrd

Dr. Edward Gehringer

Dr. Tao Xie

Dr. William Cohen

Dr. Yan Solihin
Chair of Advisory Committee
DEDICATION

To my beloved wife, to my parents, and to myself.

The quieter you become the more you can hear.

– Ram Dass
BIOGRAPHY

Xiaowei Jiang was born in Nanjing, P. R. China in 1980. He received his B.S. degree and M.S. degree in Electrical Engineering from Nanjing University, China, in year 2002 and 2005, respectively. In 2005, he worked full-time as a system engineer in ZTE Corporation, Nanjing, China, participating the development of the 3G wireless communication base-station. Xiaowei has been a Ph.D. candidate in the Department of Electrical and Computer Engineering at North Carolina State University since fall 2005. He was a Technical Staff Intern at VMware Inc, Palo Alto, CA, in the summer of 2007, and a Graduate Research Intern at Intel Corporation, Hillsboro, OR, in both summers of 2008 and 2009.

His research interests include the interaction between microprocessor architecture and Operating Systems for supporting system security and improving performance. He is also interested in the design issues of DRAM caches and tiled Chip-Multiprocessor (CMP) architectures. He has authored/co-authored several papers in the area of computer architecture and embedded systems. This dissertation fulfills the requirements of his Ph.D. degree in Computer Engineering from North Carolina State University.
ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my parents, for their selfless love and unconditional trust. Without their years of emotional and financial support, I could not have gone this far. I would also like to express my gratitude to my beloved wife, Ning Chen. She is the foundation that supports me go through my entire Ph.D. study. Nothing else to thank my family; I dedicate this dissertation to them.

I would also like to offer my sincere thanks to my advisor, Dr. Yan Solihin. I could not have completed this dissertation without his invaluable guidance and continuous support. From him, what I learnt is not only knowledge, but also the essence of how to become a successful researcher. I am also thankful to my advisory committee members, Dr. Gregory Byrd, Dr. William Cohen, Dr. Edward Gehringer and Dr. Tao Xie. I highly appreciate their important suggestions and valuable feedbacks with respect to this dissertation.

Warm thanks go to my colleagues at the Integrated Platform Research group of Intel Corporation: Donald Newell, Ravishankar Iyer, Li Zhao, Zhen Fang, Ramesh Illikkal, and Srihari Makineni, as well as Tong Li at Digital Enterprise Group of Intel Corporation, and also my former colleagues at the Software Reliability group of VMware Inc.: Matthias Hausner and Prashant Prahlad. The excellent working environment and precious comments they supplied provide the prerequisite of this
dissertation. I would like to specially thank Dr. Ravishankar Iyer and Dr. Li Zhao for their contributions to this dissertation and for their collaboration in various research projects.

Last but not the least, my gratitude also goes to the current and former fellow members at ARPERS research group: Dr. Mazen Kharbutli, Dr. Seongbeom Kim, Dr. Fei Guo, Dr. Brian Rogers, Abhik Sarkar, Aziz Eker, Fang Liu, Siddhartha Chandra, Mohit Gambhir, Devesh Tiwari, Anil Krishna, Ahmad Samih and Ganesh Balakrishnan. I am grateful to have a chance to share a part of my life with them. I specially thank Dr. Fei Guo and Fang Liu for their collaboration in my research.
# TABLE OF CONTENTS

**LIST OF TABLES** ................................................................. viii

**LIST OF FIGURES** .............................................................. ix

1 Introduction ........................................................................... 1
   1.1 Providing Operating System Survivability .............................. 3
   1.2 Improving Bulk Memory Copying and Initialization Performance ... 8
   1.3 Organization of the Dissertation ......................................... 15

2 Efficient Architecture Support for Operating System Survivability 17
   2.1 Security Model and Kernel Attacks ..................................... 18
      2.1.1 Security Model and Assumptions ................................. 18
      2.1.2 Target of Kernel Attacks ......................................... 21
      2.1.3 Mechanism of Kernel Attacks ................................... 22
   2.2 Architecture Support for Operating System Survivability ........ 25
      2.2.1 Recovery Mechanism ............................................... 25
      2.2.2 Security Attack Detection Mechanism .......................... 33
      2.2.3 Fault Isolation ..................................................... 37
   2.3 Evaluation Methodology ................................................... 41
      2.3.1 Machine Parameters ............................................... 41
      2.3.2 Benchmarks .......................................................... 41
   2.4 Evaluation ..................................................................... 44
      2.4.1 Survivability Functionality Validation ......................... 44
      2.4.2 Benchmark Characterization ..................................... 45
      2.4.3 Performance Overhead .......................................... 46
   2.5 Related Work .............................................................. 53

3 Architecture Support for Improving Bulk Memory Copying and Initial-
   ization Performance .............................................................. 57
   3.1 Related Work .............................................................. 58
   3.2 Architecture Support for Bulk Copying/ Initialization ............ 62
      3.2.1 Overview of Bulk Copying and Initialization Instructions ... 62
      3.2.2 Cache Bottlenecks .................................................. 64
      3.2.3 Pipeline Bottlenecks .............................................. 67
      3.2.4 Dealing with Pipeline Bottlenecks ............................. 71
   3.3 Micro-architecture Design of FastBCI ................................ 78
3.3.1 Micro-architecture of BCIE ........................................ 78
3.3.2 Mechanisms of BCIE ............................................. 80
3.3.3 Handling Cache-block Alignment Issues ....................... 87
3.4 Evaluation Methodology .............................................. 90
  3.4.1 Benchmarks. ....................................................... 90
  3.4.2 Machine Parameters. ........................................... 92
3.5 Evaluation ............................................................ 93
  3.5.1 Performance on Micro-benchmarks ............................ 93
  3.5.2 Real Application Performance ................................. 95
  3.5.3 Secure Deallocation Performance ............................ 103
  3.5.4 Performance of OS Survivability Scheme using FastBCI for Checkpoint Creation ................................................. 105
4 Conclusions ..................................................................... 109
  4.1 Efficient Architecture Support for Operating System Survivability .. 109
  4.2 Architecture Support for Improving Bulk Memory Copying and Initialization Performance ..................................... 110

Bibliography ................................................................. 112
LIST OF TABLES

Table 2.1 Machine configurations and parameters. .............................. 42

Table 3.1 Machine configurations and parameters. .............................. 91
LIST OF FIGURES

Figure 1.1 memcpy implementation in various instruction sets. For PowerPC (SSE2), the source address is in R2 (esi), the destination address is in R3 (edi), and the region length is in the count register (ecx). .................. 10

Figure 1.2 Limit speedups of applications with bulk copying engine, assuming perfect instruction granularity, perfect pipeline, and perfect cache. Details of applications and machine parameters can be found in Section 3.4. .............. 12

Figure 2.1 Call chain of system call (a) and exception handling (b) or other kernel routines................................................................. 23

Figure 2.2 Illustration of Linux kernel elf_core_dump vulnerability .............. 24

Figure 2.3 Code segment from strncpy_from_user, showing the breakpoint location that triggers checkpoint creation at line [**].......................... 27

Figure 2.4 Checkpoint Creation Modes: software memory copying (a), CPTCR instruction (b), and bulk memory copying using memory-side Copy Engine [57, 59](c). ................................................................. 31

Figure 2.5 Illustration of Protection Cache Structure........................................ 36

Figure 2.6 Protection Cache Overflow Mechanisms: no overflow protection (a), overflow to Bloom Filter (b), and conversion to multiple Bloom Filters showing two bloom filters per set (c). ......................................................... 37

Figure 2.7 Number of checkpoint creations per billion instructions (NCPBI) (a), and Average checkpoint size in kilobytes (b).................................. 45

Figure 2.8 Average checkpoint size in kilobytes for various benchmarks.......... 46

Figure 2.9 Execution time overhead on instruction per cycle(IPC) of checkpoint creation for various benchmarks................................................. 48
Figure 2.10 Percentage of increase in number of memory references due to various attack detection schemes. For all schemes, the hardware structure for storing protection information on chip is fixed at 3.75KB. 49

Figure 2.11 Performance overhead of attack detection scheme using protection cache with Backup Bloom Filter or Multiple Bloom Filters. 51

Figure 2.12 Overall performance overhead of OS survivability scheme using memory-side Copy Engine [57, 59] for checkpoint creation and using protection cache with Multiple Bloom Filters for attack detection. 52

Figure 3.1 The effect of ROB-clog (a) and dependence (b) bottlenecks on pipeline performance. Execution without bottlenecks (c). 69

Figure 3.2 Cycle Per Instruction (CPI) breakdown due to regular processor execution & stalls, and stalls due to ROB clog and dependence bottlenecks. Details of applications and machine parameters can be found in Section 3.4. 70

Figure 3.3 Cumulative percentage of instruction distances of back-to-back copying instructions for various benchmarks. 79

Figure 3.4 Illustration of Bulk Copying and Initialization Engine (BCIE). 80

Figure 3.5 Handling regions that are not page aligned. 83

Figure 3.6 Illustration of fine grain dependence checking. 85

Figure 3.7 Handling regions that are not block-aligned. 88

Figure 3.8 Normalized execution time per copying operation (a), and per initialization operation (b). 94

Figure 3.9 Speedup ratios of various schemes. 96

Figure 3.10 Speedup ratios (a) and the number of L2 cache misses (b) of Apache with various cache affinity options. 98

Figure 3.11 Speedup ratios (a) and the number of L2 cache misses (b) of Iperf with various cache affinity options. 100

Figure 3.12 Speedup ratios (a) and the number of L2 cache misses (b) of Iozone3 with various cache affinity options. 101
Figure 3.13 Speedup ratios over loop-based implementation for 8MB L2 cache with various cache affinity options.................................................. 103

Figure 3.14 Execution time overheads for heap-intensive benchmarks with various implementations of secure deallocation. (Noc = non-cacheable, Neu = cache neutral, and C = cacheable) .......................................................... 104

Figure 3.15 Overall performance overhead of OS survivability scheme using Copy Engine [57, 59] or FastBCI with optimal cache affinity option combination for checkpoint creation.......................................................... 107
Chapter 1

Introduction

Motivation. Computer systems are increasingly becoming more interconnected and being used for critical computation tasks. Along with such changes, more security attacks have appeared that exploit software vulnerabilities and design errors. Attacks are targeted towards user applications, as well as the Operating System (OS). At the core of an OS is the OS kernel, a fundamental software layer between the hardware and user applications, which serves to provide machine abstraction and security protection for user applications. The vulnerabilities of the kernel to security attacks are quite prevalent and many new ones are continuously discovered and reported by US-CERT almost every week [83].

In the future, kernel security will be of even greater concern. The kernel source code is growing in size and complexity to satisfy the functional requirements of the OS, and this complexity provides a fertile ground for future vulnerabilities [52, 70].
In addition, sensitive data is increasingly stored in computer systems and a successful attack on the kernel provides the attacker with unrestricted access to the system’s data, so the incentive to attack the kernel will grow accordingly. Finally, the number and variety of kernel modules are growing. A module is a piece of code that can be loaded/unloaded into the kernel on demand, in order to extend the functionality of the kernel without recompiling or rebooting the system. It is estimated that kernel modules represent 70% of the Linux kernel code [41]. One popular type of modules is device drivers. Quite often these device drivers are programmed with performance and compatibility as the main goals, without sufficient attention to security and reliability [3, 4, 7]. As a result, they tend to have more vulnerabilities than the kernel itself, with a study reporting that Linux device drivers have a bug frequency that is 3–7× higher than the rest of the OS [16].

A successful security attack on the kernel has a profound impact. The kernel runs at the highest processor privilege level and can manipulate hardware resources freely. Therefore, a successful kernel security attack exposes the entire computer system to malicious intruders. Even an unsuccessful attack or a Denial of Service attack can easily crash the kernel. Kernel crash is highly undesirable because of the damages it causes, such as the termination of all running processes and a long period of unavailability due to subsequent reboot.
1.1 Providing Operating System Survivability

Due to vulnerabilities of the OS and the dire consequences of successful security attacks on it, researchers have for a long time studied ways to provide survivability for the OS or the system. Survivability is the ability to operate in the presence of security faults [24]. The premise of survivability is that despite our best effort in making systems secure, security faults are inevitable [23]. Given that security faults will occur, survivability seeks to detect such occurrence, isolate the fault, and recover from it. Hence, a survivability scheme consists of three components that work together: (1) Security attack detection mechanism, (2) security fault isolation, and (3) recovery mechanism that resumes normal system operation.

Supporting survivability is challenging. One reason is that survivability cannot be trivially constructed from combining a security scheme and a recovery scheme, due to several factors. First, many security protection mechanisms are simply incompatible with survivability. Many of them were originally designed for protecting a user application and not the kernel. In a user program, the goal of an attack is typically to alter the program control flow or reveal sensitive data, rather than to crash the program. Indeed, the goal of many security protection mechanisms is to convert control flow hijacking attempts into application crash [54, 76, 80], since application crash is relatively acceptable than hijacking or information loss. Attacks on the kernel often have a goal of crashing it, so if the kernel crashes as a result, the attackers have achieved
their objective.

Secondly, although there are abundant recovery schemes that have been proposed for recovering from *hardware faults* (such as [27, 33, 38, 53, 68, 74, 77] among others), they are not capable of recovering from security faults. Traditional recovery schemes provide the ability to save and recover state efficiently. The recovery process is simple: rolling back the program execution to a point prior to the fault and *identical re-execution* of the program (possibly at a different node in a multiprocessor machine). In security attacks, such rollback and identical re-execution will simply replay the attack. Hence, unlike traditional fault recovery techniques, kernel survivability needs *non-identical re-execution*, in which the kernel is restored into a “good” state but the fault has to be isolated and neutralized before re-execution. In addition, traditional fault recovery schemes treat the system as consisting of a *single state*. However, the kernel state consists of states from *multiple* user processes that interact with it, hence these various states must be separated from each other. When only one process is attacked and the attack propagates to the kernel, the correct recovery behavior is not to restore the entire kernel state, but rather to restore the kernel state of that user process without affecting the kernel state for other user processes.

There have long been active researches in survivability [19, 20, 31, 36, 39, 55, 61, 62, 73]. Unfortunately, current survivability techniques are expensive and/or not sufficiently robust. They primarily rely on a *replica approach*, in which multiple non-identical or diverse instances of a system or a program are employed. The drawback
of the replica approach is that running \( N \) replicas require \( N \) times as many resources. Besides being a very expensive approach, a more fundamental weakness is that the effectiveness of the approach relies on the assumption that replicas do not share the same vulnerabilities. In practice this assumption can be hard to fully achieve [24]. Independent software teams still tend to make similar wrong assumptions, leading to common bugs [51], while naturally diverse systems may turn out to rely on the same (vulnerable) libraries [1].

This dissertation goes beyond past studies and presents an efficient architecture support for OS survivability. Our mechanism includes three inseparable components that work together: attack detection, fault isolation, and recovery mechanisms. A malicious (or attacked and subverted) application interacts with the kernel via system calls or exception handlers. Hence, we structure our survivability mechanism around system call and exception handler boundaries. To support recovery, a checkpoint is automatically created prior to the kernel serving a system call or an exception. We distinguish the kernel state of different user processes and the checkpointed state is specific to the user process making the system call or raising the exception.

If an attack is detected before the system call/exception handling is complete, the kernel state is rolled back to the previously created checkpoint. Since a checkpointed kernel state is user process-specific, the recovery does not affect the progress of other user processes. To avoid the repeat of an attack, after kernel state rollback, the user process that is involved in the attack is suspended, while other system operations are
resumed.

Our survivability scheme also includes a security attack detection mechanism. Attack detection mechanism determines the security fault coverage that the survivability scheme can provide. While many existing attack detection mechanisms (e.g. Mondrian Memory Protection (MMP) support [34]) can be easily integrated into our survivability scheme, in this dissertation we present a low performance overhead attack detection mechanism using word-granularity write protection with architecture support. Our attack detection mechanism detects contiguous buffer overflow attacks, which is the most popular form of kernel attacks. It also serves as a fault isolation mechanism to mark the boundary of kernel stack and kernel pool of a user process, so that an attack through a user process cannot corrupt the kernel state of other user processes. In the future, if other types of attacks become prevalent and new attack detection mechanisms emerge, they can still be easily integrated into our survivability scheme to provide more comprehensive security fault coverage.

Overall, our OS kernel survivability scheme makes the follow contribution:

1. *Low performance overhead*. Because kernel is performance sensitive, the survivability scheme should not result in a large performance overhead. We achieve low performance overhead through efficient architecture support.

2. *Automatic*. The kernel recovery from security attacks should not be conditional upon human intervention because a long period of unavailability may be the
goal of an attack. In our design, kernel recovery is automatically triggered when a security attack is detected, and the detection and recovery events are logged and users alerted immediately.

3. *No recompilation.* The survivability mechanism should be integrated easily into an existing kernel without requiring recompilation. We assume that the kernel source code is not always available, and hence the ability to recompile kernel cannot be assumed. In our design, checkpoint creation and deletion, protection manipulation, and state rollback are achieved through dynamic instrumentation of the kernel at system call boundaries. This dynamic instrumentation avoids the need to recompile the kernel and preserves the functional stability of the kernel because its code is preserved.

We evaluate the performance of the proposed mechanism on a full system simulator running the Linux OS. We test the protected Linux with a range of real world kernel attacks, and confirm that all the attacks are detected, and the OS survives through all the attacks and keeps its normal function intact. Although the hardware support we propose is quite simple, the average and worst-case performance overheads due to our survivability mechanism are low, at only 2.1% and 4.7%, respectively.
1.2 Improving Bulk Memory Copying and Initialization Performance

The checkpointing-based security recovery mechanism extensively relies on bulk (large-region) memory copying and initialization operations to checkpoint the healthy system states. Through our experiments, we find that it contributes a significant portion of (on average 87% for kernel intensive benchmarks we evaluated) of performance overhead to our OS survivability scheme. Therefore, more carefully-designed architectural innovations are needed to improve the performance of bulk memory copying and initialization operations.

Besides being used in the recovery mechanism of our survivability scheme, we find that bulk memory copying and initialization is in fact one of the most ubiquitous operations performed in current computer systems by both user applications and OSes. Critical OS functions such as buffer management and virtual memory management rely heavily on memory copying and initialization. For example, kernel buffer copying (through calling `memcpy`) and user-kernel buffer copying (through calling `copy_from_user` and `copy_to_user` [30]) are performed extensively to support TCP/IP processing and file I/O operations. Many TCP/IP intensive applications (e.g., Apache web server) indirectly spend a significant portion of execution time performing memory copying. A typical OS also performs bulk initialization and copying to support various virtual memory management functions such as page allocation,
copy-on-write, and swapping pages in and out of the physical memory. User applications also perform bulk memory copying and initialization for buffer management and string handling, typically through calling \texttt{memcpy} and \texttt{memset} \cite{12}.

The impact of supporting efficient bulk memory copying and initialization is twofold. First, it can significantly reduce the performance overhead associated with checkpoint creation and hence improve the performance of our OS survivability scheme. Secondly, it can also improve the performance of OSes and user applications that rely on bulk memory copying and initialization in general.

In current systems, there are at least three ways memory copying is supported at the instruction level (Figure 1.1): \emph{explicit loop} – using a loop of loads and stores, \emph{implicit loop} – using an instruction that is expanded into a series of loads and stores at instruction decode time (e.g., \texttt{rep movsd} in x86 \cite{21, 48} and \texttt{mvcl} in IBM S/390 \cite{42}), and \emph{vector extension} – using vector load and store instructions.

Comparing the three schemes, it is clear that the explicit loop and implicit loop implementations suffer from a granularity inefficiency problem, in that an excessive number of TLB accesses and cache accesses occur during copying. Instead of checking the TLB once for each page involved, each load and store instruction incurs a TLB access. Similarly, rather than accessing the cache once for each cache block involved, each load and store instruction incurs a cache access. In this regard, vector instructions achieve better (but not perfect) granularity efficiency since each load or store can work on data as wide as the width of the vector registers, e.g. 128 bits in SSE2.
**Explicit Loop**

<table>
<thead>
<tr>
<th>PowerPC:</th>
<th>Vector Extension</th>
<th>Implicit Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>loop: lwz R1, 0(R2) addi R2, R2, 4 stw R1, 0(R3) addi R3, R3, 4 bdnz loop</td>
<td>loopt: movaps xmm0,0(esi) add esi, $16 movaps 0(edi),xmm0 add edi,$16 sub ecx,$16 jnz loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IBM S/390:**

|  | la R2, dst_addr mov esi, src_addr |
|  | la R3, dst_len mov edi, dst_addr |
|  | la R4, src_addr mov ecx, len |
|  | la R5, src_len rep movsd |

**x86:**

|  | mvc1 R2, R4 |

Figure 1.1: `memcpy` implementation in various instruction sets. For PowerPC (SSE2), the source address is in R2 (esi), the destination address is in R3 (edi), and the region length is in the count register (ecx).
Future vector instructions may increase the width even more (e.g. Intel Larrabee will likely support 512-bit loads/stores [63]).

However, an efficient memory copying and initialization requires more than just granularity efficiency. In fact, surprisingly, we found that granularity efficiency alone does not improve the performance of memory copying by much. The overall performance of memory copying is impacted by how the code that follows the copying performs, more than by the efficiency of the memory copying operation itself. This is illustrated in Figure 1.2, which shows limit speedups of various copying-intensive benchmarks such as Apache web server (sender and receiver side), iperf network benchmarking tool (sender and receiver side), and iozone I/O performance tool, and their average. The “perfect granularity” bars show speedups when each kernel buffer copying function is replaced with a single copying instruction, assuming such an instruction can handle buffers with an arbitrary width and alignment. Furthermore, in the perfect granularity, we assume that the TLB and caches are only accessed as few times as possible, i.e. the TLB is checked once for each page involved while the cache is accessed once for each cache block involved. The “perfect pipeline” bars show limit speedups when the memory copying instructions do not restrict younger instructions from executing and retiring, except when they are data-dependent on the copying instruction. Finally, the “perfect cache” bars show limit speedups when the code that follows memory copying operations does not suffer from cache misses when it accesses the copying regions, and does not suffer from extra cache misses when it
accesses non-copying regions. “Perfect cache” represents the case in which memory copying produces all the useful prefetching effect for the code that follows the copying but none of the harmful cache pollution effects.

Figure 1.2: Limit speedups of applications with bulk copying engine, assuming perfect instruction granularity, perfect pipeline, and perfect cache. Details of applications and machine parameters can be found in Section 3.4.

Interestingly, the figure shows that perfect granularity alone contributes to only 8% speedup on average. However, when the copying instructions do not restrict the execution of younger, non-dependent instructions, the average speedup jumps threefold to 24%. When the copying instructions do not incur cache pollution and introduce a beneficial prefetching effect to younger instructions, the average speedup jumps almost fivefold to 37%. The figure illustrates that while granularity efficiency is important, the real key to significantly improving the performance of copying/initialization is removing pipeline and cache bottlenecks of the code that follows copying operations.

In this dissertation, we seek to provide an instruction and architecture support for Fast Bulk Memory Copying and Initialization (FastBCI). FastBCI consists of a scalar
instruction that can perform copying/initialization involving a large memory region. Unlike an implicit loop approach, the instruction is not expanded into loads and stores. Rather, it is executed on an engine that performs the copying/initialization and keeps track of its progress.

Furthermore, we found two pipeline bottlenecks that severely restrict the potential performance of a copying instruction. The first is that copying/initializing a large memory region takes a long time, and as a result, the copying instruction stalls at the head of the reorder buffer (ROB), eventually stalling the entire pipeline. To get around this, we design a mechanism that allows the copying instruction to retire early prior to completion, but safely. Moreover, we also discuss correctness criteria needed for retiring copying instructions early, and present one efficient implementation.

The second pipeline bottleneck we discovered arises from a situation in which dependent instructions often stall waiting for the completion of a copying instruction, and they increase the critical path of instruction execution. Clearly, the larger the copying region is, the longer it takes for the copying instruction to complete, and the more likely some dependent instructions stall in the pipeline. To avoid such stalls, we propose a mechanism to track which blocks have completed their copying, and allow dependent instructions to wake up. As a result, as copying progresses, more younger instructions avoid stalls and wake up, execute, and retire.

Figure 1.2 also illustrates the importance of the cache effect of memory copying on the code following the copying operations. To address this issue, we allow cache
affinity options to be specified as a part of the copying instruction parameters. We allow three option values that can be applied individually to the source and destination copying regions: *cacheable* (the region is brought into the cache as it is accessed), *non-cacheable* (the region is not brought into the cache and is evicted if all or parts of it are cached), and *cache neutral* (no new blocks from that region are brought into the cache but existing blocks are retained in the cache).

Overall, our innovation on improving bulk memory copying and initializing performance makes the following contributions:

- It shows that the key to improve the performance of copying/initialization is solving the pipeline and cache bottlenecks of the code that follows copying operations.

- It analyzes the micro-architecture and multiprocessor correctness requirements for mechanisms that avoid the pipeline bottlenecks through allowing a copying instruction to retire early.

- It proposes an architecture support for efficient bulk copying and initialization, achieving granularity efficiency while avoiding the pipeline and cache bottlenecks. To evaluate its performance, we use a cycle-accurate full system simulator with an out-of-order processor and two-level caches, running Linux OS. On micro-benchmark evaluation, FastBCI outperforms current implementations for a wide range of region sizes, by up to $2 \times$. On real benchmark evaluation, for the
benchmarks we evaluated, on average FastBCI achieves a 23.2% speedup with pipeline improvement, very close to the perfect pipeline case. Using FastBCI to perform checkpoint creation in our OS survivability scheme, the performance overhead is further reduced to an average of 1.0%.

- It proposes and evaluates cache affinity options including cacheable, non-cacheable, and cache neutral, as parameters of copying instructions. It shows that cache affinity options affect performance tremendously, with different regions favoring different affinity options. On average, the best affinity option outperforms the worst affinity option by 10%. Interestingly, we found that the cache neutral option gives the most robust performance for the benchmarks that we evaluate.

### 1.3 Organization of the Dissertation

The rest of this dissertation is organized as follows: Chapter 2 describes our contribution of the OS survivability scheme in detail. The chapter presents the attack model and attack assumptions that we use (Section 2.1), illustrates our OS survivability scheme in detail (Section 2.2), describes our evaluation environment (Section 2.3), presents and analyzes the evaluation results (Section 2.4), and describes prior related works (Section 2.5).

Chapter 3 describes our innovations on improving bulk memory copying and
initialization performance. It describes several related prior works (Section 3.1), illustrates the architecture support for FastBCI (Section 3.2), details the micro-architecture design (Section 3.3.1), presents the evaluation environment (Section 3.4) and discusses the evaluation results (Section 3.5).

Finally, Chapter 4 summarizes our findings and concludes this dissertation.
Chapter 2

Efficient Architecture Support for Operating System Survivability

This chapter is organized as follows: Section 2.1 describes the security attack model and attack assumptions that we use, Section 2.2 discusses the design of our OS kernel survivability scheme in details, Section 2.3 presents the evaluation environment, and Section 2.4 presents and analyzes evaluation results. Finally, Section 2.5 discusses related works.
2.1 Security Model and Kernel Attacks

In this section, we present the security attack model and attack assumptions that we use for our survivability scheme (Section 2.1.1), as well as the targets (Section 2.1.2) and mechanisms (Section 2.1.3) of kernel security attacks.

2.1.1 Security Model and Assumptions

The goal of a kernel attack can be to either crash the kernel or to execute malicious code at the kernel’s privilege level. We assume that the attacker can achieve these goals by exploiting vulnerabilities in the kernel, so our scheme must provide survivability for both attack goals.

Attacks on the kernel can be launched remotely by attackers through attacking a vulnerable application. After the attackers gain control of the application, it can be used to attack the kernel. Attacks on the kernel can also be launched locally if attackers (e.g. malicious insiders) already have access to the machine. To cover both scenarios, we assume that the attack can be carried out through vulnerable as well as malicious applications.

Furthermore, we assume that the kernel and kernel modules such as device drivers are vulnerable but not malicious. Dealing with malicious modules is very difficult because such code already runs in the privileged mode. Fortunately, OS developers often apply mechanisms to authenticate module signatures to avoid accepting a mod-
ule developed by unknown parties.

There are various categories of software vulnerabilities that can be exploited by an attacker. In a buffer overflow vulnerability, the lack of buffer bound checking allows a long string input to overwrite adjacent bytes beyond the buffer. In an integer overflow vulnerability, a signed-integer value is not checked for a negative value that indicates an overflow before it is used. In a format string vulnerability, the format string in the printf family of functions is taken as an input parameter, which may result in a read from or a write to a location specified in the format string argument. These vulnerabilities are all common in user applications. Since buffer overflow occurs due to buffer copying, it typically causes a contiguous overwrite of bytes adjacent to the buffer. Integer overflow and format string vulnerabilities, on the other hand, can sometimes be exploited to perform a random overwrite, i.e. overwrite any location chosen by the attacker.

Detecting various attacks that exploit various categories of security vulnerabilities requires a comprehensive detection mechanism (e.g. Mondrian Memory Protection (MMP) [34]). However, such a comprehensive detection mechanism is obtained at the cost of significant performance overhead (22% slowdown for kernel intensive benchmarks). We observed that contiguous overwrites attack is the type of attack that has the most occurrence in kernel security attacks. This is due to several reasons. First, while random overwrites are quite common for attacks on user applications, they are seldom attempted for the kernel because of the lack of opportunities to do so.
In the kernel, in theory the printk function can be invoked without specifying the format string, which exposes it to the format string vulnerability. In practice such vulnerabilities rarely exist. This is because printk is mainly used internally by the kernel to print out debugging information, and is not called directly from outside the kernel. In addition, printk debugging printouts typically uses explicit (hard-coded) format strings, so there is little opportunity for the attacker to provide their own format strings to it. Furthermore, opportunities to exploit integer overflow for random overwrites are also very limited because integer values are rarely used to directly index the memory through array indexing. Due to the limited size of kernel stack, arrays are rarely used. However, integer overflow may be exploited in conjunction with buffer handling, and hence can cause contiguous overwrites. Consequently, in contrast to regular user applications, the kernel typically only has vulnerabilities for attacks that perform contiguous overwrites.

As a result, in this dissertation we focus on providing attack detection mechanism for contiguous overwrite attacks. We present a low performance overhead detection mechanism that can detect contiguous overwrites before the attack contaminates beyond the buffer boundary (Details in Section 2.2.2). However, if other categories of kernel attacks become prevalent in the future and new architectural detection mechanisms become available, they can still be easily integrated to our survivability scheme.
2.1.2 Target of Kernel Attacks

Based on the memory locations that the attack tries to overwrite, we categorize kernel attacks into *kernel stack tampering* and *kernel pool tampering*. Both kernel stack and pool reside in the kernel space and are vulnerable to attacks.

**Kernel stack tampering.** A user process switches to execute as a kernel process when it does a system call or an exception is raised. Each kernel process has a small stack in kernel space for holding local variables and execution context information. To simplify stack management, the kernel often relies on a fixed size stack, such as 8KB on a 32-bit machine and 16KB on a 64-bit machine [79]. Because local variables in the stack are adjacent to critical kernel data, such as a process’ `thread_info` field at the end of the kernel stack, a buffer overflow in this stack can cause significant damage by overwriting this critical data.

**Kernel pool tampering.** Similar to the heap memory, the kernel dynamically allocates and frees data structures in a space referred to as the *kernel memory pool*. A pool is a collection of more specialized storage referred to as *caches*. A cache is a collection of contiguous physical pages, and has two types. A *specific cache* stores one type of objects. Each type of kernel data structure, such as `inode`, has a specific cache associated with it. A *generic cache* stores multiple types of objects such as buffers requested by a kernel process. An object is a granularity of allocation and de-allocation, and it is stored in a structure referred to as a *slab*, which is similar to
a heap chunk. Slabs in a cache are laid out contiguously without protection between them. This provides a vulnerability because many slabs contain buffers that receive input from the user process, and a buffer overflow in one slab can corrupt neighboring slabs that may contain critical data.

Finally, some of the kernel space resides in the virtual memory area (VMA) [65]. We note, however, that this virtual memory area is not used often for storing critical kernel data. On the other hand, it allows a simple and effective security protection using non-accessible pages to surround it buffers [65].

2.1.3 Mechanism of Kernel Attacks

A kernel attack starts from the attacker gaining control of a user application. The attacker then continues to attack the kernel through the interaction of the kernel and the application. In general, the kernel attack consists of three steps. The first step (trap-to-kernel) seeks a way to elevate the execution privilege level by trapping into the kernel. The trap-to-kernel step can be carried out through a system call, or directly through a kernel exception handler, as illustrated in Figure 2.1.

Figure 2.1a shows the call chain example of a kernel routine which is invoked by the user application through the open system call. The call invokes a trap handler. The trap handler switches into the privileged mode and calls a system call handler such as sys_open, which in turns calls a kernel function filp_open. In this call chain, user arguments are passed through the entire sequence. The attacker attempts
to attack the kernel by passing invalid or malicious arguments. Another point of entry for the trap-to-kernel step is when a kernel function is triggered by an exception or interrupt (Figure 2.1b). For instance, a core dump handler is triggered when a user application crashes. While no user arguments are accepted, these kernel functions still read environment data from the user space, allowing the attacker to affect the functions by controlling the data that will be read by them.

Figure 2.1: Call chain of system call (a) and exception handling (b) or other kernel routines.

Once a trap to the kernel occurs, the kernel function executes with the arguments passed by the user process. The second step (vulnerability exploitation) then exploits vulnerabilities or bugs in the kernel code. For example, if the kernel has a buffer overflow vulnerability, a lack of bound check on a long argument string may cause an overflow to bytes adjacent to a buffer in the kernel stack or pool.

In order to give a more concrete illustration, we detail an example real-world attack called the elf-core-dump attack [6]. Figure 2.2 shows the code for kernel function
elf_core_dump, which is triggered when an ELF executable crashes during execution, and which dumps the process’ memory image to the disk. The code copies arguments from the crashed process’ stack to a kernel slab. The next line of code has a vulnerability in that while `len` is checked against its upper-bound value (ELF_PRARGSZ), it is not checked against its lower-bound value. Hence, through integer overflow, an attacker can set a negative value for `len`, and that value would then be passed to the buffer copying function `copy_from_user`, which overflows the buffer due to the lack of buffer bound check. We note that once the vulnerabilities are discovered, the kernel can be patched to remove them. However, the kernel and its modules consist of millions of lines of code and it is very difficult to identify and remove all vulnerabilities before the code is deployed.

```c
static int elf_core_dump(long signr,
    struct pt_regs *regs, struct file * file){
    ...
    int i, len;
    len = current->mm->arg_end -
        current->mm->arg_start;
    if (len >= ELF_PRARGSZ)
        len = ELF_PRARGSZ - 1;
    copy_from_user(&psinfo.pr_psargs,
        (const char *)current->mm->arg_start, len);
    ...
}
```

Figure 2.2: Illustration of Linux kernel elf_core_dump vulnerability
2.2 Architecture Support for Operating System Survivability

In this section, we will describe the components of our survivability scheme in the following order: recovery (Section 2.2.1), detection (Section 2.2.2), and fault isolation (Section 2.2.3).

2.2.1 Recovery Mechanism

To design a checkpointing and recovery mechanism for our scheme, we first address the questions of when to create a new checkpoint and what state should be part of the checkpoint, when a checkpoint can be deleted, where the checkpoints are stored, and how to perform checkpoint operations efficiently.

Checkpoint Creation

The checkpoint should be taken when good state is available, preferably just before an attack may occur. Our analysis in Section 2.1 indicates that entry points for attacks are system call or exception handlers. Consequently, we create a new checkpoint for kernel stacks at the entry point to each system call and exception handler function. Since the objects in kernel pool are typically modified through calling kernel buffer management functions (e.g. memcpy, strncpy), the checkpoint for an object in the kernel pool is created whenever such functions are invoked.
Checkpoint Components

We should checkpoint the state that is the potential target of modifications in an attack. One challenge is that unlike hardware fault tolerance mechanisms, we must distinguish the kernel state of different user processes, and the checkpoint should only capture the kernel state of the user process making the system call or exception. The kernel state of a user process includes a kernel stack, and parts of the kernel pool that are written due to the system call or exception handling. As described in Section 2.1, the same kernel state is also the primary targets for overwrites or tampering. Hence, our checkpoint consists of three components: kernel stack, slabs that reside in the kernel pool, and the process’ execution contexts that include register values and pid of the process. For all these components, we need to know their ranges of addresses (base address and offset) in order to copy them into the checkpoint.

For the kernel stack, the offset is always fixed because the size of kernel stack is static (8KB for a 32-bit machine, and 16KB for a 64-bit machine). Since the kernel stack is allocated when the process is created, its base address is always known prior to the process making any system calls, so its address range to be checkpointed is determined statically. However, slabs in the kernel pool are allocated dynamically, and which slab would be overwritten by system call handling is not known until a kernel function that will overwrite them is called. Hence, we must incrementally checkpoint each slab as it is discovered. To achieve this, we must monitor the address ranges in
the kernel pool written during system call or exception handling just before they are overwritten. One challenge is that there is a very high number of kernel functions that attempt to write to the kernel pool memory. Fortunately, we found that all of them rely on a set of common kernel library functions that perform buffer management, such as `strncpy_from_user` for copying contiguous bytes from the user space to a buffer in a kernel slab. A segment of this function is shown in Figure 2.3. To figure out the buffer parameters passed to this function, we analyze buffer management functions and insert a breakpoint at line [**]. When the breakpoint is encountered in execution, we can discover the base address and offset of the buffer by looking up the values in register `edi` and `ecx`, respectively.

Figure 2.3: Code segment from `strncpy_from_user`, showing the breakpoint location that triggers checkpoint creation at line [**].

Note that these buffer management functions cannot be patched easily to make them secure. They have no way to know whether the buffer address parameter they receive is valid or not, so it is up to the caller functions to supply valid address ranges
to the buffer management functions. Under attacks, the buffer address range may even go beyond the slab that contains the buffer. Unfortunately, there are many such callers, and some of them lack the necessary checking that make sure that they pass valid buffer ranges to the buffer management functions, such as illustrated in Figure 2.2. However, with our checkpointing scheme, even if buffer ranges passed to buffer management functions are invalid, we always create the checkpoint of that invalid address range to undo subsequent overwrites to it.

Checkpoint Management

The next challenge that needs to be addressed is how many checkpoints need to be kept in the system and for how long. This depends on how much time the attack detection mechanism needs to detect an attack after the first kernel state modification is attempted by the attacker. Our survivability scheme naturally is capable of accommodating many existing attack detection mechanisms. However, to make the challenge reasonable, we need to pick detection mechanisms that give reasonable properties. The example attack detection (Section 2.2.2) provides two essential guarantees. First, an overwrite that begins in a given kernel stack or slab is detected as soon as it extends beyond that stack or slab. Second, overwrites that occur solely within a kernel stack or a kernel slab are detected before the system call or exception handler returns control to the user process. These two guarantees allow us to limit the time and life-span of our checkpoints. The maximum number
of checkpoints that need to be kept is bounded by the maximum call chain depth when servicing a system call or exception, and the checkpoints can be organized in a *checkpoint stack*. When a system call or exception handler is invoked, we push a new kernel stack checkpoint onto the checkpoint stack, and each time subsequent kernel functions attempt to write to a buffer in the kernel pool, the buffer is also pushed to the checkpoint stack. Just before the kernel handler returns to user space, an additional scan of the kernel stacks and pools involved in the checkpoint can be performed to detect an attack. If an attack is detected, we pop a checkpoint from the stack and restore its state, and repeat this pop-and-restore until the stack is empty. If no attack is detected, the checkpoint stack is simply de-allocated.

We note that the checkpoint stack is essential for ensuring survivability. Hence, it needs to be stored in memory securely. For this, a virtual memory area of the kernel is allocated for storing the checkpoint stack. To protect this checkpoint stack from attacks or unintentional tampering, we employ two protection mechanisms. First, the checkpoint stack area is surrounded by non-accessible pages, so that any intentional or accidental buffer overflow from outside the checkpoint stack area would trigger an exception before it can write to the checkpoint stack. In addition, we write-protect the pages that keep the checkpoint stack. Only when the stack is modified this write protection is removed, and after the modification the write protection is restored.
Dynamic Instrumentation for Checkpoint Management

We mention in Chapter 1 that one of the design criteria is that we should avoid OS source code modifications and recompilation. Checkpoint creation, checkpoint stack management, checkpoint deletion, and breakpoints must be weaved dynamically to an existing kernel binary. To achieve that, we use dynamic interception through Kernel Dynamic Probes (Kprobes) [60], a mechanism to register kernel execution breakpoints. Kprobes has been the default kernel configuration option since Linux kernel version 2.6.9, so it is widely available in most Linux systems.

Our breakpoint handlers consist of several types. The first type is prefix handlers which are inserted into the system calls, exception handlers and other kernel functions that can serve as the entry points to the kernel, and are executed prior to the functions themselves. Prefix handlers are used for triggering checkpoint creations. The second type is a suffix handlers which are inserted into the end of system calls, exception handlers and other kernel functions that can serve as the entry points to the kernel, and are executed after the execution of the functions themselves, but before the functions return. Suffix handlers are used for performing checkpoint stack de-allocation. We also use exception handlers that are invoked when a detected attack raises an exception, and are used for restoring the kernel state. Finally, PC handlers are invoked when instructions with the specified PC are invoked, and are used for identifying buffer ranges in the kernel pool to be checkpointed.
Architecture Support for Checkpoint Creation

A software-only checkpoint uses a loop of loads and stores to copy the kernel state into the checkpoint area. It may have a non-negligible performance impact because checkpoint creation cannot be overlapped with regular computation (Figure 2.4(a)). We explore several possible architectural mechanisms for checkpointing to determine if they would be useful.

![Diagram showing checkpoint creation modes](image)

Figure 2.4: Checkpoint Creation Modes: software memory copying (a), CPTCR instruction (b), and bulk memory copying using memory-side Copy Engine [57, 59](c).

Our first attempt is to extend the ISA by adding a new CPTCR instruction which saves the processor context (registers) and a specified range of memory addresses. This instruction saves instruction fetch bandwidth, but can be internally unrolled into loads and stores or micro ops. We do not add an instruction for rollback because rollbacks are expected to be rare events. To overlap checkpoint creation with regular
computation at the processor, we add a checkpoint address range register (CARR), which records the range of addresses (base address and offset) from which checkpoints are being created. When a checkpoint creation is initiated, its address range is recorded in a CARR and the hardware begins to copy data to the checkpoint in the background. Meanwhile, the processor can continue executing instructions. However, if the processor writes to an address that conflicts with the CARR, the processor stalls until the checkpoint is completed (Figure 2.4(b)). Once the checkpoint creation is over, the CARR is marked as invalid, and the stalled processor is allowed to resume execution. Although this scheme is promising, in practice we found that the conflicting store occurs almost as soon as checkpoint creation starts, because the checkpoint creation occurs right before the corresponding kernel stack or slab are to be written. As a result, there is only a very limited overlap between checkpoint creation and regular kernel execution. In addition, there is still a large number of memory locations which are brought into the cache although they are not going to be re-used soon.

To avoid this cache pollution and improve overlap, we use the copying support proposed in [59], in which a copy engine is added for bulk copying in the memory controller (Figure 2.4(c)). Source addresses that across page boundaries are split and send to the copy engine page by page. A pending copy table in the processor records the copying region to detect conflicts. In a bulk copy operation, the source and destination address ranges are specified to the memory controller so it can copy the bytes from the source range to the destination range without involving the processor
and polluting the cache. At the start of checkpoint creation, the system enters a *bulk copy setup* phase, in which the source and destination address ranges are written to an on-chip Copy Control Unit (CCU) for virtual-to-physical address translation. The cache controller writes back (without replacing) all cached lines in that address range. During this setup time, a conflicting store from the processor would result in stalling the processor. After the write backs are completed, the system enters *bulk copy* phase. In this phase, the copy engine located in the memory controller performs the actual copying from the source to destination address ranges. The processor is allowed to resume execution if it has been stalled previously. However, if cache blocks in the source address range are modified, which is indicated by a match to the pending copy table, these modified blocks are not allowed to be written back to the memory, to ensure the integrity of the checkpoint.

### 2.2.2 Security Attack Detection Mechanism

One important component of our survivability scheme is attack detection mechanism. Our survivability scheme naturally provides the capability to accommodate many existing attack detection mechanisms. However, to restrain the performance overhead that can be incurred by the attack detection mechanism, it should be carefully chosen. As discussed in Section 2.1, contiguous overwrite attacks are the main threat to the kernel security. In this section, we show an example of detection mechanism that deals with contiguous overwrite attacks.
To guarantee an overwrite is detected as soon as it extends beyond the stack or slab boundary and is detected before system call or exception handling returns, we place write-protected delimiters around a kernel state to prevent an overwrite to continue past the delimiters. To minimize the storage overhead of storing delimiters and to avoid complicating the kernel memory allocators, we use word-size delimiters around each kernel slab, and word-alignment of slabs. This requires each word to have its own write protection bits. This protection would ensure that a buffer overflow beyond a kernel slab would promptly trigger an exception. To detect attacks on the stack, we simply make return addresses and other critical thread information non-writable. To add these delimiters without modifying the kernel source code, we insert prefix and suffix breakpoints to kernel slab allocation routine (\texttt{kmalloc}). The prefix handler intercepts the requested allocation size, while the suffix modifies the pointer returned by the allocation routine to adjust for the alignment and delimiter.

In the main memory, we associate one write-protection bit per kernel state word and the bits are linearly stored in a bit vector format in the main memory. The protection bit area is allocated at the virtual memory area and since its security is critical to the correctness of our attack detection, so it is surrounded by write-protected pages, and the protection area itself has temporal write protection that is only removed when it is updated.

Note that every store by the processor needs to be checked against this protection information in order to determine if the store is valid or should trigger exception.
Reading the protection bits in memory for each store instruction is obviously unacceptable. Therefore, we try to cache some of the protection bits in an on-chip structure. We observe that at any given time, the number of write-protected words is quite small, indicating that a small cache may be sufficient to keep most of the protection information for these words. We also observe that write-protected words are scattered around with a limited spatial locality. As a result, rather than caching the protection bit vectors, we simply cache the address of each individual word that is write-protected in a small protection cache, as shown in Figure 2.5. Therefore, a protection cache is basically a set-associative tag array for write-protected words. If there are more write-protected words than the set associativity of a set, the overflow bit is set and the information is stored into the protection bit vector in main memory. The protection cache is indexed by XORing all sections of the address bits of a store. The 30-bit tags in the indexed set are then compared to the word tag part of the store address. A match indicates that a store to a write-protected word is attempted, hence an exception is triggered. However, when the overflow bit is one, a mismatch is quite costly as it means that the store address must be checked against the protection bit vector information in the main memory.

To reduce the cost of a mismatch, we rely on two overflow mechanisms, as shown in Figure 2.6. The first overflow mechanism (Figure 2.6(b)) utilizes a bloom filter [22] for each cache set to summarize protection information that overflows the protection

\[1\] The purpose of this XOR-based indexing is to makes sure that all sets are uniformly utilized.
Figure 2.5: Illustration of Protection Cache Structure.

cache. The protection cache’s associativity is halved in order to make space for the bloom filter. On a mismatch at an overflowed cache set, the store address is matched against the bloom filter for that set. Since a bloom filter has no false negatives, a mismatch means that the store is not to an address that is write-protected. However, a match in the bloom filter may be a false positive, so the address needs to be checked against the protection bit vector in memory.

The second mechanism keeps the protection cache. However, on overflow, several cache lines in the set are grouped together and transformed into a bloom filter (Figure 2.6(c)). On another overflow, we have two choices: store the new address in the existing bloom filter, or group other cache lines in the set to make a new bloom filter for the new overflow. On the surface, aggressively turning cache lines into bloom filters seem unnecessary. However, further mathematical analysis reveals that the
combined false positive rate is minimized when we use the aggressive approach to ensure that the number of addresses kept by all bloom filters are roughly the same. We find that when the addresses are mapped uniformly to all bloom filters, the false positive rate increases much more slowly than when most of them are mapped to just one bloom filter. Hence, we conclude that the protection cache that converts to multiple Bloom Filter is the superior approach. Each bloom filter is a counting bloom filter [56] so that when an address’ write protection is removed, its address can be removed from the bloom filter without going to the main memory.

2.2.3 Fault Isolation

As long as the detection mechanism ensures that kernel state of different user processes are isolated against each other, we can always recover the kernel state of each individual user process from security attacks. When a security attack is detected,
an exception is raised and our exception handler is invoked. The handler rolls back the kernel state to the clean state prior to the system call or exception handler that causes the attack. Then the handler calls a kernel function `kill_proc` that suspends the user process by sending a `SIGSTOP` signal to the process, and invokes scheduler to continue normal operation by scheduling another process. The handler also logs the fact that an attack detection and kernel recovery have occurred. Optionally, the handler can be made to dump the memory image of the process for further analysis, and alarm users about the events.

Additionally, there are special cases that need to be taken care of when isolating faulty processes.

**Process dependencies.** In the presence of process dependencies, executing processes that have dependencies on the faulty process may result in unexpected consequences. Therefore, such processes must be identified and isolated as well. Since process dependencies can be built explicitly through Inter Process Communication (IPC) or child process creation (e.g. `fork()`), as well as built implicitly through data dependencies on shared data of the OS kernel, our fault isolation mechanism must be able to identify and isolate processes that have both explicit and implicit dependencies on the faulty process.

To identify processes that have explicit dependencies, we simply instrument system calls that handle IPC creation (e.g. `sys.pipe`) and process creation (e.g. `sys.fork`). Process dependencies built explicitly using such system calls are recorded in a data
structure and maintained along with the checkpoints. Once the kernel recovery occurs, the data structure is scanned to look up processes that have dependencies with the faulty one. A SIGSTOP signal is sent to those processes as well.

To identify processes that have implicit dependencies, a naive approach is to track data dependencies by instrumenting each load and store instructions. However, such an approach may incur tremendous performance overhead since it slows down all regular load and store instructions. We observe that implicit process dependencies only affect dependent process in the presence of kernel preemption on multi-threaded OSes. The notion is that if a process is not preempted in the middle of serving a system call, once being detected to be faulty and rolled back, processes that have data dependencies on it still views a consistent kernel state. Therefore, rather than tracking data dependencies, we instrument OS scheduler to track and record the kernel preemption of processes. Once an attack is detected, processes that preempt the faulty process are also rolled back and their system calls are re-executed. Note that the checkpoint management mechanism is slightly affected by this. Rather than being destroyed when system call returns, checkpoints maintained for processes that preempt another process must be kept until the system call of preempted process returns.

**Interrupts.** An interrupt may occur in the middle of the system call of a faulty process. Once the faulty process is detected and rolled back, the effect of the interrupt handling is also rolled back as a result. To deal with this problem, we simply record
the occurrence and execution context of interrupts through kernel instrumentation and re-execute interrupts after the kernel recovery.
2.3 Evaluation Methodology

2.3.1 Machine Parameters

We use a full system simulator based on Simics [75] to model an out-of-order 4GHz x86 processor with issue width of 4. For L1 instruction and data caches, we assume 32KB write-back caches with 64-byte block size, and access latencies of 2 cycles. The L2 cache is 8-way and has 1MB size, 64-byte block size, and an access latency of 8 cycles. The memory access latency is 300 cycles (or 75ns), and we model a split transaction bus with 6.4 GB/s peak bandwidth. For the protection caching support, we use a 3.75KB structure that is either configured entirely as a protection cache, as half a protection cache and half a bloom filter, or as a protection cache that can transform into multiple bloom filters (Section 2.2.2). The protection cache has 16 ways and each block stores a 30-bit tag. Each bloom filter entry is 120-bit wide and has six keys. Table 2.1 presents the machine configurations and parameters of the simulated machine.

2.3.2 Benchmarks

To evaluate our OS kernel survivability scheme, we use all 16 C/C++ benchmarks from the SPEC2000 benchmark suite [82] with reference inputs: ammp, art, bzip2, crafty, eon, equake, gap, gcc, gzip, mcf, mesa, parser, perlbmk, twolf, vortex, vpr. Since out-of-order processor simulation in Simics can be over ten times slower than in-
order processor simulation, we first run all benchmarks with in-order processor model, and then select seven of them that show the highest checkpoint creation frequencies for further simulation with the out-of-order processor model.

To stress the performance of our scheme, we add four *kernel intensive* benchmarks, including *Apache Web Server v2.0* [11] with *ab* workload, a network benchmarking tool *iperf* [10] with 300KB window size, and two Unix Tools: *find* and *du*. For *find*, we use the command `find /usr -type f -exec od \{} \;`, which searches the /usr directory and dumps the file content. For *du*, we use the command `du -h /usr` to summarize the disk usage of each file in /usr directory. For SPEC2000, *find* and *iperf*, we simulate 1B instructions after skipping the initialization phase. For *du* and *apache*, we simulate all dynamic instructions.

**Real World Attacks.** To validate the functionality of our survivability mechanism,
we use two real-world kernel vulnerability exploits. The first one is the \textit{elf\_core\_dump} attack described in Section 2.1 which attempts to crash the kernel. The second one is the “Raw Device Block Device Driver Vulnerabilities” attack [8], which is more malicious because it attempts to elevate privilege level of the process. To add variation, we write a device driver program that is vulnerable to buffer overflow and the number of bytes overflow can be varied, and it attempts to crash the kernel. We call the last attack \textit{drivercrash} attack.

\textbf{Operating System.} We choose Fedora Core 4 (FC4) [15], with the default Linux kernel v2.6.11. Kernel breakpoint support comes by default with FC4. Although checkpoint creation, detection, recovery, and fault isolation are all automatic, currently breakpoint handlers are still programmed and inserted to kernel manually. Hence, we limit the system calls, exception handlers and kernel functions that are instrumented. To approximate realistic performance overheads, we instrument the top twenty most frequently invoked ones. They contribute to more than 90\% of all system call and exception handler instances and hence provides a good coverage. In addition, we instrument eight other kernel functions which have been reported as vulnerable.
2.4 Evaluation

2.4.1 Survivability Functionality Validation

First, we validate the three attacks \((\text{elf\_core\_dump}, \text{pktcdvd}, \text{and drivercrash})\) on an unmodified FC4 Linux. Both the \text{elf\_core\_dump} and \text{drivercrash} crash the kernel, while \text{pktcdvd} successfully allows execution of arbitrary code with the kernel level privilege. This is a significant problem, because many production systems that use FC4 Linux are completely open to these attacks. We then boot the same Linux kernel on Simics, repeat the attacks, and observe the same outcome.

We then integrate our survivability mechanism into the simulated machine. We repeat all attacks, with various overflow amounts for the \text{drivercrash} attack. In all these attacks, illegal overwrites are detected immediately when they exceed the bounds and write into write-protected words. Automatic recovery occur in that case and the faulty process is correctly suspended, while the kernel successfully survives the attacks and retains its normal functioning. After each recovery, we continue the simulation for eight more simulated machine hours to see if there is any sign of system instability, and we found none.

This result is significant, especially considering that the survivability of the Linux kernel is achieved without any modifications to the kernel, with a relatively small programming effort.
2.4.2 Benchmark Characterization

**Checkpoint creation frequency.** To determine how frequently benchmarks would cause checkpoint creations, we measure the *number of checkpoint creations per billion instructions* (NCPBI). NCPBIs of all the SPEC benchmarks and kernel intensive (*KI*) benchmarks are shown in Figure 2.7(a), with their separate averages. NCPBIs for SPEC2000 benchmarks have an average of 151, which corresponds to one checkpoint creation of every 6.5 million instructions. In contrast, all kernel intensive benchmarks show much higher NCPBIs compared, ranging from a minimum 8158 (find) to a maximum 21632 (du), with an average of 16516, which corresponds to one checkpoint creation for every 60 thousand instructions. Thus, due to frequent system calls, kernel intensive benchmarks create checkpoints roughly two orders of magnitudes more frequent compared to SPEC2000 benchmarks. We pay particular attention to benchmarks with high NCPBIs because they stress the performance of our survivability mechanism.

![Figure 2.7: Number of checkpoint creations per billion instructions (NCPBI) (a), and Average checkpoint size in kilobytes (b).](image-url)
Checkpoint sizes. Figure 2.7(b) shows the average size of created checkpoints. Since each kernel stack checkpoint is always 8KB, the variation comes from the sizes of checkpoints from the kernel pool memory. The figure shows that a checkpoint size is typically a few kilobytes. Combined with a checkpoint interval of every 60 thousand instructions, kernel intensive benchmarks would incur a large checkpoint footprint if these checkpoints are cached when they are created.

For the rest of the evaluation, we focus on the benchmarks that stress our hardware mechanisms. So we choose all kernel intensive benchmarks and seven benchmarks with the highest NCPBIs: gap, gcc, gzip, mcf, perlbmk, twolf, and vortex.

2.4.3 Performance Overhead

Having validated the functionality of the survivability mechanism, we now turn to the performance overheads of our survivability mechanism. Since the performance overheads of our scheme come from checkpoint management (mostly from checkpoint
creation and very little from checkpoint stack de-allocation) and attack detection, we first show the checkpoint management and attack detection overheads separately, then show the performance overheads of combining both of them to fully support survivability.

**Checkpoint Creation Evaluation**

Figure 2.9 compares the execution time overheads of NULL breakpoint handlers (Kprobes), software-only checkpoint creation (SW), and memory-side Copy Engine [57, 59] support (CopyEng). Kprobes represent a kernel instrumented with all breakpoint handlers that we use for other cases, but we remove the code in each handler’s body, which essentially measures the performance overheads of only the dynamic instrumentation. For each benchmark, all configurations simulate a fixed number of number of *user* instructions plus instructions from the breakpoint handlers and checkpoint management. Unrelated kernel instructions are simulated but not counted in deciding when to stop the simulation.

The figure shows that Kprobes dynamic instrumentation shows largely negligible overheads compared to an un-instrumented (and unprotected) kernel. However, due to the more frequent system calls, it is clear that kernel intensive benchmarks suffer larger instrumentation overheads, with an average of 0.5% vs. 0.2% for SPEC benchmarks. The negative performance overhead of -2.3% in apache is mainly caused by
the variation in context switching effect with other processes (such as OS daemons) that also run and skew our measurement slightly. With software-only checkpoint creation, the figure shows much smaller overheads in SPEC benchmarks (average of 1.0%) than in kernel intensive benchmarks (average of 19.1%). Because the software mechanism uses a loop of loads and stores to do the copying for checkpoint creation, checkpoint creation is not overlapped with regular computation. These checkpoints will not be accessed again unless there is an attack that requires kernel state rollback. Finally, the memory-side Copy Engine avoids cache pollution and overlaps most of checkpoint creation with regular computation. As a result, it achieves very small overheads, with an average of 1.5% for SPEC benchmarks and only 2.6% for kernel intensive benchmarks. The worst case overhead is also low (4% for iperf).
Attack Detection Evaluation

We now show the evaluation results for our attack detection mechanism. For comparison, we also implement Mondrian Memory Protection (MMP) to compare with our protection cache scheme. Figure 2.10 shows the percentage increase of memory references (or L2 cache misses) due to various attack detection mechanisms.

![Figure 2.10: Percentage of increase in number of memory references due to various attack detection schemes. For all schemes, the hardware structure for storing protection information on chip is fixed at 3.75KB.](image)

The Mondrian bars show that using MMP results in a significant increase of memory references compared to an unprotected system, especially for kernel intensive benchmarks. This is due to the insufficient spatial locality that results in a high Protection Look-aside Buffer (PLB) miss rate. Note that without bloom filters, each PLB miss must result in memory access to fill the PLB with the protection information. Furthermore, looking up protection information in memory sometimes requires multiple memory accesses to traverse the multi-level protection table.
Our protection cache with backup bloom filter organization\textit{(BackupBF)} significantly reduces the extra memory references to an average of 0.05\% for SPEC benchmarks and 1.6\% for kernel intensive benchmarks. This is because in contrast to regular caching which causes memory access on a cache miss, with a bloom filter memory access is incurred only if there is an address that matches the bloom filter. Since bloom filter false positive rates can be made much lower than protection cache miss rates, it reduces the number of memory accesses. Finally, we show our protection cache that converts to multiple bloom filters per set on overflows\textit{(MultiBF)}. As we showed in Section 2.2.2, MultiBF always incur smaller false positive rates than BackupBF. The reduced false positive rates result in a lower number of memory accesses, on average only 0.02\% for SPEC benchmarks and 0.4\% for kernel intensive benchmarks. Even the worst case (iperf) shows less than 1\% increase in memory accesses.

Comparing our schemes to MMP, our scheme works better because of two factors. First, MMP relies on spatial locality of the protection information and stores both write-protected and non write-protected word information on chip in its PLB. In contrast, our protection cache only stores the write-protected word information. Since at any given time there is a lot less write-protected words than non write-protected words, our protection cache is more efficient. In addition, bloom filters are able to summarize protection information in the memory quite well, so they reduce the frequency of checking the protection bit vector in memory. Finally, we use counting bloom filters rather than plain bloom filters, which enable on-chip protection infor-
information to be modified (address deleted or added) without involving the main memory most of the time.

We now show the execution time overheads of our attack detection schemes. We assume 3.75KB protection cache can be accessed in one cycle. The base case is an unprotected system. BackupBF shows an average performance overhead of 1% for SPEC benchmarks and 5.3% for kernel intensive benchmarks, with a worst case of 10.2% for iperf. These high overheads despite small increases in the number of memory references are due to the long latency to read, locate, modify, and write protection bit vectors in the main memory. As expected, the multiple bloom filter scheme (MultiBF) outperforms BackupBF scheme in all cases, and reduce the overheads to 0.8% for SPEC benchmarks and 1.9% for kernel intensive benchmarks.

Figure 2.11: Performance overhead of attack detection scheme using protection cache with Backup Bloom Filter or Multiple Bloom Filters.
Overall Performance Overheads of Survivability

Figure 2.12 shows the total execution time overheads of the full survivability mechanism, which includes the memory-side Copy Engine hardware support for checkpoint creation, and protection cache with multiple bloom filters for attack detection. Because the overheads of checkpoint creation and attack detection mechanisms are both low, the combined scheme still achieve low overheads. The average overheads are 1.3% for SPEC benchmarks and 3.7% for kernel intensive benchmarks, with a worst case overhead of 4.7% for iperf. iperf spends 99% of its execution time in the kernel mode due to its very frequent system calls, hence we believe that very few applications would be as kernel intensive as iperf, so the 4.7% worst-case overhead probably applies to a wide range of applications as well. Additionally, the rollback handler will not incur any performance overheads unless an attack is detected, which is a rare occurrence.

Figure 2.12: Overall performance overhead of OS survivability scheme using memory-side Copy Engine [57, 59] for checkpoint creation and using protection cache with Multiple Bloom Filters for attack detection.
2.5 Related Work

**Attack Detection, Avoidance, and Diagnostic.** The common practice to improve the security of the OS today is to rely on patches, essentially software update that users must install in their system. Unfortunately, patches do not provide complete protection, let alone survivability. In practice patches are often created after vulnerabilities have been found or exploited, and users often apply patches too late because they fear kernel instability or are simply reluctant to install patches promptly.

Many attack detection and avoidance schemes have been proposed, such as StackGuard [37], PointGuard [25], SmashGuard [40], Stack Ghost [64], and Transparent Runtime Randomization [54]. However, they do not provide the ability to recover from a security attack. Many of them simply convert an attack into a crash, which is incompatible with survivability.

Linux Kernel Crash Dump [5] takes a snapshot of system state taken at the time the kernel crashes. While it provides useful information to track the root cause of the crash, it does not prevent kernel crashes.

**Hardware Fault Tolerance.** Many hardware mechanisms to provide fault tolerance have been proposed, such as [27, 33, 38, 53, 68, 74, 77]. As mentioned in Chapter 1, fault tolerance techniques are not directly applicable to survivability techniques. This is due to different correct recovery behaviors for hardware faults and for security faults. In security faults, rolling back the state of the system to one
prior to an attack and identical re-execution of the program starting from that good state will only repeat the attacks. Hardware fault tolerance techniques also treat the system as consisting of a single state. However, for kernel state recovery, one must consider that the kernel state is affected by multiple user processes that interact with it. When only one process is affected by a security fault, only the kernel state of that process has to be rolled back and recovered. State of other user processes should not be impacted. As a result, our recovery mechanism is different than hardware fault tolerance mechanisms in that we distinguish multiple kernel states and the recovery process prevents security attacks from being replayed.

There are also mechanisms to tolerate software faults. Nooks [69] is a mechanism to survive the OS from device driver failures, using an extra software layer between the kernel and device drivers. Nooks does not protect the kernel itself, and rely on identical re-execution, so the recovery process could repeat security attacks if they cause the failures. Indra [84] relies on hardware fault tolerance technique to recover from security faults on network applications. Hence, it relies on identical re-execution and treats the system as having a single state, which are not appropriate for kernel survivability.

Fine-Grain Memory Protection. There are also studies on fine-grain protection such as Mondrian Memory Protection (MMP) [34]. MMP divides the memory into different protection domains, where permission control bits are assigned at the granularity of individual words. Permission bits are stored in permission tables, whose
entries can be cached on chip to improve performance. Mondrix [35] is an OS that uses MMP to enforce isolation between kernel modules by loading them into different protection domains. Switch gates are placed on the first instruction of a routine by programmers, to serve as the point where a caller enters callee’s domain or callee returns. A thread can only manipulate its own stack designated by stack permission registers. While Mondrix improves kernel security by providing fault isolation, it is a detection scheme inherently rather than survivability scheme and does not provide recovery and isolation from security faults. Security faults could still occur within a single protection domain and when they occur, the kernel cannot recover from them.

Overall, we note that attack detection and avoidance schemes attempt to improve the security of the kernel by reducing the probability of security faults. In contrast, we assume that despite our best effort in making the kernel secure, security faults are inevitable [23]. Since security faults would eventually occur, we seek to make the kernel survivable.

There have also been user-level survivability schemes such as DYBOC [81] for recovering an application from security attacks. DYBOC assumes that most functions have transaction semantics, in that the program would execute correctly if a function is executed completely or not executed at all. DYBOC uses an assumption that the system has a single state, hence DYBOC cannot be applied for kernel survivability. Finally, perhaps the most relevant prior work to ours is Recovery Domains [18], a recovery-based scheme that can recover the kernel states from unexpected run-time
error. As a pure software scheme, Recovery Domains slows down the common case attack-free execution by between 8% to $5.6 \times$. Moreover, it is not aware of the dependencies among faulty and non-faulty processes while isolating faults. In contrast, our scheme provides an efficient architecture support for OS kernel survivability and only incurs less than 5% performance overhead. While recovering from security attacks, our scheme is capable of isolating the faulty processes from non-faulty ones.
Chapter 3

Architecture Support for
Improving Bulk Memory Copying
and Initialization Performance

This chapter is organized as follows: Section 3.1 discusses related works, Section 3.2 describes the architecture support for FastBCI, Section 3.3 illustrates the micro-architecture design of FastBCI and Section 3.4 presents the evaluation environment. Finally, Section 3.5 discusses evaluation results.
3.1 Related Work

DMA and memory-side copy engine. An alternative way to accelerate memory copying is to use DMA. Traditional DMA engines (e.g. ISA or PCI DMA [29]) allow copying between device and memory but not between memory and memory. More recent DMA engines support bulk data movement between two memory regions (e.g. Intel I/OAT [45]). It resides in the chipset of a computer system to offload the memory-to-memory copying operations. However, such a DMA engine still requires (1) a long DMA channel setup latency through non-cacheable memory accesses, (2) an interrupt-based completion, which requires an expensive pipeline flush and interrupt handling, and (3) OS involvement the provides device driver support for the engine. Thus, from performance stand point, such overheads cannot be easily amortized if the copying region is relatively small, e.g. a few kilobytes or less. Moreover, there are other practical limitations such as requiring physical addresses that prohibits user applications which view their address space in terms of virtual addresses from benefiting from it.

Some recent DMAs support bulk data movement between the main memory and on-chip software-controlled scratchpad memory (e.g. Cell processor [66] and others [50]). Cell DMA can operate at the user level and does not require a long setup latency. However, it requires software-controlled scratchpad memory, which is not available in most general purpose processors that rely on hardware-controlled cache.
In addition, it still relies on interrupt-based notification for data transfer completion, has no fine-grain dependence checking (incurring pipeline bottleneck), and is asynchronous. We compare FastBCI’s performance with zero setup-latency DMA in Section 3.5.

Zhao et al. [59, 58] proposed an instruction support and hardware engine for performing memory copying. The engine includes an off-chip memory-side component called Copy Engine (CE) which performs copying, and an on-chip component called Copy Control Unit (CCU). When a copying instruction is encountered, CCU accepts virtual addresses, translates them into physical addresses, and sends them off to the CE in the memory controller. The CE acquires necessary ownership of the copying regions by sending invalidations to caches, and then performs the copying by directly reading from and writing to the main memory. While copying is in progress, any loads or stores made by the processor to such addresses are stalled until the copying is fully completed. Conceptually, CE can be thought of as an instruction-level cache coherent DMA, and falls in between traditional DMAs and our FastBCI scheme. Like FastBCI, CE supports a bulk copying instruction and an engine to perform bulk memory copying. However, in contrast to FastBCI, CE does not address pipeline and cache bottlenecks for code that follows copying operations, which Figure 1.2 points out as the key for improving copying performance. For example, CE stalls all dependent loads/stores until the copying is fully completed. Being memory-side, CE also always requires caches to flush blocks in the copying region prior to the start of copying.
Such a policy negates the potential beneficial effect of bringing blocks in the copying region into the cache, and cause extra cache misses when the following code accesses the copying regions in the future. In contrast, FastBCI deals with both pipeline and cache bottlenecks by keeping track of which cache blocks have completed their copying, allowing dependent loads/stores to proceed even when copying for other blocks has not completed (non-blocking policy), as well as supporting three cache affinity options. As a result of these differences, our evaluation (Section 3.5) shows that FastBCI achieves roughly three times the speedups achieved by CE (23.2% vs. 7.9% on average) across benchmarks we tested.

**Cache Affinity Flexibility.** In current systems, while copying instructions lack such flexibility, instructions that control cache affinity are already provided. For example, Power and Intel architectures [43, 48] have instructions to flush or prefetch specific cache blocks. IA64 [47] load/store instructions can be augmented with cache hints that specify the cache level into which data should be fetched. Load/store instructions in SSE4 [46] (e.g. movntdq and movntdqa) allow non-cacheable loads and stores using a streaming load and a write-combining buffer. The fact that various instruction-level supports are provided in several processors highlights the feasibility and importance of such flexibility on performance. In FastBCI, cache affinity flexibility is provided for both the source and destination regions and each region can be specified individually as cacheable, non-cacheable, or cache neutral.

Cache affinity flexibility may also be provided beyond the instruction level. For
example, direct cache access (DCA) is an architecture that allows the Network Interface Card (NIC) to directly place inbound packets in the processor’s cache [78]. Finally, there are also techniques to accelerate TCP/IP processing (such as on-chip network interface [71] and Remote DMA [13]). Such architectures and techniques have been shown to accelerate TCP/IP processing. Compared to them, FastBCI is a more general support in that it deals more with accelerating memory copying and initialization rather than TCP/IP processing only. However, FastBCI can also be used to accelerate TCP/IP processing.

**Instruction-Support for Bulk Memory Copying.** As mentioned briefly in Chapter 1, in current architectures, instructions that can operate on wide data are typically vector or vector-extension instructions [46, 72, 26]. Vector loads/stores can work on data as wide as the width of the vector registers. However, vector loads/stores require data to be aligned at the respective width [17], making them unsuitable for use in scalar data which may not be aligned at vector width.

It is likely that instructions that can perform a large-region copying have been implemented in some CISC (Complex Instruction Set) architectures. Our FastBCI instructions can be thought of as an example of such CISC-flavored instructions. However, we showed in Figure 1.2 that such instructions do not improve performance much unless pipeline and cache bottlenecks of code that follows are removed. To our knowledge, FastBCI is the first to propose mechanisms to remove such bottlenecks for bulk copying instructions.
3.2 Architecture Support for Bulk Copying/Initialization

This section describes our FastBCI in detail. It starts from an overview of bulk copying and initialization instructions (Section 3.2.1), followed by discussions on how cache bottleneck (Section 3.2.2) and pipeline bottlenecks (Section 3.2.3 and 3.2.4) are handled in FastBCI.

3.2.1 Overview of Bulk Copying and Initialization Instructions

To achieve maximum granularity efficiency, ideally there should be a single instruction to copy or initialize a large memory region. Such instructions can be one of instructions already used in current architectures (e.g., _mvcl_ or _rep movsd_ in Figure 1.1), or be new ones. For ease of discussion, but orthogonal to the design of FastBCI, we assume the following three-register operand instructions are available:

\[
\text{BLKCPY } \text{Reg}_\text{SRC}, \text{Reg}_\text{DEST}, \text{Reg}_\text{LEN} \\
\text{BLKINIT } \text{Reg}_\text{INITVAL}, \text{Reg}_\text{DEST}, \text{Reg}_\text{LEN}
\]

where BLKCPY copies \text{Reg}_\text{LEN} (i.e. value stored in register \text{Reg}_\text{LEN}) number of bytes from a region that starts at address \text{Reg}_\text{SRC} to a region that starts at address \text{Reg}_\text{DEST}, and BLKINIT initializes \text{Reg}_\text{LEN} number of bytes in a region that starts at address \text{Reg}_\text{DEST} using an initialization string value stored in \text{Reg}_\text{INITVAL}. 
In actual implementations, the instruction format may differ depending on the constraints of a particular ISA. The source and destination regions in copying may or may not overlap. To be consistent with most current implementations [2, 44], our discussion we assume they are not overlapped. Furthermore, BLKINIT can be thought of as a special case of BLKCPY where the source region is a register value rather than a memory address. Architecture support for a bulk copying instruction can also be used to support a bulk initialization instruction. Hence, from this point on, our discussion will center around memory copying.

In traditional implicit loop implementations, the copying instruction is fetched and decoded into a loop of loads and stores. Each load and store then incurs a TLB access to check the validity of the page and obtains its physical address, and incurs a cache access to read value into a register or write from a register. Such an approach has poor granularity efficiency. For example, if a load/store granularity is 8 byte, for each 4KB-size page copied, we would incur $2 \times \frac{4096}{8} = 1024$ TLB and cache accesses. If instead, the copying instruction is executed in a special functional unit (an “engine”), we can improve the granularity efficiency by checking the TLB only once for each page involved, and performing the copying at the granularity of cache blocks. With the latter approach, copying a 4KB region only incurs 2 TLB accesses for the source and destination pages (a reduction of 99.8%) and 128 cache accesses assuming 64-byte cache block size (a reduction of 87.5%) \(^1\). Hence, FastBCI adopts

\(^1\)The cache data path width of a processor determines the actual reduction in the number of cache accesses.
the latter approach.

### 3.2.2 Cache Bottlenecks

As shown in Figure 1.2, one of the keys to improving memory copying performance is solving the cache performance bottlenecks of the code that follows memory copying operations. The huge performance improvement in the “perfect cache” is by assuming that copying produces beneficial prefetching effect (i.e. later accesses to the source and destination regions result in cache hits), but none of the cache pollution effect (i.e. later accesses to blocks that were in the cache prior to copying result in cache hits). While “perfect cache” is optimistic, it highlights the upperbound performance improvement of maximizing the prefetching effect while minimizing the cache pollution effect.

Whether the prefetching or pollution effect is stronger depends on the temporal reuse patterns of the application that uses copying. Caching new blocks from the copying regions will cause cache pollution if the processor needs them less urgently than the blocks they replace, but will cause a helpful prefetching effect if the processor needs them more urgently than blocks they replace. For example, in a network application in which an outgoing packet is copied from a buffer to the network interface memory-mapped buffer, the destination region will not be accessed by the processor, hence it is likely beneficial to keep the region uncached. But if an incoming packet is copied from the memory-mapped buffer to a destination buffer, the destination region
will soon be accessed by the processor, and hence it is likely beneficial to fetch it into the cache during copying.

Considering this, we propose to augment the bulk copying instruction with parameters that can specify *separately* the cache affinity policy for the source and destination regions. In FastBCI, the affinity policy specifies whether a region should be *cacheable* (the region is brought into the cache as it is accessed), *non-cacheable* (the region is not brought into the cache and is evicted if all or parts of it are cached), or *cache neutral* (no new blocks from that region are brought into the cache but existing blocks are retained in the cache). A cacheable affinity intends to allow an application to maximize the prefetching effect of copying, whereas a non-cacheable affinity intends to allow an application to minimize the cache pollution effect of copying. A cache neutral affinity achieves a balance in the sense that it does not produce any prefetching or pollution effect. The affinity options can be embedded as parameters into the copying instruction using unused operands or opcode bits.

There are several factors that determine which affinity option will perform best for a particular pair of source or destination regions. One factor is the likelihood of the region to be needed by the processor right after copying is completed. The higher the likelihood, the more attractive the cacheable option becomes. Another factor is the temporal locality of data that was already cached prior to copying. The higher the temporal locality, the more cache pollution it has if a copying region is fetched into the cache. Another important factor is the cache capacity compared to
the working set of the application. A larger capacity can tolerate cache pollution better, and increase the likelihood that a cache-allocated region remains in the cache when the processor accesses it. Due to the combination and interaction of these factors, we find that typically, the source and destination regions require different affinity options to achieve the best performance, which argues for the need to provide flexible and separate affinity options for source and destination regions. In contrast, current approaches do not offer cache affinity options. The loop approach always allocates both regions in the cache. DMA or Copy Engine [59], on the other hand, always makes both regions non-cacheable since DMA and Copy Engine are memory side devices. Our evaluation results (Section 3.5) demonstrate that using the same cacheable or non-cacheable affinity option for both the source and destination regions usually result in suboptimal performance (they yield 7% and 10% lower application speedups compared to the optimal options, respectively).

Note that the affinity flexibility also eases the interaction with the I/O system. If copying involves a memory-mapped I/O region, specifying a non-cacheable affinity helps in ensuring that the region is not cached at the end of copying, and avoids cache coherence problems with I/O devices.

To support the non-cacheable and cache neutral options for the copying region, the cache controller needs to be modified slightly. For a non-cacheable source region, if a block is already cached, it is read and then evicted from the cache. If it is not found in the cache, it is read from memory but not allocated in the cache. For a
cache neutral source region, if a block is already cached, it is left where it is. If it is not found in the cache, it is fetched and read but not placed in the cache. For a non-cacheable destination region, if a block is already cached, it is overwritten and immediately evicted and written back. If it is not already cached, an entry in the write combining buffer can be created with the copied value, so bandwidth is not wasted for fetching the block. For a cache neutral destination region, if a block is already cached, it is overwritten. If it is not already cached, an entry in the write-combining buffer is created containing the copied value. It is possible to let the affinity policy be applied differently to different cache levels, but in this dissertation we assume it is applied to all cache levels.

### 3.2.3 Pipeline Bottlenecks

Let us now investigate the pipeline performance of bulk copying instructions. When the instruction is encountered in the pipeline, let us assume it is sent off to a special "functional unit" (or engine) that performs the actual copying. The granularity efficiency of the instruction allows the copying to be performed without occupying much pipeline resources. Unfortunately, while the copying instruction itself is executed efficiently, the code that follows copying may not benefit much from it. Figure 3.1(a) and (b) illustrate this problem using a bulk initialization instruction. In Figure 3.1(a), the bulk initialization instruction validates a page it wants to write to, then performs writes to each block. Some writes may take longer than others due
to cache misses (shown as longer lines). Unfortunately, the instruction may take a long time to complete due to writing to a large number of blocks, and hence it stays in the ROB of the processor for a long time. Since instruction retirement is in order, younger instructions cannot retire either, and eventually the pipeline is full, at which point no new instructions can be fetched. We refer to this bottleneck as the ROB-clog bottleneck.

The pipeline bottleneck described above can be avoided if we allow the bulk copying instruction to retire early, e.g. right after page validation is completed (Figure 3.1(b)). Assuming such an early retirement is safe to do (a big if for now), another pipeline bottleneck may occur. A younger instruction (instruction B in the figure) stalls in the pipeline waiting for the initialization completion. The stall can again propagate and increases the critical path of execution, or even clog the pipeline. We refer to this bottleneck as the dependence bottleneck.

If the above two bottlenecks are avoided, then instructions can flow through the pipeline without being blocked by the copying instruction or by instructions that depend on the copying instruction. In Figure 3.1(c), instruction B, which depends on the completion of initialization of block 2, can wake up and execute once block 2 is initialized. This improves performance as instruction B will no longer stall for a long time in the pipeline.

To get an idea of the importance of the two pipeline bottlenecks, Figure 3.2 shows the breakdown of cycle per instructions (CPI) due to processor execution and
Figure 3.1: The effect of ROB-clog (a) and dependence (b) bottlenecks on pipeline performance. Execution without bottlenecks (c).
regular stalls such as structural, data dependence, branch mis-prediction (bottom section), and due to copying-specific stalls: ROB clog bottleneck (middle section), and dependence bottleneck (top section). The figure shows that on average, 12.2% of the CPI stall is due to copying-specific stalls (7.8% from ROB clog bottleneck and 4.4% from dependence bottleneck). To understand why copying-specific CPI stalls are high, we find that for the tested applications, a copying on average takes 9,086 cycles to complete. A high-performance processor pipeline can support a few hundred in-flight instructions, which is only enough to keep the processor busy for a few hundred cycles, before stalling for the remaining thousands of cycles. In addition, the average distance between a copying instruction to the first dependent instruction is only 16 instructions. Thus, even if we solve the ROB clog bottleneck, stalls due to dependence delay occurs soon after.

Figure 3.2: Cycle Per Instruction (CPI) breakdown due to regular processor execution&stalls, and stalls due to ROB clog and dependence bottlenecks. Details of applications and machine parameters can be found in Section 3.4.
### 3.2.4 Dealing with Pipeline Bottlenecks

Comparing the ROB-clog bottleneck and the dependence bottleneck, the dependence bottleneck can be dealt with more easily because it involves dependence checking mechanisms rather than instruction retirement mechanisms. However, solving the dependence bottleneck without solving the ROB-clog bottleneck unlikely improves performance much because the ROB is more likely to be clogged by the copying instruction before it is clogged by (younger) dependent instructions. Hence, first we must solve the ROB clog bottleneck before we can solve the dependence bottleneck.

There are several ways to remove the ROB-clog bottleneck. The first option is to remove the guarantee that instruction retirement implies the completion of the instruction execution, i.e. by making the bulk copying instruction *asynchronous*. This allows the instruction to be retired ahead of its completion, freeing the ROB. However, using asynchronous copying instruction complicates programming because programmers must explicitly test the completion of the copying instruction (e.g., by polling). Moreover, polling for completion introduces overheads, and these overheads preclude a cost-effective solution for avoiding the dependence bottleneck problem since it requires fine-grain dependence tracking.

Another possible solution to allow the copying instruction to be retired prior to its completion is by treating the retirement as speculative, and using a checkpoint and rollback mechanism to rollback and re-execute it when a dependent instruction
uses data that has not been produced by the copying instruction. Unfortunately, a checkpoint and rollback mechanism is quite expensive to support. In addition, if the distance between a copying instruction and dependent instructions is short (on average, only 16 instructions apart), this will cause frequent and persistent rollbacks.

With FastBCI, we adopt a new, non-speculative approach. The key to our approach is while we allow the copying instruction to retire ahead of its completion, we provide an *illusion of completion* to other instructions. There are several requirements to providing an illusion of completion:

1. Exception completeness requirement: any exception that a copying instruction may raise must have been handled prior to retiring the instruction, and no conditions that may raise a new exception can be permitted.

2. Consistency model requirement: a retired copying instruction must be consistent with load/store ordering supported by the architecture.

In addition to the two requirements needed for providing an illusion of completion, we also employ an additional *non-speculative* requirement that simplifies the implementation of copying: the effect of copying should not reach the memory system before the copying instruction is verified to be non-speculative, i.e. after exceptions and branch mis-predictions in older instructions have been handled.
**Non-Speculative Requirement**

The non-speculative requirement deals with when the copying instruction should be allowed to start its execution. One alternative is to let it start right away as soon as its operands are ready. However, this means that we must provide an ability to roll it back if it turns out to be speculative, e.g. it lies in the wrong branch path. Supporting such a rollback capability is difficult since the instruction may write to a large number of cache blocks and these writes must be either buffered or canceled.

Hence, FastBCI adopts a simpler alternative that avoids the reliance on roll back altogether: it allows the Bulk Copying and Initialization Engine (BCIE) to be programmed, but it holds off the BCIE from performing the actual copying until the copying instruction has been verified to be non-speculative. This means all older instructions must have been verified exception free and have their branch mis-predictions handled. To achieve that, we *conservatively* wait until the copying instruction reaches the ROB head before firing off the BCIE. While this may introduce some delay in starting the copying, a delay of up to tens of cycles is insignificant compared to the performance gain obtained by FastBCI.

**Exception Completeness**

It is straightforward to observe that without exception completeness and ordering, the illusion of completion for a copying instruction is broken. For example, an exception raised by a copying instruction after it retires leaves the system in an in-
consistent state, i.e. it appears as if the copying instruction has not completed or has failed, but some of its younger instructions have retired.

To provide the illusion of completion, FastBCI allows a copying instruction to be retired only after it is verified to be exception-free or all its exceptions have been handled. When a copying instruction is executed, the Bulk Copy and Initialization Engine (BCIE) is simply programmed with the required parameters, but is not fired off. After the instruction is verified to be exception free, or its exceptions have been completely handled, then the instruction can be retired and the actual copying can start (the BCIE is fired off). Hence, the effect of copying does not reach the memory system until after the instruction can no longer raise an exception.

Whether a copying instruction will raise an exception or not can be verified by checking the TLB. If the translation exists and page permission allows the respective operations (read permission for the source region and write permission for the destination region), then the copying instruction will not generate an exception. Note, however, that both the page permission and the physical address of pages in the source and destination regions must remain valid for the entire copying operation. This requires the pages to be locked to prevent the OS from swapping the pages out to disk, or perform an operation that changes the permission of the pages, until the completion of copying.

Due to the requirements to handle all exceptions prior to retiring a copying instruction, and locking pages until the completion of copying, the implementation
can be made easier if the maximum region size is limited. In FastBCI, we limit the maximum copying region size to be the smallest page size, i.e. 4KB. Hence, page validation incurs at most four TLB misses (two pages for each of the source and destination regions if they are not page-aligned). In addition, the granularity efficiency loss compared to an unbounded copying is negligible since most of the granularity efficiency gain is obtained going from word/double-word-sized load/store to a page-sized copying granularity.

Finally, there are events that can potentially break the illusion of completion, such as interrupts, context switches, thread spawning, etc. This requires the processor to arrive at a consistent state because the interrupt handling or the new thread may read or modify data in the copying regions. If any of these events occurs when a copying instruction is already retired but not completed, the event handling is delayed until copying is fully completed.

**Illusion of Completion in Consistency Models**

The illusion of completion for a copying instruction that is retired early also applies in the context of a multiprocessor system. Let us consider both the cache coherence issue as well as the memory consistency model.

The engine that performs copying (BCIE) communicates with cache controllers to send read/write requests for each block involved. Coherence requests are generated by cache controllers in response to the requests, as in a regular multiprocessor sys-
tem. Therefore, cache coherence is handled properly by existing coherence protocols without changes.

FastBCI allows copying to different blocks to proceed out of order and allows younger loads/stores to issue before an older copying instruction fully completes. Therefore, FastBCI fits naturally with relaxed consistency models such as _weak ordering_ and _release consistency_. In these models, it is assumed that programs are properly synchronized, and ordering among loads and stores are not enforced except at synchronizations or memory fences. A copying instruction can be thought of loads and stores surrounded by synchronizations/fences. Therefore, the copying instruction is prevented from being issued when an older synchronization/fence has not performed. In addition, a younger synchronization/fence is prevented from being issued until all older copying instructions have fully performed. Such a restriction conforms to these consistency models.

More restrictive consistency models rely on strict ordering of ordinary loads and stores, such as _sequential consistency_ (SC) or _processor consistency_ (PC), hence another approach needs to be employed. We start from an observation that the main problem with the copying instruction execution is that it is not atomic, i.e. when a younger load/store issues, the reads/writes of an older copying instruction have not fully completed. To conform to SC or PC, the illusion of completion of a copying instruction also needs to be maintained with respect to older and younger loads/stores.

To achieve the illusion of completion, after page validation, the processor issues
read (read-exclusive) requests for source (destination) in order to obtain read (write) permission for the source (destination) region. Since there may be many block addresses in one region, one way to achieve this efficiently is to send the address ranges to other processors. Other processors react by downgrading the affected cache blocks to a clean-shared state (if modified or exclusive) in response to a read request, or invalidating them (if valid) in response to a read-exclusive request. After all appropriate ownerships are obtained by the processor which will execute the copying instruction, the copying/initialization is started.

If later other processors access an address in the copying regions that requires coherence state upgrade, an intervention or invalidation request is sent to the processor that executes the copying instruction. Upon receiving the request, if the copying instruction is already retired but not fully completed, the processor must provide the illusion of completion. If copying for the requested block is completed (a block-granularity copying progress tracking is assumed), the request is processed. Otherwise, the request is either buffered until when the block’s copying is completed, or negatively acknowledged so that the request can be retried. Which solution is more appropriate depends on various factors, for example the negative acknowledgment approach is more applicable to a distributed shared memory multiprocessor.
3.3 Micro-architecture Design of FastBCI

In this section, we discuss the micro-architecture design of our FastBCI. It starts from discussion on the architecture design of the engine (BCIE) that executes the copying instruction (Section 3.3.1), followed by the three-step sequential copying algorithm of BCIE (Section 3.3.2) and discussion on how alignment issues are handled in FastBCI (Section 3.3.3).

3.3.1 Micro-architecture of BCIE

We now discuss the architecture design of the engine that executes the copying instruction. The architecture of the engine, which we refer to Bulk Copying and Initialization Engine (BCIE) is illustrated in Figure 3.4. It has a main control unit (MCU). MCU controls other components in the BCIE, interfaces with TLB for page validation and interface with caches to perform copying.

BCIE needs to track the progress of page validation (to determine when the instruction can be retired) and copying (to determine the completion of copying and to support fine-grain dependence checking). Page validation progress is tracked by using Page Validation Status Registers (PVSRs), while copying progress is tracked using Copy Status Registers (CSR) at the page level and using Copy Status Table (CST) at the block level for outstanding pages.

PVSRs keep the range of addresses that have not yet been validated separately for
the source and destination regions. Read permission is checked for the source region while write permission is checked for the destination region. Since copying region may not be page-aligned, a source and destination region may span up to two pages each.

![Fig 3.3](image)  
**Figure 3.3:** Cumulative percentage of instruction distances of back-to-back copying instructions for various benchmarks.

Experimentally, we have not observed the need for supporting concurrent execution of multiple copying instructions. Figure 3.3 shows the distance of back-to-back copying instructions of benchmarks that we evaluated, with y-axis being the cumulative percentage and x-axis being the distance in terms of number of instructions. As shown in Figure 3.3, on average 82% of back-to-back copying instructions are separated by over 5000 instructions, and only 9% of them are separated by less than 2000 instructions. Hence, a copying instruction is likely already completed before the next one arrives at the pipeline. Therefore, we keep the hardware simple by keeping only
one set of PVSRs, CSR, and CST.

```
MCU = Main Control Unit
PVSRs = Page Validation Status Registers
CSR  = Copy Status Registers
CST  = Copy Status Table
```

Figure 3.4: Illustration of Bulk Copying and Initialization Engine (BCIE).

### 3.3.2 Mechanisms of BCIE

A copying instruction flows through the pipeline in three sequential steps: in the start-up step, the instruction is issued and BCIE is programmed; in the validation step, page validation is carried out; and in the finishing step, copying is performed. FastBCI does not require copying regions to be aligned. For now, we assume that they are aligned at cache block boundaries, and defer the discussion for non-cache block aligned regions to Section 3.3.3.
The Start-up Step

The goal of the start-up step is to determine when the copying instruction can be issued. Like regular instructions, a copying instruction waits until all its register operands are ready before issuing. It also waits if the BCIE is occupied by an older copying instruction. After register dependences and structural hazard are resolved, the copying instruction is issued to the BCIE to program it with the instruction parameters, but BCIE does not fire off until after the validation step is completed (Section 3.2.4). For example, the PVSR and CSR are initialized with the initial address ranges of the source and destination regions specified in the instruction.

The Validation Step

In the validation step, the permission of pages in the range of addresses in the PVSRs are checked in the TLB to validate whether reads (writes) are allowed for the source (destination) regions. The physical addresses are obtained as well. PVSRs keeps the range of addresses that have not been validated, and they are continually updated each time a page is validated. If an exception occurs, the exception bit of the instruction in the ROB entry is raised. After the exception is handled, the copying instruction is re-executed. The completion of page validation is detected when PVSRs contain an empty address range.

Recall that validated pages must remain valid until the copying is completed (Section 3.2.4). One way to avoid the risk that the page is swapped out or its permission
changed is to lock the page in the physical memory. Normally, to lock a page, the OS must be invoked to manipulate its page lock data structure [30]. When an OS wants to swap out a page, it checks whether the page is locked. Meanwhile, an invalidation is sent to the TLB. To avoid the OS involvement in page locking, we augment each TLB entry with a lock bit and set the bit for the duration of the copying operation. If the OS wants to swap out a page, the TLB will receive invalidation, and at this time TLB checks the lock bit of the entry. If set, the invalidation fails, and the OS reacts by setting the lock bit of the page in its data structure. Similarly, when a TLB entry with a lock bit set is evicted, the OS must also be invoked so it can lock the page. With this technique, most copying operations only involve the TLB but not the OS. To support this technique, the OS is modified slightly to react to a failed invalidation and TLB replacement of entries with lock bit set by performing the page locking.

The Finishing Step

After page validation, the actual copying can be started. The instruction also becomes retireable. BCIE uses CSR and CST to keep track of the copying progress at page and cache block level, respectively. If the copying regions cross page boundary, the copying is broken into smaller subcopying operations, each of which only involves one source page and one destination page (a page pair). Since we limit the maximum region size to be the size of a page, a copying operation is broken down into at most three subcopying operations, as illustrated in Figure 3.5. Whenever a subcopying
operation completes, the CSR is updated by bumping the source and destination base addresses up to the next page pair.

Figure 3.5: Handling regions that are not page aligned.

The page pairs in the subcopying operation are tracked independently from one another in the CST. Each page pair is allocated an entry in the CST, hence the CST has three entries to keep track of up to three page pairs. Each entry in the CST has a valid bit (V), the virtual and physical base addresses of the source and destination pages, mask bits indicating blocks that belong to the source or destination region in the pages (Mask-Src and Mask-Dst), and bit vectors that track copying progress at the cache block level for both the source and destination pages (Figure 3.3.2(b)). We refer to the bit vector as Copy Progress Bit Vector (CPBV). The CPBVs (CPBV-Src and CPBV-Dst) have as many bits as cache blocks in a page, e.g. 64 bits for a system with 4KB page and 64-byte cache block size. CPBVs are initially cleared to zero when a page pair is allocated in a CST entry. When a block in the source has been copied to the corresponding block in the destination, the corresponding bit in source and destination’s CPBV is set to ‘1’. When all bits in the CPBV that are not masked out have a value of ‘1’, the copying for the page pair is completed. The
page pair is deallocated from the CST and the CSR is updated to reflect it. When the region length field in the CSR becomes zero, the entire copying has completed, and the BCIE is freed.

By checking the CPBV s of each CST, the MCU keeps track of blocks that still need to be copied, and generate appropriate read and write requests to the cache controller. The cache controller performs copying according to the cache affinity options specified. The MCU may simultaneously send several read/write requests to the cache controller, and based on whether the cache blocks are found in the cache, some blocks may be copied sooner than others. The CPBV allows blocks in a page to be copied in parallel and completed out of order. In addition, having multiple CST entries also allows blocks from different page pairs to be copied in parallel and complete out of order. However, completed page pairs are reflected on the CSR sequentially.

The CSR, CST and its CPBV s also serve as a way to check for memory dependencies between a copying instruction and younger load/store instructions. For a younger load, its address must be checked against the destination region, while for a younger store, its address must be checked against source and destination regions. The check is performed when the BCIE has an outstanding copying instruction, and in parallel with regular load/store queue accesses. Since BCIE only has a few small registers and a small table, the dependence checking latency can be hidden by the overlapping it with load/store queue access. Moreover, in most cases, only the CSR
Figure 3.6: Illustration of fine grain dependence checking.
needs to be checked to resolve dependences (the CST is only checked occasionally).

Figure 3.3.2 illustrates an example of how dependences with younger loads can be
checked by the BCIE. Figure 3.3.2(a) shows a copying operation with the source region
that is page aligned and the destination region that is not page aligned, and with the
region size of 4KB. The copying is broken up into two subcopying operations, and
let us assume that subcopying operation 1 is completed while subcopying operation
2 is in progress. Consequently, the address range stored in the CSR (Figure 3.3.2(b))
only includes those of “Src 2” and “Dst 2”. An entry for subcopying operation 2
is allocated in the CST, and it contains bit vectors for the source (CPBV-Src) and
destination (CPBV-Dst) which track which cache blocks in the source and destination
pages have completed copying/initialization.

Suppose there are four younger load instructions (A, B, C, and D) illustrated in
Figure 3.3.2(c). Instruction A loads from an address that is not in the copying region,
so it is an independent instruction. B loads from the first byte of “Dst 1” and hence
has a dependence with the copying instruction. However, address 0x3F00 does not
intersect with the address ranges in the CSR because copying for the address has
been completed by subcopying 1 operation. Instruction C loads from an address that
is in the address range stored in the CSR. So it is checked further against the entry
in CST. Since address 0x4000 is the first byte in “Dst 2” page, it falls into the first
cache block of that page. The first bit in CPBV-Dst is a ‘1’ which indicates that
the block has completed copying, so its dependence is resolved and instruction C can
issue and execute. Finally, instruction D loads from address 0x4040 which is the 65th byte of “Dst 2” page. It corresponds to the second cache block of that page, and since the second bit in CPBV-Dst is a ‘0’, the block has not completed copying. Hence, instruction D waits until the dependence is resolved. Note that the per-block copying progress tracking and dependence checking removes the dependence pipeline bottleneck. In contrast, DMA and Copy Engine do not have such a mechanism, so instructions B, C, and D would all stall until copying is fully completed.

### 3.3.3 Handling Cache-block Alignment Issues

We have discussed the BCIE design when the regions involved in copying/initialization are not page aligned. If they are not aligned at cache block boundaries, an additional mechanism is needed to handle them in order to still permit copying/initialization at the cache block granularity. For brevity, we only discuss the high-level idea.

When the source and destination regions are not aligned at cache block boundaries, it is likely that we need to read two cache blocks from the source in order to produce one cache block (or a part of it) for the destination. Thus, to produce a destination block, the appropriate bytes must be selected from both source cache blocks and combined together. Figure 3.3.3(a) shows that to select appropriate bytes, we employ two bitmasks, one for each source cache block and one for each destination block in the subcopying region, with each bit indicating whether a corresponding byte should
Figure 3.7: Handling regions that are not block-aligned.
be selected from the block (‘1’) or not (‘0’). We found that the bitmasks show a repeated pattern and which bitmasks to use for a cache block can be determined by the its address. Hence, at most four distinct 64-bit alignment bitmasks for the source cache blocks are required. To reduce storage overheads, rather than storing four 64-bit bitmasks for the destination cache blocks, we store four 6-bit shift values that indicate how much data from the source block needs to be shifted to generate the needed part of a destination block. The four source alignment bitmasks and the four destination shift value are generated at the beginning of finishing step, stored in the CST entry and used repeatedly.

The logic that performs the byte selection from source blocks and generates the destination block is shown in Figure 3.3.3(b). After bitmask generation (Circle 1), the four bitmasks are input into a mux, and one of them is selected based on the address of the source blocks (Circle 2a and 2b). The output is the selected source alignment bitmask (64 bits) plus a 6-bit destination shift value. Then, source blocks are input into the Select&Shift logic, which uses the bitmask to select the appropriate bytes from the source block, and the 6-bit shift value to shift the result (Circle 3a and 3b). Finally, the two blocks are ORed together to produce the destination block (Circle 4).
3.4 Evaluation Methodology

3.4.1 Benchmarks.

We utilize micro-benchmarks to measure copying latencies and real benchmarks to measure the overall performance improvement of FastBCI. For the micro-benchmarks, we build a simple program that performs repeated copying or initialization by calling standard memory copying and initialization functions (e.g. \texttt{memcpy} and \texttt{memset}) in glibc \cite{12} library.

To measure the overall performance improvement that can be obtained by FastBCI, we apply it to buffer management functions in Linux kernel (Fedora Core 4 distribution (FC4) \cite{15} with kernel version 2.6.11 \cite{14}). The kernel buffer management functions that we modified include \texttt{copy\_from\_user}, \texttt{copy\_to\_user}, \texttt{memcpy} and \texttt{memset}. To stress the performance of FastBCI, we use three buffer-intensive benchmarks: Apache HTTP Web Server v2.0 \cite{11} with \texttt{ab} workload \cite{11}, a network benchmarking tool \texttt{Iperf} \cite{10} with 264KB TCP window, and an I/O performance modeling tool \texttt{iozone3} \cite{9} with tests performed on a 4 MB file residing on an \texttt{ext3} filesystem. We run the benchmarks from beginning to the end except for \texttt{iperf} in which we skip the first 2 billion instructions and simulate the next 1 billion instructions. For \texttt{apache} and \texttt{iperf}, we separate the performance measurement of the sender side and the receiver side because they exhibit different performance behaviors.

In addition to the extensive use of bulk memory copying and initialization in
current systems, future systems may increasingly rely on it. In a security feature, secure deallocation [49], the possibility of information leakage is reduced by limiting the lifetime of heap data is by zeroing deallocated chunks, which relies on bulk memory initialization. To evaluate the performance of our approach for applications with secure deallocation support, we choose to use eight C/C++ heap-intensive benchmarks: \textit{boxed, cfrac, deltaBlue, espresso, lindsay, LRU\textsc{sim}, richards, and roboop}. These benchmarks are widely used for testing heap management implementations due to their high allocation/deallocation rates [28, 32, 67]. For all heap-intensive benchmarks, we choose a deallocation intensive program phase and run 1 billion instructions.

Table 3.1: Machine configurations and parameters.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Core</td>
<td>out-of-order super-scalar, 4-wide, 4GHz 32-entry LSQ, 128-entry ROB, 16-entry MSHR bimodal branch predictor</td>
</tr>
<tr>
<td>I &amp; D TLBs</td>
<td>64-entry, fully-associative, 1-cycle access latency</td>
</tr>
<tr>
<td>L1 I &amp; D caches</td>
<td>32KB, 4-way, 64-byte block, 2-cycle hit latency</td>
</tr>
<tr>
<td>L2 cache</td>
<td>1MB, 8-way, 64-byte block, 8-cycle hit latency</td>
</tr>
<tr>
<td>BCIE</td>
<td>3-entry CST</td>
</tr>
<tr>
<td>Memory</td>
<td>300-cycle access time, maximum sustainable bandwidth at 6.4GB/Sec</td>
</tr>
</tbody>
</table>
3.4.2 Machine Parameters.

We build FastBCI model on top of a full system simulator Simics [75] with improved sample-micro-arch-x86 processor module and g-cache-ooo cache module. We simulate a desktop-like machine with an 4-wide out-of-order superscalar processor with 4GHz frequency and x86 ISA. The processor has a 32-entry load/store queue, 128-entry reorder buffer, bimodal branch predictor, 16-entry MSHR, and non-blocking caches. The I-TLB and D-TLB have 64 fully associative entries with 1 cycle access latency. The L1-instruction and L1-data caches are 32KB write-back caches with 64-byte block size, and an access latency of 2 cycles. The L2 cache is unified, 1MB size, 8-way associativity, 64-byte block size, and has an 8-cycle access latency. The memory access latency is 300 cycles, and the system bus has a 6.4 GB/s peak bandwidth. The BCIE has a 3-entry CST, and the total size overheads for PVSRs, CSR, and CST are less than 1KB. PVSRs, CSRs and CST, are accessible in 1 cycle, and each modulus calculation in BCIE takes 1 cycle latency as well. The machine configurations and parameters are shown in Table 3.1.
3.5 Evaluation

This section presents performance evaluation of FastBCI on various cases, starting with microbenchmark-based characterization (Section 3.5.1), followed by buffer management in TCP/IP processing (Section 3.5.2), secure deallocation in heap-intensive applications (Section 3.5.3) and OS survivability scheme using FastBCI for checkpoint creation (Section 3.5.4).

3.5.1 Performance on Micro-benchmarks

We use the micro-benchmarks described in Section 3.4 to evaluate the performance of FastBCI. Figure 3.8(a) shows the execution time for each copying operation performed by calling `memcpy` (which copies at the word granularity), or by utilizing FastBCI instructions (`FastBCI`); While Figure 3.8(b) shows the execution time for each initialization operation performed by calling `memset` or by utilizing FastBCI instructions (`FastBCI`). The copying/initialization region size is varied from 2 bytes to 2 MB (512 pages), and for each size, the execution time is normalized to the `memcpy` case. For FastBCI, we set the cache affinity of both source and destination to “cacheable” so that its cache behavior is the same with those of `memcpy` and `memset`. For FastBCI, the execution time is measured until the entire initialization/copying is completed.

Figure 3.8(a) shows that FastBCI reduces the execution time per copying for all
Figure 3.8: Normalized execution time per copying operation (a), and per initialization operation (b).
region sizes by between 3% to 55%. At large region sizes, FastBCI’s granularity efficiency outperforms even more due to much fewer instructions, TLB, and cache accesses. The variability of how much FastBCI outperforms a traditional `memcpy` is actually due to the non-linear performance of the traditional `memcpy`; FastBCI’s latency per copying operation is a near-linear function of the region size. Similar trend is also observed in the initialization case shown in Figure 3.8(b).

3.5.2 Real Application Performance

To evaluate FastBCI performance in real applications, we apply the bulk copying instruction for Linux kernel (v2.6.11) buffer management. Figure 3.9 shows various speedup ratios of various benchmarks under different schemes: traditional loop-based implementation (Traditional), copy engine with synchronous bulk copying instructions [59] (CopyEng), an optimistic DMA with 0-cycle setup overhead and 0-cycle interrupt handling cost (0-DMA), our FastBCI scheme with cacheable affinity for all regions (FastBCI), and our FastBCI scheme with the best cache affinity options (FastBCI+CAF), obtained by exhaustively trying all cache affinity options and choosing the best performing one for each benchmark. To make the comparison more fair, CopyEng and 0-DMA assume cacheable copying regions.

Figure 3.9 shows that FastBCI outperforms traditional implementation, copy en-

---

2 As memory side devices, Copy Engine DMA actually require uncached regions. Using uncached regions, CopyEng and 0-DMA’s average speedups are lower: 5.7% and 12.8%, respectively.
Figure 3.9: Speedup ratios of various schemes.

gine, and an optimistic DMA in all cases. The copy engine gives 7.9% speedup over conventional loop-based approaches on average due to its granularity efficiency over traditional loop-based implementation. However, the bulk copying instructions quite frequently clog the ROB, and hence the speedups of the copy engine are limited. 0-DMA performs better than copy engine, but even with highly optimistic assumptions, it does not perform as well as FastBCI, reaching an average speedup of 17.1%. Note, however, that the average copying region sizes are relatively small: ranging from 300 bytes to 1.9KB. Thus, in reality, a DMA engine will not be able to amortize its setup and interrupt completion handling overheads well. Through early retirement and per-block copying progress tracking and dependence checking, FastBCI significantly outperforms other alternatives for all benchmarks. The average speedup is 23.2% (roughly 3× of CopyEng and 1.5× of 0-DMA). With the best cache affinity
options applied to each benchmark, the speedups further improve to an average of 32.1\% (roughly 4× of CopyEng and 2× of 0-DMA). Overall, the results demonstrate that pipeline and cache bottlenecks are the biggest performance roadblocks to a bulk copying instruction. Solving them gives a significant performance improvement over just naively providing better granularity efficiency through a bulk copying instruction support.

**Cache Affinity Flexibility Effectiveness.** The speedup provided by FastBCI is very close to the “perfect pipeline” case in Figure 1.2 (23.2\% vs. 24\%). The speedup provided by FastBCI+CAF, however, is less close to the “perfect cache” case in Figure 1.2 (32.1\% vs. 37\%). Thus, it is important to understand better the relationship of cache affinity performance with the behavior of the application.

Figure 3.10(a) shows the speedup ratios of apache achieved by FastBCI for various cache affinity options over a traditional loop implementation. The notation on the X axes of A_B corresponds to A being the affinity option for the source (C = cacheable, NEU = cache neutral), and B being the affinity option of the destination. Note that the non-cacheable option is not shown because it is consistently outperformed by the cache neutral option for both source and destination regions.

Figure 3.10(a) shows that cache affinity options affect the receiver and sender sides of apache differently, so we will discuss them separately. At the receiver, (NEU,NEU) outperforms other affinity options (21\% speedup vs. 18\% speedup in C,NEU, 17\%
Conceptually, with cacheable source and destination regions, during copying they are brought into the cache. If a region has good temporal locality, future cache accesses will find blocks of that region already fetched into the cache, reducing future number of cache misses. However, the figure shows that the more we make regions cacheable, the worse the speedup becomes.

To understand why this is so, we collect the number of L2 cache misses for the code not including misses caused by the copying instructions themselves (Figure 3.10(b)). The number of misses is broken into two components: \textit{OrigMiss} shows the number of original misses that remain after a cacheable option is applied, and \textit{HarmMiss} shows the number of new misses that are caused by using the cacheable option, which
represents the harmful cache pollution incurred by bringing copying region into the cache. The number of L2 cache misses is normalized to the NEU_NEU case.

Figure 3.10(b) shows that while caching the source or destination region provides some prefetching effect, as evident by the reduction in original misses, it also produces cache pollution. Since the cache pollution effect is stronger, the new extra misses outnumber the reduction in original number of misses. To understand why this is the case, in TCP/IP processing, the receiver side receives packets by copying them from an OS kernel buffer (which is not reused after the copying) to a user buffer (which is reused by the application after the copying). Hence, caching the source region mostly pollutes the cache. Caching the destination seems fruitful but unfortunately the system call involved (recv or read) consists of a long chain of kernel routines before and after the copying. Such kernel routines accesses a lot of data within the kernel stack that have better temporal locality than the copying region. So before the user has a chance to reuse the destination, many of the destination region blocks are already replaced. As a result, caching the destination also leads to cache pollution.

Now let us consider the sender side. At the sender side, NEU_C outperforms all other options (25% speedup vs. 22% speedup in NEU_NEU, 21% in C_C, and 14% in C_NEU). The figure shows that the reason NEU_C performs the best is because its number of original L2 cache misses is reduced by 4.4% without adding many new harmful misses (only 1.7%). The reason is that in TCP/IP processing at the sender side, data is copied from a user buffer to an OS kernel buffer. The source region is not
reused so caching it causes cache pollution (as in C,NEU). However, the destination region is immediately reused because the TCP/IP stack performs further processing on the newly copied packets, so caching it is beneficial (as in NEU,C).

Figure 3.11: Speedup ratios (a) and the number of L2 cache misses (b) of Iperf with various cache affinity options.

Figure 3.11 shows the results for iperf with the same format as in Figure 3.10. Comparing apache, iperf shows identical trends in terms of the relative performance of different cache affinity options, partially because they rely on the same kernel functions for TCP/IP processing. One difference is that iperf shows much higher speedups overall ranging from 26% to 46% at the receiver side, and 25% to 40% at the sender side. Another difference is that the performance variation across cache affinity options is also higher. This is mainly caused by larger copying region sizes, which
make the FastBCI even more efficient compared to a loop-based implementation. For example, the average copying region size is 1.4 KB in iperf receiver side compared to an average of 900 bytes in apache receiver side.

Figure 3.12: Speedup ratios (a) and the number of L2 cache misses (b) of Iozone3 with various cache affinity options.

Figure 3.12 shows the speedup ratios and the number of L2 cache misses for iozone3 with the same format as Figure 3.10. Iozone3 is unique in that it performs a file operation and not related to TCP/IP operations. The figure shows that NEU_NEU outperforms other cache affinity options with a speedup of 29% vs. 24% in C_NEU, 26% in NEU_C, and 21% in C_C, which is also explainable by the number of L2 cache misses. To understand why this is the case, iozone3 copies data from the user heap to a destination region in kernel slabs [30] that will later be transferred to the disk by
the DMA. The source is not reused much after the copying because different regions in the heap are involved in different instances. The destination is not reused much because it is eventually invalidated from the cache for the DMA transfer. However, the source and destination regions do have some temporal reuse locality as shown by the decrease in L2 cache misses, however the blocks that they replace have a much higher temporal locality that caching them incurs new cache misses more than it eliminates old cache misses.

**Sensitivity to a Larger L2 Cache Size.** Figure 3.13 shows the performance of FastBCI with various cache affinity options for a larger, 8MB L2 cache size. The figure shows almost identical relative speedup patterns as in a 1MB L2 cache. What is markedly different is that the relative difference in speedups of different cache affinity options has diminished significantly. The reason for this is that an 8MB cache can tolerate cache pollution much better than a 1MB L2 cache, hence cacheable source or destination, or both source and destination, perform very close to the best affinity option. Overall, in both 1MB and 8MB L2 caches, the cache neutral option for both source and destination regions shows a robust performance that is comparable to the best affinity options.
Figure 3.13: Speedup ratios over loop-based implementation for 8MB L2 cache with various cache affinity options.

3.5.3 Secure Deallocation Performance

With an efficient support for bulk memory copying and initialization operations, many security and reliability features can be supported at very low cost. For example, in secure deallocation [49], heap chunks are zeroed out at deallocation time to reduce their life time and reduce the possible of information leaks. As pointed in that study, if deallocated chunks are not immediately zeroed out, data can remain in the memory almost indefinitely, for as long as days and weeks, even across system reboots. Sensitive data such as passwords, social security numbers, credit card numbers are stored in memory by programs such as web browsers, and can be leaked due to programmer’s error or unexpected interaction of features such as core dump and logging. Clearly, zeroing out deallocated chunks at the time of deallocation is a common sense step to reduce security vulnerability of programs.
While it was reported that the overheads of secure deallocation are typically small [49], we find that for heap-intensive applications the execution time overheads can become substantial. We implement secure deallocation heap library using a loop-based memory initialization implementation by calling \texttt{memset} functions to clear out deallocated heap chunks. Then, we also implement a library function in which the calls to \texttt{memset} are replaced by FastBCI’s BLKINIT instruction. Figure 3.14 shows the execution time overheads for heap-intensive benchmarks with secure deallocation support using a loop-based implementation, versus FastBCI with various cache affinity options (Noc = non-cacheable, Neu = cache neutral, and C = cacheable).

![Figure 3.14: Execution time overheads for heap-intensive benchmarks with various implementations of secure deallocation. (Noc = non-cacheable, Neu = cache neutral, and C = cacheable)](image)

One observation from the figure is that secure deallocation performed by FastBCI\textsubscript{C} incurs negligible performance overhead across all eight benchmarks, with
an average of 1.5\% and the worst case of 5.4\% in roboop, whereas secure deallocation performed by \texttt{memset} incurs noticeable performance overhead with an average of 7.2\% and the worst case 22.2\% in roboop. In roboop, a deallocation request of 30 bytes is made every 1400 cycles. richards has no slowdown since it makes no deallocation requests. The results again demonstrates the performance advantages of our FastBCI scheme.

The second observation is that different cache affinity options again show different performance. FastBCI\_Noc shows a larger execution time overhead compared to FastBCI\_C, on average 3.6\% vs. 1.8\%. The reason is that a freed chunks tend to be reallocated immediately and thus have a very high temporal locality. Keeping them non-cacheable leads to future cache misses, while caching them avoids future cache misses. On the other hand, FastBCI\_Neu and FastBCI\_C perform almost identically. The reason for this is that deallocated heap chunks tend to be already in the cache when deallocated, so both schemes keep them in the cache without incurring cache pollution.

### 3.5.4 Performance of OS Survivability Scheme using FastBCI for Checkpoint Creation

The major source of performance overhead in our OS survivability scheme is the checkpoint creation of the recovery mechanism. We measured that on average 87\% of
the performance overhead comes from checkpoint creation even using the memory-side Copy Engine scheme [57, 59]. To further reduce the performance overhead of our OS survivability scheme, we apply FastBCI for checkpoint creation to the survivability scheme.

Figure 3.5.4 shows the execution time overhead of our OS survivability scheme using Copy Engine (labeled as CopyEng) or our FastBCI scheme (labeled as FastBCI) for checkpoint creation. Each bar is further broken into the execution time overhead incurred by the attack detection mechanism (Detection) and checkpoint creation (Ckpt). As demonstrated in Section 2.4, using protection cache with multiple bloom filters for attack detection obtains the best performance, as a result we apply this approach in the experiment. Furthermore, we exhaust all possible cache affinity combinations for the source and destination regions in FastBCI, and present the best-performing one in the figure.

Figure 3.5.4 shows that using FastBCI for checkpoint creation in our survivability scheme achieves better performance than using a memory-side Copy Engine in general. On average, it reduces the checkpoint creation performance overhead from 1.8% to 0.8%, and hence reduces the overall performance overhead from 2.1% to 1.0%. For kernel intensive benchmarks, the overhead performance overhead is reduced from 3.7% to 1.8%, with the worst case of iperf being reduced from 4.7% to 2.5%. This again demonstrates the performance effectiveness of removing the pipeline and cache
Figure 3.15: Overall performance overhead of OS survivability scheme using Copy Engine [57, 59] or FastBCI with optimal cache affinity option combination for checkpoint creation.
bottlenecks in bulk memory copying and initialization operations. We also found that setting the checkpointing source region to be cacheable and the destination region to be cache neutral obtains the best performance in all cache affinity option combinations we tested. The reason is that in the checkpoint creation, the source region resides in the kernel stack or kernel pool that is to be accessed immediately after checkpoint creation completes, while the destination region resides in the checkpoint memory region that is rarely touched unless a real kernel security attack is detected and rollback is needed. Therefore, caching the source region is fruitful while caching the destination region easily pollutes the cache.
Chapter 4

Conclusions

4.1 Efficient Architecture Support for Operating System Survivability

In this dissertation, we have presented the need for ensuring OS survivability in the presence of security attacks. We have also presented a recovery-based survivability that includes various mechanisms that work together: attack detection, recovery, and fault isolation. We have shown that through simple but carefully-designed architecture support, the execution time overheads of our survivability scheme can be kept at under 1% even for kernel intensive benchmarks.

We have validated the functionality of the survivability mechanism through two real world kernel attacks and one variable buffer overflow driver attack. We found that
although the attacks are not new (more than one year old), surprisingly many Linux systems are still completely unprotected against these attacks. In general, systems remain unprotected from some exploits until security patches are eventually released and installed by users of the system. Therefore, we believe that it is important to have a survivability mechanism that does not require any modifications to the kernel source code or kernel recompilation. Our mechanism achieves that with a relatively simple architecture support.

4.2 Architecture Support for Improving Bulk Memory Copying and Initialization Performance

In this dissertation, we have also shown that the key to significantly improving the performance of bulk copying_INITIALIZATION instructions is removing pipeline and cache bottlenecks of the code that follows copying operations. We have proposed and presented a novel architecture support that achieves granularity efficiency of a bulk copying_INITIALIZATION instructions without their pipeline and cache bottlenecks (FastBCI). To overcome the pipeline bottlenecks, FastBCI relies on a non-speculative early retirement of the copying instruction, and fine-grain tracking of copying progress. To overcome the cache bottlenecks, FastBCI allows three cache affinity options that can be applied individually to the source and destination regions: cacheable, non-cacheable, and cache neutral. When applied to kernel buffer management, we showed
that on average FastBCI achieves anywhere between 23% to 32% speedup ratios, which is roughly $3 \times 4 \times$ of an alternative scheme, and $1.5 \times 2 \times$ a highly optimistic DMA engine with zero setup and interrupt overheads. When applied to our OS survivability scheme, we showed that the performance overhead of our OS survivability scheme is reduced to 1% on average.
Bibliography


