Architectural Support to Accelerate Fine-Grain Program Monitoring

THÈSE N^{O} 6257 (2014)

PRÉSENTÉE LE 3 OCTOBRE 2014

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE D'ARCHITECTURE DE SYSTÈMES PARALLÈLES

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Sotiria FYTRAKI

acceptée sur proposition du jury:

Prof. M. Grossglauser, président du jury
Prof. B. Falsafi, directeur de thèse
Dr E. Bugnion, rapporteur
Dr Ph. Gibbons, rapporteur
Prof. S. Kaxiras, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2014

Acknowledgements

First and foremost, I would like to thank my academic advisor Babak Falsafi for giving me the opportunity to make this journey. Over years, Babak taught me everything that I will ever need for my career, always aiming to get the very best out of me. Babak always provided me with an excellent working environment at Parallel Systems Architecture Laboratory (PARSA), École Polytechnique Fédérale de Lausanne. Additionally, Babak introduced Sushi to me and other PARSA fellows during a memorable outing at SushiAnn, New York. Thank you for everything!

I would like to thank Edouard Bugnion, Phillip B. Gibbons and Stefanos Kaxiras for serving on my thesis committee, and for improving my thesis with their insightful comments. Additionally, I would like to thank Matthias Grossglauser for serving as the jury president for my thesis exam.

I would like to thank Boris Grot and Evangelos Vlachos for our excellent collaboration. Working with them was a great experience that fueled my graduate studies, and helped me to grow academically. Apart from being great colleagues, Boris and Evangelos have also been good friends. Thank you both!

Special thanks go to Onur Kocberber for our great collaboration during our junior years at PARSA. Additionally, I would like to thank Onur for his friendship and for accompanying me to HPCA 2014, in Orlando.

I am grateful for having great colleagues at EPFL. I will remember Stavros Volos for being a good friend and for providing excellent feedback on my research and various drafts over years. I would like to thank Djordje Jevdjic and Cansu Kaynak for being good friends and excellent proof-

readers for my papers. I would also like to thank the rest of my friends and colleagues at PARSA group for providing invaluable feedback on my work, and for attending my practise talks: Mike Ferdman, Al Mutaz Adileh, Effi Georgala, Javier Picorel Obando, Alexandros Daglis and Pejman Lotfi-Kamran. Additionally, I would like to thank Rodolphe Buret for providing technical support, Stéphanie Baillargues and Valérie Locca for providing administrative advice, and Ousmane Diallo for his advice on job finding. I am also grateful to Manos Athanassoulis and Ioannis Alagiannis; during these years, Manos, Ioannis and I exchanged useful information related to swiss and EPFL bureaucracy.

During my undergraduate studies at Technical University of Crete, Greece, I had the opportunity to work with excellent researchers, and take classes with inspiring teachers. Specifically, I would like to thank Dionisios Pnevmatikatos, who supervised my undergraduate and master theses, Apostolos Dollas, and Kostas Kalaitzakis for providing me with a strong background, and for encouraging me to pursue a Ph.D. degree.

I am lucky to have close friends that have shared good and bad times with me. Many thanks go to my childhood friends Eleftheria Kili, Stelina Tsafaraki and Theodosia Fiotodimitraki, my friends from my undergraduate studies Angelos Arelakis, Giorgos Smaragdos and Panos Ainalis, and my recent friends Yiannis Sotiropoulos, Kostas Matakos, Kostas Sougias and Margarita Maouni.

I am deeply and forever indebted to my parents, Giorgos Fytrakis and Katerina Antoniou-Fytraki, for their love, support and encouragement throughout my entire life, and to my sister, Lia Fytraki, who always makes my life brighter and colorful.

I would like to dedicate this thesis to Yiannis Papadakis for this love, support and patience, and for our motorcycle rides throughout all these years.

Finally, I would like to express my appreciation to Jean-Eudes Ranvier for helping me with the French version of my thesis abstract. I gratefully acknowledge the funding support from Swiss National Science Foundation, Project No. 200021_140551/1.

Abstract

Software robustness is an ever-challenging problem in the face of today's evolving software and hardware that has undergone recent shifts. The increase in computational power has been accompanied by an unprecedented increase in the occurrence of software bugs and vulnerabilities that not only leads to breaches in privacy or financial loss, but may eventually cause catastrophic failures.

Instruction-grain program monitoring is a powerful technique to detect and mitigate bugs. Instruction-grain monitors track the execution of individual instructions to identify anomalous behavior. Tracking instruction execution at speed requires custom hardware which can only detect a particular bug or class of bugs. Firmware allows for flexibility to detect a variety of bugs but comes with a 10x slowdown in execution, while without hardware support the slowdown can be as high as 100x. Although general monitoring tools with low runtime overhead would significantly assist the debugging process, none of the techniques available today provide a flexible solution with affordable performance degradation.

This thesis proposes architectural support for a flexible at-speed Filtering Accelerator for Decoupled Event processing, or FADE. FADE is based on the observation that much of that instruction-grain monitoring overhead can be virtually eliminated because either bugs are rare (e.g., most application accesses go to an allocated memory region), or the applications have an expected behavior that requires no monitoring action (e.g., applications mostly operate on non-pointer data).

Based on these observations, the monitoring activity in response to common application activity can be *filtered* thereby significantly reducing the runtime overhead of software monitors for a variety of memory, security, and concurrency bugs. To allow for flexibility, the unfiltered application activity is delegated to software for further processing. Unlike prior work on architectural support for monitoring that necessitated a dedicated core to run the monitors software, this thesis shows that when filtering most of the monitor's activity with complexity-effective hardware structures, there is no need for a duplicate set of hardware resources. As parallel software constitutes a large fraction of modern software, the thesis develops hardware extensions to support the monitoring of single- and multi-threaded applications alike.

Keywords: software robustness, application monitoring, support for single- and multi-threaded applications, architectural extensions, filtering accelerator, software bugs, debugging.

Résumé

La robustesse du logiciel est un problème difficile due à la constante évolution des logiciels et du matériel. L'augmentation de la puissance de calcul a été accompagnée par une augmentation sans précédent du nombre de bugs logiciels et de failles qui non seulement conduisent à des violations de vie privée ou des pertes financières, mais peuvent également provoquer des défaillances catastrophiques.

Le suivi des programmes par évaluation des instructions est une technique prometteuse permettant de détecter et atténuer les bugs. Le suivit des programmes permet de suivre l'exécution des instructions individuelles afin d'identifier un comportement anormal. Effectuer ce suivi sans altérer la vitesse d'exécution des programmes nécessite un matériel sur mesure qui ne peut détecter qu'un bug ou une catégorie de bugs. Les approches basées sur des hardwares généralistes permettent de détecter une plus grande variété de bugs, mais sont accompagnés de ralentissement de l'ordre d'un facteur 10. Cependant, sans support matériel, le ralentissement peut être de l'ordre d'un facteur 100. Bien que des outils de suivi impactant faiblement le temps d'exécution aideraient considérablement le processus de débogage, aucune des techniques disponibles aujourd'hui à un prix abordable ne fournit une solution flexible sans dégrader les performances.

Cette thèse propose une extension architecturale pour un accélérateur de filtrage découplant le traitement des évènements/instructions (FADE : Filtering Accelerator for Decoupled Event Processing). FADE est basé sur l'observation que beaucoup de ralentissements liés au suivi des programmes peuvent être évités pour deux raisons : 1) Les bugs logiciels sont rares (ex. la plupart des accès mémoires se font dans des zones allouées préalablement. Les dépassements de mémoires

sont donc relativement rares.), 2) Les programmes ont des comportements ne requérant qu'un suivi basique (ex. Les programmes travaillent rarement avec des pointeurs).

Sur la base de ces observations, le suivi des programmes déclenché par la plupart des évènements générés par l'application peuvent être filtrés afin de réduire de manière significative le ralentissement due au logiciel de suivi pour plusieurs types de bugs (bug de mémoire, bug de sécurité, bug de concurrence). Dans un but de flexibilité, les évènements non filtrés sont délégués à un logiciel. Contrairement aux approches précédentes d'extension architecturales permettant le suivie des programmes qui accaparaient un cœur complet, cette thèse montre que le filtrage de la plupart des évènements grâce à un hardware efficace, permet de s'affranchir du cœur dédié. Etant donné que les applications multithread constituent une part importante des applications modernes, cette thèse propose une extension matérielle pour supporter le suivit programmes mono et multithread.

Mots-clefs: robustesse du logiciel, suivi des programmes mono et multithread, extension architecturale, accélérateur de filtrage, bugs logiciels, débogage.

Table of Contents

Acknowledgements ..... **iii**

Abstract ..... **vii**

Résumé ..... **ix**

Table of Contents ..... **xi**

List of Figures ..... **xv**

List of Tables ..... **xix**

Chapter 1 Introduction **1**

1.1 The Multi-Core Era 2

1.2 How to Mitigate Bugs 4

1.3 Prior Work on Dynamic Instruction-Grain Monitoring 6

1.4 Thesis Contributions 8

Chapter 2 Background **11**

2.1 Instruction-Grain Monitoring 11

2.2 Metadata Organization 13

2.3 Studied Monitors 14

Chapter 3 Why Filter? **19**

3.1 Generalized Monitor Functionality 19

3.2 Filtering Common Application Events 22

3.3 Implications of Filtering on Monitoring Systems' Design 24
3.3.1 Fast Monitoring 24
3.3.2 Flexible Monitoring 25
3.3.3 Resource-Efficient Monitoring 25
3.4 Summary 28

Chapter 4 Event Management 29

4.1 Motivation 29
4.2 A Quantitative Analysis 31
4.2.1 System Description 31
4.2.2 Event Producer 32
4.2.3 Event Queue 34
4.2.4 Filtering Accelerator 36
4.2.5 Unfiltered Event Queue and Consumer 37
4.3 Summary 38

Chapter 5 FADE 41

5.1 Baseline Filtering Accelerator 43
5.1.1 Filtering Unit 44
5.1.2 Stack-Update Unit 49
5.2 Non-Blocking FADE 49
5.2.1 Observations 49
5.2.2 Extensions to the Baseline Pipeline 51
5.3 Methodology 52
5.4 Evaluation 55
5.4.1 FADE versus Unaccelerated System 55
5.4.2 Performance for Different Core Types 58
5.4.3 Single-Core versus Two-Core System 59
5.4.4 Benefits of Non-Blocking Filtering 61
5.4.5 Area and Energy Efficiency 61
5.5 Summary 61

Chapter 6 Parallel FADE 63

6.1 Background: Event Order 64

6.2 Baseline Parallel Monitoring System 65

6.3 Accelerating Parallel Monitoring 68

6.3.1 Parallel FADE 68

6.3.2 Comparison to Prior Work 70

6.4 Parallel FADE's Design 71

6.4.1 Dependence Recorder 71

6.4.2 Dependence Checker & Progress Publisher++ 71

6.4.3 Dependence Queue 74

6.5 Accesses to Shared Metadata 75

6.5.1 Metadata Coherence 75

6.5.2 Racing Metadata Accesses 76

6.6 Evaluated Systems 77

6.7 Methodology 78

6.8 Evaluation 80

6.8.1 Monitoring Load 80

6.8.2 Filtering Efficiency 82

6.8.3 Parallel FADE versus Unaccelerated System 83

6.8.4 Dedicated Monitoring Core versus HW thread 86

6.8.5 Scalability Analysis 87

6.8.6 Discussion 89

6.9 Relaxed Memory Models 90

6.10 Summary 91

Chapter 7 Related Work 93

7.1 Software Monitoring Systems 93

7.1.1 Dynamic Binary Instrumentation Approaches 93

7.1.2 Software Monitoring Tools for Sequential Bugs 94

7.1.3 Software Monitoring Tools for Concurrency Bugs 96

7.2 Hardware-Based Monitoring Systems 98

7.2.1 Hardware Support for Dynamic Information Flow Tracking 98
7.2.2 Specialized Hardware-Based Monitoring Systems 100
7.2.3 Monitoring Systems Using Hardware of Contemporary Processors 101
7.2.4 Watchpoint-Based Monitoring Systems 103
7.2.5 Systems Implementing the Monitor on a Different HW Substrate 103
7.2.6 Hardware-Assisted Multi-Cores 104
7.3 Support for Parallel Applications 105
7.3.1 Software Approaches 106
7.3.2 Hardware Approaches 106
7.3.3 Deterministic Record & Replay 108

Chapter 8 Conclusions 109
8.1 Future Directions 111

Bibliography 113

Curriculum Vitae 129

List of Figures

FIGURE 1: The first computer bug. Source: U.S. Naval Historical Center Online Library Photograph. 2

FIGURE 2: End of Dennard scaling in 2004. Note that the number of cores per chip is one until 2004. Source: NRC. 3

FIGURE 3: A simple memory checker. 5

FIGURE 4: A simple instruction-grain monitor. 12

FIGURE 5: (a) Application address space layout. (b) Metadata organized in an one-level metadata map (on the left), and a two-level metadata map (on the right). The figure has been initially presented in the LBA paper [23]. 13

FIGURE 6: (a) A SW handler for a load instruction event. (b) A SW handler for a stack-update event. 20

FIGURE 7: The sources of instruction-grain monitoring slowdown for instruction and stack-update events. 21

FIGURE 8: (a) A two-core monitoring system, and (b) a single-core monitoring system with filtering support. 26

FIGURE 9: The core utilization in the two-core monitoring system. 27

FIGURE 10: Log-Based Architectures (LBA) overview. 30

FIGURE 11: A monitoring system with filtering support. 32

FIGURE 12: Breakdown of application IPC to monitored and unmonitored, averaged across benchmarks for each monitor (a) for an in-order core, and (b) for a 4-way OoO core. 33

FIGURE 13: Breakdown of application IPC to monitored and unmonitored per-benchmark for (a) AddrCheck and (b) MemLeak. 34

FIGURE 14: The occupancy of an infinite event queue for (a) AddrCheck and (b) MemLeak... 35

FIGURE 15: The effect of event queue size on performance for MemLeak. 36

FIGURE 16: (a) Cumulative distribution of distances between unfiltered events for MemLeak. (b) Unfiltered burst size for all monitors and benchmarks. 38

FIGURE 17: Filtering Unit pipeline. Striped structures show pipeline extensions for Non-Blocking Filtering. 44

FIGURE 18: Event table entries. The size of an event table entry is 96 bits. 45

FIGURE 19: Event entry format. 45

FIGURE 20: The MD cache in the cache hierarchy. 46

FIGURE 21: The internals of filter logic. 47

FIGURE 22: Evaluated systems. 52

FIGURE 23: Performance of FADE compared to the unaccelerated system. 56

FIGURE 24: Performance of the single-core monitoring system for different core types..... 58

FIGURE 25: (a) Performance of the single- versus two-core monitoring systems with FADE. (b) Core utilization in the two-core system. (c) Performance benefits of Non-Blocking Filtering. 60

FIGURE 26: Event order under parallel monitoring. To maintain correctness, the monitoring process has to follow the commit and dependence order of the application instructions. 65

FIGURE 27: The baseline monitoring system. The striped structures allow the monitoring process to follow the application dependence order. 67

FIGURE 28: Numbered the two requirements to ensure correctness of the monitors execution with Parallel FADE (including Non-Blocking Filtering support): (1) The events are delivered to the filtering accelerator in commit and dependence order. (2) The event progress is advertised in commit order. 69

FIGURE 29: Leveraging coherence activity to infer inter-thread dependences. 72

FIGURE 30: A snapshot of the progress queue and the timestamp of the last filtered event, along with published progress. 73

FIGURE 31: The event queue entries that carry dependences. The graph shows ranges (e.g., 6% means >3% and up to 6%) 74

FIGURE 32: The dependence queue design for four monitored threads. 75

FIGURE 33: Evaluated System I: The monitor runs on a dedicated monitoring core. 77

FIGURE 34: Evaluated System II: The monitor runs on a dedicated HW thread. 77

FIGURE 35: Parallel FADE: Breakdown of application IPC to monitored and unmonitored: (a) averaged across benchmarks for each monitor, and per-benchmark for (b) TaintCheck and (c) MemLeak. 81

FIGURE 36: Performance of Parallel FADE compared to the unaccelerated system. 84

FIGURE 37: (a) Performance of a dedicated monitoring core versus a dedicated monitoring HW thread. (b) Core utilization in the two-core system. 87

FIGURE 38: Performance of (a) the unaccelerated system and (b) Parallel FADE for 2, 4 and 8 monitoring pairs running on 2, 4 and 8 multi-threaded cores, respectively. 88

FIGURE 39: The function that calculates the Euclidean distance in streamcluster benchmark.... 89

List of Tables

Table 1: Eight cases of access interleavings. All accesses are to the same shared variable. Subscript r denotes remote interleaving access; superscript i and p denotes one access and its preceding access from the same thread. The table has been presented in the AVIO paper [65]. 16

Table 2: Filtered instruction events. 22

Table 3: Monitors functionality and the associated clean checks and redundant updates. M stands for metadata of memory and registers. The events are based on the SPARC ISA. 23

Table 4: Simulation setup. 53

Table 5: System setup. 78

Table 6: Filtering efficiency in Parallel FADE. 82

Chapter 1

Introduction

Fabrication technology advancements have led to an unprecedented increase in computational power enabling the deployment of a wide range of services that have substantially improved the quality of life. For over four decades, we have seen an exponential proliferation of digital computing platforms penetrating all aspects of a modern life including government, commerce, entertainment, health and social infrastructure. However, the growth of computational power is accompanied by high software complexity and concerns about software robustness [26].

Bugs are prevalent in modern software, not only decreasing productivity, but also introducing vulnerabilities that can lead to security and privacy breaches and catastrophic system failures [104]. The term *bug* originates back in 1946, when an actual moth was found trapped in the Mark II Aiken Relay Calculator while the machine was being tested at Harvard University, on September 9th, 1945. The operators tapped the moth to the computer log and wrote the comment: “First actual case of bug being found” (Figure 1) [78].

Over time, bugs have had severe consequences ranging from financial loss to loss of human lives. The list of the infamous bugs is long including (1) Therac-25 radiation therapy machine, which was directly responsible for patients deaths, in the 1980s; (2) Ariane 5 rocket, an $1 billion prototype of the European Space Agency, which has been destroyed in less than one minute after launch, in 1996; (3) Northeast Blackout, which has been associated with an immense financial damage of 6$ billion, in 2003; (4) Knight Capital's computer bug, which cost over 440$ million, in 2012.

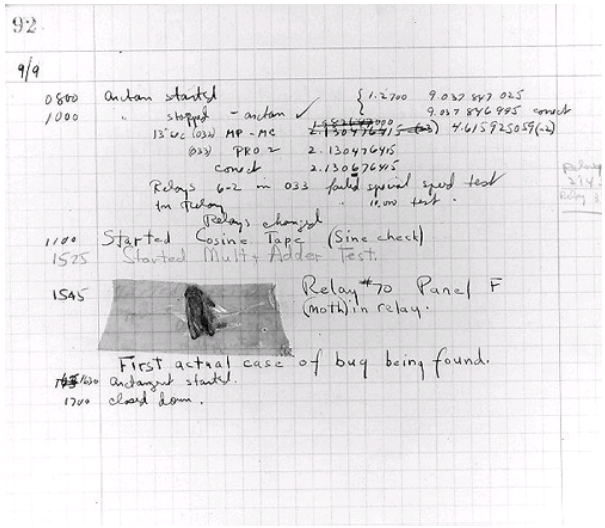

FIGURE 1: The first computer bug. Source: U.S. Naval Historical Center Online Library Photograph.

Unfortunately, debugging (i.e., the art of diagnosing the source of a bug and fixing it) is a difficult and time-consuming task. In the late 1940s, Maurice Wilkes, computer pioneer, described his realization that much of the rest of his life would be spent finding mistakes in his own programs. To ease the debugging process and to assist the developers, academia and industry have proposed a large body of debugging techniques.

1.1 The Multi-Core Era

For the past three decades Dennard scaling (1974) [34] along with Moore's law (1965) [72] have driven the development of computer systems. Moore's law provided the processor designers with a doubling number of transistors per unit area every 18 months. Most importantly, Dennard's scaling enabled designers to improve performance by increasing the frequency of a single core per chip. Consequently, newer processor generations allowed single-threaded applications to run faster, meeting the demand for additional computational power.

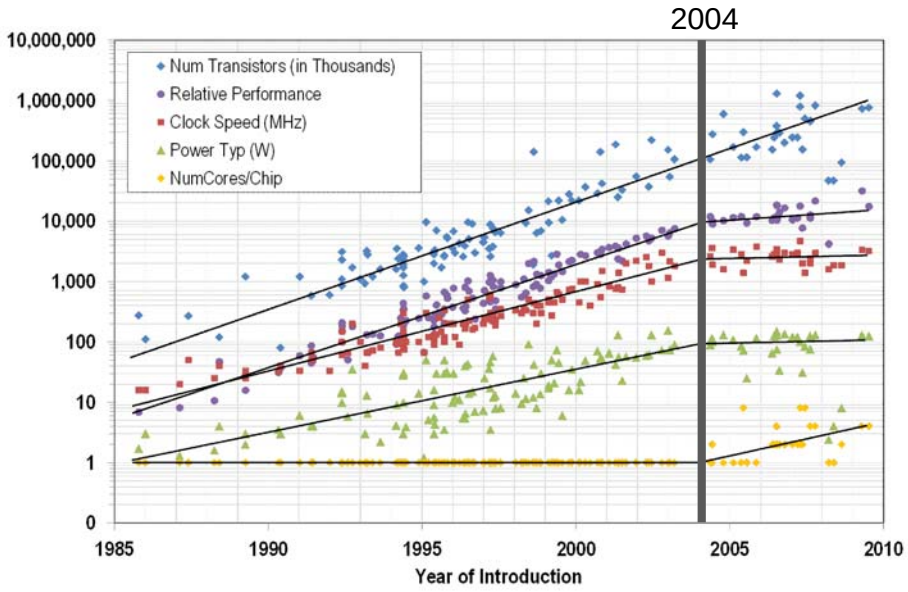

FIGURE 2: End of Dennard scaling in 2004. Note that the number of cores per chip is one until 2004. Source: NRC.

To meet the heat dissipation and cooling constraints, while improving the aggregate chip performance, computer system designers turned to multicore architectures. As shown in Figure 2, the number of cores per chip has been increasing as of 2004. Today, multicores have invaded all segments of the processors market including servers, desktops, even mobile phones.

Multicore chips can be an attractive solution given that software can scale accordingly. Unfortunately, the development of parallel software has not been commensurate with the proliferation of multicore chips. Both, writing parallel code from scratch and parallelizing a piece of serial code are notoriously difficult tasks that require substantial manual effort. Although the efficient (ideally automated) development of scalable parallel code remains an open research problem, providing robust and practical debugging tools is determinant towards this direction.

1.2 How to Mitigate Bugs

A number of complementary techniques have been proposed to assist developers in finding software bugs. These can be broadly categorized into *static* tools, *symbolic* tools, *post-mortem* tools and *dynamic* tools. Static tools (e.g., RacerX [38], CP-Miner [61], MUVI [63], RELAY [113]) aim at identifying bugs before the application executes. Symbolic tools (e.g., Cloud9 [15], KLEE [18], S2E [25], Bitblaze [101], ESD [123]) leverage symbolic values, instead of the actual values, so as to analyze a program and enumerate possible execution paths. Post-mortem tools (e.g., LXR [4], BugNet [77], FDR [118], RTR [120]) attempt to identify what went wrong after the application crashes. Dynamic tools (e.g., Valgrind [81], Purify [51], Eraser [93], CCured [28], PIN [69]) monitor the application as it executes so as to identify what went wrong during a specific run. In this thesis, we study dynamic techniques that observe programs behavior at runtime and we focus on application written in unmanaged languages (i.e., C, C++), while managed languages (e.g., C#) are out of our scope.

Dynamic tools with the ability to monitor programs at the granularity of individual instructions[1] possess a unique advantage stemming from their access to detailed runtime events, such as memory references, control flow and runtime inputs. Instruction-grain monitoring tools allow for the development of a wide range of bug-finding tools, and can effectively handle anomalous application behavior ranging from memory bugs, such as memory leaks [71], to concurrency bugs, such as atomicity violations [65]. Hereafter, we refer to these tools as *monitors*.

The monitors rely on *metadata*, which is per memory location and/or register information related to the bug-finding task. The monitor performs metadata checks to ensure that certain

1. Although dynamic tools commonly monitor fine-grain events (i.e., instructions), certain tools may monitor coarse-grain events. For instance, Dimmunix [55] that allows programs to develop resistance against already observed deadlocks, takes a monitoring action upon less frequent events, such as locks acquisition and release.

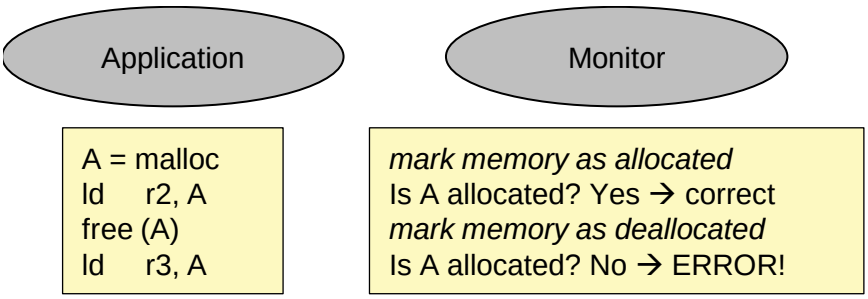

FIGURE 3: A simple memory checker.

invariants hold, for each application *event of interest*. In Chapter 3, we study a diverse set of monitors (detailed in Section 2.3), and we show that commonly monitors take actions in response to three categories of application events: (1) instruction events (e.g., add, load), (2) stack-update events (i.e., events that modify the stack pointer, such as function calls and returns), and (3) other events (e.g., memory allocation routines). For instance, a monitor that checks whether application accesses go to an allocated memory region, as the one shown in Figure 3, keeps information about the allocation status of each memory location when the application executes malloc-like and free-like routines. As shown in the figure, when the application generates a memory instruction (i.e., load or store), the monitor obtains and checks the status of the associated memory location. In general, monitoring belongs to *tagged memory* approaches, which rely on metadata to infer meta-information about applications. Other tagged-memory use cases include but are not limited to security [59], reliability [100], performance bugs [52], garbage collection [53], and transactional memory [20].

Without loss of generality, we consider two design points: *serial monitoring* and *parallel monitoring*. Under serial monitoring, a single-threaded monitor observes the behavior of a single-threaded application. Under parallel monitoring, the application and the monitor are multithreaded processes running on a CMP. Each monitoring thread is associated with an application thread.

Parallel monitoring entails more challenges compared to serial monitoring due to its inherent concurrency [112], but also allows for the development of a much broader class of monitors, such as data race [41, 93] and atomicity violation [40, 65] detectors. Under serial monitoring, the only requirement to guarantee correctness is to process application's activity in program order, which is trivially satisfied by processing the application's dynamic instructions in commit order. However, under parallel monitoring, this task is complicated because the monitoring threads have to consider the relative order of concurrent application events. An additional requirement for parallel monitoring is to ensure synchronization and atomicity for monitor's accesses at low cost. To allow for wide adoption, a monitoring approach should support both serial and parallel monitoring.

1.3 Prior Work on Dynamic Instruction-Grain Monitoring

A number of projects have targeted effective instruction-grain monitoring. Here, we briefly summarize the main attributes of software-only and hardware-assisted schemes; a comprehensive discussion is provided in Chapter 7.

Software-only monitoring frameworks rely on Dynamic Binary Instrumentation (DBI) [79] to implement the monitoring functionality dynamically. Specifically, DBI (discussed in more detail in Section 7.1) rewrites the original application code and instruments it with the monitoring code. These frameworks (e.g., Valgrind [81], PIN [69], DynamoRIO [13], DTrace [19]) enable the development of flexible, programmable, and accurate monitoring tools to detect memory access violations [51], atomicity violations [40, 65], etc. However, this flexibility comes at a steep performance penalty of 10-100x [81], since for common application events, a software handler is dispatched to check and/or update metadata. The overhead comes from (1) the instrumentation (i.e., saving and restoring registers for the application and the monitor), (2) the resource sharing between the application and the monitor, and (3) the execution of the monitoring handlers [24].

Algorithm-specific optimizations that try to reduce the performance overhead, still incur substantial slowdown (e.g., 8x for a race detector [41]).

To mitigate the performance bottleneck, researchers have investigated *hardware-assisted* solutions. *General hardware-assisted* schemes [24, 96, 112] provide hardware support for dispatching software handlers, thus eliminating the instrumentation overhead, but do not help in mitigating the handler execution time, resulting in up to 10x slowdown versus unmonitored code [23, 24]. An alternative general hardware-assisted scheme (to be studied in the future) is one that combines Dynamic Binary Translation (DBT)[2] [16, 8] with hardware support. In contrast to DBI, which is a heavy-weighted process, DBT translates the original binary through a "thin" indirection layer, but without the instrumentation overhead, thus resulting in marginal performance degradation. As a result, DBT could be used along with hardware support to lower the monitoring slowdown while maintaining flexibility.

A number of proposals aimed to bring down the performance cost through *specialized hardware* targeted at specific monitors (e.g., [35, 36, 46]). To avoid the shortcomings of monitor-specific tools, prior work proposed *hardware-based reconfigurable* tools [33, 57]. Although these tools offer flexibility, their wide adoption is limited because they are programmed in low-level hardware languages that escape the comfort zone of most programmers.

Our goal is to provide a general hardware-assisted monitoring system that combines *flexibility* and *low slowdown*, the positive attributes of prior work, with *resource efficiency.* Specifically, we would like to offer flexibility similar to software-only schemes [13, 19, 69, 81] and low slowdown similar to systems relying on specialized hardware [32, 35, 36, 46, 108], while at the same time reducing the amount of resources (i.e., a separate core [22, 32, 96]) dedicated to the monitoring task.

---

2. Popular dynamic binary translators include VMware binary translator [16], which supports x86-to-x86 binary translation, and QEMU [8], which supports cross-platform binary translation.

1.4 Thesis Contributions

This thesis provides a practical and general monitoring system for both single- and multi-threaded applications. The statement of the thesis reads as follows:

Thesis Statement

Identifying and filtering common monitoring activity with simple monitor-agnostic hardware enables the design of fast, flexible and resource-efficient monitoring systems for single- and multi-threaded applications.

By studying a set of monitors and through cycle-accurate simulations we demonstrate that:

- Instruction and stack-update events are the main contributors to the monitoring slowdown. While instructions dominate the monitor's execution profile, stack updates account for up to 17% of the execution time, thus representing an attractive acceleration target. Instruction events require fine-grained accesses to monitor's metadata, most of which can be filtered through (1) hardware-executed checks of metadata state against an invariant, and (2) detection and elimination of redundant updates that leave the metadata state unmodified. Across a diverse set of monitors targeting from memory bugs to atomicity violations, the filtering efficiency for instruction events is 84-99%. Stack-update events perform bulk metadata initialization in response to function calls and returns and can be efficiently handled with a simple state machine in hardware. Although the thesis focuses on SPARC ISA, our prior study (BugSifter [43]) shows that our observations also hold for x86 ISA.

- While 84-99% of the monitored events can be filtered in hardware, the rest of the events are delegated to software thus allowing for full flexibility and generality. Unlike prior work that dedicated a whole core to the monitoring task, we show that a dual-threaded core provides sufficient resources for the application and the monitor. In doing so, we show that 59-96% of the time one of the two cores is idle in a two-core monitoring system with filtering support.
- By studying a broad range of bug-finding functionality and applications, we make the following observations: (1) The monitoring load rarely exceeds one event per cycle even with an aggressive OoO core producing events. (2) Both filterable and unfilterable events arrive in bursts that must be buffered to reduce stalls due to backpressure. (3) Shallow queues of 16 to 32 events are sufficient for this purpose and allow for decoupling of the filtering accelerator from the core running the application.
- To maintain a high filtering rate, filtering has to happen concurrently with the processing of unfiltered events, a task that is complicated due to data dependencies between unfilterable and subsequent filterable events. To decouple filtering and the processing of the unfiltered events, we observe that there is only minimal state that is critical for deciding if a dependent event is filterable. We show that this state can be updated for unfilterable events directly in the accelerator with simple hardware extensions. We name our technique *Non-Blocking Filtering*.
- Building on our observations, listed above, we develop an architecture, along with full microarchitectural support, for a flexible at-speed Filtering Accelerator for Decoupled Event processing, or FADE. FADE is fully programmable and can support a broad range of monitoring tasks with high filtering coverage and low hardware overhead. FADE supports Non-Blocking Filtering that dynamically resolves dependencies between unfilterable events and subsequent events, eliminating data-dependent stalls and maximizing accelerator's performance. Using full-system cycle-accurate simulation, we show that FADE is highly efficient, filtering out 84-99% of

events that would otherwise be handled in software, thereby reducing the application slowdown to only 1.2-1.8x (versus 1.6-7.4x for unaccelerated execution). In the 40nm technology, an instance of FADE requires 0.12mm^2 of area and 273mW of power at peak per core.

- We present *Parallel FADE* a parallel monitoring accelerator that allows for the design of flexible, fast and resource-efficient parallel monitoring systems. Parallel FADE combines Non-Blocking Filtering, the state-of-the-art hardware filtering technique to accelerate monitoring, with the necessary hardware extensions to handle the inherent concurrency of parallel applications, overall reducing the design complexity over prior parallel monitoring accelerators.
- To showcase the applicability of FADE (including Non-Blocking Filtering support) in the context of parallel monitoring, we provide a formal proof and we perform an experimental study. Our study of a suite of diverse monitors and a number of multi-threaded benchmarks shows that Parallel FADE filters 81-99% of events that would otherwise be handled in software and reduces the slowdown to an average of only 1.1-1.8x (versus 1.9-11.5x for unaccelerated execution), thus making monitoring practical.

The rest of this thesis is organized as follows. In Chapter 2, we give background on instruction-grain monitoring and the debugging tools used in this study. In Chapter 3, we show why filtering enables the design of a fast, flexible and resource-efficient monitoring system. In Chapter 4, we study the event management in monitoring systems with filtering support. In Chapter 5, we introduce FADE's micro-architecture, along with Non-Blocking Filtering. In Chapter 6, we present Parallel FADE. Finally, we discuss related work in Chapter 7, and we conclude in Chapter 8.

The material in Chapter 4 and Chapter 5 was previously presented in the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA 2014): "Sotiria Fytraki, Evangelos Vlachos, Onur Kocberber, Babak Falsafi and Boris Grot. *FADE: A Programmable Filtering Accelerator for Instruction-Grain Monitoring*".

Chapter 2

Background

2.1 Instruction-Grain Monitoring

A number of tools have been proposed to assist developers in finding bugs. These can be grouped into five general categories: (1) static tools (e.g., RacerX [38], CP-Miner [61], MUVI [63], RELAY [113]), (2) post-mortem tools (e.g., LXR [4], BugNet [77], FDR [118], RTR [120]), (3) dynamic tools (e.g., Valgrind [81], Purify [51], CCured [28], PIN [69]), (4) symbolic tools (e.g., Cloud9 [15], KLEE [18], S2E [25], Bitblaze [101], ESD [123]), and (5) model checking tools (e.g., BLAST [9], CHESS [74], Java Pathfinder [110]). While these tool categories can be considered complementary, dynamic tools with the ability to monitor at the granularity of individual instructions possess a unique advantage stemming from their access to detailed runtime events, such as memory references and information flow. This capability affords a wide range of powerful bug-finding tools, generally referred to as *monitors*, that span the spectrum from frequently occurring memory bugs to hard-to-reproduce concurrency bugs. In addition to facilitating bug finding at development time, instruction-grain monitors may be useful in the field by enabling on-the-fly recovery from errors, reducing susceptibility to security exploits, and improving damage confinement.

In general, instruction-grain monitors work by maintaining certain *invariants* and checking that these invariants hold for each application *event of interest*. Invariants might specify that every accessed memory location has been allocated and initialized, or that the value used as a jump target is not suspicious. Monitors take actions in response to three categories[3] of application events:

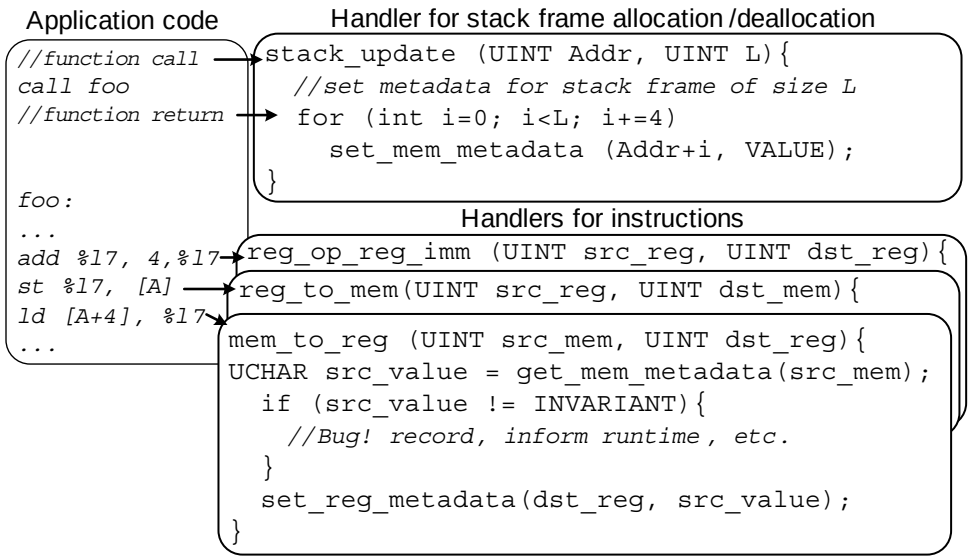

FIGURE 4: A simple instruction-grain monitor.

(1) instruction events (e.g., add, load), (2) stack-update events (i.e., events that modify the stack pointer, such as function calls and returns), and (3) other events (e.g., memory allocation routines). To assist analysis, monitors maintain bookkeeping information, or *metadata*, about application memory and registers. Depending on the event, the relevant metadata are checked against the invariant and/or updated with a new value.

A code snippet for a representative monitor, along with a slice of monitored application code, is shown in Figure 4. The monitor performs propagation-based analysis used by a number of bug-finding tools (e.g., MemCheck [81], which checks whether every referenced memory location has been initialized). In the example, each application instruction triggers a software handler associated with the monitor. For each of the instruction's source operands, the handler accesses and checks the metadata. If the metadata value differs from the invariant (e.g., a referenced memory location has not been allocated or initialized), an action is taken to inform the user and/or the run-

3. Although this thesis focuses on SPARC ISA, our prior study (BugSifter [43]) shows that these categorization also holds for x86 ISA.

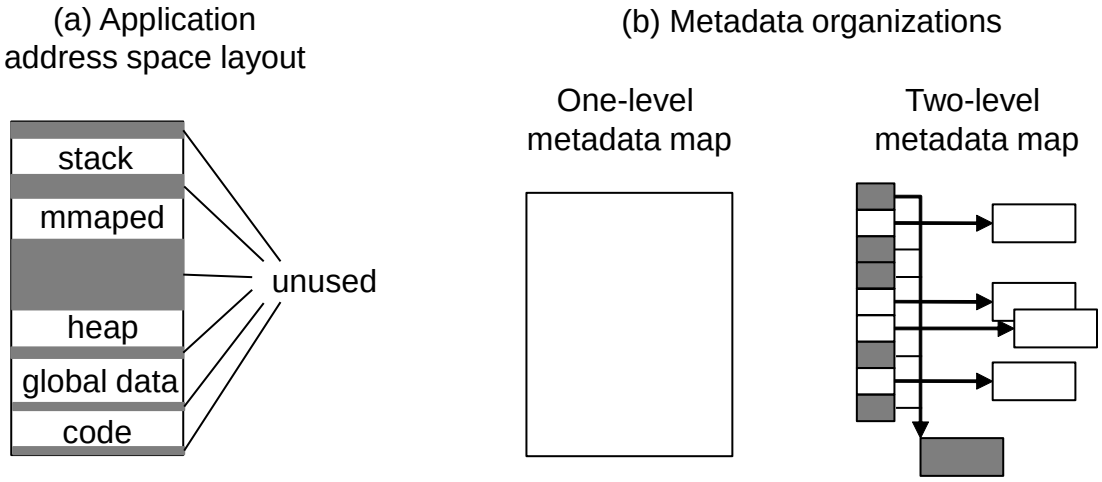

FIGURE 5: (a) Application address space layout. (b) Metadata organized in an one-level metadata map (on the left), and a two-level metadata map (on the right). The figure has been initially presented in the LBA paper [23].

time. The handler also updates the metadata for destination operands based on the metadata state of the source operands.

2.2 Metadata Organization

Prior monitoring (software and hardware) frameworks organize metadata either in an one-level metadata map [107, 108], as shown in Figure 5(b)-left, or in a two-level metadata map [22, 81], as shown in Figure 5(b)-right. The two metadata organizations are also discussed in the LBA paper [23].

The one-level design uses a contiguous memory region, which is allocated when the monitor is initialized. This design allows for simple metadata lookups, because the metadata can be obtained directly through a simple pointer arithmetic operation, given an application address. Although this design point simplifies metadata addressing, it allocates a significant amount of virtual memory for the metadata unnecessarily. The two-level design resembles page tables and requires an intermediate translation step before obtaining the metadata address given an applica-

tion address. However, this design only allocates the metadata being accessed by the monitor, thus being more efficient.

2.3 Studied Monitors

We use a suite of five diverse monitors. These monitors effectively cover a broad range of memory, security, and concurrency bugs.

AddrCheck [81] checks whether every memory access is to an allocated memory region. AddrCheck monitors non-stack memory accesses. It maintains one bit of metadata per application byte to encode the two possible states: *allocated* or *unallocated*. The tool keeps track of the calls to allocation routines (e.g., malloc(), free()) and update the associated metadata. For instance, when processing a malloc (free), AddrCheck sets the metadata corresponding to the malloced (freed) memory to allocated (unallocated).

MemCheck [81] extends AddrCheck to detect the use of uninitialized values. MemCheck maintains two bits per application byte; one bit to encode the *accessibility* status of a memory location, similar to AddrCheck, and one bit to encode the *initialization* status of a memory location. These two bits encode three metadata states (i.e., *unallocated*, *uninitialized*, and *initialized*).

The accessibility bits are updated as in AddrCheck. The initialization bits are cleared after free function calls and they are set when a constant is written to a memory location. When processing an instruction, MemCheck propagates metadata values from the source operand(s) to its destination operand. The destination operand becomes uninitialized, if at least one of the source operands is uninitialized.

MemCheck performs metadata checks to ensure correctness. When loading an uninitialized value, MemCheck does not issue an error (e.g., copying a partially initialized structure). However,

MemCheck detects an error when an uninitialized value is used in critical ways — i.e., being dereferenced as a pointer, used in conditional tests, or passed into system calls.

TaintCheck [82] is a security monitoring tool that checks for overwrite-related security exploits (e.g., due to buffer overruns, format string vulnerabilities). It performs propagation tracking to monitor the use of spurious data throughout the program execution. Program input data, such as data from the network, are marked as suspect or *tainted*. TaintCheck intercepts the necessary library calls (e.g., read, write) so as to mark the associated metadata as tainted. An error is raised, if tainted data are used in critical ways, such as in jump target addresses, or system call arguments. Although TaintCheck has two metadata states (*untainted* and *tainted*), we use one byte of metadata per application word, in order to avoid sub-byte access cost for common four-byte application operations.

*MemLea*k [71] uses a reference counting algorithm to identify leaked heap objects (i.e., objects that are no more reachable through an application pointer). MemLeak maintains one metadata word per application word, which is a pointer to the context of the corresponding malloc and a null value otherwise. The context includes an allocation ID, the PC, and a reference counter. The allocation ID is a unique identifier assigned to an object at the time of allocation.

MemLeak performs propagation tracking of pointer values throughout program's execution. Loading a pointer to a register and using the pointer to generate a new address, propagates a pointer status to the destination register. When a pointer is written back to memory (on the heap) a new reference is created, thus increasing the corresponding reference counter. When this action overwrites the pointer to a another object, the reference counter of this object is decreased. Pointers stored in the stack do not require reference counting, as they get overwritten when the stack frame is popped. A memory leak is identified if an object's reference counter reaches the zero value, while the object is not deallocated.

Table 1: Eight cases of access interleavings. All accesses are to the same shared variable. Subscript *r* denotes remote interleaving access; superscript *i* and *p* denotes one access and its preceding access from the same thread. The table has been presented in the AVIO paper [65].

#	Interleaving	Description	Serializability	Equivalent Serial Access	Problem
0	$read^p$ $read^i$ $read_r$	two reads interleaved by a read	YES	$read^p$ $read^i$ $read_r$	N/A
1	$write^p$ $read^i$ $read_r$	read after write interleaved by a read	YES	$write^p$ $read^i$ $read_r$	N/A
2	$read^p$ $read^i$ $write_r$	two reads interleaved by a write	NO	N/A	The interleaving write makes the two reads have different views of the same memory location
3	$write^p$ $read^i$ $write_r$	read after write interleaved by a write	NO	N/A	The local read does not get the local result it expects
4	$read^p$ $write^i$ $read_r$	write after read interleaved by a read	YES	$read_r$ $read^p$ $write^i$	N/A
5	$write^p$ $write^i$ $read_r$	two writes interleaved by a read	NO	N/A	Intermediate result assumed to be invisible to other threads is read by a remote thread
6	$read^p$ $write^i$ $write_r$	write after read interleaved by a write	NO	N/A	The local write relies on a value from the preceding local read that is then overwritten by the remote write
7	$write^p$ $write^i$ $write_r$	two writes interleaved by a write	YES	$write^p$ $write_r$ $write^i$	N/A

AtomCheck [65] detects atomicity violations by checking access interleavings. For this purpose, AtomCheck keeps information for the last access by each thread to each memory location, maintaining one piece of metadata per application word. The main structures are (1) a global table to keep the status bit (Private/Shared) and the id of the thread that last referenced each memory location, and (2) local per-thread tables to keep the type (Read/Write) of the last access by each thread. Implementation-wise, AtomCheck encodes the thread status bit and the thread id in one byte.

For each memory access, AtomCheck performs one of the following actions: (1) If the status bit indicates that the data are thread-private, no further action is required. (2) If the status bit indicates that the data are shared, AtomCheck first checks the global table and the current thread id. If they match (i.e., the memory location was previously referenced by the same thread), which happens in the common case, a simple software handler is dispatched to update the metadata of the local table for the current thread with the type of the last access (Read/Write). (3) Otherwise, a complex handler is dispatched to check if there is a potential atomicity violation. In the last case, where there are consecutive accessed to a memory location by two different threads, there are eight possible interleavings, shown in Table 1. If the interleaving is unserializable, a potential atomicity violation is identified and the program counter of the instruction is reported.

In AtomCheck, the metadata are initially marked as uninitialized. Upon their first access, they are marked as thread-private. Any consequent access by a different thread, sets the metadata to the shared state.

Chapter 3

Why Filter?

In this Chapter, we motivate our design choices that lead to a fully generalized monitoring accelerator allowing for low runtime and resource overhead. First, we identify common high-level functionality inherent in a wide range of monitors, targeting memory, security and concurrency bugs. Next, we provide intuition on how filtering can eliminate much of the run-time overhead associated with commonly occurring monitoring activities. Finally, we discuss the implications of filtering on the design of monitoring systems.

3.1 Generalized Monitor Functionality

The existence of different bug types dictates that monitors should be specialized for each particular type of a bug. Moreover, for a given bug type, several bug-finding algorithms may exist that differ in their coverage guarantees, resource requirements, implementation complexity, etc. Despite the resulting diversity of bug-finding tools and algorithms, we find that virtually all monitors have functionally-similar characteristics at a high level. These can be summarized as follows, with Figure 6 serving as an illustrative example.

Simple checks and updates for instruction events: The bulk of monitoring activity in response to individual application instructions involves some combination of metadata accesses, metadata checks against an invariant, and metadata updates. As most instructions in ISAs of contemporary general-purpose processors operate on one or two source operands and update one destination operand, the per-instruction handlers typically manipulate three small pieces of metadata

(a)

Handler for a load instruction (e.g., ld [A+4], %l7)

```
mem_to_reg (UINT src_mem, UINT dst_reg){
UCHAR src_value = get_mem_metadata(src_mem);
  if (src_value != INVARIANT){
    //Bug! record, inform runtime, etc.
  }
  set_reg_metadata(dst_reg, src_value);
}
```

(b)

Handler for a stack frame allocation/deallocation

```
stack_update (UINT Addr, UINT L)
{
  //set metadata for the stack frame
  //the stack frame's size is L
  for (int i=0; i<L; i+=4)
    set_mem_metadata (Addr+i, VALUE);
}
```

FIGURE 6: (a) A SW handler for a load instruction event. (b) A SW handler for a stack-update event.

or less (for their source and destination operands), with each metadata item associated with a given application register or memory location. Figure 6(a) shows a representative software handler for a load instruction.

Bulk updates for stack-update events: Software engineering practices call for abstraction and encapsulation of functionality, leading to software with many short functions and frequent function invocations at execution time. When the stack pointer of the application is adjusted, a frame is allocated/deallocated on the application stack (upon a function call/return). We refer to both types of activity as *stack updates*. Stack updates must be shadowed by the monitor to properly track what memory has been allocated to an application. As a result, each function call and return event in the application triggers a handler in the monitor that sets a region of metadata memory to a known value (e.g., *allocated+uninitialized* upon a call, *unallocated* upon a return). Figure 6(b) shows a software handler for a stack update instruction.

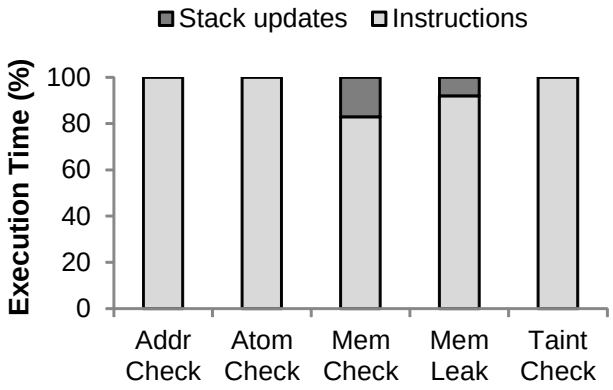

FIGURE 7: The sources of instruction-grain monitoring slowdown for instruction and stack-update events.

Complex or uncommon functionality for other events: Occasionally, monitors invoke functionality that differs from the two cases above. This happens whenever the application performs a high-level event of interest (e.g., malloc or free), initializes the metadata, or when a bug is found.

Figure 7 breaks down the monitors' execution time into (1) simple metadata checks and updates for application instruction events, and (2) metadata bulk updates in response to applications' stack frame allocations and deallocations. In this graph, we do not show complex events as they account for less than 5% of the run time for all monitors. The five studied monitors, described in Section 2.3, cover a broad spectrum of bugs. The benchmarks along with our methodology[4] can be found in Section 5.3.

As the figure shows, monitoring of instructions dominates the execution profile; however, stack updates consume up to 17% of the execution time in two out of five monitors due to a large number of instructions (over 100, on average) committed by the stack update handlers iterating through a memory region.

4. The monitors run on the baseline Log-Based Architectures system [22], also shown in Figure 8(a).

Table 2: Filtered instruction events.

AddrCheck	AtomCheck	MemCheck	MemLeak	TaintCheck
99.5%	85.5%	98.0%	87.0%	84.0%

3.2 Filtering Common Application Events

In this Section, we analyze the monitors behavior in response to the two common application events types: instructions and the stack updates. We show that the majority of the common application events can be filtered.

Instruction events: For instruction events, we make two observations: First, most of the time applications behave as expected and the metadata match the expected invariant (e.g., memory accesses reference memory that has been allocated and initialized). We refer to these events as *clean checks*. The associated handlers do not affect the outcome of the monitoring algorithm, as they just confirm that the application behaves as expected. Second, propagation event handlers that copy metadata values from source to destination operands commonly update the metadata with the same value, because metadata are stable (e.g., memory that has been initialized remains initialized while the actual value in application memory may change). We call these events *redundant updates*, as they do not affect the metadata state. As the instruction events that fall into either clean checks or redundant updates do not change the monitoring outcome and the monitor's state, they can be filtered.

Table 2 shows the percentage of instruction event handlers that fall into either the clean check or redundant update category. The benchmarks along with our methodology can be found in Section 5.3. For AddrCheck, almost all instruction events result in clean checks, because applications access allocated memory. For MemCheck the vast majority of instruction events (98%) are either clean checks or redundant updates as most application data are initialized. For MemLeak 87% of the events are clean checks because most of the applications data are not pointers. In con-

Table 3: Monitors functionality and the associated clean checks and redundant updates. M stands for metadata of memory and registers. The events are based on the SPARC ISA.

Monitor functionality	Clean Checks/Redundant Updates
AddrCheck	
The metadata values are checked to detect unallocated memory accesses.	Example: `ld %rd, mem(saddr)` clean check if (M[saddr] == *allocated*)
AtomCheck	
The metadata values are checked to detect potential atomicity violations through access interleavings.	Example: `ld %rd, mem(saddr)` clean check if (M[saddr] == thread-private)
TaintCheck	
1) The metadata values are propagated through instructions, such as ld. 2) An error occurs when tainted data are used in critical ways, such as jump targets.	1) Example: `ld %rd, mem(saddr)` redundant update if (M[saddr] == M[rd]) 2) Example: `jne %rs` clean check if (M[rs] == *untainted*)
MemCheck	
1) The metadata values are propagated through instructions, such as add. 2) An error occurs when uninitialized data are used in critical ways, such as library call arguments.	1) Example: `add %rd,%rs1,%rs2` redundant update if (M[rd] == (M[rs1] & M[rs2])) 2) Example: `ld %rd, mem(saddr)` clean check if (M[rd] == M[saddr] == *initialized*)
MemLeak	
1) The metadata values (allocation ID/pointer status) are propagated through instructions, such as ld.	1) Example: `ld %rd, mem(saddr)` clean check if (M[rd] == M[saddr] == non-pointer)

trast, TaintCheck's rate is lower (84%) than that of MemLeak, as TaintCheck performs value propagation that results in long propagation chains with a higher frequency of metadata updates. For AtomCheck, over 85% of events are clean checks as most application data are consecutively accessed by the same thread and do not risk an atomicity violation. Overall, we show that 84-99% of instruction event handlers result in either clean checks or redundant updates. In Table 3, we present examples of clean checks and redundant updates for the five studied monitors.

Stack update events: Stack update events, namely function calls and returns, contribute up to 17% of the monitoring execution time, as shown in Figure 7. We observe that these handlers set a range of metadata to a predefined value and do not check for bugs directly. Our proposal is to

accommodates stack update events in a dedicated hardware unit that updates metadata in bulk, thereby filtering the associated software handler dispatch.

Prior work proposing architectural support for monitoring has largely ignored the acceleration of *stack-update* events; Prior work either assumes a hardwired policy for all stack accesses (e.g., all memory references to the stack access initialized data) [96], or does not provide acceleration for bulk updates, thus suffering from the associated runtime overhead [23, 48]. In our prior study, BugSifter [43], we show the runtime overhead due to the lack of hardware support for bulk updates for the five studied monitors running on the LBA framework [23].

3.3 Implications of Filtering on Monitoring Systems' Design

In the previous sections, we identified common high-level functionality inherent in a wide range of monitors, and we showed that the majority of the monitored events can be filtered without affecting the monitors coverage. In this Section, we explain why filtering allows for the design of a monitoring system with three key characteristics: (1) fast, (2) flexible, and (3) resource-efficient.

3.3.1 Fast Monitoring

We envision a Filtering Accelerator, with monitor-agnostic hardware, that filters the application event stream for diverse monitors based on our observations in Section 3.2. The accelerator handles instructions and stack updates, as they are the main contributors to the monitoring slowdown.

To handle instruction events, the Filtering Accelerator includes a simple event filtering mechanism in hardware that elides the execution of costly SW handlers for clean checks (i.e., when the metadata match an invariant), and redundant updates (i.e., when the metadata of the source and destination operands are the same). To accommodates stack-update events, the Filter-

ing Accelerator includes a dedicated hardware unit that performs multi-block writes. Our observation is that stack update handlers set a large range of metadata to a predefined value, but do not check for bugs directly.

Overall, the proposed accelerator can filter 84-99% of the common monitoring activity in hardware, thus eliminating most of the runtime overhead and allowing for fast instruction-grain monitoring.

3.3.2 Flexible Monitoring

While the Filtering Accelerator can handle the common case in hardware, the unfiltered events require further processing. We identify two possible options for the processing of the unfiltered events: (1) to implement the monitoring functionality on reconfigurable fabric, and (2) to implement the monitoring functionality in software, which executes on a general-purpose core.

Reconfigurability offers flexibility, but limits wide adoption because it requires programming in a low-level hardware language that escapes the comfort zone of most programmers. An additional limitation is that a reconfigurable fabric, with an internal clock rate that is typically well under a gigahertz, may struggle to keep up [33] with today's processors commonly running at multi-gigahertz frequencies.

Our preferred approach is to write the monitor on mainstream programming languages and development tools. This guarantees maximum monitoring flexibility through software execution for any unfiltered event, and accessibility to a broad range of developers.

3.3.3 Resource-Efficient Monitoring

To allow for full flexibility, the SW handlers for the unfiltered events are executed on a general-purpose core, in contrast to other approaches that implement the monitor on specialized HW.

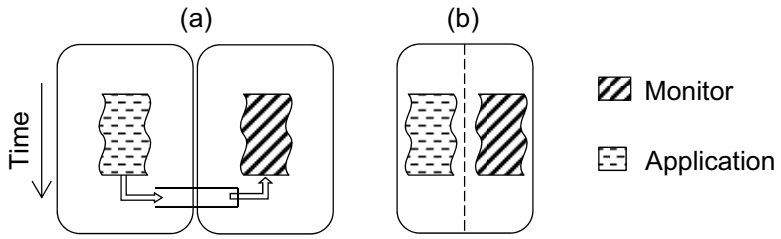

FIGURE 8: (a) A two-core monitoring system, and (b) a single-core monitoring system with filtering support.

To guarantee isolation and containment, the monitor and the application execute in a separate process We consider two possible integration options for the Filtering Accelerator, as shown in Figure 8.

The *two-core monitoring system* (Figure 8(a)) executes the application and the monitor on separate cores, with an intermediate buffer to communicate the application activity to the monitor [23]. In this system, the Filtering Accelerator inspects the application events and decides whether an event requires further processing. Once the event queue is full, the application core stalls.

Upon an unfiltered event, a software handler is executed on the monitor's core. Because the handler updates metadata state potentially read by subsequent events, filtering stops during the software handler execution. When the handler execution completes, filtering resumes, draining the event queue and allowing the application to make progress. In Chapter 5, we propose Non-Blocking Filtering, a technique that overcomes this restriction and allows for the overlapped execution of the filtering process and the SW handlers for unfiltered events.

Figure 9 shows the execution time breakdown in the two-core system for our five monitoring tools (our methodology is detailed in Section 5.3). The execution time is broken down into three categories: cycles in which (1) the application core is idle due to a full event queue; (2) the

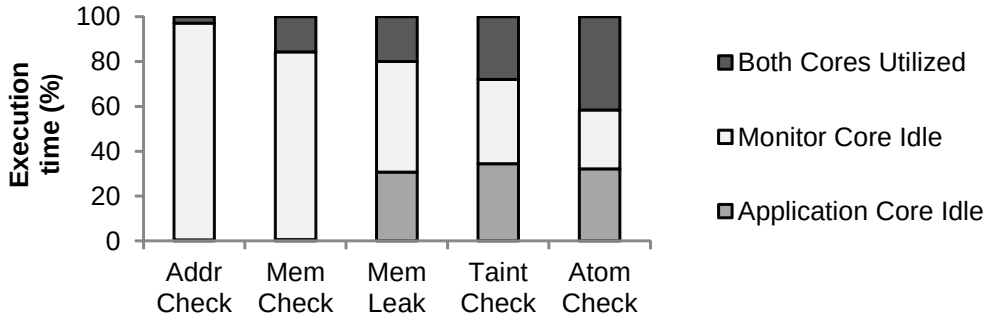

FIGURE 9: The core utilization in the two-core monitoring system.

monitor core is idle because events are filtered by the Filtering Accelerator; and (3) both, the application and monitor cores are not idle.

As the figure shows, 59% to 96% of the time one of the two cores is idle, because the two-core accelerated system is either effectively filtering the incoming event stream (idling the monitor core) or spends a significant amount of time in handler execution (stalling the application core). With both cores utilized just 16% of the time, on average, the benefit of the second core is clearly limited.

In order to reduce the resource overhead of the dedicated monitoring core, we propose a *single-core monitoring system* (Figure 8(b)) based on a dual-threaded core. In this system, the application runs on one hardware thread and the monitor on another. The hardware cost of a second hardware thread is low, especially with a simple hardware threading model (i.e., fine-grain thread interleaving instead of simultaneous issue/execute from both threads). Most importantly, performance can approach that of a two-core system, because the execution time is typically dominated by either the application (when the filtering rate is high) or the monitor (when unfiltered events are frequent), as shown in Figure 9.

3.4 Summary

In this chapter, we showed that the majority of the monitoring slowdown is due to instruction and stack-update events. For frequently occurring instruction events, we make two observations: (1) most of the time applications behave as expected (i.e., invariant checks succeed), and (2) the monitor's metadata do not need to be updated (i.e., most updates are redundant). These observations allow the vast majority of the costly software handlers invoked in response to instruction events to be filtered out. For stack updates, we show that they constitute up to 17% of the monitor's execution time. To handle these events, we propose a HW mechanism that can update metadata in bulk.

While 84-99% of the monitored events can be filtered in HW, the rest of the events require further processing. To allow for full flexibility, we choose to delegate the unfiltered events to SW. Unlike prior work that dedicated a whole core to the monitoring task, we show that a dual-threaded core provides sufficient resources for the application and the monitor.

Chapter 4

Event Management

In Chapter 3, we showed that the majority of the application events can be filtered, as they change neither the monitor's state nor the monitoring outcome, thus allowing for the design of a fast, flexible and resource-efficient monitoring system. In this chapter, we discuss the event management in such a monitoring system.

To motivate our analysis, we start with a discussion of the most closely related work (Log-Based Architectures) that reveals three important implications on the design of monitoring systems. Then, taking these implications into consideration, we study the event management in the proposed monitoring system. In doing so, we analyze the monitored and the unfiltered event generation rates in order to estimate the queueing requirements and the filtering rate requirements.

4.1 Motivation

Log-Based Architectures (LBA) [22, 23, 24] is an event-based monitoring system that captures application events in hardware and communicates them to a neighboring core on a CMP platform to perform monitoring. LBA includes three monitoring acceleration mechanisms: two accelerators to filter the event stream (a discussion of the differences between these two accelerators and our work can be found in Section 7.2.6), and one accelerator to reduce the length of software handlers executed for unfiltered events. Although, LBA is a good starting point towards generalized hardware for monitoring systems, it comes with certain shortcomings regarding event management, as we explain in the rest of this section.

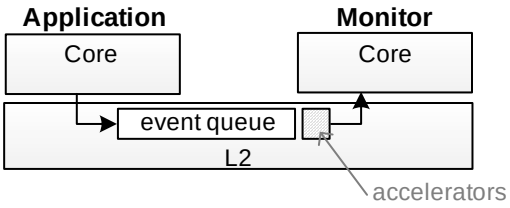

FIGURE 10: Log-Based Architectures (LBA) overview.

Figure 10 shows an overview of Log-Based Architectures. The application generates events and enqueues the events of interest in the event queue. The accelerators dequeue and examine the events from the queue. If the events can be filtered, no further action is required. Upon an unfiltered event, the filtering process has to stall until the processing of the event by the monitor's software completes. As inter-event dependencies mandate in-order processing, this system fully exposes the slowdown due to unfiltered events.

Implication 1 (decoupling): *To hide the monitoring slowdown due to unfiltered events, the filtering process should be decoupled from the processing of unfiltered events, so that the two can overlap in time. We revisit this implication in Section 5.2 and we show performance results in Section 5.4.4.*

LBA proposes two monitoring accelerators that process event streams generated by in-order cores. However, the event generation rate is expected to be higher, when the application is running on more aggressive cores. To quantify the effect of the core type, we measure the event generation rate of an in-order and a 4-way OoO core. We find that the aggressive core produces 2x more events compared to the in-order core (the complete analysis follows in Section 4.2.2).

Implication 2 (accelerators design): *The pressure on the accelerators increases when aggressive cores produce the event stream. Thus, aggressive cores should be studied, so as to ensure the wide applicability of an acceleration technique.*

LBA employs a large queue that can accommodate up to 64K events. However, the queue can assist with the monitoring slowdown, only when the event generation rate is lower than the event consumption rate.

For this discussion, we consider two systems: (1) the baseline LBA system without acceleration [24], and (2) the LBA system including the three accelerators [22, 23]. In both systems, the consumption rate is limited by the frequency of software handlers invocation, as the filtering process has to stall upon an unfiltered event. In the baseline LBA, the consumption rate is always lower than the event generation rate, because a software handler is dispatched for each monitored event. The accelerated LBA system increases the consumption rate due to the filtering support. However, the consumption rate is still lower than the event generation rate, with the exact ratio depending on the filtering ratio and the cost of processing an unfiltered event.

Implication 3 (queueing): *Queueing cannot hide the monitoring slowdown due to unfiltered events, if the filtering process has to stall upon each unfiltered event.*

4.2 A Quantitative Analysis

4.2.1 System Description

Taken into consideration the implications of our analysis in Section 4.1, we propose a monitoring system for decoupled event processing. In Figure 11, we show the main entities involved in the event processing flow. The *application* generates events as instructions retire and enqueues the events of interest (i.e., *monitored events*) in the *event queue*. The rest of the events (i.e., *unmonitored events*) do not require further processing. The *filtering accelerator* (*FA*) dequeues events from the head of the event queue and checks whether the filtering condition is satisfied. If so, events are *filtered* and no further action is required. As further processing is necessary for the rest of the events (i.e., *unfiltered events*), the filtering accelerator places them into the *unfiltered event*

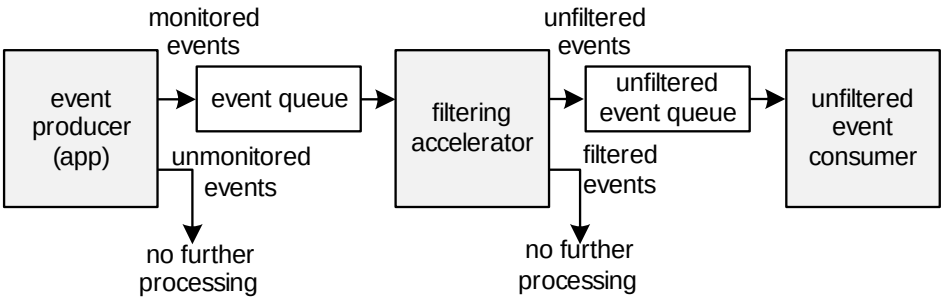

FIGURE 11: A monitoring system with filtering support.

queue. Finally, the *unfiltered event consumer* dequeues and handles the unfiltered events completing the monitoring analysis.

4.2.2 Event Producer

As the application instructions retire, they generate events. However, monitoring analyses do not require all application events to be processed. As a result, software [81] and hardware [23, 33] monitoring frameworks include support to eliminate[5] the unmonitored events. We define *monitoring load* as the ratio of monitored events to all committed instructions.

Based on the types of the monitored instruction events, monitoring analyses can be broadly categorized into two types: *memory tracking*, which process only memory instructions, and *propagation tracking*, which may track any instructions types and propagate a metadata value from the source operand(s) to the destination operand. The exact instruction types being monitored depend on the monitor's task. For instance, *MemLeak* [71], which identifies memory leaks, monitors instructions that may propagate a pointer value, such as arithmetic and load/store instructions, but eliminates floating-point instructions.

5. The term filtering has been used in prior work [33] to refer to elimination of unmonitored events. We do not use the term filtering in this context because no monitoring task is associated with unmonitored events.

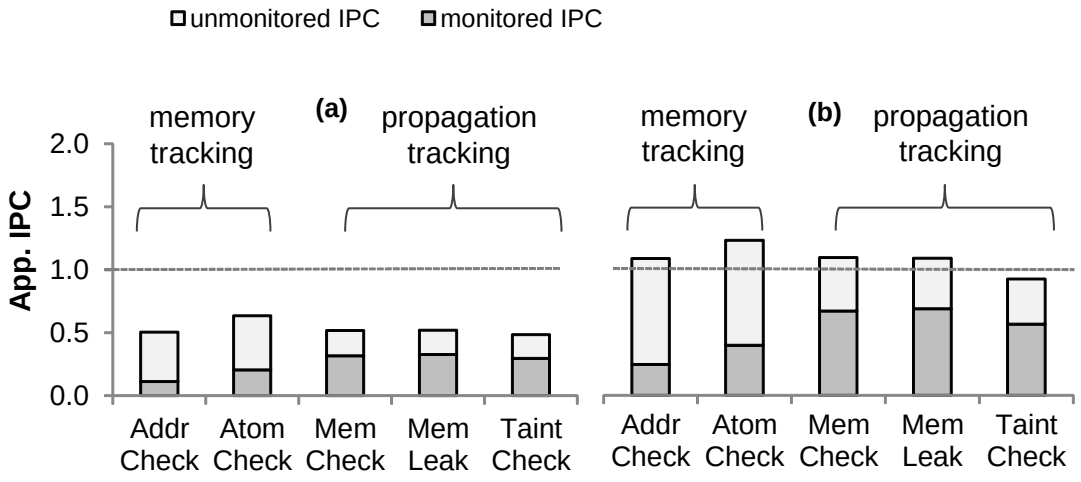

FIGURE 12: Breakdown of application IPC to monitored and unmonitored, averaged across benchmarks for each monitor (a) for an in-order core, and (b) for a 4-way OoO core.

To quantify the load on different monitors, we measure the applications' monitored IPC on an in-order core (Figure 12(a)) and an aggressive 4-way OoO core (Figure 12(b)). We detail the benchmarks and monitors in Section 5.3. Figure 12 shows per-monitor results averaged across benchmarks. For instance, for AddrCheck in Figure 12(b), the average application IPC (including both monitored and unmonitored instructions per cycle) is 1.1, out of which 0.4 (monitored instructions per cycle) require a monitoring action to be taken. We find that the applications produce 2x more events when running on the aggressive core compared to the in-order core. Therefore, we focus our analysis on applications running on the aggressive core (Figure 12(b)), as they stress the accelerator with a higher event generation rate.

In general, the monitoring load of memory-tracking monitors is lower compared to the monitoring load of propagation-tracking monitors, because propagation-tracking monitors tend to process more events. As a result, the former have a low monitored IPC (up to 0.4 event per cycle), while the opposite holds for the latter (up to 0.68 event per cycle).

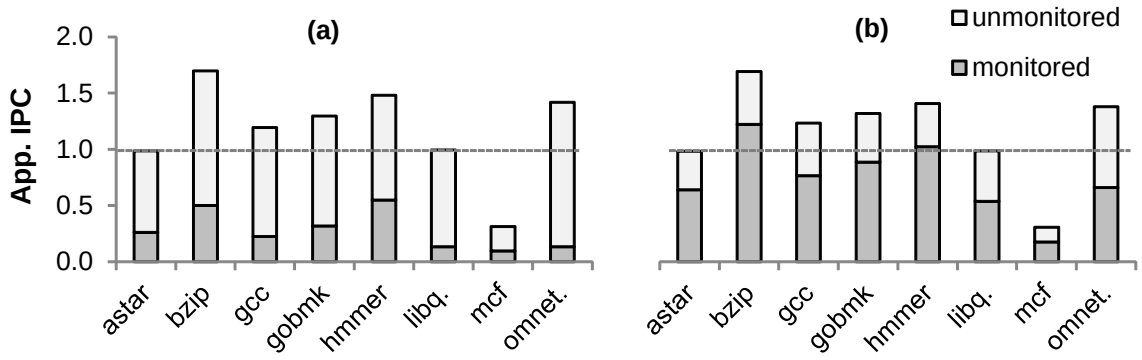

FIGURE 13: Breakdown of application IPC to monitored and unmonitored per-benchmark for (a) AddrCheck and (b) MemLeak.

Figure 13(a), shows the per-benchmark results for *AddrCheck*, a memory tracking monitor, which checks whether an access goes to allocated memory [81]. For all benchmarks, the monitored IPC is significantly below 1.0, with an average of 0.24. In contrast, Figure 13(b) shows the per-benchmark results for MemLeak, a propagation tracking monitor. While most benchmarks also have a monitored IPC of below 1.0, with an average of 0.68, the monitored IPC of MemLeak is 2.8x higher than AddrCheck, underscoring the differences in monitoring load.

The monitored IPC indicates the event generation rate of the applications and dictates the rate at which events must be consumed by the filtering accelerator. The presented analysis shows that the monitored IPC is below 1.0 for a range of monitors, even when the event stream is produced by an aggressive OoO core. We thus conclude that *a filtering accelerator with a processing capability of one event per cycle can keep up with the event producer.*

4.2.3 Event Queue

We next examine the buffering requirements between the event producer and the filtering accelerator. For the purpose of our study, we assume a filtering accelerator that processes one event per cycle and has an infinite event queue. In Figure 14(a, b), we present the cumulative dis-

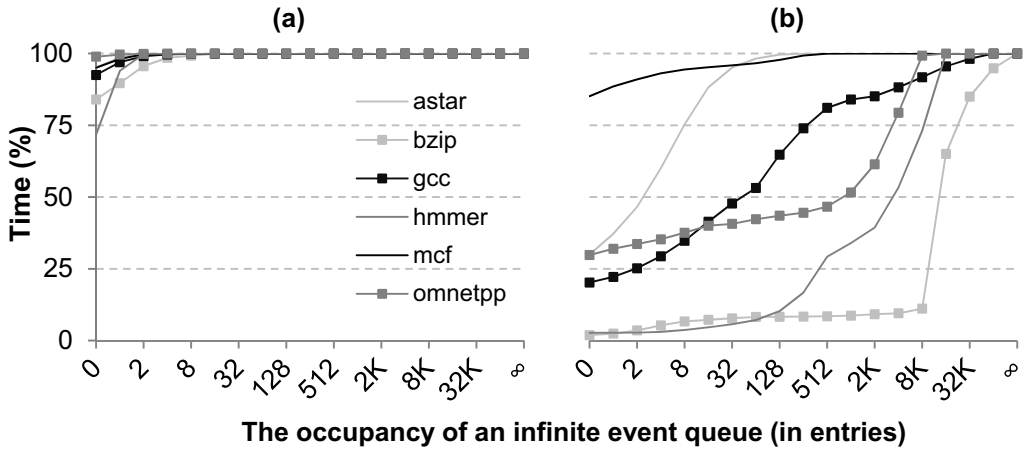

FIGURE 14: The occupancy of an infinite event queue for (a) AddrCheck and (b) MemLeak.

tribution of the event queue's occupancy for (a) AddrCheck, a memory-tracking monitor, and (b) MemLeak, a propagation-tracking monitor, on an aggressive 4-way OoO core.

For memory-tracking monitors (Figure 14(a)), the monitored IPC is low, resulting in small bursts of events that can be captured in an 8-entry queue. For propagation-tracking monitors (Figure 14(b)), the monitored IPC is considerably higher, resulting in longer bursts. Depending on the benchmark's monitored IPC, the queueing requirements range from 128 entries (mcf – low monitored IPC) to 8K entries (omnetpp – higher monitored IPC). For benchmarks with a monitored IPC greater than one, such as bzip, queueing cannot help, as the filtering rate (1.0 event per cycle) is below the event generation rate (1.2 events per cycle).

We next compare the performance loss stemming from finite queues over an infinite event queue. We evaluate two queue sizes: (1) 32K entries, which can accommodate the bursts based on our analysis, and (2) 32 entries, which is a practical-sized queue. In Figure 15, we present results for MemLeak, a monitor that exerts the greatest pressure on the queue due to its high monitored IPC. We observe that the 32K-entry queue can fully accommodate the bursts (resulting in no slowdown) for all benchmarks but bzip and gcc, corroborating the burstiness analysis in Figure 14(b).

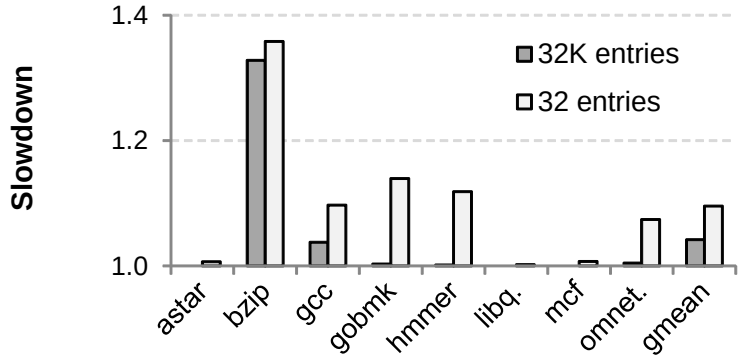

FIGURE 15: The effect of event queue size on performance for MemLeak.

Meanwhile, a much smaller queue of only 32 entries results in a slowdown that ranges from none (mcf, astar, libq.), to 1.17x (gombk). Queueing can only provide negligible performance improvement for bzip (monitored IPC over 1.0) resulting in a 1.33x slowdown for a 32K-entry queue and a 1.36x slowdown for a 32-entry queue. For gcc, queueing reduces the slowdown from 1.1x (32-entry queue) to 1.04x (32K-entry queue). We conclude that *a small (e.g., 32-entry) event queue allows for insignificant slowdown caused by bursts.*

4.2.4 Filtering Accelerator

The filtering accelerator aims at reducing the overhead of common monitoring activities (discussed in Chapter 3), which mainly happen in response to two categories of application events: (1) instructions, (2) function calls and returns. The monitors also process high-level events (e.g., malloc, fopen, mmap). The filtering accelerator does not target high-level events, as they are infrequent and require complex handling.

The vast majority of monitoring activity is due to instruction events requiring accesses, checks, and updates to the metadata of the instruction operands. Nearly all remaining monitoring activity is due to the allocation (deallocation) of stack frames on the application stack upon func-

tion calls (returns). Stack updates must be shadowed by the monitor to properly track which portion of the application memory has been allocated. Therefore, the monitor sets a region of metadata memory to a known value (e.g., *allocated and uninitialized* on a call, *unallocated* on a return).

While instructions dominate the execution profile, in two out of five studied monitors stack updates consume up to 17% of the execution time and represent an attractive acceleration target (Chapter 3).

4.2.5 Unfiltered Event Queue and Consumer

Events that cannot be handled by the filtering accelerator (i.e., unfiltered events) require further processing by the monitoring system. An ideal unfiltered event consumer should be able to support a wide variety of monitoring tools for comprehensive bug coverage. As discussed in Chapter 3, we employ a general-purpose core to process the unfiltered events.

Nearly all unfiltered events arise as a result of (1) memory allocation, deallocation, or initialization; and (2) traversals of tainted data structures or files in taint-tracking monitors. In general, these actions involve multiple memory words and, as a result, trigger a burst of metadata updates that cannot be filtered.

Figure 16(a) plots the distance, as a cumulative distribution, between unfiltered events for MemLeak. Results are similar for other monitors. The distance is measured in events. We observe that two unfiltered events are typically separated by up to 16 filterable events. Based on this analysis, we define an *unfiltered burst* as a sequence of unfiltered events, each of which is separated by at most 16 filterable events. Figure 16(b) shows the average burst size (measured in unfiltered events) for each monitor and benchmark pair. We observe that the bursts are small, with an average

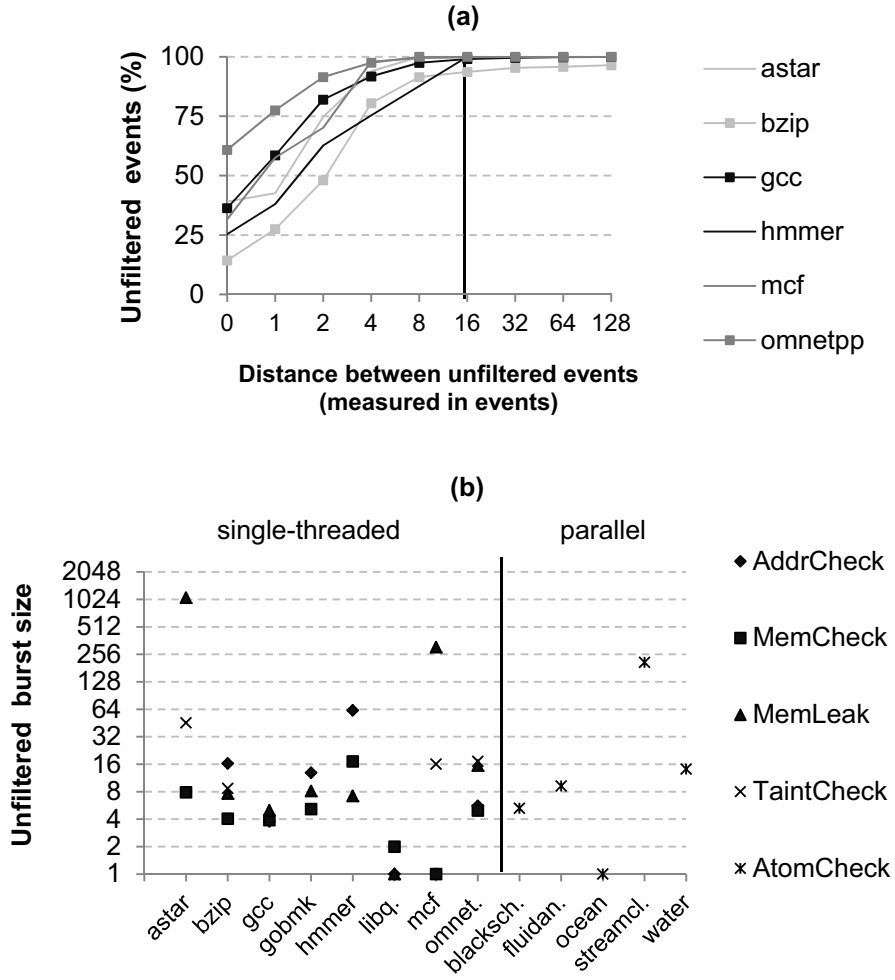

FIGURE 16: (a) Cumulative distribution of distances between unfiltered events for MemLeak. (b) Unfiltered burst size for all monitors and benchmarks.

size of 16 or fewer unfiltered events for the majority of benchmarks and monitors. We thus conclude that *a small (e.g., 16-entry) unfiltered event queue is effective at accommodating the bursts.*

4.3 Summary

In this chapter, we studied the event management in a monitoring system with filtering support. An important implication of our analysis in Section 4.1 is that performing filtering concur-

rently with the processing of unfiltered events is essential to reduce the monitoring slowdown. We revisit this topic in Section 5.2 and we show performance results in Section 5.4.4.

In Section 4.2, our study of a broad range of applications and monitors shows that the monitoring load rarely exceeds one event per cycle even with an aggressive OoO core producing events. Event production is bursty, mandating queueing for pending events; however, a small queue is sufficient for good performance. Unfiltered events are also bursty and are sparsely spaced within an otherwise filterable event stream.

Overall, these results point to a programmable filtering accelerator able to keep up with an average monitoring load of one event per cycle, capable of filtering concurrently with unfiltered event processing, and loosely coupled through shallow queues to both application and monitoring systems.

Chapter 5

FADE

Filtering has been a recurring theme in related work, done concurrently with this work, as a way to reduce the monitoring slowdown. Related work has also identified the potential of filtering and has introduced hardware-based mechanisms to achieve high monitoring performance [35, 96, 107]. However, related work treats filtering as a trade-off between flexibility and performance. Filtering mechanisms that achieve high efficiency and low runtime overhead are focused on a narrow set of monitoring analyses (e.g., only taint flow analysis [107], or only memory safety analysis [35]). Filtering mechanisms that aim at high flexibility do not lower the slowdown uniformly across monitors [23, 43]. Moreover, a number of existing filtering proposals either require intrusive modifications to the core microarchitecture (e.g., a new pipeline stage [107]) or have high resource overheads, needing a dedicated core for the monitoring task [23, 96].

We make the observation that filtering does not have to trade flexibility for performance, and can be effective at accelerating a wide range of monitoring tools. Furthermore, filtering can be independent of the underlying system and monitoring architecture while accommodating different design points in terms of the core microarchitecture and the execution substrate for processing of unfilterable events.

The main contribution of this chapter is in generalizing and extending prior point solutions into a programmable filtering accelerator that can support a broad range of monitoring tasks with high filtering coverage and low hardware overhead. To that extent, in this chapter, we develop an architecture, along with full microarchitectural support, for a flexible at-speed *Filtering Accelera-*

tor for Decoupled Event processing, or *FADE*. FADE is fully programmable and can support a broad range of monitoring tasks with high filtering coverage and low hardware overhead.

FADE's design is guided by our observations from the previous chapters that can be summarized as follows:

- The average monitoring load rarely exceeds one event per cycle, indicating that a single-issue filtering accelerator with a throughput of one event per cycle suffices.
- Instruction and stack-update events dominate the monitoring load. Instruction events require fine-grained accesses to monitor's metadata, most of which can be filtered through (1) hardware-executed checks of metadata state against an invariant, and (2) detection and elimination of redundant updates that leave the metadata state unmodified. Stack-update events perform bulk metadata initialization in response to function calls and returns and can be efficiently handled with a simple state machine.
- Maintaining a high filtering rate requires that filtering takes place concurrently with the processing of unfiltered events.
- Both filterable and unfilterable events arrive in bursts that must be buffered to reduce stalls due to backpressure. Shallow queues of 16 to 32 events are sufficient for this purpose and allow for decoupling of the filtering accelerator from the core running the application.

Based on these observations, we propose a pipelined microarchitecture affording a peak filtering rate of one application event per cycle (when all events are filtered). The latter suffices to keep up with an aggressive OoO core running the monitored application.

To maintain a high filtering rate, filtering has to happen concurrently with the processing of unfiltered events, a task that is complicated due to dependencies between unfilterable and subsequent filterable events. To decouple filtering and the processing of the unfiltered events, we observe that there is only minimal state that is critical for deciding if a dependent event is filterable. We show that this state can be updated for unfilterable events directly in the accelerator with simple hardware extensions. We name our technique *Non-Blocking Filtering* and we described it in detail in Section 5.2. FADE supports Non-Blocking Filtering that dynamically resolves dependencies between unfilterable events and subsequent events, eliminating data-dependent stalls and maximizing accelerator's performance.

Using full-system cycle-accurate simulation, we show that FADE is highly efficient, filtering out 84-99% of events that would otherwise be handled in software, thereby reducing the application slowdown to only 1.2-1.8x (versus 1.6-7.4x for unaccelerated execution). In the 40nm technology, FADE requires 0.12mm^2 of area and 273mW of power at peak.

In the rest of this Chapter, we first present the baseline FADE design that stops or blocks upon an unfiltered event (Section 5.1), and then extend it to support Non-Blocking Filtering (Section 5.2) that allows for the overlapped execution of the filtering process and the handling of unfiltered events in SW.

5.1 Baseline Filtering Accelerator

The baseline Filtering Accelerator is composed of two building blocks: (1) the *Filtering Unit*, which filters instruction events (Section 5.1.1), and (2) the *Stack-Update Unit*, which accelerates stack-update events (Section 5.1.2). Without loss of generality, we assume that unfiltered events are processed in software on a general-purpose core.

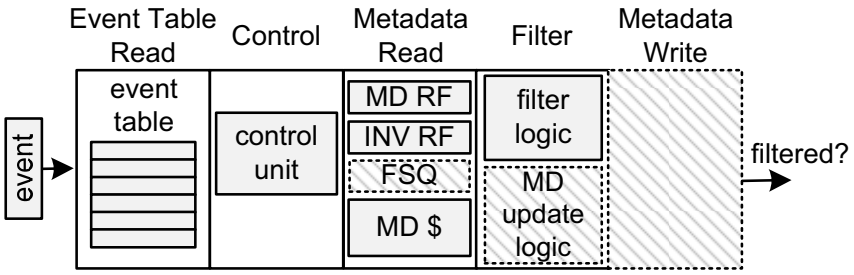

FIGURE 17: Filtering Unit pipeline. Striped structures show pipeline extensions for Non-Blocking Filtering.

5.1.1 Filtering Unit

To elide software execution, the Filtering Unit supports two filtering actions, clean checks *(CC)* and redundant updates *(RU)*. Clean checks are based on the observation that most of the time applications behave as expected and the metadata match the expected invariant (e.g., memory references are to initialized memory). Redundant updates are based on the observation that metadata are stable as propagation handlers commonly update the metadata with the same value (e.g., initialized memory remains initialized even when the actual value in application memory changes). We discussed clean checks and redundant updates in detail in Chapter 3.

The Filtering Unit handles an instruction event either as a clean check or as a redundant update. To maximize flexibility and applicability, the Filtering Unit implements three modes of operation: (1) *Single-shot filtering* either performs a clean check or identifies a redundant update, (2) *Multi-shot filtering* chains multiple single checks together to determine whether an event is filterable, (3) *Partial filtering* filters a part of the software handler functionality in hardware, thus reducing the handler's length.

FADE's hardware is fully programmable and allows for per-event definition of the filtering rules. In FADE, programmability is achieved by configuring two structures: (1) the *event table*, which includes per-event filtering rules, and (2) the Invariant Register File (*INV RF*), which keeps

eventID / next entry	valid s1	valid s2	valid d	mem s1	mem s2	mem d	MD bytes s1	MD bytes s2	MD bytes d	mask s1	mask s2	mask d	CC	INV id s1	INV id s2	INV id d	RU	Non-Block. INV id	Non-Block. Op	MS	next entry	P	handler PC
ld mem , rd	1		1	1			1		1	0xff		0xff	1	2		2							PC_1
ld mem , rd	1		1	1			1		1	0xff		0xff					1						PC_3
ld mem, rd	1			1			1			0x80			1	2						1	A		
A	1			1			1			0x7f			1	3									PC_2

FIGURE 18: Event table entries. The size of an event table entry is 96 bits.

invariant values related to the monitoring task (e.g., *unallocated, allocated,* and *initialized* states for MemCheck). These structures are memory-mapped and programmed on a per-application basis.

Figure 17 shows the baseline filtering pipeline, which consists of four stages. Note that striped structures, including the *Metadata Write* stage, are only for Non-Blocking Filtering as discussed in Section 5.2. The pipeline works as follows. First, the filtering rules are read from the event table. Next, the *control unit* uses the event information and the filtering rules to produce the control signals for subsequent stages, in *Control* stage. Then, the Filtering Unit accesses the metadata register file (*MD RF*) and a dedicated metadata cache (*MD cache*) to obtain metadata. The Filtering Unit may also access the INV RF to obtain monitor-specific invariants, if necessary. Finally, in the *Filter* stage, the *filter logic* checks whether the filtering condition is satisfied.

field	bits
event ID	6
src1 reg	5
src2 reg	5
dest reg	5
app addr	32
app PC	32

FIGURE 19: Event entry format.

Stage 1: Event Table Read. The filtering accelerator dequeues an event (Figure 19) from the event queue and accesses the event table with the event ID to obtain the event's filtering rules. An event table entry (Figure 18) includes the following information for each operand (i.e., *s1*, *s2* and *d*): (1) the *valid* bit and the *mem* bit to denote the evaluated operands and the memory operands, respectively; (2) the number of *MD bytes* to be evaluated; (3) a *mask* to extract

the appropriate bits. Each entry also includes the *PC* of the software *handler* to be invoked for unfiltered events.

Each entry includes the *CC* bit and the *INV id* for clean checks, and the *RU* field for redundant updates. The *INV id* indicates the invariant registers (one for each operand) to be used upon a clean check. The *RU* field encodes three options. In case of one source operand, the source metadata are directly compared to the destination metadata. In case of two source operands, the source metadata are composed using either OR or AND and then compared to the destination metadata. The rest of the fields are described later.

Stage 2: Control. The control unit processes the information obtained from the event table and uses combinational logic to generate control signals for subsequent stages (e.g., filter logic mux controls, selects and enables for MD RF).

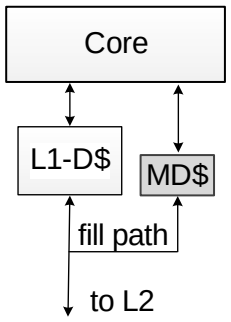

FIGURE 20: The MD cache in the cache hierarchy.

Stage 3: Metadata Read. The Filtering Unit accesses the MD RF, the INV RF and the MD cache (shown in Figure 20) to obtain metadata and invariants values.

Metadata accesses (for memory operands) necessitate a translation step from the application to the monitor address space before accessing the MD cache. The memory operands (due to loads, stores) include virtual application addresses, because the event stream is generated at the application side, while the memory metadata are allocated in the monitor's virtual address space (a feature that enhances system security and reliability). In FADE, we fold the address translation into the MD cache access. The TLB of the MD cache, similar to the M-TLB [23], contains the translation from a virtual application page to the physical page that contains the associated memory metadata. The TLB is software managed by the user-space monitor (same as the M-TLB [23]). To enable the software handlers to leverage the metadata cache, we extend the

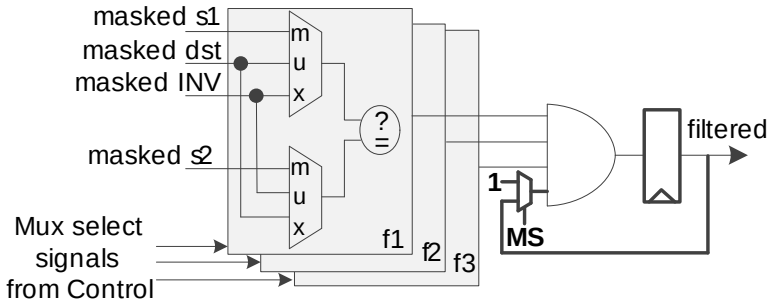

FIGURE 21: The internals of filter logic.

ISA with new Load Metadata and Store Metadata instructions. Conventional loads and stores access the L1-D unaffected.

Stage 4: Filter. The Filtering Unit supports three modes of operation to filter events.

Single-shot Filtering. In a single cycle, the Filtering Unit compares up to three distinct operand metadata to an invariant (clean check), or compares the operand metadata to each other (redundant update).

Examples of single-shot filtering are shown in the first two entries of Figure 18. The first event table entry corresponds to a load instruction for MemLeak. FADE handles the event as a clean check (CC=1) and filters the event when both operands are not pointers. In doing so, the metadata of the event operands (i.e., the memory operand s1 and the register operand d) are compared to the *non-pointer* invariant, which is stored in the third entry of the INV RF (INV id=2). The evaluated metadata are one byte (MD bytes=1). The second event table entry corresponds to a load instruction that is handled as a redundant update.

Figure 21 details the *filter logic*, which is organized as three identical two-operand comparison blocks (labeled f1, f2, and f3 in the figure). Each block can compare any one of three event operands (i.e., s1, s2, and d) to another operand or to an invariant. Together, the three blocks allow

for a single-cycle evaluation of the most complex single-shot condition (i.e., comparing each of the three operands – s1, s2, and d – to a different invariant).

Multi-shot Filtering. To accommodate complex monitors that require multiple checks to determine whether an event is filterable, FADE supports multi-shot filtering. The Filtering Unit processes multi-shot events in multiple cycles by performing one check per cycle, and maintains one entry in the event table per check, thus keeping each entry simple. To encode multi-check events, each event table entry requires two additional fields (shown in Figure 18): (1) the *next entry* field, which contains a pointer to the next entry in the event table; and (2) the *multi-shot bit* (MS), which enables multiple checks to be chained by allowing the previous filtering outcome to be considered in the final filtering outcome. As shown in Figure 21, the associated circuit (in bold) includes a clocked register and a multiplexer, controlled by the MS bit.

Partial Filtering. Partial filtering affords a part of the handler functionality to be executed in hardware, reducing the length of the software handler. A software handler may first perform a check and based on the check's outcome, executes either an update or a more complex routine including multiple checks and updates. FADE accelerates such cases by performing the initial check in hardware. To support partial filtering, each event table entry includes a *partial* bit *(P)* (shown in Figure 18), which drives the selection of the handler PC.

An example of partial filtering appears in AtomCheck, where the filter logic checks whether a shared memory location was last referenced by the same thread. Commonly, the check succeeds, and a simple software handler is dispatched to update metadata. Otherwise, a complex handler runs to check whether there is a potential atomicity violation. While both cases require software execution, the hardware check eliminates the code associated with the check itself, control flow, and register spills and fills.

5.1.2 Stack-Update Unit

Stack-update events, which set consecutive metadata addresses to a predefined value in response to allocation/deallocation of an application stack frame, are handled in FADE via a dedicated Stack-Update Unit (SUU). The SUU implements a finite state machine that takes the stack frame's starting address and length as parameters to calculate the address(es) of the metadata block(s) covered by the stack frame. The SUU issues writes to the MD cache to set the target range of addresses to one of two predefined values (one value on function calls and another on function returns), which are stored in the INV RF.

The SUU can use either the block-wide or sub-block interface of the metadata cache to minimize the number of write operations; this interface is similar to that in contemporary processors that support write combining to reduce write activity in L1-D. The number of metadata cache block writes is dictated by the length of the stack frame, encoded in the event descriptor.

Overall, FADE handles stack-update events in hardware by (1) configuring the filter table to recognize them, (2) forwarding them directly to the stack update unit, and (3) executing them using the range update FSM, thus completely eliding software intervention.

5.2 Non-Blocking FADE

5.2.1 Observations

Due to true dependencies between monitored instructions, baseline FADE must stall filtering when an unfiltered event is encountered. Filtering resumes when the monitoring system completes the unfiltered event processing and the updated metadata become available. This organization penalizes performance because filtering and execution of unfiltered event handlers cannot overlap.

To overcome the serial processing of unfiltered events and subsequent dependent events, we make a critical observation: while monitors often maintain detailed metadata to support complex monitoring analyses, there is a subset of metadata, which we call *critical*, that includes sufficient information to decide whether a subsequent dependent event is filterable. Importantly, this critical state can be updated for unfilterable events directly in hardware in the Filtering Unit. These updates are non-speculative and are based on predefined rules that can be implemented in simple hardware.

For instance, for MemLeak, which performs reference counting to identify memory leaks, an event is filterable when its operands are not pointers. Therefore, just checking the *pointer/non-pointer* status of a memory location or a register suffices to make the filtering decision. For example, in case of a load instruction, if the source memory location has a pointer status, the destination register obtains a pointer status as well. However, to perform reference counting, MemLeak maintains additional metadata per register and memory location, which consist of a pointer to the context (explained in Section 2.3) of the corresponding malloc. While fundamental to MemLeak's monitoring algorithm, these additional metadata are non-critical from the perspective of the filtering task. A detailed description of critical/non-critical metadata for the studied monitor is provided in Section 5.3.

Overall, we observe that (1) there is critical (minimal) state that can be checked to determine the filtering outcome in a non-speculative way, and (2) this state can be updated in simple hardware based on simple pre-defined rules. Based on these observations, the filtering decision and the handling of unfiltered event can be decoupled, thus enabling the design of a *Non-Blocking* filtering unit that can continue filtering past an unfiltered event.

5.2.2 Extensions to the Baseline Pipeline

Figure 17 shows the pipeline extensions (striped) to support Non-Blocking Filtering. We introduce two new structures; the *metadata (MD) update logic*, which performs updates to the filtering-critical metadata for unfilterable events, and the *filter store queue* (*FSQ*), which stores the updated memory metadata. We also introduce a new pipeline stage, *Metadata Write*, where updates to metadata take place.

Processing of instruction events. Consider an unfilterable event that just enters the pipeline. The processing in the first three stages (Event Table Read, Control, and Metadata Read) is the same as in the baseline pipeline. In the Filter Stage, while the filtering condition is evaluated, the MD update logic computes the new value for the filtering-critical metadata. The new metadata value is subsequently used only if the filtering condition evaluates to false, indicating an unfilterable event. Otherwise, the new metadata value is discarded.

To determine the logic for critical metadata updates, we observed that critical metadata have minimal state and their propagation follows simple rules. Based on the studied monitors, we provide support for the following rules: (1) propagating the source metadata (s1 or s2) to the destination; (2) composing the new destination metadata from the two source metadata using OR or AND; (3) setting the destination metadata to a constant value, which is stored in an INV register denoted by the *Non-Blocking/INV id* field in the event table (see Figure 18); and (4) conditionally performing one of the above actions after comparing the source operands to each other, to the destination, or to a constant.

In the *Metadata Write* stage, the Filter Unit commits updated metadata to the MD RF (for register) or to the FSQ (for memory). Subsequent events with a true dependence on the updated metadata can then obtain them from the MD RF or the FSQ in Metadata Read stage. For memory metadata, the FSQ is searched in parallel with the MD cache. If a matching FSQ entry is found, it

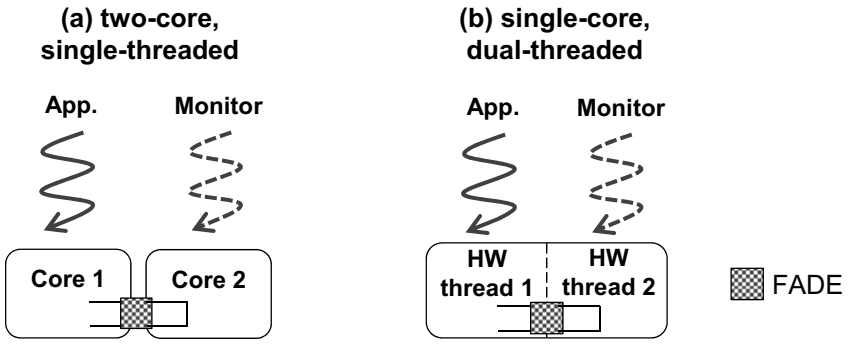

FIGURE 22: Evaluated systems.

is used to satisfy the dependence; otherwise the metadata from the cache are used. To accommodate back-to-back dependencies, forwarding from the Metadata Write stage to the Filter stage is supported.

Eventually, the unfiltered event handler executes and updates both the critical and the non-critical metadata for registers and memory. Once the handler completes, the MD cache contains the updated value for the critical memory metadata (if any) and the corresponding FSQ entry is discarded. Subsequent accesses to these metadata are served by the MD cache.

Processing of stack-update events. As stack updates change the metadata state, filtering must stop upon a stack-update event to allow the SUU to set the stack frame metadata. Moreover, as pending unfiltered events may reference stack frame-related metadata, the unfiltered event queue must be drained by the consumer prior to stack-update processing.

5.3 Methodology

Evaluated designs. We evaluate two FADE-enabled systems, shown in Figure 22. The *two-core monitoring system* (Figure 22(a)) executes the application and monitor threads on separate cores to maximize concurrency [23]. Filtering takes place next to the monitor core. The *single-core monitoring system* (Figure 22(b)) is based on a fine-grained, dual-threaded core with a dedi-

Table 4: Simulation setup.

Parameter	Value
Core type	in-order, 1-way
	lean OoO, 2-way/48-entry ROB
	aggr. OoO, 4-way/96-entry ROB
ISA	SPARC v9 [84]
L1 caches	32KB, 2-way, 64B block, 2-cycle latency
Shared L2	2MB, 16-way, 64B block, 10-cycle latency
DRAM	90-cycle latency

cated hardware thread for the application and monitor processes. This design point minimizes resource requirements, but results in higher slowdown because the core resources are shared between the application and monitor.

We also evaluate two unaccelerated systems, similar to the single- and two-core systems presented in Figure 22 but without FADE. In these systems, the application and the monitor communicate through a single queue.

System configuration. Table 4 summarizes the configuration of the evaluated systems. Additionally, FADE-enabled systems have a 4KB, two-way MD cache with one-cycle access latency, and a 16-entry Metadata TLB. The event table has 128 entries, covering the heavily used subset of the modeled ISA. The event queue and the unfiltered event queue is 32 and 16 entries, respectively. Unless, otherwise specified, experiments use Non-Blocking FADE.

Simulation. We use Flexus [114] for cycle-accurate full-system simulation. Flexus extends Simics with timing models of multithreaded cores, caches, and interconnect. For our evaluation, we use the SMARTS sampling methodology [117]. Our samples are drawn over one billion instructions of the monitored application. As our benchmarks are organized as a collection of loops, we sample over an execution interval that covers multiple iterations. For the parallel benchmarks, we follow the same approach to cover a representative part of the benchmark's parallel section. For each measurement, we launch simulations from checkpoints with warmed caches

(including the MD cache), and run 100K cycles to achieve a steady state of detailed cycle-accurate simulation before collecting measurements for the next 50K cycles. For the two-core system experiments with a 64K-entry event queue (in Chapter 3), we run for 1 million cycles to ensure that the event queue has the correct state and we take measurements in the subsequent 500K cycles.

Power and Area. To estimate FADE's area and power, we synthesize our VHDL implementation with Synopsys Design Compiler. We use TSMC 45nm technology (core library: TCBN45GSBWP, V_{dd}: 0.9V) scaled down to 40nm half node, and target a 2GHz clock frequency. For the MD cache, we estimate area, power, and latency with Cacti 6.5 [73].

Monitors. We revisit the five studied monitors, introduced in Section 2.3, so as to discuss the critical and non-critical metadata on a per-monitor basis.

AddrCheck [81] checks whether memory accesses are to an allocated region. The critical metadata encode two states *(allocated or unallocated)* per memory location, while the non-critical metadata include book-keeping information for bug reporting.

MemCheck [81] extends AddrCheck to detect the use of uninitialized values, and *Taint-Check* [82] detects overwrite-related security exploits. For critical metadata, MemCheck has three metadata states (i.e., *unallocated*, *uninitialized,* and *initialized*) and TaintCheck has two metadata states (i.e., *untainted* and *tainted*). Non-critical metadata may include information related to origin tracking [11] or other bookkeeping information.

MemLeak [71] identifies memory leaks through reference counting. The critical metadata consist of the *pointer/non-pointer* status of each register and memory word. Non-critical metadata consist of a pointer to the corresponding malloc's context and a null value otherwise. The context includes a unique ID, PC, and a reference counter.

AtomCheck [65] detects atomicity violations by checking access interleavings. For this purpose, it keeps track of the last access by each thread to each application memory location. AtomCheck maintains one byte of critical metadata per application word with the *thread status bit* and the *thread id*. Furthermore, it keeps non-critical metadata including the type (Read/Write) of the last access by each thread in local per-thread tables. AtomCheck is accommodated by Partial filtering, as explained in Section 5.1.1.

Benchmarks. For all monitors, except AtomCheck, we use the SPEC2006 integer benchmarks with reference inputs. These CPU-intensive benchmarks stress the monitoring system with a high event generation rate. For TaintCheck, we use the benchmarks (astar, bzip, mcf, omnetpp) that have tainting propagation initiated by file reads and we exclude the rest. For AtomCheck, we use five multithreaded benchmarks: water and ocean from the SPLASH suite [116]; and blackscholes, streamcluster, and fluidanimate from the PARSEC suite [10]. Each benchmark has four threads that run on one core in a time-sliced manner. All benchmarks use 32-bit binaries.

5.4 Evaluation

5.4.1 FADE versus Unaccelerated System

Figure 23 depicts the performance of FADE versus the unaccelerated monitoring system. In both systems, application and monitor tasks execute in dedicated hardware threads of a dual-threaded 4-way OoO core. Performance is normalized to an unmonitored (application-only) system.

In general, for the unaccelerated systems, we observe an average slowdown of 4.1x, across monitors. For memory-tracking monitors (AddrCheck, AtomCheck), the average slowdown is 2.5x, while for propagation-tracking monitors (MemCheck, MemLeak, TaintCheck), the slow-

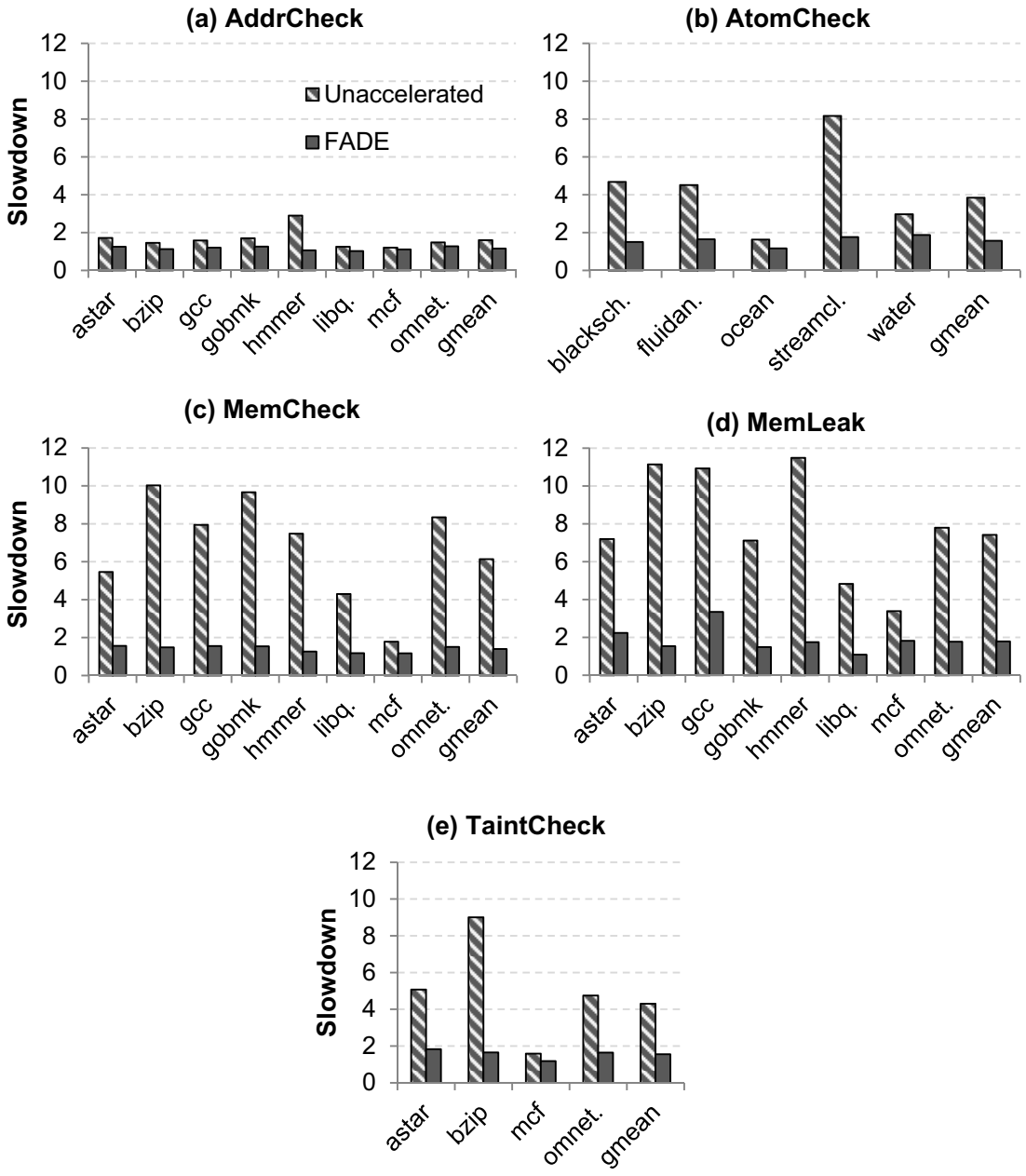

FIGURE 23: Performance of FADE compared to the unaccelerated system.

down is 5.8x. FADE reduces the slowdown significantly for all monitors, with an average slowdown of 1.5x. FADE's slowdown is 1.3x and 1.6x for memory- and propagation-tracking monitors, respectively.

Figure 23(a) shows AddrCheck's performance, which is generally good on both systems as the monitor just processes non-stack memory instructions. In the unaccelerated system, AddrCheck's slowdown ranges from 1.2x to 2.9x, with an average of 1.6x. FADE reduces the slowdown to an average of 1.2x by filtering out nearly all monitored events.

Figure 23(b) presents results for AtomCheck. Although AtomCheck is a memory-tracking monitor with a low event generation rate, it has an average slowdown of 3.9x (8.2x max) in the unaccelerated system because the events are costly due to numerous monitoring actions. In contrast, FADE benefits from a high filtering ratio, resulting in an average slowdown of 1.6x (1.9x max).

Figure 23(d) shows the results for MemLeak, a heavy-weight propagation-tracking monitor. In the unaccelerated system, we observe slowdown ranging from 3.4 to 11.5x, with an average of 7.4x. We note that the benchmarks with a high *monitored* IPC (e.g., 1.2 for bzip) generate events faster than those with a low *monitored* IPC (e.g., 0.2 for mcf), resulting in higher slowdown due to the increased pressure on the monitor. FADE significantly reduces the slowdown to an average of 1.8x, thanks to its high filtering ratio and the hardware-accelerated stack-update unit. The highest slowdown is observed on astar (2.2x) and gcc (3.3x), which are characterized by a low filtering ratio (70%) and must frequently drain the unfiltered event queue at function call/return boundaries (Section 5.2.2).

Finally, FADE reduces the slowdown to an average of 1.4x for MemCheck, Figure 23(c), (similar to MemLeak) and 1.6x for TaintCheck, Figure 23(e), (similar to AtomCheck). Across the five evaluated monitors, FADE reduces the monitoring slowdown to an average of 1.5x, versus 4.1x for the unaccelerated system.

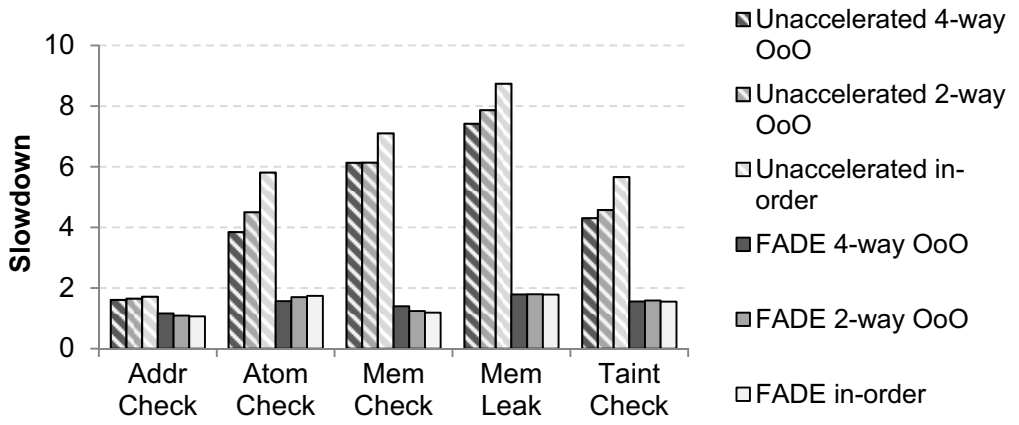

FIGURE 24: Performance of the single-core monitoring system for different core types.

5.4.2 Performance for Different Core Types

To better understand the effects of core microarchitecture on monitoring performance, we evaluate the unaccelerated and FADE-enabled systems with different core types. Figure 24 summarizes the performance for three core microarchitectures; in-order, 2-way OoO, and 4-way OoO averaged across all benchmarks.

For the unaccelerated monitoring systems (dashed bars), we observe a reduction in performance ranging from 7% to 51% for simpler core microarchitectures as compared to the 4-way design. Although the applications generate up to 2x fewer events per cycle on the in-order core than on the 4-way OoO core**,** each event handler executes up to 3x faster on 4-way OoO because event handlers consist of instruction sequences with high cache locality, resulting in high IPC on aggressive cores. Thus, we conclude that monitors are sensitive to the core micro-architecture.

In the FADE-enabled system (solid bars), performance is less dependent on the core type. For example, MemCheck performs marginally better on the simple microarchitecture (average

slowdown of 1.2x on in-order versus 1.4x on 4-way OoO), showing that filtering leaves little work for the monitor core and the core microarchitecture is less important.

5.4.3 Single-Core versus Two-Core System

Prior work [23, 112] suggested utilizing otherwise idle cores to accelerate the monitoring task. However, our analysis in Chapter 3 shows that when the majority of the events can be filtered, the second core is most of the time underutilized. In this Section, we evaluate the performance of the single- and two-core monitoring systems, so as to show that the performance benefits of the second core are limited, as expected, and that a multithreaded core provides sufficient resources for both the application and the monitor.

Figure 25(a) compares the performance of single-core (dual-threaded) and two-core monitoring systems. Both are FADE-enabled and feature a 4-way OoO microarchitecture. Indeed, the results indicate that the two-core design outperforms the single-core option by only 15% on average (28% max) by eliminating resource contention between monitor and application threads.

For clarity, we show the break-down of the two-core system utilization in Figure 25(b) (similar to Figure 9 from Chapter 3). The execution time is broken down into three categories: cycles in which (1) the application core is idle because the event queue is full, (2) the monitor core is idle because FADE filters all events, and (3) both application and monitor cores are utilized. As the figure shows, 48% to 97% of the time, one of the two cores is idle, as either FADE filters the incoming event stream (idling the monitor core), or the monitor core processes unfiltered events (backpressuring the application core). With both cores utilized only 22% of the time, on average, the benefit of the second core is clearly limited.

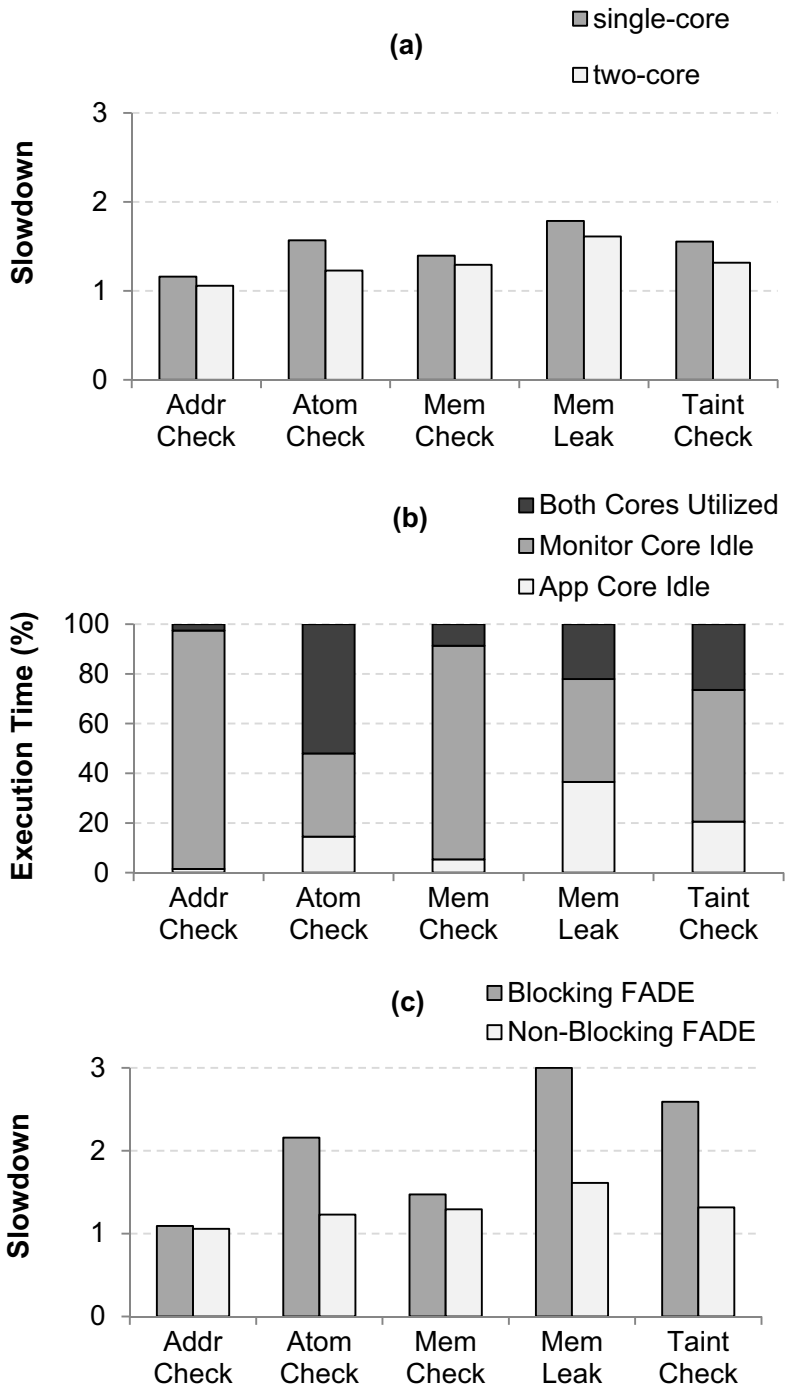

FIGURE 25: (a) Performance of the single- versus two-core monitoring systems with FADE. (b) Core utilization in the two-core system. (c) Performance benefits of Non-Blocking Filtering.

5.4.4 Benefits of Non-Blocking Filtering

To show the benefits provided by Non-Blocking Filtering, Figure 25(c) compares the performance of Non-Blocking FADE (used in the studies above) to the baseline FADE that stalls on each unfiltered event.

We observe that Non-Blocking Filtering improves the performance by 2x for AtomCheck, MemLeak and TaintCheck, which have relatively low filtering ratios (<87%), and by 1.1x for AddrCheck and MemCheck, whose filtering ratio is high (>98%). The benefit of Non-Blocking FADE comes from overlapping the filtering actions with the unfiltered events processing.

5.4.5 Area and Energy Efficiency

To model FADE's area and power costs, we synthesized our RTL design in TSMC 40nm technology, targeting a clock frequency of 2GHz. Our design includes a 128-entry event table, a 32-entry event queue, and a 16-entry unfiltered event queue. Synthesis results show a peak power consumption of 122mW and an area of 0.09mm^2 with 20% of the area dedicated to logic and the rest to memory and latches. *Filter Store Queue* (*FSQ*) is in the critical path of our design. To estimate the area and power requirements of the 4KB MD cache, we use CACTI. We find the area cost of the cache to be 0.03mm^2, peak power of 151mW, and an access latency of 0.3ns.

5.5 Summary

This chapter introduced FADE, a Filtering Accelerator for Decoupled Event monitoring. The proposed design exploits common behavior across monitors to provide simple, programmable hardware for handling common application events while delegating infrequent complex events to software for maximum flexibility. To maximize throughput and avoid stalls in the presence of unfiltered events, FADE employs Non-Blocking Filtering — a hardware-assisted mechanism for

concurrent processing of filterable and unfiltered events. Our results showed that FADE can reduce the slowdown to an average of only 1.2-1.8x, thereby making instruction-grain monitoring practical.

Chapter 6

Parallel FADE

Efficient instruction-grain monitoring of parallel applications is a necessity for the development of correct parallel code, which corresponds to a large fraction of today's software. However, solutions developed for single-threaded applications are not directly applicable to parallel applications, because the inherent concurrency of the latter introduces additional complexity. In addition to the tasks performed by sequential monitoring frameworks, parallel monitoring frameworks have to faithfully replay the inter-thread application order, so as to maintain the correct view of the application execution.

Our goal is to provide flexible parallel monitoring at low runtime and resource overhead by parallelizing our FADE design, which has significant advantages over prior monitoring accelerators (Chapter 5). However, related work [112] in this direction showed that parallelizing hardware monitoring accelerators is a difficult task because (1) the initial sequential accelerators require extensive modifications, and (2) the accelerators integration to the rest of the system is complicated. Unlike prior work, we show that the adoption of FADE to parallel monitoring is straightforward, thus simplifying the design of the proposed monitoring system.

We propose *Parallel FADE* a CMP-based monitoring system where each core is equipped with a parallelized instance of FADE. Parallel FADE is an end-to-end monitoring system, unique in offering all of the following features; Parallel FADE: (1) supports parallel applications running on a CMP, (2) targets a wide range of monitoring tasks, (3) supports Non-Blocking Filtering, (4) reduces the resources dedicated to the monitoring task from a core to just a hardware thread per

monitored application thread (effectively doubling chip's throughput), and (5) supports aggressive OoO cores that stress the accelerator with a high monitoring load.

Parallel FADE combines the techniques proposed in this thesis to reduce the resource- and runtime-overhead of instruction-grain monitoring, along with mechanisms proposed in prior work [112], and enhances them as necessary to ensure correctness. To that extent Parallel FADE relies on (1) application coherence activity [56, 112, 118, 120] to infer inter-thread dependences, and (2) synchronization-free fast paths [112] to access shared metadata at low runtime overhead.

In this chapter, we make the following contributions:

- We propose a parallel monitoring accelerator that not only has significant performance benefits over prior accelerators, but also allows for the design of less complex parallel monitoring systems.
- We provide a formal proof showing the applicability of FADE with Non-Blocking Filtering support in the context of parallel monitoring.
- We propose a storage-efficient structure for keeping the dependences (i.e., metadata indicating the happens-before relationship between application events) in the accelerator.
- Using a suite of diverse monitors and a number of multi-threaded benchmarks, we show that Parallel FADE filters 81-99% of events that would otherwise be handled in software and reduces the slowdown to an average of only 1.1-1.8x (versus 1.9-11.5x for unaccelerated execution), thus making parallel monitoring practical.

6.1 Background: Event Order

Instruction-grain monitoring checks application instructions in *program order*, and ensures that a program invariant holds (e.g., the application does not access unallocated memory). For single-threaded applications, the program order is the same as the *commit order* of the application

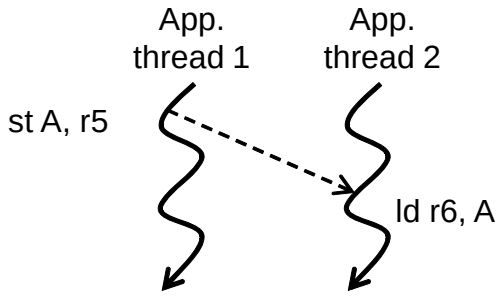

FIGURE 26: Event order under parallel monitoring. To maintain correctness, the monitoring process has to follow the commit and dependence order of the application instructions.

instructions. However, the commit order alone is not sufficient for shared memory multi-threaded applications, because the parallel monitor has not only to process the events of each application thread in commit order, but also to follow the happens-before relationship (or *dependence order*) of instructions from different threads. Dependences between application instructions from different threads are created, as the application threads communicate by reading and writing values to a common shared address space.

Using TaintCheck as an example, Figure 26 shows a dependence (or a happens-before relationship) between two threads due to a load and a store to the same address. In this example, the store instruction writes a tainted value to memory location A at time T, and subsequently, the load instruction reads this tainted value at local time T'. To maintain the correct application view, the monitor has to process the load instruction *after* the store instruction. Otherwise, the monitor will observe an untainted value, leading to a wrong monitoring outcome.

6.2 Baseline Parallel Monitoring System

In this section, we present a formal definition of our baseline parallel monitoring system. Our baseline is similar to the unaccelerated system presented in ParaLog [112].

Definition 1 (baseline system). The application and the monitor are multithreaded processes running on a CMP under Sequential Consistency (we discuss accesses to shared metadata in Section 6.5). Each monitoring thread processes the event stream generated by an application thread. Together, these two threads form a monitoring pair and communicate through an event queue. In Figure 27, we present our baseline system focusing on one of the system's tiles. The monitoring system is equipped with a dependence recorder (Definition 5), a dependence checker (Definition 6) and a progress publisher (Definition 7) per monitoring pair.

Definition 2 (commit order). The commit order is the per-thread commit order of the application dynamic events.

Definition 3 (dependence order). The dependence order is the happens-before relationship of concurrent application events from different threads.

Definition 4 (correctness). To maintain the correct view of the application state, the monitor has to maintain the event program order while processing application events – i.e., the monitor has to process the application events in an order that reflects the commit order (Definition 2) and the dependence order (Definition 3).

Definition 5 (dependence recorder). For each pair, the dependence recorder observes the coherence activity of the application thread so as to infer the event dependence order (Definition 3), and records the dependences in the dependence queue. The exact mechanisms are described in Section 6.4.

Definition 6 (dependence checker). For each pair, the dependence checker processes the event stream (stored in the event queue) and the dependences (stored in the dependence queue) in commit order. The dependence checker ensures that the dependence order is maintained — i.e., if there is a dependence from event i (thread t) to event i' (thread t'), this component ensures that the event i' is delivered to the monitor thread t', only after the monitor thread t has completed the processing of

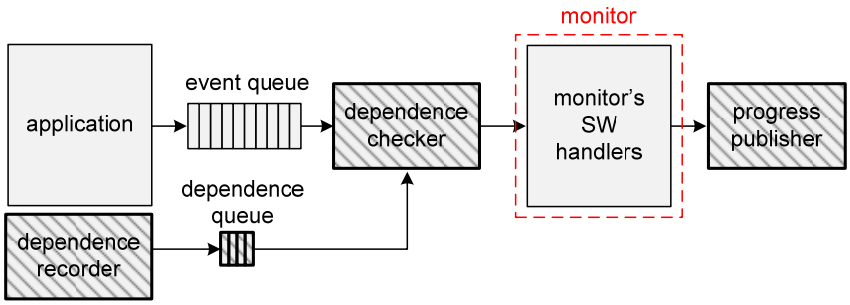

FIGURE 27: The baseline monitoring system. The striped structures allow the monitoring process to follow the application dependence order.

the event i. Once the event dependences (if any) have been satisfied, the event is delivered to the monitor.

Definition 7 (progress publisher). For each pair, the progress publisher keeps track of the events processed by the monitor, and commits the completion time of the last completed event to the system's global state. After this point, there are no further updates associated with this event.

Lemma 1. The baseline parallel monitoring system (Definition 1) can faithfully replay the application event order, thus guaranteeing correctness (Definition 4).

Proof: The baseline system processes the events of each thread in their commit order, thus satisfying the commit order requirement (Definition 2). Regarding dependence order, the monitor has to process the event A of thread t, before the event B of thread t', if A happened before B in the application space. As under sequential consistency, the associated memory references cannot be reordered, the coherence activity should indicate that A happened before B. Otherwise, either the memory consistency or the coherence implementation is faulty. Thus, the baseline system guarantees the dependence order requirement (Definition 3), overall ensuring correctness (Definition 4).

6.3 Accelerating Parallel Monitoring

In this section, we present Parallel FADE, a parallel monitoring system with FADE extensions. Parallel FADE supports Non-Blocking Filtering, a novel filtering technique that reduces the overall monitoring slowdown by up to 2x (Chapter 5). Non-Blocking Filtering decouples the filtering process and the processing of unfiltered events in SW, thus allowing for their overlapped execution in time.

As the inherent concurrency of parallel monitoring introduces additional complexity, in this section, we provide a formal proof that shows the applicability of Non-Blocking Filtering to Parallel FADE. In doing so, we show that monitors execution in Parallel FADE is equivalent to the monitors execution in the baseline system (Section 6.2). Then, we discuss prior work showing that Parallel FADE allows for the design of less complex monitoring systems.

6.3.1 Parallel FADE

Definition 8 (Parallel FADE). Parallel FADE extends the baseline parallel monitoring system (Definition 1) with FADE including Non-Blocking Filtering support. Figure 28 presents Parallel FADE focusing on one of the system tiles. The figure shows the original FADE system (Figure 11) enhanced with the extensions (striped) to support parallel monitoring.

Definition 9 (progress publisher++). As Parallel FADE supports Non-Blocking Filtering, event processing may complete out of order. The progress publisher (Definition 7) is extended to take into account the out-of-order event completion. Therefore, the completion of an event is committed to the system's global state iff there is no older event being processed in the monitoring pair.

Definition 10 (Metadata manipulation under Non-Blocking Filtering). Parallel FADE operates on

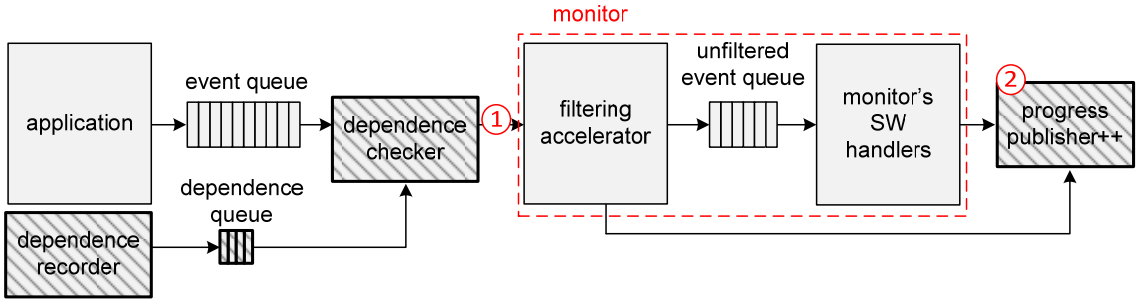

FIGURE 28: Numbered the two requirements to ensure correctness of the monitors execution with Parallel FADE (including Non-Blocking Filtering support): (1) The events are delivered to the filtering accelerator in commit and dependence order. (2) The event progress is advertised in commit order.

critical metadata (as defined in Section 5.2). Parallel FADE reads and writes register metadata from/to its local structures. Parallel FADE loads memory metadata from the system's global state. Parallel FADE writes the updated memory metadata in a local queue.

Corollary 1: Parallel FADE with Non-Blocking Filtering support does not update system's global state.

Definition 11 (Metadata manipulation in SW) The system's global state is updated iff a SW handler for an unfiltered event is executed.

Lemma 2: Given that (1) the events are issued to the filtering accelerator in commit and dependence order, and (2) the system is equipped with progress publisher++, monitors execution in Parallel FADE is equivalent to monitors execution in the baseline system.

Proof: When the events are processed out of order in Parallel FADE due to Non-Blocking Filtering, the progress publisher++ ensures that their completion is advertised in order, the same as in the baseline system.

As the filtering accelerator is placed right after the dependence checker, the events are guaranteed to be delivered to the accelerator in commit and dependence order. Therefore, event order is

followed as in the baseline system.

Once program order has been satisfied, the filtering accelerator processes the event stream and creates ordered (according to Definition 2 and 3) batches of unfiltered events, without modifying the global state (Corollary 1). Then, the unfiltered events are processed in monitor's SW, updating the system's global state (Definition 11) in commit and dependence order, as in the baseline system. Therefore, monitors execution in Parallel FADE is equivalent to the baseline system. ■

6.3.2 Comparison to Prior Work

ParaLog [112] is the most closely related work, because it studies the parallelization of sequential hardware monitoring accelerators. These sequential accelerators (i.e., Unary Inheritance Tracking, Idempotent Filter and Metadata-TLB) were initially introduced in Log-Based Architectures (LBA) [23]. We focus this discussion on Unary Inheritance Tracking (Unary IT), a representative accelerator from the LBA paper.

Unary IT maintains a metadata register file that associates each architectural register with the address from which it inherits its value. An FSM evaluates an incoming event and either updates the inherits-from metadata in the register file of Unary IT, or delegates the event to the monitor's software. As Unary IT keeps *addresses instead of values*, the consumption of the actual metadata value can be delayed compared to the time that the accelerator processed the event. Thus, when an instruction event carries a dependence, the dependence is effectively propagated to subsequent instruction events that inherit from this event.

To maximize the benefits of Unary IT, ParaLog has to advertise the completion of an event, only after the processing of all events that inherit from this event has been completed. As this policy may deadlock the system, ParaLog needs to include mechanisms for deadlock detection and avoidance. Clearly, dependence propagation greatly complicates the design of this system. In con-

trast, the parallelization of FADE (discussed in the previous Section) does not require any of these mechanisms, thus allowing for the design of a much simpler parallel monitoring system.

6.4 Parallel FADE's Design

6.4.1 Dependence Recorder

To support dependence recording, we employ a design similar to ParaLog [112]. Each core maintains a local instruction *counter* of the last dynamic instruction committed at this processor. Additionally, each L1-D cache block is enhanced with a field that records the *timestamp* corresponding to the last block access. A timestamp is a tuple consisting of the thread id and the counter associated with an application instruction, {tid, counter}. The timestamp uniquely identifies each instruction of the multi-threaded application. Finally, the L2 cache blocks are enhanced to carry timestamp information upon an L1-D block eviction. When an instruction carries dependences to instructions from other application threads, the dependence recorder captures and records the timestamps to the dependence queue.

Figure 29 shows the happens-before relationship between the instructions I1 and I2 of two application threads and how the coherence activity can be leveraged to infer this dependence. As the figure shows, the timestamp of the cache block for address A is updated with the value {1, T}, when the store instruction is executed, and is carried along with the coherence messages. Finally, the dependence recorder obtains and appends the timestamp in the dependence queue.

6.4.2 Dependence Checker & Progress Publisher++

Parallel FADE has to ensure that the dependences of an event have been satisfied before delivering the event to the filtering accelerator. In other words, Parallel FADE has to ensure that I1

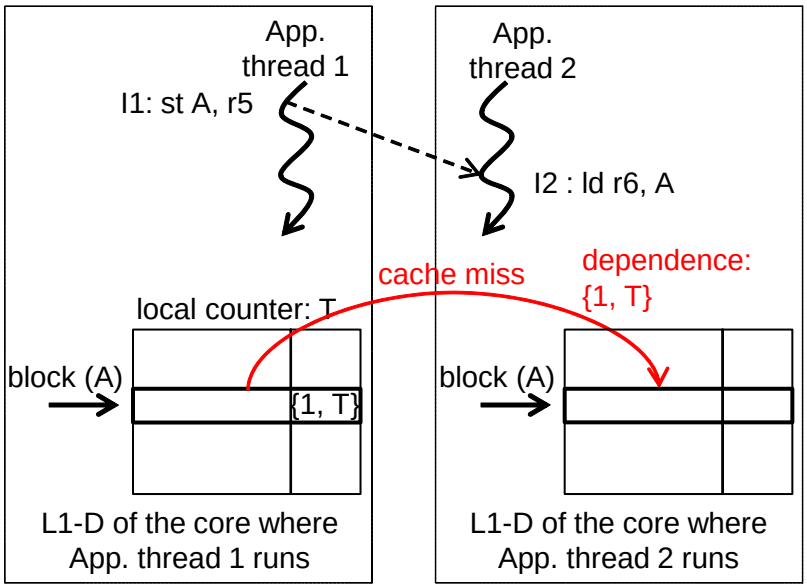

FIGURE 29: Leveraging coherence activity to infer inter-thread dependences.

(thread t) is processed before I2 (thread t'), when there is a happens-before relationship between I1 and I2. To support this functionality, we extend the hardware coordination mechanism proposed in ParaLog [112]. Similar to ParaLog, the monitor threads share a memory-mapped table of *progress identifiers*, indexed by the thread identifiers. The progress identifier of a monitoring thread is the timestamp value of the last application event (of this thread), whose processing has been completed in the monitoring system.

Progress Publisher++. The progress publisher updates the progress identifier of a monitoring thread as the events processing completes. To design the progress publisher correctly, we need to take into account that Parallel FADE supports Non-Blocking Filtering.

Non-Blocking Filtering decouples the filtering process and the execution of SW handlers for unfiltered events, thus allowing for their overlapped execution and out-of-order completion. Although, the events processing may complete out of order, the events progress has to be published in (commit) order.

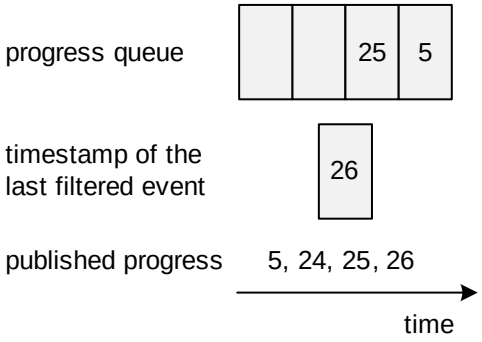

FIGURE 30: A snapshot of the progress queue and the timestamp of the last filtered event, along with published progress.

To advertise the events progress in (commit) order, we extend the baseline progress publisher with the *progress queue* and the *timestamp of the last filtered event*. The progress queue is a hardware structure that keeps the timestamps of the unfiltered events whose SW handlers are currently under execution. The size of the progress queue is determined by the size of the core's reorder buffer and the length of the SW handler in instructions. We find that the smallest SW handlers are 10 instructions long. Therefore, a progress queue with 10 entries suffices to keep the unfiltered events' timestamps for a 100-entry reorder buffer, typical in aggressive OoO cores. Figure 30 shows an example of the published progress given a snapshot of the progress queue and the timestamp of the last filtered event.

Dependence Checker. The dependence checker ensures that event dependences are satisfied before delivering the events to the filtering accelerator. When the dependence checker processes a dependence, such as {tid, T}, it uses the tid to index the hardware table with the progress identifiers and extract the thread's progress. If the progress is greater than T, the event is delivered to the filtering accelerator. Otherwise, the filtering process has to stall. In this case, the dependence checker periodically re-reads the progress, until the desired value is reached. To access the progress identifiers, each core maintains a hardware pointer to the progress identifier table (the *progress base register*).

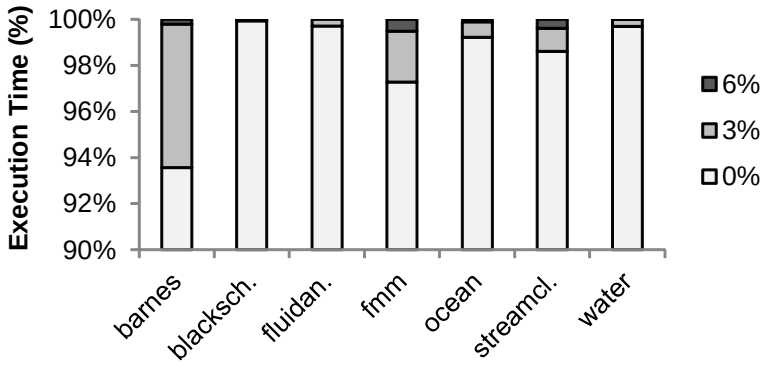

FIGURE 31: The event queue entries that carry dependences. The graph shows ranges (e.g., 6% means >3% and up to 6%)

6.4.3 Dependence Queue

After capturing the dependences in the dependence recorder and before processing them in the dependence checker, the dependences have to be buffered. In this section, we present our observations that lead to the design of a practical storage-efficient structure for storing the dependences.

The single-core FADE design (Chapter 5) keeps per event information (e.g., event operands, the effective memory address) in the event queue. A naive approach would be to extend the event queue entries to accommodate the dependences. However, we observe that the number of instructions that carry dependences is dictated by the application L1-D misses. As a result, the majority of instructions *do not carry* dependences.

In Figure 31, we show the percentage of the event queue entries that carry dependences for MemLeak, a representative monitor. Indeed, our results show that the majority of the time, the event queue keeps events without dependences. Even when the queue keeps events with dependences, the latter correspond to a small fraction of the total queue entries (up to 6%).

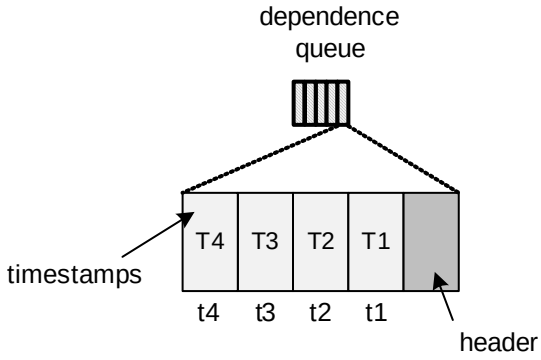

FIGURE 32: The dependence queue design for four monitored threads.

Based on this observation, we introduce a separate structure, the *dependence queue*, to store the dependences (Figure 32). Each entry of the dependence queue has fixed size and includes (1) a header with the timestamp of the instruction associated with the dependence, and (2) a list consisting of one timestamp per monitored application thread.

6.5 Accesses to Shared Metadata

As the monitor is a parallel process itself, the monitor's threads access shared metadata while performing the monitoring task. In this section, we discuss two important aspects of Parallel FADE related to shared metadata: (1) metadata coherence, and (2) concurrent metadata accesses.

6.5.1 Metadata Coherence

Parallel FADE introduces a metadata caching structure per core, for which coherence has to be preserved. The metadata caches can be naturally integrated to the cache hierarchy and are backed up by the L2 cache similar to the L1-D.

6.5.2 Racing Metadata Accesses

Parallel FADE has to ensure that concurrently executing monitoring threads do not corrupt their metadata due to improper synchronization. Under Sequential Consistency, synchronized access to shared metadata is guaranteed, given that application reads translate to metadata reads [112]. When this condition holds, the read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependences for the metadata accesses are already handled properly because Parallel FADE enforces the inter-thread event order. Although this condition holds for most monitors, in certain cases, such as AtomCheck, an application read may translate to a metadata write. As inter-event ordering is enforced by tracking coherence activities, it cannot capture read-after-read (RAR) dependences.

When an application read translates to a metadata write, there are racing metadata updates due to concurrent reads. As this scenario is not handled by the monitoring hardware, we need to study the monitor's algorithm and ensure that the monitor produces an equivalent result when processing concurrent application reads in any order. For instance, while monitoring an application, AtomCheck may encounter any of the eight accesses interleavings shown in Table 1. We observe that in case of concurrent application reads, the order in which the reads are processed by the monitor does not change the monitoring outcome.

In addition to algorithm-level guaranties, Parallel FADE has to preserve metadata atomicity in case of RAR dependences for monitors such as AtomCheck. Atomicity is necessary to prevent metadata corruption, when events with RAR dependences are processed concurrently by the monitoring threads. There are two case that we need to consider: (1) metadata updates in SW, and (2) metadata updates in Parallel FADE. In the first case, explicit synchronization is required in the SW handler, as the monitor may execute a number of instructions while updating metadata. Please note that the update of filtering critical metadata in the SW handlers is performed by a single instruction (atomically). Otherwise, Parallel FADE would have to be synchronized with the execution of the

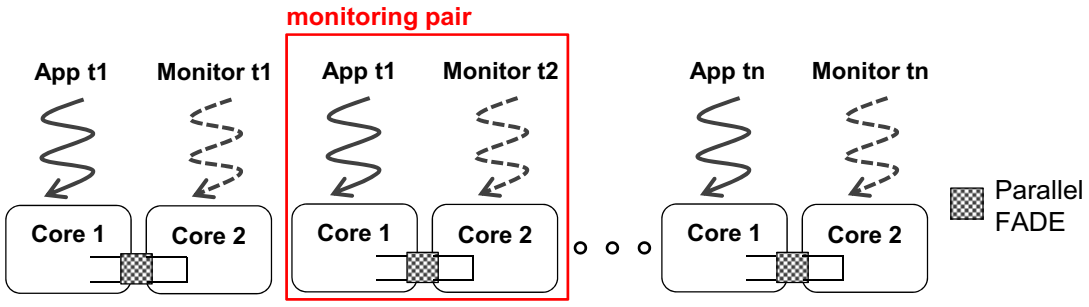

FIGURE 33: Evaluated System I: The monitor runs on a dedicated monitoring core.

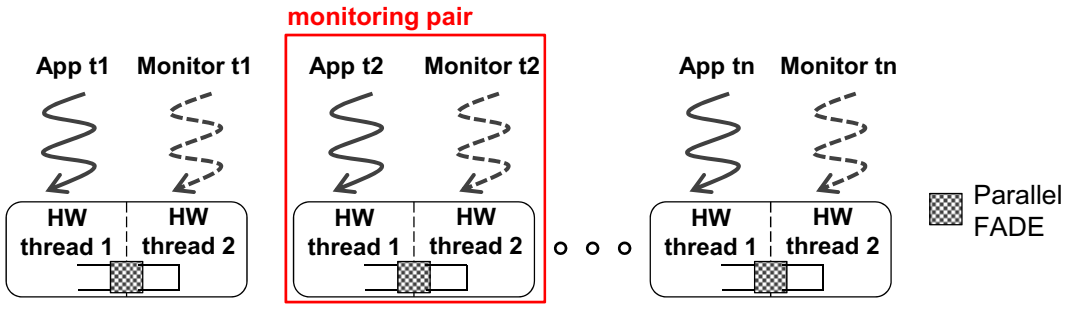

FIGURE 34: Evaluated System II: The monitor runs on a dedicated HW thread.

critical section in the SW handlers. In the second case, we observe that the metadata update is performed by a single memory access, in which case atomicity is trivially guaranteed.

6.6 Evaluated Systems

We evaluate two FADE-enabled parallel monitoring systems that afford high flexibility by handling unfiltered events on a general purpose CMP. Each application thread is monitored by a monitor thread. Together, these two threads form a *monitoring pair* (Figure 33, Figure 34). To guarantee isolation, we execute the application and the monitor in separate processes. To guarantee error containment the monitor synchronizes with the application at system call boundaries.

Table 5: System setup.

Parameter	Value
Core type	4-way/96-entry ROB
ISA	SPARC v9 [84]
L1 caches	32KB, 2-way, 64B block 2-cycle latency
Shared L2	2MB per core, 16-way, 64B block, 10-cycle latency
DRAM	90-cycle latency

The first monitoring system (Figure 33) executes the application and monitor threads on separate cores to maximize the concurrency, similar to ParaLog [112] and Resolve [111]. In this system, filtering takes place next to the monitor (consumer) core. We model a dedicated event queue separating the application (producer) core and the filtering accelerator; however, a memory-mapped queue [23] is also a viable design point.

The second monitoring system (Figure 34) is based on a dual-threaded core with a dedicated hardware thread for each of the application and the monitor processes. This design point minimizes resource requirements, but exposes the slowdown due to unfiltered events as core resources are shared between the application and the monitor. OS support is needed to ensure that each monitoring pair is restricted to run on the same multi-threaded core.

The unaccelerated versions of these systems are similar but without FADE. In these systems, the application and the monitor communicate through a single queue, instead of two queues (i.e., event queue and unfiltered event queue).

6.7 Methodology

System configuration. Table 5 summarizes the configuration of the evaluated systems. Additionally, FADE-enabled systems have a 4KB, two-way metadata cache with one-cycle access latency, and a 16-entry M-TLB. The event table has 128 entries, covering the heavily used subset

of the modeled ISA (SPARC). The event queue, the unfiltered event and the dependence queue is 32, 16 and 4 entries, respectively. For our experiments, we use Non-Blocking Filtering.

Unless otherwise specified, we evaluate two monitoring systems that execute the studied benchmarks with four application threads: (1) an eight-core CMP with four application threads (as in Figure 33), and (2) a four-core CMP with two-way multi-threaded cores (as in Figure 34).

Simulation. We use Flexus [114] for cycle-accurate full-system simulation. Flexus extends Simics [109] with timing models of multi-threaded cores, caches, and interconnect. We extend Flexus to model the parallel monitoring architecture described in this chapter.

For our evaluation, we follow the SMARTS sampling methodology [117], and the execution samples are selected to cover a representative part of the application's parallel section. In all benchmarks studied, the parallel section dominates the application execution time. For parallel sections organized as multiple iterations, we cover at least one iteration, and for some benchmarks two iterations. For the rest of the benchmarks that are not organized in this fashion, we cover at least 1 billion instructions. For each measurement, we launch simulations from checkpoints with warmed caches (including the metadata cache), and run 100K cycles to achieve a steady state of detailed cycle-accurate simulation before collecting measurements for the next 50K cycles.

Benchmarks. We include benchmarks from the SPLASH-2 [116] and PARSEC [10] benchmark suites. We use the following inputs: PARSEC (input: simlarge), FMM (input: 64K particles), Ocean (input: 1026x1026 matrix), and water (input: 2197 mols).

6.8 Evaluation

6.8.1 Monitoring Load

We measure the applications monitored IPC, so as to quantify the monitoring load of different monitors. In Chapter 4, we defined the monitoring load as the ratio of monitored instructions to all committed instructions. In Figure 35(a), we present the per-monitor results averaged across benchmarks. For instance, for AddrCheck, the average application IPC (including both monitored and unmonitored instructions per cycle) is 1.30, out of which 0.19 (monitored instructions per cycle) require a monitoring action to be taken.

AddrCheck and AtomCheck are memory tracking monitors, which only process memory instructions. AddrCheck has a lower average monitored IPC (0.19 event per cycle) compared to AtomCheck (0.43 event per cycle), because AddrCheck does not monitor memory accesses to stack. For both monitors, the monitored IPC is significantly below 1.0.

The rest of the monitors are propagation tracking monitors, which may track any instruction type based on their monitoring analysis. Out of all studied monitors, MemCheck and TaintCheck have the highest monitoring load (1.06 events per cycle on average). In Figure 35(b), we present the per-benchmark results for TaintCheck. The results for MemCheck are similar. We observe that for most of the benchmarks the monitored IPC is around 1.0 event per cycle, apart from two benchmarks (fmm and streamcluster), which have a monitored IPC of 1.45 event per cycle.

Figure 35(c) shows the per-benchmark results for MemLeak. The monitoring load of MemLeak is lower (0.47 event per cycle) compared to MemCheck and TaintCheck (1.06 event per cycle), because MemLeak does not need to monitor most of the FP instructions in order to identify memory leaks. Under MemLeak, FP arithmetic and FP load instructions do not need to be moni-

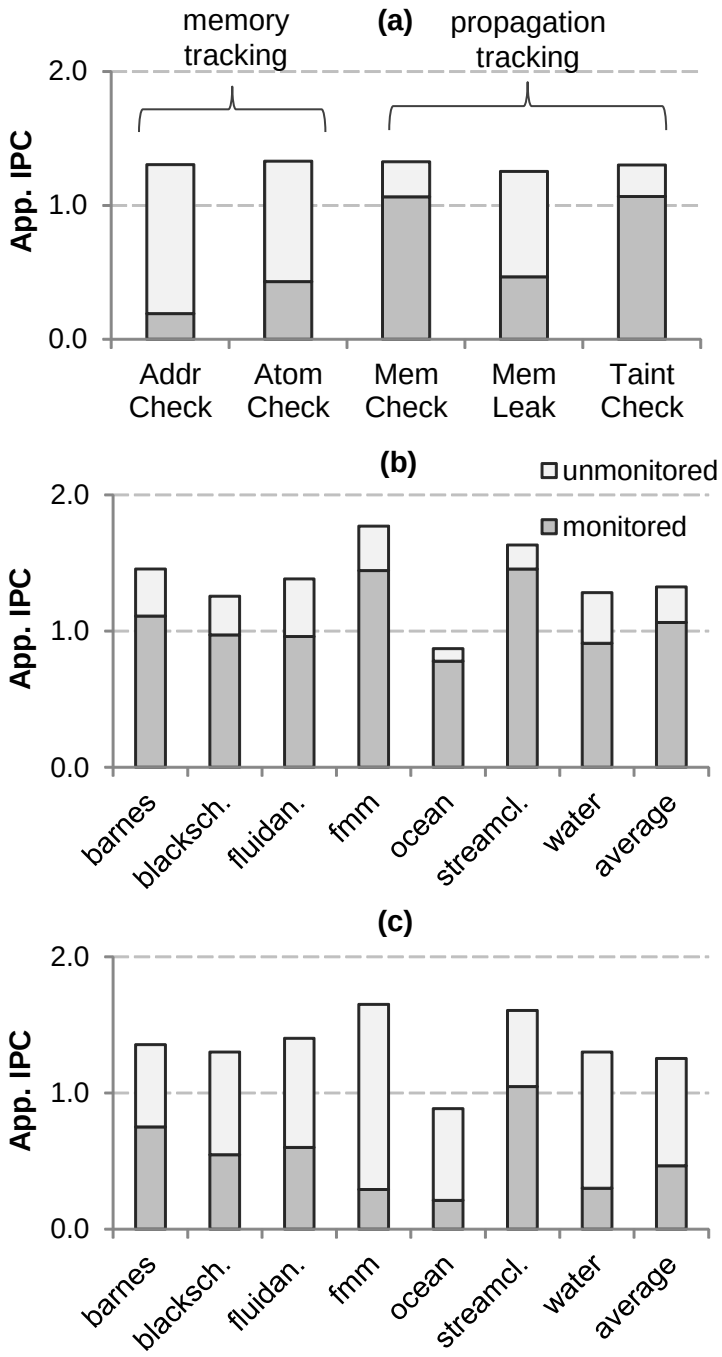

FIGURE 35: Parallel FADE: Breakdown of application IPC to monitored and unmonitored: (a) averaged across benchmarks for each monitor, and per-benchmark for (b) TaintCheck and (c) MemLeak.

Table 6: Filtering efficiency in Parallel FADE.

AddrCheck	99.8%
AtomCheck	90.0%
MemCheck	97.0%
MemLeak	81.0%
TaintCheck	95.0%

tored at all, while FP store instructions are monitored so as to set the accessed memory location to a clean (non-pointer) value.

As expected, the monitored IPC of MemLeak is lower for *floating point* benchmarks (studied in this chapter), compared to *integer* benchmarks (studied in Chapter 4). For instance, the studied integer benchmarks have a monitored IPC of 0.68 event per cycle (with an overall IPC of 1.10 events per cycle), on average, while the studied FP benchmarks have a monitored IPC of 0.47 event per cycle (with an IPC of 1.25 events per cycle), on average.

In summary, our results show that, on average, the monitored IPC is significantly below 1.0 event per cycle for three out of the five monitors (AddrCheck, AtomCheck, and MemLeak) and slightly higher than 1.0 event per cycle for the remaining two monitors (MemCheck and TaintCheck). Our study in Chapter 4 showed that FADE can significantly reduce the slowdown of monitors with such monitored IPCs, given a high filtering rate. Our performance results for Parallel FADE, presented later in this Section, corroborate our previous study.

6.8.2 Filtering Efficiency

Table 6 shows that FADE filters 81-99% of all instruction event handlers. AddrCheck has the highest filtering ratio because the vast majority of application accesses goes to allocated memory. MemLeak has the lowest filtering ratio, because the studied applications operate on big arrays allocated through malloc, thus requiring frequent metadata updates.

6.8.3 Parallel FADE versus Unaccelerated System

Figure 36 depicts the performance of Parallel FADE versus the unaccelerated monitoring system. In both systems, the application and monitor tasks execute on dedicated hardware threads of a dual-threaded 4-way OoO core (similar to Figure 34). Performance is normalized to an unmonitored (application-only) system.

In general, for the unaccelerated systems, we observe an average slowdown of 5.4x, across monitors. For memory-tracking monitors (AddrCheck, AtomCheck), the average slowdown is 2.5x, while for propagation-tracking monitors (MemCheck, MemLeak, TaintCheck), the slowdown is 8.9x. Parallel FADE reduces the slowdown significantly for all monitors, with an average slowdown of 1.6x. Parallel FADE's slowdown is 1.3x and 1.6x for memory- and propagation-tracking monitors, respectively.

Figure 36(a) shows AddrCheck's performance, which is generally good on both systems as the monitor just processes non-stack memory instructions. In the unaccelerated system, AddrCheck's slowdown ranges from 1.5x to 3.1x, with an average of 1.9x. Parallel FADE reduces the slowdown to an average of 1.05x by filtering out nearly all monitored events.

Figure 36(b) presents results for AtomCheck. Although AtomCheck is a memory-tracking monitor with a low event generation rate, it has an average slowdown of 3.4x (5.5x max) in the unaccelerated system because the events are costly due to numerous monitoring actions. In contrast, Parallel FADE reduces the slowdown to 1.8x, on average. The highest slowdown is observed for fmm (2.5x) as it has the lowest filtering rate (78%).

Figure 36(c) shows the results for MemCheck, a heavy-weight propagation-tracking monitor with an average monitored IPC of 1.06 event per cycle. In the unaccelerated system, the slowdown ranges from 8 to 16x, with an average of 11.5x. The highest slowdown observed for: (1)

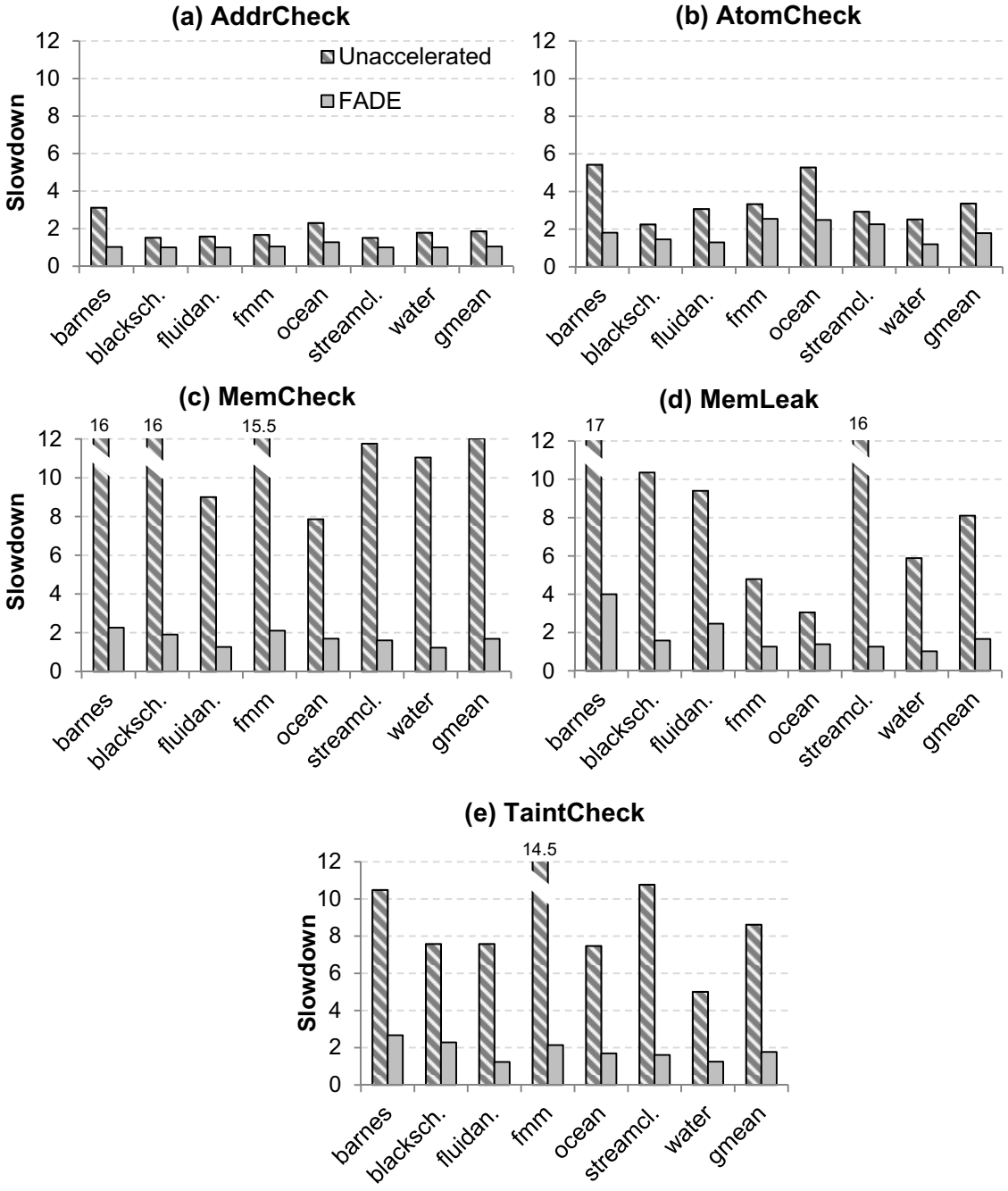

FIGURE 36: Performance of Parallel FADE compared to the unaccelerated system.

barnes and blackscholes, which have frequent stack updates, and (2) for fmm which has the highest monitored IPC. Parallel FADE significantly reduces the slowdown across benchmarks to an average of 1.7x thanks to the high filtering ratio and the stack update unit. The highest slowdown is observed for barnes (2.3x) and fmm (2.1x). Barnes must frequently drain the unfiltered event queue at function call/return boundaries (Section 5.2.2). Fmm has a high monitored IPC of 1.44 events per cycle, thus stressing Parallel FADE with a high monitoring load.

Figure 36(d) shows the results for MemLeak. The slowdown in the unaccelerated system ranges from 3 to 17x, with an average of 8.1x. We note that the monitored IPC of MemLeak is different compared to the rest of the propagation tracking monitors (MemCheck and TaintCheck) and is dictated by the percentage of floating point instructions in the instruction stream (Section 6.8.1). As a result, the lowest slowdown is associated with ocean and fmm, which have the lowest monitored IPC, 0.21 and 0.29 event per cycle, respectively. The highest slowdown is for barnes, which has the highest monitored IPC (0.75 event per cycle) and frequent stack updates.

Parallel FADE reduces the slowdown to an average of 1.66x, with the highest slowdown observed for barnes (4.0x) and fluidanimate (2.46x). Barnes is characterized by the highest monitored IPC (0.75 event per cycle), a low filtering ratio (78%), and frequent stack updates. Fluidanimate is characterized by the second higher monitored IPC (0.6 event per cycle) and a low filtering ratio (75%). Streamcluster's slowdown is high (8x) but can be reduced to 1.3x by taking a closer look at the code. We include the improved results here, and we discuss streamcluster's behavior in detail in Section 6.8.6.

Figure 36(e) shows the results for TaintCheck. In the unaccelerated system, the slowdown ranges from 5 to 10.5x, with an average of 8.6x. Although TaintCheck has the same monitored IPC as MemCheck, TaintCheck's monitoring algorithm does not require to process stack updates. As a result, the slowdown in the unaccelerated system is lower, especially for benchmarks such as barnes (10.5x) and blackscholes (7.6x) with frequent stack updates. The slowdown for fmm is sim-

ilar for MemCheck (15.5x) and TaintCheck (14.5x), as the high slowdown stems from the high monitored IPC (1.44 events per cycle), which is the same for both monitors. Parallel FADE reduces the monitoring slowdown for all the benchmarks to an average of 1.7x due to the high filtering ratio, which is 95% on average.

6.8.4 Dedicated Monitoring Core versus HW thread

In Section 5.4.3, we evaluated the performance of a single-core monitoring system (with a dedicated HW thread) versus a two-core monitoring system (with a dedicated monitoring core), both equipped with FADE, and we showed that the performance benefits of the second core are limited thanks to the high filtering rates. In this section, we repeat this study for the equivalent parallel monitoring systems, which are shown in Figure 33 and Figure 34, respectively.

Figure 37(a) compares the performance of the two systems. The results indicate that the dedicated-core design outperforms the dedicate-thread design by only 7% on average (19% max). Figure 37(b) shows the break-down of the two-core system utilization. The execution time is broken down into four categories: cycles in which (1) the application core is idle because the event queue is full, (2) the monitor core is idle because Parallel FADE filters all events, (3) both application and monitor cores are utilized, and (4) the log is empty. The log can be found in an empty state, when the application is scheduled off, due to backpressure from monitor. This happens for the monitors having the lowest filtering rates (i.e., AtomCheck with 90% and MemLeak with 81%).

As the figure shows, both cores are utilized only 15% of the time, as either Parallel FADE filters the incoming event stream (idling the monitor core), or the monitor core processes unfiltered events (backpressuring the application core). These results corroborate our prior study in Chapter 5, showing that the benefit of the second core is limited.

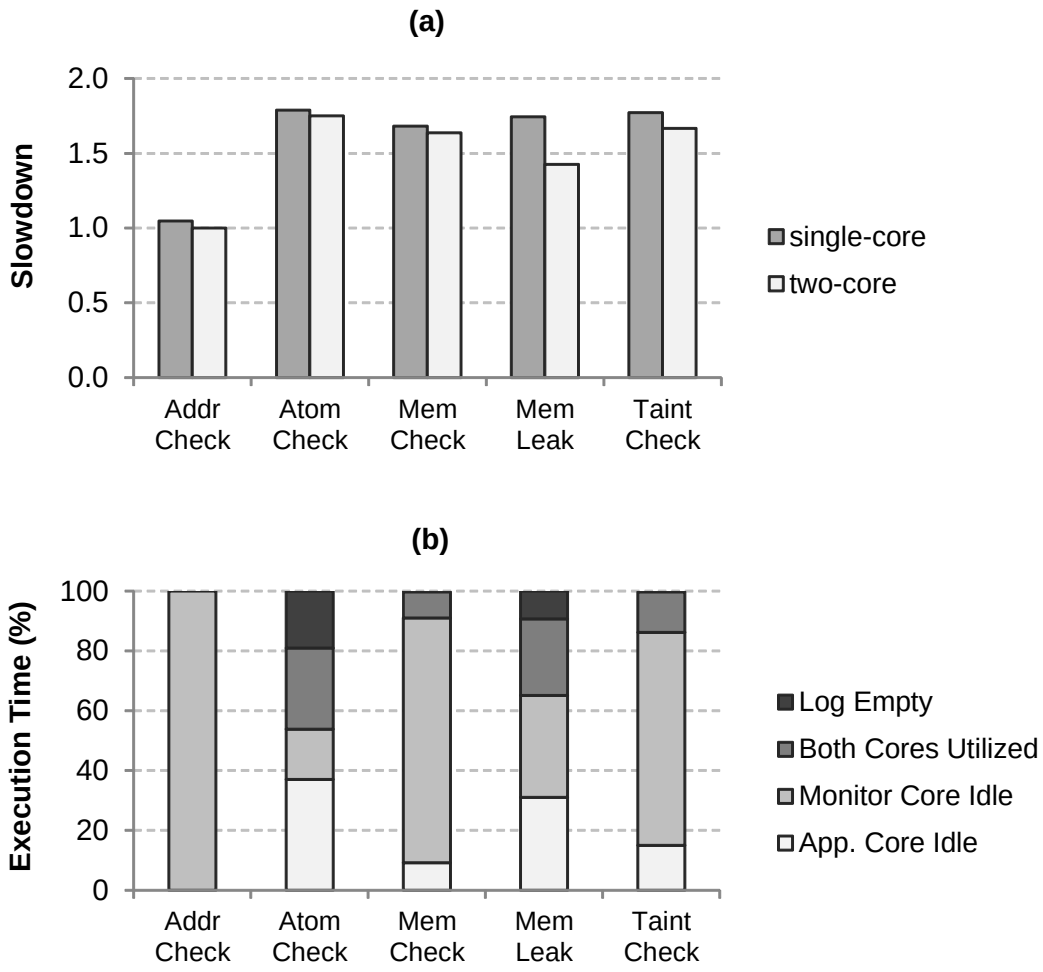

FIGURE 37: (a) Performance of a dedicated monitoring core versus a dedicated monitoring HW thread. (b) Core utilization in the two-core system.

6.8.5 Scalability Analysis

In this section, we show the performance of the unaccelerated system (Figure 38(a)) and Parallel FADE (Figure 38(b)) for 2, 4 and 8 monitoring pairs, running on 2, 4 and 8 cores, respectively. Both systems' performance shows good scalability for all monitors, but AtomCheck.

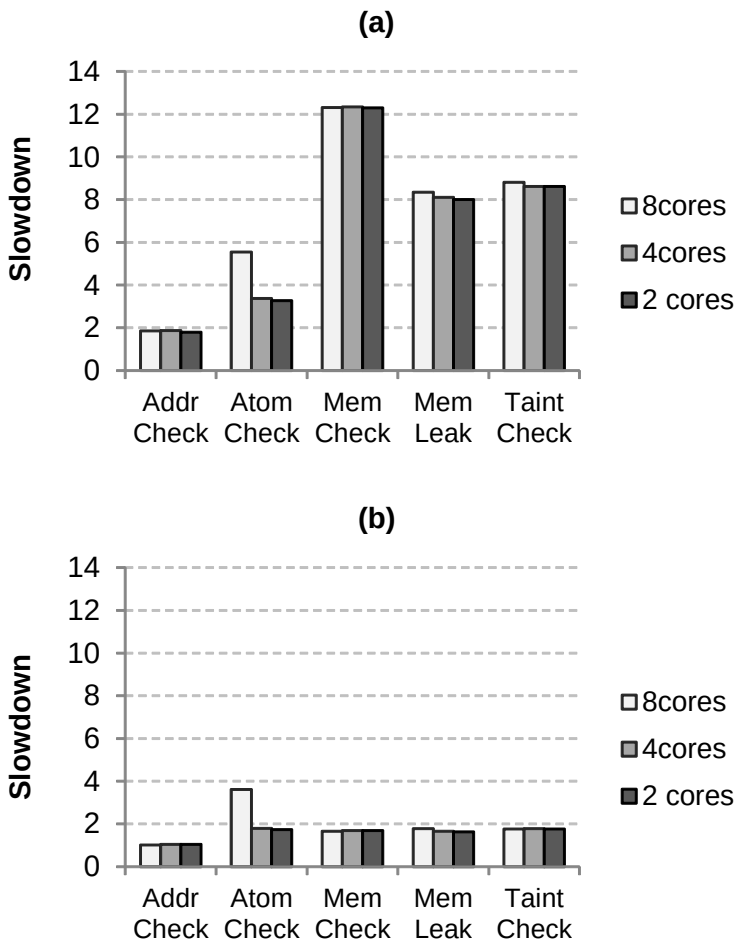

FIGURE 38: Performance of (a) the unaccelerated system and (b) Parallel FADE for 2, 4 and 8 monitoring pairs running on 2, 4 and 8 multi-threaded cores, respectively.

AtomCheck's performance does not scale to 8 cores for the unaccelerated system and Parallel FADE, because the number of AtomCheck's SW structures increases with the number of the monitored application threads, unlike the rest of the monitors. Specifically, AtomCheck maintains one local per-thread table (more details on AtomCheck's metadata structures can be found in Section 2.3). Regarding Parallel FADE, although the filtering rate for 8 threads is the same as the filtering rate for 4 and 2 threads, the slowdown is higher (by almost 2x), following the trends in the baseline system.

```
from streamcluster benchmark: streamcluster.cpp
/* compute Euclidean distance squared between two points */
float dist(Point p1, Point p2, int dim)
{
    int i;
    float result=0.0;
    for (i=0;i<dim;i++)
      result += (p1.coord[i] - p2.coord[i])*(p1.coord[i] - p2.coord[i]);
    return(result);
}
```

FIGURE 39: The function that calculates the Euclidean distance in streamcluster benchmark.

6.8.6 Discussion

While monitoring *streamcluster* with MemLeak, we observed that the benchmark has a high ratio of unfiltered events (48%). As this ratio, significantly differs compared to the rest of the monitors and benchmarks, we wanted to identify the source of this behavior.

Streamcluster frequently calculates the euclidean distance between two points. The corresponding piece of code is shown in Figure 39. By observing the assembly of this function, we found that most instructions cannot be filtered because they operate on pointer data. This happens for three reasons; First, p1 and p2 are pointers. Thus, when calculating p1.coord[i] - p2.coord[i], the code operates on pointer data. Second, although the outcome of the subtraction p1.coord[i] - p2.coord[i] could be reused in the same iteration, it is calculated twice through a similar instruction sequence that (again) operates on pointer data. Third, although p1.coord and p2.coord could be reused across iterations, they are re-calculated on each iteration.

This code could have been improved in at least two ways in order to assist the filtering task: (1) the outcome of the subtraction p1.coord[i] - p2.coord[i] could be reused in the same iteration,

and (2) p1.coord and p2.coord could be reused across iterations (while now they are calculated twice on each iteration).

Although, this low-level transformation of the code could have significantly reduced the pointers manipulation, there are two higher level observations that can be leveraged to completely filter the monitoring of the whole function. First, this function only generates temporary values that do not update the monitor's memory metadata. Second, this function does not propagate a pointer status, when it returns.

With a filtering ratio of only 52%, streamcluster experiences a slowdown of 8x (with an initial slowdown of 16x). However, when filtering the instructions of the function shown in Figure 39, *without affecting the monitoring outcome*, the slowdown is reduced to just 1.27x. Although, our extensive study of a large number of benchmarks and monitors shows that low filtering ratios are rare, our results for streamcluster show that programmer's hints could further reduce the monitoring slowdown.

6.9 Relaxed Memory Models

In this chapter, we discussed parallel monitoring under Sequential Consistency (SC). Under SC, coherence activity is sufficient to replay the application event order (ParaLog [112], Kannan [57]). However, coherence activity alone does not suffice to replay multi-threaded application execution under more relaxed memory models. As shown in prior work (ParaLog [112], Kannan [57], Resolve [111]), dependence-cycle deadlocks can be generated because memory references can be reordered in memory, with the exact reorderings being defined by the consistency model. For instance, Total Store Order (TSO) allows loads to bypass stores to unrelated addresses and to obtain their value from the memory or the store buffer.

ParaLog [112] and Kannan et al. [57] discuss monitoring under TSO and propose hardware similar to the mis-speculation detection mechanisms in SC systems (e.g., MIPS R10000 [121]), so

as to identify SC violating instructions. TSO violates SC semantics in two cases: (1) instruction reorderings, and (2) store atomicity violations, as shown by Arvind and Maessen [5]. In these cases, the monitoring frameworks handle SC-violating instructions properly, so as to avoid deadlocks.

The frameworks discussed so far (ParaLog [112], Kannan et al. [57]) can not handle more relaxed memory models, such as SPARC Relaxed Memory Order (RMO) [84], because they allow for additional memory reorderings. In contrast, Resolve [111], a recent monitoring framework, can handle dependence cycles and avoid deadlocks under more relaxed memory models through a software/hardware approach that relies on the application dataflow graph to identify and handle dependence cycles.

To accelerate parallel monitoring under more relaxed memory models, Parallel FADE can be extended with the mechanisms described in prior work (ParaLog [112], Resolve [111]). The resulting monitoring system has to ensure that (1) the events are delivered to the filtering accelerator in commit and dependence order (after handling potential dependence cycles), and (2) the events progress are advertised in commit order. These two requirements were discussed in Section 6.3 (also shown in Figure 28).

6.10 Summary

This chapter introduced Parallel FADE, a parallel monitoring accelerator that allows for the design of fast, flexible and resource-efficient monitoring systems. Our framework combines Non-Blocking Filtering, the state-of-the-art hardware filtering approach to accelerate monitoring, with the necessary hardware extensions to handle the inherent concurrency of parallel applications, overall reducing the complexity of prior parallel monitoring systems.

Using a suite of diverse monitors and a number of multi-threaded benchmarks, we showed that Parallel FADE filters 81-99% of events that would otherwise be handled in software and

reduces the slowdown to an average of only 1.1-1.8x (versus 1.9-11.5x for unaccelerated execution), thus making monitoring practical.

Chapter 7

Related Work

There has been a large body of work on application monitoring aiming to assist programmers with software development. Due to the plethora and diversity of bugs in the increasingly complex today's software and the importance of robust software in modern societies, the monitoring approaches differ in the degree of flexibility, speed and the proposed hardware modifications. In this Chapter, we provide an overview of the related work by discussing software monitoring systems in Section 7.1, and hardware monitoring systems in Section 7.2. Then, in Section 7.3, we discuss monitoring systems that provide support for parallel applications.

7.1 Software Monitoring Systems

In this section, we discuss Dynamic Binary Instrumentation approaches (Section 7.1.1), and then, we provide an overview of software-only tools for sequential bugs (Section 7.1.2) and concurrency bugs (Section 7.1.3).

7.1.1 Dynamic Binary Instrumentation Approaches

Software monitoring frameworks rely on Dynamic Binary Instrumentation (DBI) [13], a technique that inserts additional code into an application, so as to monitor its behavior. There are two approaches [69] to DBI: *probe-based* and *jit-based*. The probe-based approach dynamically replaces the original application instructions with new instruction sequences (i.e., *trampolines*)

that branch to the instrumentation code. Example systems include Dyninst [14], Vulcan [102] and DTrace [19].

The jit-based systems perform *just-in-time compilation* to translate the original application code and save the resulting code, so at to re-use it in the future (*i.e., code caching*). In literature, this process is also referred to as *process virtualization* [12]. To implement the monitoring functionality, these systems instrument the binary as necessary after the translation step and before saving the (translated and instrumented) instruction sequence in the code cache. Example systems include Valgrind [81], PIN [69], Strata [94], DynamoRIO [13] and DIOTA [70].

Each framework supports a number of platforms and operating systems. For instance, PIN, developed and distributed by Intel, can monitor applications running on Linux or Windows on x86 machines, while Valgrind targets more platforms including x86, AMD64, ARM, PPC, MIPS. Additionally, these frameworks may come with a number of monitoring tools. For instance, Valgrind's current distribution (Valgrind 3.9.0) includes a number of production quality tools, such as a memory error detector and a cache profiler. Although the associated monitoring slowdown can be as high as 100x [1], software frameworks have been widely adopted because they provide flexible, programmable and accurate monitoring, thus showcasing the need for general-purpose monitoring support.

7.1.2 Software Monitoring Tools for Sequential Bugs

In this Section, we discuss representative tools that target a wide range of sequential bugs (e.g., memory safety violations, memory access violations).

Memory safety tools are based on one of the three following approaches [35]: (1) the red-zone approach, (2) the object lookup approach, and (3) the fat-pointer approach. Tools based on the red-zone approach, such as Purify [51] and Valgrind's MemCheck [81]**,** surround an array with red-zone blocks that indicate invalid memory blocks. Accesses to the red zones flag a violation.

However, this heuristic can only identify invalid accesses close to the object. Tools based on the fat-pointer approach, such as CCured [28], SafeC [6] and Annelid [80], enhance pointers with additional information (i.e., their bounds). These metadata are updated during execution (e.g., pointer arithmetic) and checked when a pointer is dereferenced. Finally, tools based on the object lookup approach, such as Jones and Kelly tool [54] and the tool proposed by Ruwase et al. [92], track the size of each object (e.g., using a splay tree), and perform out-of-bound checks upon pointer dereferences.

Type-safety tools can assist programmers to debug applications written in low-level languages that do not provide type safety guarantees. For instance, Loginov et al. propose a tool [62] that enforces type safety upon a memory access, through checks inserted at compile time. Hobbes [17] dynamically tracks the type of each variable (based on operations performed on the value) and detects subsequent operations that are incompatible with the inferred type. DynCompB [49] is a related tool that dynamically infers abstract types so as to assist program comprehension and derive invariants.

Memory access errors were among the first bugs to be studied by the debugging experts leading to the development of well-known tools, such as Purify [2, 51] and Valgrind's MemCheck [81]. Purify inserts the checks statically, while Valgrind's MemCheck instruments the binary with checks dynamically. Although, the implementation details may differ, both tools rely on a similar algorithm that has been described in Section 2.3.

A number of tools target *overwrite-related security exploits*. Certain tools, such as StackGuard [30] and LibSafe [7] target specific exploits. While others, such as LIFT [89] and TaintCheck [82], are based on dynamic information flow tracking [103] and can detect overwrite-related security attacks independent of the specific vulnerability being exploited.

Finally, there are tools that *infer invariants* for program's variables, thus assisting programmers to better understand their code and identify buggy behavior. Daikon [39] is an early system

that stores all the values taken by program variables throughout its execution, and analyzes them off line to extract a set of invariants. DIDUCE [50] is another software tool that dynamically monitors the range of values taken by an application variable, thus inferring one invariant per memory instruction.

7.1.3 Software Monitoring Tools for Concurrency Bugs

In this section, we discuss tools that target the three main categories [64] of concurrency bugs: data races, atomicity violations and order violations.

Data race detection has received a lot of attention from the research community in the past years. In general, race detectors are categorized to *precise*, when they do not report false alarms, and to *imprecise* otherwise. Typically, precise detectors, such as DJIT+ [86], rely on vector clocks (VCs) to infer the happens-before relationship [60] of memory accesses. However, a precise race detector may employ a different algorithm. For instance, GoldiLocks maintain a set of “synchronization devices” per memory location [37].

To tackle the runtime and memory overhead of precise datarace detectors researchers proposed imprecise algorithms. Eraser (also referred to as LockSet) [93] is a well-known imprecise race detection algorithm that issues a warning, if there is no lock consistently held when accessing a particular memory location. To optimize for precision, performance or both, researchers proposed hybrid schemes that combine lockset and VC algorithms. Race detectors in this category include MultiRace [87] and RaceTrack [122].

FastTrack (2009) [41] has been a milestone in datarace detection, allowing for precise detection similar to VCs but without the associated overheads. FastTrack is based on the observation that the full expressiveness of vector clocks is infrequently necessary. To further reduce the runtime overhead of race detection other approaches are necessary, such as hardware support (e.g., Radish [36]), or crowdsourcing (e.g., RaceMob [58]).

Atomicity violations are an important class of concurrency bugs that can manifest even in data race-free programs. Atomicity, also referred to as *serializability*, is a property that holds for several concurrently executed actions, when their data manipulation effect is equivalent to that of a serial execution of them [65]. Here, we discuss dynamic detectors that do not require code annotation.

Atomizer [40] leverages the lockset algorithm [93] to infer synchronization between threads and relies on simple heuristics to identify atomic blocks. As a result, its synchronization knowledge is limited by the lockset algorithm. SVD [119] is a subsequent detector that automatically infers atomic regions based on data and control dependencies. SVD reports bugs when such regions are interleaved by unserializable writes. However, SVD is limited to a specific set of unserializable interleavings. A subsequent work, AVIO [65], defines the complete set of unserializable access interleavings to shared variables, and proposes a tool that can identify all possible interleavings, thus extending SVD's accuracy. Although, the tools discussed so far do not miss a true atomicity violation, they may produce long warning reports, thus placing a large burden upon the programmer. Veldrome [42] reduces the false positives generated by others detector, but may miss a real violation (false negative).

Order violations are the third category of concurrency bugs. An order violation occurs when a programming assumption on the order of certain events is not guaranteed during the implementation. For instance, a thread accesses an object, before the object's creation by another thread. Order violation bugs can manifest even in a program without atomicity violations. So far, there has been only a few proposals targeting order violations, such as Bugaboo [66] and DefUse [98].

Bugaboo [66] proposes context-aware communication graphs to detect a number of concurrency bugs including order violations. Specifically, Bugaboo collects communication graphs from multiple executions and uses invariant-based techniques to detect anomalies. DefUse [98] is another invariant-based bug detection tool, which targets a number of sequential and concurrency

bugs including order violations. DefUse automatically extracts invariants during a training phase and then uses them to detect bugs. For instance, DefUse leverages the Local/Remote (LR) invariant that indicates whether a read uses a value produced by the local or by a remote thread. DefUse detects a bug when the inferred invariant is violated.

7.2 Hardware-Based Monitoring Systems

A number of proposals have sought to provide hardware support for a variety of monitoring tools. To ease description, we group prior work into categories, and then we discuss individual solutions.

7.2.1 Hardware Support for Dynamic Information Flow Tracking

Early hardware-only proposals implement the monitor directly in hardware and hardwire the monitoring policy. Examples include data race detection [125], and propagation tracking [31, 103]. HARD [125] implements the lockset algorithm in hardware so as to enable data race detection at low overhead. HARD extends each L1 and L2 cacheline with bloom filters to store lock sets and performs the necessary set operations with specialized bitwise logic. Minos [31] is a propagation tracking monitoring system that protects the integrity of control flow data. Control flow data are any data loaded into the program counter upon a control transfer. In doing so, Minos extends all critical pipeline structures (e.g., register file, reorder buffer), and the on-chip caches with one metadata bit per 32-bit application word. In Minos, a separate monitoring pipeline processes the metadata transparently to the application execution on the main pipeline. In parallel with Minos, Suh et al. [103] developed a similar monitoring system. Their work mainly focuses on the size of the metadata footprint.

RIFLE [106] provides architectural support for information flow tracking similar to Minos [31] and the work by Suh et al., but focuses on preventing the illegitimate use of privileged data. In

doing so RIFLE, maintains labels (metadata) for all program's data, and a set of legitimate flows (pairs of labels), that determine how information can flow. To ensure security, RIFLE verifies that a program only contains legitimate flows. For instance, a store with its source operand and its destination memory location associated with the labels l1 and l2, respectively, is legitimate, only if the flow l1 -> l2 is a legitimate flow.

MemTracker [108] and FlexiTaint [107] append an in-order monitoring pipeline to the processor's out-of-order pipeline. The monitoring pipelines support up to four bits of metadata per application word. While MemTracker [108] checks only memory accesses, similar to HeapMon [96], FlexiTaint also includes support to track taint propagation. Specifically, FlexiTaint employs rule-based filtering to determine whether taint propagation can be performed in dedicated logic delegating only uncommon functionality to software.

Another work, Raksha [32] proposes an architecture for software security based on dynamic information flow tracking. Raksha includes a separate pipeline that propagates and checks the metadata transparently to the main processor's pipeline. Registers, caches, and memory are extended with fixed 4-bit tags. Similar to FlexiTaint, Raksha can execute software handlers without the overhead of operating system traps so as to complement the hardware-based analysis. The authors showcase the system's bug-finding capability through an FPGA prototype.

SIFT [85] proposes a monitoring system tailored for tainting propagation. The application and the monitor run on an 8-way (POWER-like) core with SMT support. SIFT executes only one (specialized) monitoring instruction per application instruction. Although, SIFT combines an aggressive core with monitor-specific hardware the slowdown is still around 20%.

Hardbound [35] provides spatial memory safety for C/C++ programs through architectural support for bounded pointers. Upon application's memory allocation, the metadata for pointers bounds are initialized by software. Then, the metadata are propagated in hardware and checked when a pointer is dereferenced, transparently to the application's execution. Just instrumenting

malloc-related library calls is sufficient to enforce spatial memory safety for heap objects. However, compiler support is needed for complete safety including arrays allocated in the stack.

WatchDog [76] also aims at providing spatial memory safety for C/C++ programs. Upon application's memory allocation, WatchDog generates a unique identifier which is associated with the pointer to the allocated memory region. On each memory access, the identifier's validity is verified through the *lock and key identifier* technique, which implements the check with just a load and a comparison operation. The identifier consists of two components: (1) a key (a 64-bit unsigned integer) and (2) a lock (a 64-bit address which points to a location in memory). An identifier is valid, when the value contained in the lock location is equal to the key's value.

Although, certain monitoring systems include programmable structures [32, 107, 108], the supported monitors are limited by the implementation complexity. As the monitoring pipelines can only accommodate fixed-sized metadata per application word, they cannot implement monitoring algorithms with more complex metadata, such as AtomCheck, bounded pointers [35] and vector clocks [95].

7.2.2 Specialized Hardware-Based Monitoring Systems

DISE [29] proposes hardware support for DBI. In doing so, DISE augments the processor pipeline and injects microcode to implement the monitoring functionality for each application instruction. Although this approach is flexible, it incurs high runtime overhead as each application instruction results in a sequence of monitoring instructions inserted into the pipeline. AccMon [124] proposes a heuristic to perform PC-based invariant detection. This system builds on the observation that a given memory location is accessed by only a few instructions under normal execution. As a result outlier instructions can indicate memory corruption, buffer overflow, or other memory-related bugs. Testudo [47] is a hardware approach for security analysis based on statistical sampling. Testudo distributes the monitoring load to multiple runs (users) by only analyzing a

few tagged variables during any particular run. To gain high coverage, Testudo requires a sufficient population of users sampling different (random) sets of variables. Radish [36] is a software/ hardware approach for vector-clock based race detection, that maintains metadata in L1-D caches in order to reduce the number of data race checks in software. Radish requires specialized logic for SIMD-style vector clock computations.

Atom-Aid [68] is inspired by prior work enforcing consistency at a coarse grain in order to bridge the performance gap between strict and relaxed memory models (e.g., BulkSC [21]). Atom-Aid leverages transactional memory to executes a chunk of instructions atomically. This way Atom-Aid reduces the degree of memory interleavings thus guaranteeing atomicity implicitly. ColorSafe [67] targets atomicity violations due to multiple variable, in contrast to AtomAid that focuses only on single variables. First, ColorSafe assigns a color to a set of variables and treats them as a single unit thereafter. Then, ColorSafe detects multi-variable violations through unserializable access interleavings to data with the same color. Although this system is presented as a solution for debugging and deployment alike, its use with deployed code seems impractical as the application data cannot be colored in an automated way.

Mondrian memory protection [115] is a fine-grain protection scheme allowing for multiple protection domains. In contrast to earlier proposals, that enforce protection at the page level, Mondrian allows for permissions control at the granularity of individual words. Mondrian employs compression to reduce the metadata space overhead, and two-level metadata caching to reduce the runtime overhead.

7.2.3 Monitoring Systems Using Hardware of Contemporary Processors

SafeMem [88] leverages Error Correction Code (ECC) available in off-the-shelf hardware to develop a heuristic for the detection of memory leaks and memory corruptions. For instance, to detect memory leaks, SafeMem builds on the observation that most objects have an expected life-

time. Thus, objects significantly exceeding their expected lifetime are considered to be suspects (i.e., potential memory leaks). SafeMem employs ECC to further monitor the suspect objects, so as to prune false positives. A false positive is identified upon a subsequent access to a suspect object (i.e., the object is still in use).

Greathouse et al. [46] propose a demand-driven race detection system that performs monitoring activity upon inter-thread sharing reflected in coherence events. To keep track of interesting coherence events, this work leverages performance counters, available in contemporary modern processors. Although this approach benefits applications with little sharing, it can result in inaccuracies (miss some races), and performance degradation due to false sharing.

PBI (a production-run failure diagnosis system) [3] samples hardware performance-counter events at run time and uses statistical processing to discover instructions related to failures. To detect concurrency bugs, such as atomicity and order violations, PBI leverages performance counters that indicate the cacheline state (i.e., Modified, Exclusive, Shared, Invalid) before a memory access. To detect sequential bugs (i.e., deviations from the intended execution path), PBI leverages performance counters indicating whether a branch is taken or not taken.

Another work by the same authors [4] makes the observation that often bugs have short propagation distance, thus collecting information close to program's failure can be sufficient to diagnose the bugs root. To identify deviations from the intended execution path, this work leverages existing hardware, named the Last Branch Record (LBR), and records the last few taken branches. To help diagnosing concurrency bugs, this work proposes simple hardware extensions, named Last Cache-coherence Record (LCR), that record the last few cache accesses with specified coherence states. Although, these heuristics are effective for bugs causing program failures, they are not generally applicable to other monitoring algorithms (e.g., memory leaks) or when the monitored programs do not crash.

7.2.4 Watchpoint-Based Monitoring Systems

iWatcher [126] checks memory accesses that belong to pre-specified (i.e., *watched*) memory ranges but cannot support propagation tracking monitors, such as MemCheck. Unlimited Watchpoints [48] also trigger monitoring activity when a watched memory location is accessed. However, in contrast to iWatcher, this system allows for fine-grain manipulation of the watched memory locations, by storing the associated metadata on a Range Cache [105]. As Range Caches can summarize ranges of metadata having the same values, there are performance benefits when the monitor's metadata show good spatial locality. However, slowdown increases when byte-level watchpoints are necessary, or when the monitor performs frequent range updates that may spawn multiple ranges (e.g., stack updates). Sentry [99] proposes an accesses control mechanism based on memory tagging that is employed either to enforce memory protection or to provide watchpoint-based debugging.

7.2.5 Systems Implementing the Monitor on a Different HW Substrate

Prior work has proposed the use of on-chip reconfigurable fabric to implement the monitoring functionality [33, 57]. These systems can accommodate multiple monitors but face two issues: (1) they require low-latency access to metadata, which is challenging for large metadata (e.g., a vector clock per word for FastTrack [41]) and (2) they require the reconfigurable fabric to be clocked at frequency comparable to the monitored core. FADE can assist these monitors (1) by identifying whether an event is filterable by accessing much less metadata kept in an auxiliary map (common case is encoded with one-two bits), and (2) by filtering a large portion of application events allowing the reconfigurable fabric to run at lower frequency.

Introspective cores [75] advocate that although monitoring hardware can significantly increase the developers' productivity, the rest of the users should not pay the additional cost associated with this hardware. Thus, this work proposes implementing the monitoring functionality on

a separate logic die stacked vertically on the processor die using 3D IC technology. The focus is on the impact of the monitoring layer to the chip's design in terms of the area, power and temperature.

7.2.6 Hardware-Assisted Multi-Cores

INDRA [97], HeapMon [96] and Log-Based Architectures (LBA) [23] are monitoring frameworks that capture application events in hardware and communicate them to a neighboring core on a CMP to perform the monitoring task. LBA is the most related to our work as it also includes three mechanisms aiming to accelerate a number of monitors. The first accelerator, unary inheritance tracking (IT), targets the overhead of TaintCheck and MemCheck. Unary inheritance tracking uses hardware to accelerate only instructions with one source operand, adding extra overhead or sacrificing bug coverage for two-source operand instructions. The second accelerator, idempotent filter (IF), stores addresses to filter redundant metadata checks, but requires frequent storage flushing (upon stack-update, malloc, free, etc.). The third accelerator, a metadata TLB, maps an application address to a memory metadata address in hardware. Although the combination of the three accelerators reduces the slowdown of the targeted monitors, it has the following shortcomings compared to FADE: (1) may sacrifice bug coverage (unary inheritance tracking for TaintCheck), (2) has lower filtering rate resulting in higher monitoring slowdown[6], (3) does not accelerate stack updates, and (4) does not consider aggressive OoO cores that stress the accelerators with a high event generation rate.

To reduce the overhead of serial propagation tracking tools for serial applications, Parallel DIFT [90] executes a parallelized version of TaintCheck and MemCheck on the LBA framework. In doing so, parallel DIFT divides the application's log into segments and assigns each log segment to a worker core. Then, the worker cores produce monitoring summaries, which are pro-

6. In our prior study, BugSifter [43], we evaluated LBA and a comparable version of Blocking FADE. We showed that the filtering rate of LBA is lower compared to FADE (25-78% vs. 80-98%), thus resulting in higher monitoring slowdown.

cessed and merged by a master core. To generate shorter summaries and simplify the parallelization process, this system employs unary inheritance tracking, that tracks the information flow through unary operands. When the actual value cannot be determined explicitly, Parallel DIFT uses a symbolic value (e.g., a register id). Concurrently and independently, Nightingale et al. [83] proposed a similar approach to parallelize DIFT on commodity CMPs. However, this work does not employ unary inheritance tracking, thus the resulting slowdown can be significant (up to 9x for taint propagation).

Guardrail [91] is a monitoring framework for detecting bugs in device drivers at runtime, and from preventing anomalous behavior from propagating to the rest of the system. In contrast to previous proposals that focus on bug detection at the driver's interface, Guardrail performs instruction-grain correctness checking as the driver executes. The monitored events are logged and sent to another core for further processing similar to LBA [23]. Hardware support for logging is sufficient to significantly lower the overhead (at most 10%) and to allow for practical deployment for applications with low driver activity. However, the overhead is higher (60% drop in throughput for network streaming), when the drivers activity is frequent. FADE's approach could be employed to lower the runtime overhead in these cases.

7.3 Support for Parallel Applications

In this section, we discuss monitoring systems that provide support for parallel applications. Software and hardware approaches to dynamic monitoring are discussed in Section 7.3.1 and Section 7.3.2, respectively, while Section 7.3.3 is about the most closely related work on deterministic record and replay systems.

7.3.1 Software Approaches

DBI-based monitoring tools for parallel applications either time-slice the application execution and perform the monitoring analysis on a single core (e.g., Valgrind [81]), or require explicit synchronization similar to any multi-threaded application (e.g., PIN [69]). In both cases, the resulting overhead is high, thus limiting practicality and potentially masking bugs (because applications execution may be perturbed)**.**

To allow for concurrent execution of translated multi-threaded programs, Chung et al. [27] proposed the use of transactional memory along with Dynamic Binary Instrumentation. By using memory transactions to enclose the data and the associated metadata in a single atomic block, this approach detects and corrects potential races, thus ensuring correctness.

7.3.2 Hardware Approaches

Early proposal that extend the processors pipeline with a monitoring pipeline (as described in Section 7.2.1), process the metadata along with the application data in the processor's pipeline [31, 32, 103]. As data and metadata updates happen atomically no support is required to order the metadata accesses with respect to application accesses.

However, when the monitor is *decoupled* from the application's pipeline, the atomicity of the data and the associated metadata is broken. The decoupling degree differs as decoupled monitoring architectures range from FlexiTaint [107], where the monitor is implemented in a few extra pipeline stages, to our work, where the monitor runs on a separate HW thread, and LBA [23], where the monitor runs on a separate core.

In FlexiTaint, application instructions are not committed until the associated metadata accesses are performed, thus executing the data and metadata accesses atomically. However, this approach does not work for highly decoupled monitoring architectures, such as LBA [23], FADE

[44], Decoupled Co-processor [57], that have to faithfully replay applications activity in order to ensure correct monitoring.

ParaLog [112] is a CMP-based monitoring system where N application threads are being monitored by an equal number of monitoring threads, occupying 2N cores in total. ParaLog leverages coherence activity to recreate applications ordering at the monitor's side under sequential consistency and total-store-order. As memory references are further reorder under more relaxed memory models, the monitoring process may deadlock without additional monitoring support. Resolve [111] extends ParaLog to perform correct monitoring under more relaxed memory models by proposing an algorithm that detects and resolves dependence cycles resulting from the non-SC memory accesses.

Kannan [56] also leverages cache coherence to infer information related to the application inter-thread order. However, in contrast to ParaLog that records the dependences at the cores running the application threads, this system records dependences at the cores running the monitor threads. In doing so, it maintains two tables at each monitoring core: (1) a table recording the misses of the monitored application core, and (2) a table recording the invalidations served by the monitored application core. The tables record not only the address but also the application instruction associated with a coherence event. As this approach requires an associative search of the communication queue (between the application and the monitor cores) to identify the instruction associated with a coherence event, it cannot be applied to systems with large queues (e.g., 64K entries in LBA [23]).

Butterfly [45] is a monitoring framework for dynamic analysis that does not require to record inter-thread dependences, thus avoiding the associated hardware extensions and allowing monitoring under any memory consistency model. Butterfly builds on the observation that instructions executed in the distant past by other threads must have been completed after a certain period of time, given the finite buffering of instructions and memory accesses in modern pipelines. At the

same time, the relative order of concurrent instructions (i.e., instructions in the near past or near future) is considered unknown. As a result, Butterfly breaks down the monitored application into epochs, where the events of adjacent epochs are conservatively considered to be concurrent, while the events of any other epoch are considered to be properly ordered. This approach may result in false positives (increasing in number with the epoch's size) and incurs high slowdown.

7.3.3 Deterministic Record & Replay

Frameworks in this category record application or system-level activity over an execution window so as to deterministically replay this window off line, if the application crashes. Although, there has been a large body of work in this domain, we focus on the most closely related work.

Flight Data Recorder [118] (FDR) is a hardware-assisted debugging system for post-mortem analysis (i.e., after an application crashes). FDR logs thread-ordering information continuously to allows for full-system deterministic replay of multi-threaded applications running on cache-coherent multi-processors under Sequential Consistency. To infer the order of concurrent events, FDR piggybacks the necessary information on cache coherence messages. Regulated Transitive Reduction (RTR) [120] extends FDR (1) by reducing the hardware cost and the rate of the log size growth, and (2) by supporting a more relaxed memory model (Total Store Order).

BugNet [77] is another post-mortem analysis system that records information related to applications and the linked libraries but does not allow for full-system replay, as FDR and RTR. BugNet is based on the insight that checkpointing the register file contents at any point in time, and then recording subsequent load values, allows for deterministic replay of programs execution.

Chapter 8

Conclusions

This thesis proposed FADE, a Filtering Accelerator for Decoupled Event monitoring, that enables the design of efficient general monitoring systems. FADE comes to bridge the gap between prior software-only and hardware-assisted approaches through monitor-agnostic extensions that significantly reduce the monitoring slowdown for a wide spectrum of bug-finding tools, ranging from memory bugs to atomicity violations.

In Chapter 3, we showed that FADE exploits common monitoring behavior to provide simple, programmable hardware logic to accelerate bug finding. For frequently occurring instruction events, FADE takes advantage of the fact that most of the time applications behave as expected (i.e., invariant checks succeed) and that the monitor's metadata do not need to be updated (i.e., most updates are redundant). These observations allow the vast majority of the costly software handlers that are invoked in response to instruction events to be filtered out. Unlike prior work that does not monitor stack-update events, FADE performs bulk metadata updates in simple hardware to accelerate these events, which constitute up to 17% of the monitors execution time. While FADE filters 84-99% of software handlers, it maintains full flexibility and generality by supporting software handler execution for the rest of the events. Additionally, we showed that by executing software handlers for unfiltered events in a dedicated hardware context alongside the application, our proposed design approaches the performance of a separate monitoring core (with FADE extensions) without the associated resource overheads.

In Chapter 4, we discussed event management in a monitoring system with filtering support. Our study of a broad range of applications and monitors showed that the monitoring load rarely

exceeds one event per cycle even with an aggressive OoO core producing the event stream. Event production is bursty, mandating queueing for pending events; however, a small queue is sufficient for good performance. Unfiltered events are also bursty and are sparsely spaced within an otherwise filterable event stream. These results pointed to a decoupled filtering accelerator able to keep up with an average monitoring load of one event per cycle, capable of filtering concurrently with unfiltered event processing, and loosely coupled through shallow queues to the application and the monitor.

In Chapter 5, we proposed Non-Blocking Filtering a novel technique that enables filtering to happen concurrently with the processing of unfiltered events, a task that is complicated due to dependencies between unfilterable and subsequent filterable events. To decouple filtering and the processing of the unfiltered events, we showed that there is only minimal state that is critical for deciding whether a dependent event is filterable. Most importantly, this state can be updated for unfilterable events directly in the accelerator with simple hardware extensions.

Based on these observations, we proposed FADE's pipelined microarchitecture with a peak filtering rate of one application event per cycle. Our design supports Non-Blocking Filtering that dynamically resolves dependencies between unfilterable events and subsequent events, eliminating data-dependent stalls and maximizing accelerator's performance. Using full-system cycle-accurate simulation, we showed that FADE is highly efficient, filtering out 84-99% of events that would otherwise be handled in software, thereby reducing the application slowdown to only 1.2-1.8x (versus 1.6-7.4x for unaccelerated execution). In the 40nm technology, FADE requires 0.12mm^2 of area and 273mW of power at peak.

As parallel applications constitute a large fraction of modern software, in Chapter 6, we introduced a parallel monitoring accelerator, Parallel FADE, that allows for the design of fast, flexible and resource-efficient parallel monitoring systems. Specifically, we studied parallel applications running on multiple cores, on a CMP-based monitoring system, where each core is equipped

with an instance of Parallel FADE. Our framework combines Non-Blocking Filtering, the state-of-the-art hardware filtering approach to accelerate monitoring, with the necessary hardware extensions to handle the inherent concurrency of parallel applications, proposed in prior work, so as to reduce the complexity of prior parallel monitoring systems. Using a suite of diverse monitors and a number of multi-threaded benchmarks, we showed that Parallel FADE filters 81-99% of the events that would otherwise be handled in software and reduces the slowdown to an average of only 1.1-1.8x (versus 1.9-11.5x for unaccelerated execution), thus making parallel monitoring practical.

Overall, FADE and Parallel FADE can assist developers to detect a wide range of bugs at low resource and runtime overhead, while significantly reducing the associated design complexity.

8.1 Future Directions

While instruction-grain monitoring provides broad coverage by monitoring every instruction, the monitoring slowdown can be reduced by tailoring the monitoring process to the users' needs with software assistance. Program developers usually have a good understanding of their code, and can further assist monitoring by providing hints on what exactly needs to be monitored, thus avoiding the monitoring overhead associated with the whole program. For instance, software hints could be leveraged to narrow the focus of the monitoring analysis either to specific data structures, or to specific functions.

FADE is based on the observation that there is common behavior that (1) requires a minimal set of actions, and (2) can be identified through minimal state. As these observations are general enough, they could be applied to other tagged memory systems. Potential use cases include but are not limited to security, reliability, performance bugs, garbage collection, and transactional memory.

Bibliography

[1] http://valgrind.org.

[2] IBM Rational Purify. http://www.ibm.com/software/awdtools/purify/, 2005.

[3] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’13, 2013.

[4] J. Arulraj, G. Jin, and S. Lu. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’14, 2014.

[5] A. Arvind and J.-W. Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA ’06, 2006.

[6] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI ’94, 1994.

[7] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-time Defense Against Stack Smashing Attacks. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’00, 2000.

[8] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’05, 2005.

[9] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5), October 2007.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, 2008.

[11] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, 2007.

[12] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent Dynamic Instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, 2012.

[13] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[14] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14(4), November 2000.

[15] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, 2011.

[16] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Transactions on Computer Systems*, 30(4), November 2012.

[17] M. Burrows, S. N. Freund, and J. L. Wiener. Run-time type checking for binary programs. In

International Conference on Compiler Construction, CC 2003, 2003.

[18] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[19] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, 2004.

[20] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[21] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, 2007.

[22] S. Chen, M. Kozuch, P. B. Gibbons, M. Ryan, T. Strigkos, T. C. Mowry, O. Ruwase, E. Vlachos, B. Falsafi, and V. Ramachandran. Flexible Hardware Acceleration for Instruction-Grain Lifeguards. *IEEE Micro*, 29(1), 2009.

[23] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, 2008.

[24] S. Chen, B. Lin, S. W. Schlosser, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, and G. R. Ganger. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[25] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[26] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, 2001.

[27] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-Safe Dynamic Binary Translation Using Transactional Memory. In *Proceedings of the 18th International Symposium on High-Performance Computer Architecture*, HPCA '08, 2008.

[28] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the Real World. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, 2003.

[29] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, 2003.

[30] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, 1998.

[31] J. Crandall and F. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, 2004.

[32] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Com-*

puter Architecture, ISCA ’07, 2007.

[33] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’43, 2010.

[34] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET’s with very small physical dimensions. In *IEEE Journal of Solid-State Circuits*, 1974.

[35] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 2008.

[36] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-On Sound and Complete Race Detection in Software and Hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA ’12, 2012.

[37] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’07, 2007.

[38] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP ’03, 2003.

[39] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE ’99, 1999.

[40] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, 2004.

[41] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, 2009.

[42] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[43] S. Fytraki, O. Kocberber, E. Vlachos, J. B. Sartor, B. Grot, and B. Falsafi. BugSifter: A Generalized Accelerator for Flexible Instruction-Grain Monitoring. Technical Report EPFL-REPORT-187154, EPFL, 2012.

[44] S. Fytraki, E. Vlachos, O. Kocberber, B. Falsafi, and B. Grot. FADE: A Programmable Filtering Accelerator for Instruction-Grain Monitoring. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[45] M. L. Goodstein, E. Vlachos, S. Chen, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Butterfly Analysis: Adapting Dataflow Analysis to Dynamic Parallel Monitoring. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[46] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. M. Austin. Demand-driven Software Race Detection Using Hardware Performance Counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, 2011.

[47] J. L. Greathouse, I. Wagner, D. A. Ramos, G. Bhatnagar, T. Austin, V. Bertacco, and S. Pettie. Testudo: Heavyweight Security Analysis via Statistical Sampling. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41,

2008.

[48] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin. A Case for Unlimited Watchpoints. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[49] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic Inference of Abstract Types. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, 2006.

[50] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, 2002.

[51] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter 1992 USENIX Conference*, 1991.

[52] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[53] J. A. Joao, O. Mutlu, and Y. N. Patt. Flexible Reference-counting-based Hardware Acceleration for Garbage Collection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[54] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, 1997.

[55] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[56] H. Kannan. Ordering Decoupled Metadata Accesses in Multiprocessors. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[57] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '09, 2009.

[58] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[59] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch Regulation: Low-overhead Protection from Code Reuse Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, 2012.

[60] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communication ACM*, 21(7), July 1978.

[61] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation - Volume 6*, OSDI'04, 2004.

[62] A. Loginov, S. H. Yong, S. Horwitz, and T. W. Reps. Debugging via Run-Time Type Checking. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, FASE 2001, 2001.

[63] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007.

[64] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on

Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 2008.

[65] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems,* ASPLOS XII, 2006.

[66] B. Lucia and L. Ceze. Finding Concurrency Bugs with Context-aware Communication Graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[67] B. Lucia, L. Ceze, and K. Strauss. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-variable Atomicity Violations. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[68] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, 2008.

[69] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[70] J. Maebe, M. Ronsse, and K. D. Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Compendium of Workshops and Tutorials Held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques, 2002*, 2002.

[71] J. Maebe, M. Ronsse, and K. D. Bosschere. Precise detection of memory leaks. In *Proceedings of the 2nd International Workshop on Dynamic Analysis*, 2004.

[72] G. E. Moore. Cramming more components onto integrated circuits. In *Electronics, 38(8)*, 1965.

[73] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, 2007.

[74] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[75] S. Mysore, B. Agrawal, N. Srivastava, S.-C. Lin, K. Banerjee, and T. Sherwood. Introspective 3D Chips. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, 2006.

[76] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, 2012.

[77] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, 2005.

[78] Naval Surface Warfare Center Computer Museum at Dahlgren Virginia.

[79] N. Nethercote. *Dynamic binary analysis and instrumentation*. PhD thesis, University of Cambridge, 2004.

[80] N. Nethercote and J. Fitzhardinge. Bounds-Checking Entire Programs Without Recompiling. In *the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, SPACE 2004, 2004.

[81] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary In-

strumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 2007.

[82] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, NDSS '05, 2005.

[83] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, 2008.

[84] Oracle. *The SPARC Architecture Manual: Version 9*. 2000.

[85] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri. SIFT: A Low-Overhead Dynamic Information Flow Tracking Architecture for SMT Processors. In *Proceedings of the 8th International Conference on Computing Frontiers*, CF'11, 2011.

[86] E. Pozniansky and A. Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, 2003.

[87] E. Pozniansky and A. Schuster. MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. *Concurrency and Computation: Practice and Experience - Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 19(3), March 2007.

[88] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, 2005.

[89] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th*

Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39, 2006.

[90] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing Dynamic Information Flow Tracking. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, 2008.

[91] O. Ruwase, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry. Guardrail: A High Fidelity Approach to Protecting Hardware Devices from Buggy Drivers. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, 2014.

[92] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, NDSS '04, 2004.

[93] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4), 1997.

[94] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and Reconfigurable Software Dynamic Translation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, 2003.

[95] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, 2009.

[96] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM Journal of Research and Development*, 50(2), 2006.

[97] W. Shi, H. Lee, L. Falk, and M. Ghosh. An Integrated Framework for Dependable and Re-

vivable Architectures Using Multicore Processors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, 2006.

[98] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I Use the Wrong Definition?: DeFuse: Definition-use Invariants for Detecting Concurrency and Sequential Bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, 2010.

[99] A. Shriraman and S. Dwarkadas. Sentry: Light-weight Auxiliary Memory Access Control. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[100] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-Effective Multicore Redundancy. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, 2006.

[101] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, 2008.

[102] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[103] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, 2004.

[104] The MITRE Corporation. Common vulnerabilities and exposures (CVE). http://cve.mitre.org.

[105] M. Tiwari, B. Agrawal, S. Mysore, J. Valamehr, and T. Sherwood. A Small Cache of Large

Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, 2008.

[106] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajani, and D. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, 2004.

[107] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation. In *Proceedings of the 14th IEEE International Symposium on High-Performance Computer Architecture*, HPCA '08, 2008.

[108] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, 2007.

[109] Virtutech simics. http://www.virtutech.com/.

[110] W. Visser, C. S. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, 2004.

[111] E. Vlachos, S. Fytraki, P. B. Gibbons, M. Kozuch, and B. Falsafi. Resolve: Enabling Accurate Parallel Monitoring under Relaxed Memory Models. EPFL Technical Report: EPFL-REPORT-197953, March 2014.

[112] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[113] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE ’07, 2007.

[114] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4), 2006.

[115] E. Witchel, J. Cates, and K. Asanovic. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, 2002.

[116] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA ’95, 1995.

[117] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA ’03, 2003.

[118] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA ’03, 2003.

[119] M. Xu, R. Bodik, and M. D. Hill. A Serializability Violation Detector for Shared-memory Server Programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’05, 2005.

[120] M. Xu, M. D. Hill, and R. Bodik. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, 2006.

[121] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2), April 1996.

[122] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, 2005.

[123] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

[124] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, 2004.

[125] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, 2007.

[126] P. Zhou, F. Uin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, 2004.

Curriculum Vitae

Sotiria Fytraki

RESEARCH BACKGROUND

My research centers around multiprocessor systems, with an emphasis on at-speed program monitoring and hardware support for robust software.

ACADEMIC BACKGROUND

Ph.D. in Computer Science (2008 – 2014)

École Polytechnique Fédérale de Lausanne (EPFL)

Title: “Architectural Support to Accelerate Fine-Grain Program Monitoring”

Advisor: Prof. Babak Falsafi

M.Sc. in Computer Engineering (2006 - 2008)

Technical University of Crete (TUC)

Title: “ReSim, a Trace-Driven, Reconfigurable ILP Processor Simulator”

Advisor: Prof. Dionisios Pnevmatikatos

Five-year Diploma in Electronics and Computer Engineering (2001 - 2006)

ECE Department, Technical University of Crete (TUC)

DISTINCTIONS

• Award from Technical Chamber of Greece, for being ranked third best in my year

• Award from Women’s Engineering Association, for being ranked first best woman graduate in my year

• Graduate Fellow, Technical University of Crete, Fall Semester 2006

• Diploma GPA: 8.75/10.0, Honors (third best in my year)

CONFERENCE PUBLICATIONS

• S. Fytraki, E. Vlachos, O. Kocberber, B. Falsafi, and B. Grot. “FADE: A Programmable Filtering Accelerator for Instruction-Grain Monitoring”. To appear in Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture, February 15-19, 2014, Orlando, Florida, USA.

• S. Fytraki and D. Pnevmatikatos. “ReSim, a trace-driven, Reconfigurable ILP Processor Simulator”. In Proceedings of the 10th Conference on Design, Automation & Test in Europe, April 20-24, 2009, Nice, France.

TECHNICAL REPORTS

• S. Fytraki, O. Kocberber, E. Vlachos, J. B. Sartor, B. Grot, and B. Falsafi. “BugSifter: A Generalized Accelerator for Flexible Instruction-Grain Monitoring”

• E. Vlachos, S. Fytraki, M. A. Kozuch, P. B. Gibbons and B. Falsafi. “Resolve: Enabling Accurate Parallel Monitoring under Relaxed Memory Models”

GRANTS

• Co-author of “At-Speed Program Monitoring (almost) for Free”. PI Babak Falsafi. Swiss National Science Foundation, 163K CHF ($174K USD), EPFL, Duration: 2012 - 2015.

WORKING EXPERIENCE

• Research Assistant, Parallel Systems Architecture Laboratory (PARSA), EPFL (2008 – present)

• Research Assistant, Microprocessor and Hardware Laboratory (MHL), TUC (2006 – 2008)

TEACHING EXPERIENCE

Teaching Assistant, EPFL

Computer Architecture I (Fall 2013)

Advanced Multiprocessor Architecture (Fall 2012)

Logic Systems II (Spring 2011)

Introduction to Multiprocessor Architecture (Spring 2010)

Teaching Assistant, Technical University of Crete (TUC)

Digital Design (Fall 2007)

Advanced Logic Design (Spring 2006)

Advanced Computer Architecture (Fall 2006)