Enhancing Trace Generation-based Software Debugging Infrastructure for Physical and Emulated Development Platforms

Bojan Mihajlović

Department of Electrical and Computer Engineering
McGill University
Montreal, Canada

December 2014

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

© 2014 Bojan Mihajlović
Acknowledgments

I am grateful to my supervisors, Željko Žilić and Warren Gross, for all of the mentorship and valuable advice they have provided over the years. I am glad to have had such a productive and collegial relationship with both of them, which afforded me both the freedom to explore different ideas, and provided me with useful feedback in the projects that I pursued. I also appreciate the generous hospitality of Ing-Jer Huang, who, by inviting me to visit his lab in 2011, allowed me to form ideas that would ultimately find their way into this thesis. I would also like to thank my thesis reviewers and committee members for their constructive criticism and comments.

Many thanks to Ben Nahill, Gabi Sarkis, Ashraf Suyyagh, and Jean-Samuel Chenard for taking the time to brainstorm various ideas with me at one time or another. Thank you to Luca Montesi, Steven Ding, James Grabowski, Jason Tong, François Leduc-Primeau, and Marwan Kanaan for all of the interesting conversations that we had during our downtime. I would like to extend a special thanks to Dimitrios Stamoulis, who was a great co-working companion over the course of my writing a large part of this thesis. I would also like to thank Alexandre Raymond and François Leduc-Primeau for providing the French translation of the thesis abstract, and Jasmina Vasiljević for contributing her thoughts on an important question.

I am especially grateful to the people who made a positive impression on me in my most formative years. I thank my parents for imparting upon me the importance of education, and for their unrelenting support of my life goals and career ambitions. I feel fortunate that my father exposed me to world of electronics and computing at an early age, which planted the seeds of interest in my chosen field of study. I owe a particular debt of gratitude to my aunt and uncle, Jelena and Zlatko, for their patience, understanding, and persistence during a time when I needed it the most. I am thankful for having known the late Dan Henderson, who made me believe that I belonged in the world of engineering. I am also grateful for the motivation provided by Branko Radišić at an important juncture in my life.

I would also like to thank Blagojče Pogačevski, whom I firmly believe contributed to a self-fulfilling prophesy by long ago nicknaming me “Doctor”. But most importantly, I would like to thank Sandra Medakovich for her support, patience, and sacrifice throughout the period of our lives in which I was pursuing this degree. I am both grateful and privileged to have had her by my side during this time.
Abstract

Software debugging is now widely reported to constitute the majority of software development time and cost, largely due to the effects of continuously rising software complexity. In a trend which is expected to continue, complex software faults that can require weeks to resolve are becoming increasingly commonplace. Since traditional debugging methods are considered unsuitable for resolving such faults, trace generation is being recognized as a solution to future debugging needs. This thesis presents trace generation infrastructure that includes advanced on-chip supporting hardware, complementary software tools, and interfaces to bridge the gap between them. The contributions enable or enhance the use of trace generation across a variety of software development platforms.

Even though embedded software development is increasingly performed on software emulators, many existing emulators lack trace generation capabilities. This thesis provides the ubiquitous QEMU emulator with the ability to perform trace experiments. The described extensions enable continuous instruction-level trace generation to be controlled using a standard software debugger client, which is given the ability to create tracepoints to dynamically log registers and memory addresses. The infrastructure is made aware of operating system context-switching in a unique way, which allows traces to be collected in five different modes: from all executed code, down to a single Linux process.

For hardware-based development platforms, the volume of on-chip trace data generated in real-time can exceed the ability to practically transfer or store it. This thesis presents two different schemes for the compression of execution traces, which are vital in establishing the flow of software execution post-hoc. The Architecture-Aware Trace Compression (AATC) scheme eliminates previously unidentified redundancies within typical execution traces using a combination of on-chip predictors, transforms, and encoders. The scheme achieves the highest-performance compression of any similar method, most notably by exploiting the widespread use of linked branches, as well as the compiler-driven movement of return addresses between link-register, stack, and program counter. The Multi-Architectural Trace Compression (MATC) scheme is also introduced to compress the execution traces of any fixed instruction-width soft-core processor deployed on Field-Programmable Gate Array (FPGA) platforms. A novel architecture is presented in which five parameterizable variants of a pipelined scheme allow different combinations of unused logic and memory resources to be repurposed for trace compression purposes.
Abrégé

Le déboguage est désormais largement reconnu comme la tâche coûtant le plus cher et prenant le plus de temps de développement dans le contexte de projets logiciels. Cela est fortement lié à l’augmentation de la complexité de ce type de projet. Il arrive de plus en plus souvent que la correction de défauts logiciels complexes prenne plusieurs semaines, et cette tendance s’accentue avec le temps. Puisque les méthodes de déboguage traditionnelles ne permettent plus de résoudre ces défauts, de nouvelles méthodes comme la génération de traces sont proposées pour la résolution des bogues modernes. Cette thèse propose une infrastructure de génération de traces incluant une portion matérielle avancée, des outils logiciels complémentaires, ainsi que des interfaces servant à lier ces deux éléments. Ces contributions permettent ou améliorent l’utilisation de la génération de traces à travers diverses plates-formes de développement logiciel.

Malgré que le développement de logiciels embarqués soit de plus en plus effectué à l’aide de logiciels d’émulation, ces logiciels sont souvent incapables d’effectuer la génération de traces. Cette thèse présente un ajout à l’émulateur QEMU permettant la génération de traces. Les extensions proposées permettent de générer des traces d’instructions en continu, et ce à l’aide d’un débogueur logiciel standard. Ce débogueur est doté de la capacité de créer des points de traçage pour enregistrer les registres et les adresses mémoire de façon dynamique. Cette infrastructure est de plus prévenue des changements de contexte effectués par le système d’exploitation grâce à une approche unique qui permet de récolter les traces selon cinq modes différents (soit pour tout le code exécuté, soit de façon graduellement plus précise jusqu’à ne retenir qu’un seul processus Linux).

Pour les plates-formes de développement matériel, le volume de données associé aux traces générées sur la puce en temps réel peut dépasser la capacité de transfert ou d’enregistrement disponible. Il est donc essentiel de compresser ces données pour permettre de reconstituer le fil de l’exécution du logiciel après coup. Deux approches sont présentées pour la compression des traces d’exécution. L’approche de Compression de Traces Basée sur l’Architecture (AATC) élimine certaines redondances qui étaient auparavant ignorées au sein de traces d’exécution typiques grâce à une combinaison de prédicteurs matériels, de transformées et d’encodeurs. Cette méthode permet d’obtenir une compression supérieure à toutes les méthodes comparables, notamment en tirant profit de l’ubiquité des points de branchage liés ainsi que du déplacement des adresses-retour entre le registre-retour, la pile,
et le compteur d’instructions. La seconde approche appelée Compression de Traces Multi-Architecture (MATC) permet quant à elle de compresser les traces d’exécution de tout processeur *soft-core* utilisant des instructions à taille fixe et déployé sur un circuit logique programmable (FPGA). L’architecture proposée comporte cinq variantes qui permettent divers niveaux de réappropriation des circuits logiques et des ressources mémoire inutilisés pour la compression de traces.
Contents

1 Introduction 1
  1.1 Software Debugging with Trace Generation .............................. 5
  1.2 Summary of Thesis Contributions ........................................ 9
  1.3 Self-Citations .......................................................... 11
    1.3.1 Related work ....................................................... 13
  1.4 Thesis Organization .................................................... 14

2 Background and Related Work 17
  2.1 Software Development ................................................... 17
    2.1.1 Models ............................................................ 18
    2.1.2 Practices ........................................................ 18
    2.1.3 Platforms ......................................................... 19
    2.1.4 Parallelism ....................................................... 21
  2.2 Software Debugging ...................................................... 21
    2.2.1 Debugger Programs ............................................... 22
    2.2.2 Fault types ....................................................... 23
    2.2.3 Multi-threaded Software ......................................... 24
  2.3 Traditional Software Debugging Methods for SoC Platforms .......... 25
2.3.1 Software Instrumentation .............................. 26
2.3.2 Scan-chain Methods .................................. 26
2.3.3 In-Circuit Emulation ................................. 28
2.4 Software Debugging Methods for Alternative Platforms .... 31
  2.4.1 Software-Emulated Platforms ...................... 31
  2.4.2 Hardware-Emulated Platforms ...................... 33
2.5 Debugging with Trace Generation ...................... 35
  2.5.1 Software Instrumentation ......................... 37
  2.5.2 Hardware Instrumentation ......................... 37
  2.5.3 Emulated Hardware Instrumentation ............... 38
  2.5.4 Deterministic Replay ............................. 39
  2.5.5 Triggers .......................................... 40
  2.5.6 Debugging Interfaces .............................. 42
  2.5.7 Trace Ordering ................................... 42
  2.5.8 Trace Qualification ............................... 43
  2.5.9 Trace Storage .................................... 44
  2.5.10 Trace Data Volume ............................... 46
2.6 Trace Compression .................................... 48
  2.6.1 Real-time Trace Compression ....................... 50
2.7 Generalized Software Debugging Procedure ............. 52

3 Dynamically Instrumenting the QEMU Emulator ......... 55
  3.1 Introduction ........................................ 55
    3.1.1 Contributions of this Chapter .................. 56
  3.2 The QEMU Emulator ................................ 57
    3.2.1 Instruction Translation .......................... 57
    3.2.2 Dynamic Binary Instrumentation ................. 58
  3.3 Overview of Proposed Method ........................ 60
  3.4 Modes of Operation ................................ 61
  3.5 Trace experiment control ........................... 63
    3.5.1 RSP Trace Packets .............................. 63
    3.5.2 Example Trace Experiment ....................... 65
    3.5.3 Avoiding “Memory Abort” Exceptions ............. 66
3.6 Experimental Results .................................................. 68
3.6.1 Execution Time .................................................. 68
3.6.2 Trace Volume .................................................. 71
3.7 Chapter Summary .................................................. 72

4 Architecture-Aware Real-Time Compression of ARM Execution Traces 75
4.1 Introduction .................................................. 75
4.1.1 Contributions of this chapter ........................................ 77
4.2 Architecture-Aware Trace Compression Overview .................. 78
4.2.1 Usage Scenario .................................................. 79
4.2.2 Tracking Stage .................................................. 80
4.2.3 Encoding Stage .................................................. 81
4.2.4 Limitations .................................................. 81
4.3 Branch Counters .................................................. 82
4.3.1 Direct Unconditional Branches ................................... 83
4.3.2 Direct Conditional Branches ..................................... 83
4.3.3 Indirect Unconditional Branches .................................. 84
4.3.4 Indirect Conditional Branches ..................................... 84
4.4 Link-Register and Stack Tracking ................................... 85
4.4.1 Link-Register Tracking ........................................... 86
4.4.2 Tracking Return Address Movement to/from the Stack ........ 88
4.4.3 Example: Link-Register Tracking and Stack Tracking ........ 90
4.5 Encoding Modules .................................................. 92
4.5.1 Fibonacci Encoder .............................................. 94
4.5.2 Differential BTA .............................................. 98
4.5.3 VarLen Bijection Encoder ....................................... 98
4.5.4 Move-to-Front Transform ........................................ 101
4.6 Experimental Results .............................................. 101
4.6.1 BP and BTB Performance ......................................... 102
4.6.2 VarLen Bijection Encoding Parameters ......................... 104
4.6.3 Compression Performance ......................................... 104
4.6.4 Hardware Area .............................................. 107
4.7 Chapter Summary .................................................. 108
5 Multi-Architectural Soft-Processor Execution Trace Compression 111
  5.1 Introduction .................................................. 111
    5.1.1 Contributions of this Chapter .......................... 113
  5.2 Multi-Architectural Trace Compression Overview .............. 114
    5.2.1 Usage Scenario .......................................... 115
  5.3 MATC Compression Stages ................................... 117
    5.3.1 Consecutive Address Elimination ....................... 117
    5.3.2 Finite Context Method ................................ 118
    5.3.3 Move-to-Front & Address Encoding ..................... 121
    5.3.4 Data Stream Serializer ................................ 123
    5.3.5 Run-length and Prefix Encoding ....................... 124
    5.3.6 Lempel-Ziv Encoding of the Data Stream ............... 126
  5.4 Experimental Results ........................................ 128
    5.4.1 Compression Performance ................................ 128
    5.4.2 FPGA Resource Utilization ............................. 131
  5.5 Chapter Summary .............................................. 133

6 Conclusions and Future Work 135
  6.1 Future Work .................................................. 137

Bibliography 141

List of Acronyms 154
List of Figures

2.1 Remote Debugging Scenario – Software View ............................................. 23
2.2 Execution-control software debugging of a physical SoC with single processor core through an IEEE 1149.1 (JTAG) TAP interface .......................... 28
2.3 Execution-control software debugging of a physical SoC with multiple processor cores through an IEEE 1149.1 (JTAG) TAP interface .................. 29
2.4 Execution-control software debugging of a physical SoC with single processor core through In-Circuit Emulation (ICE) ................................. 30
2.5 Execution-control software debugging of a QEMU-emulated guest processor through a virtual network interface ........................................ 33
2.6 Execution-control software debugging of an FPGA-emulated SoC with single soft-processor core through an IEEE 1149.1 (JTAG) TAP interface .... 34
2.7 Software debugging interface with both trace generation and ICE capabilities 36

3.1 Example of QEMU Dynamic Binary Instruction Translation from ARM to x86 ISAs ................................................................. 57
3.2 Functional Overview of QEMU Trace Generation ................................. 59
3.3 Memory-mapping of Linux Process IDs to ReservedAddr ..................... 64
3.4 Number of Instructions Captured by Qualification Level while Recording Execution Traces of the MiBench Benchmark Suite ...................... 73

4.1 Generalized Compression Procedure ..................................................... 77
4.2 Debugging Scenario ............................................................................. 79
4.3 AATC Tracking Module ....................................................................... 82
4.4 Branch target address volume reduction when applying Link-Register Tracking and Stack Tracking to an unweighted execution trace of the MiBench benchmark suite .................................................. 88
4.5 fAATC Encoding Module ................................................................. 92
4.6 xAATC Encoding Module ................................................................. 93
4.7 Value frequency vs. binary-encoded size of aggregated $bCnt$ and $iCnt$ symbols 94
4.8 Value frequency vs. binary-encoded size of (a) BTA and (b) differential BTA 95
4.9 Autocorrelation plot of BTA symbols and aggregated $bCnt$ and $iCnt$ symbols 96
4.10 Comparison of encoding schemes on the size of $bCnt$ and $iCnt$ symbols for an unweighted execution trace of the MiBench benchmark suite .......... 97
4.11 Comparison of encoding schemes on the size of differential BTA symbols for an unweighted execution trace of the MiBench benchmark suite .......... 99

5.1 Five-Stage Pipelined MATC Compression Scheme .......................... 115
5.2 Trace-based Debugging Scenario ..................................................... 116
5.3 Finite Context Method Implementation in MATC Variants $v0$ and $v1$ ... 121
5.4 Example of Move-to-Front array element matching an input symbol ...... 122
5.5 Run-Length & Prefix Encoding Compression Stage – Huffman Tree for Prefix Encoding ................................................................. 127
List of Tables

3.1 Modes of Operation ................................................................. 63
3.2 Execution Trace Collection Time for the MiBench Benchmark Suite on a QEMU ARM Guest Machine .................................................. 71
4.1 BP and BTB event handling ......................................................... 85
4.2 BP prediction correct or BTB hit .................................................. 85
4.3 Link-Register Tracking Block: Instruction Tracking Rules for the ARM ISA ................................................................. 87
4.4 Stack Tracking Block: Instruction Tracking Rules for the ARM ISA ................................................................. 88
4.5 Link-Register Tracking and Stack Tracking Example ......................... 91
4.6 Example of the Fibonacci encoding of decimal symbol 130 ............... 98
4.7 Example of negative-bit and bijection representations of negative numbers ................................................................. 100
4.8 Rates of Correct BP Predictions and BTB Hits ................................. 103
4.9 Software-based Execution-Trace Compression Scheme Performance Comparison with AATC (bits/instruction) ......................... 105
4.10 Hardware-based Execution-Trace Compression Scheme Performance Comparison with AATC (bits/instruction) ......................... 106
4.11 Comparison of Logic Utilization and Average Compression Performance ................................................................. 107
5.1 Example of Consecutive Address Elimination with 16-bit Addresses .......... 118
5.2 MTF/AE Compression Stage – 32-bit Address Encoding into Prefix and Data Streams ................................................................. 124
5.3 Example of Differential Address Encoding – 16-bit Addresses ................ 125
5.4 Performance of Software-based General-Purpose Compression Schemes in Compressing Execution Traces (bits/instruction) ......................... 129
5.5 Hardware-based Execution-Trace Compression Scheme Performance Comparison with MATC (bits/instruction) .......................... 130
5.6 MATC Compression Scheme Logic Utilization on an Altera Stratix III FPGA132
List of Listings

2.1 Example of Tracepoint Creation in the GDB Debugger ............... 41
3.1 Example of GDB Trace Experiment Configuration with Mode 4 PID Filter 66
3.2 Example of ASCII Output of GDB Trace Experiment from Listing 3.1 . . 67
Chapter 1

Introduction

In the past several decades, the mechanical systems that have served the fundamental roles in society have gradually been augmented or supplanted by software-based ones. As a society, we are increasingly reliant upon software systems to manage our communication, transportation, finance, health, and electric distribution systems. In fact, it is difficult to find facets of our lives that have not been permeated by some type of software. That can partly be attributed to the overwhelming benefits of software-based systems over mechanical ones. The ability to make precise calculations, rapid decisions, and enable ever more complex procedures has driven the development of software in practically every domain.

However, current events are filled with examples of prominent software failures. As we become increasingly reliant upon software systems, the consequences of software failures can be both far-reaching, and at times severe. In the summer of 2003, a software bug caused a race condition in the electrical distribution system of FirstEnergy Corporation [1]. That bug triggered an electrical blackout in northeastern North America that left approximately 50 million people without power for two days, contributed to 11 deaths, and cost an estimated $6 billion [2].

Software bugs are also finding their way into safety-critical systems previously dominated by mechanical systems. In 2013, some Toyota vehicles experienced unintended acceleration, causing collisions that resulted in at least one human death [3]. The problem was blamed on faulty firmware in the Engine Control Units (ECUs) of some vehicles, which was primarily attributed to a stack overflow. A thorough analysis of the vehicle software later revealed several other problems, including buffer overflows, unsafe typecasting, and
race conditions [3].

It can be assumed that software with such potentially catastrophic bugs is not intended to be used in critical portions of our infrastructure. The contributing factors that have allowed such failures to occur have been studied in the past, and continue to be studied [4]. It has been suggested that many of the factors can be controlled and addressed through formalized approaches to software development and project management [5]. Failings due to poor project management, sloppy development practices, and unarticulated project goals can be rectified with solutions based upon the more rigorous practices employed in other domains. However, some causes are far more difficult to manage. The solution to an “inability to handle the project’s complexity” [5] is far from straightforward.

The nature of software failure is also one that does not necessarily mirror the mechanical system it is intended to replace. A failing “analog” system can give clues as to its status, often diminishing its performance over time in order to signal the need for replacement or repair. However, a complex software project can appear to operate correctly until a specific set of circumstances is encountered that unmasks a software bug, sometimes resulting in catastrophic failure. Since many software systems lack the conception of diminished capacity, their operational state is a binary one: either operating correctly or not.

Even high-stakes software projects are not immune to failures arising from software complexity. The software being built to operate the F-35 Lightning II aircraft is a prime example. For the F-35 to become fully operational, software consisting of an estimated 24 million lines of code will need to be developed, debugged, and tested [6]. Repeated delays in meeting deadlines have primarily been attributed to “the need to fix problems and retest multiple software versions” [7], the added cost of which has been estimated at billions of dollars per year.

The overall problem of complexity within software development is well articulated in the following quote from Michael R. Lyu [8]:

“The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test, and maintain them... It is the integrating potential of software that has allowed designers to contemplate more ambitious systems encompassing a broader and more multidisciplinary scope, and it is the growth in utilization of software components that is largely responsible for the high overall complexity of many system designs.”
Several factors have conspired to create an environment in which managing software complexity is especially difficult. In an effort to reap the rewards of creating increasingly capable software, software development has been in a constant state of flux for many decades. The programming languages, development platforms, and debugging methods upon which software development in based have consistently changed, at times quite rapidly and dramatically. It is not uncommon for a modern software project to mix the code of several programming languages, developed using both object-oriented and algorithmic-oriented programming methods, written by multiple developers at different times and to varying standards. Consistent change in the industry has precluded the establishing of certain standards and best practices that are common in other domains.

The availability of increasingly powerful hardware platforms can be considered especially responsible for problems of software complexity. While microprocessors costing tens of dollars can today execute billions of operations per second, creating software that fully exploits those capabilities to reliably carry out a set of complex tasks can cost millions of dollars. The increasingly complex software projects of recent years demonstrate that with sufficient resources, creating complex software is not necessarily an impossible task. Modern software development tools include features which are intended to increase developer speed and efficiency, such as the auto-completing of source code as a developer types, or by making available large libraries of code for immediate inclusion into a project. To complete large projects under tight time frames, modern software development is almost inevitably a collaborative endeavour. To manage a large software project involving many developers, there are a myriad of tools that can organize software development into manageable blocks which can be assigned to individual developers. Revision control systems can then track who is responsible for a given block.

While there are many tools to aid the creation of large software projects, such software can be inherently more difficult to debug and test than comparatively smaller projects. A large increase in the number of possible execution paths and conceivable interactions between software components is partly responsible for that difficulty. As a result, software complexity has reached a point where the elimination of every software bug from a system is rarely attempted, since the cost of doing so is considered prohibitive [9]. Competitive pressures to develop and ship products in tight time frames can also impact when software is considered “complete”. Many software projects take the approach of fixing bugs that have the greatest impact on overall functionality, opting to resolve major bugs that are
discovered after software deployment through patching. However, certain types of software require that all bugs be resolved before deployment to ensure the safety of human beings or infrastructure, even though doing so incurs a high marginal cost [10]. It is thought that “some level of software failure will always be with us... but too many of the failures that occur today are avoidable” [5]. Prominent software failures show that the time, cost, and difficulty of debugging and testing complex software can be overwhelming or intractable, even for large organizations or governments. It is apparent that when considering the use of modern hardware platform capabilities in combination with modern software development practices, our ability to manage the inherent complexity of creating reliable software has already been surpassed.

The difficulty of debugging complex software is exacerbated by dramatic changes that have been introduced to a software development model that has been used for decades. The deployment platforms that typically execute software are composed of an increasing number of cores and must choreograph the execution of multiple software threads to achieve their objectives. Multi-threaded software development is becoming more common and is expected to become even more necessary as deployment platforms become massively parallel in order to increase overall computational efficiency and reduce overall power consumption. The coordination of large numbers of interdependent and concurrently executing software threads adds to the problems of software complexity that are faced today.

The time and cost of debugging software has risen over the past few decades, and some estimates now place the cost as high as 50–75% of the overall cost of software development [11]. That can partly be attributed to the fact that some complex bugs require days or weeks to diagnose and resolve [12]. One of the reasons that the debugging of complex software is becoming increasingly difficult is that traditional software debugging methods are ill-adapted to the debugging of such software. For many decades, execution control-based software debugging has been used to halt software execution at an opportune point, and to observe internal data structures before resuming software execution. However, complex software systems contain an increasing number of software faults that appear to manifest non-deterministically due to complex run-time environments. In many cases, halting software execution causes changes to software timing that may mask software faults by preventing their manifestation. Faults that manifest intermittently or infrequently can also be difficult to observe using execution-control methods. That is especially true of multi-threaded software, where a fault may depend on the order of execution between
1.1 Software Debugging with Trace Generation

Software threads, which may differ every time the software is invoked. For that reason, complex software faults are expected to become even more prevalent as the development of multi-threaded software increases.

Software debugging methods that are suitable for detecting all types of software faults have been proposed as alternatives to traditional debugging methods. The leading solutions are known as trace generation-based schemes, which have seen increased research, development, and use in recent years. There are still many obstacles to the widespread adoption of trace generation that must be addressed in order to consider those schemes practical for the debugging of current and future software. This thesis contributes to solving some of the fundamental issues that stand in the way of employing trace generation-based software debugging infrastructure on a variety of software development platforms. Those contributions either enable the otherwise intractable task of debugging some types of complex software, or decrease the time and cost of doing so through other means. The ultimate goal of such work is to give a software developer the ability to create reliable software in the face of increased complexity, in a reasonable amount of time, and at reasonable cost.

1.1 Software Debugging with Trace Generation

Trace generation has been used for decades to debug real-time software, which is a type of software that risks missing deadlines and generating faults if its execution is halted. It works by generating logs of debug data, known as traces, in real-time without interrupting software execution. The method has seen renewed interest as a solution to the debugging of other types of software that are intolerant of having their execution halted, including complex software and multi-threaded software. As a result, modern Systems-on-Chip (SoCs) are increasingly including on-chip trace generation hardware to support software development and debugging. Companies such as ARM have consistently expanded their debugging and trace generation Intellectual Property (IP) block offerings over the years, and many SoCs are devoting significant amounts of hardware resources to such blocks. A variety of software tools are also being actively developed to perform trace experiments, collect different types of traces, and contextualize traces to aid the software debugging process. Recent advances are helping to adapt trace generation from a niche technology into a tool for mainstream software debugging, one which helps to reduce the time and cost of debugging complex software. However, there are several outstanding problems that must
be addressed before trace generation can be applied to the range of software development platforms typically employed by developers. The following is a description of some of the outstanding problems and how they are addressed by this thesis:

1. *Excessive trace data volume*

   Hardware-based software development platforms commonly store generated traces in small on-chip buffers, whose data must be transferred off-chip after the trace experiment has completed. Since such small buffers are quickly filled by trace data, trace collection is limited to coarse-grained and infrequently-occurring events, or to fine-grained trace collection over very small periods of execution. Neither of those solutions are adequate for observing some complex software faults that manifest infrequently and require fine-grained trace collection over long periods of execution. Alternatively, continuous trace collection requires that trace data be transferred off-chip in real-time. However, generating traces of even relatively few data structures can result in a staggering volume of trace data, requiring an expensive and vast trace storage array, and high-bandwidth trace port. To enable the full capabilities of trace generation to be used for debugging current and future software faults, various approaches aim to compress trace data before it is streamed off-chip. Chapters 4 and 5 introduce trace compression schemes for what is arguably the most useful type of trace that can be collected, one which reveals the path of execution taken through the software-under-debug. The schemes target different software development platforms, and achieve superior trace compression rates for their respective platforms. Such high-performance trace compression is expected to become increasingly more important as complex software faults requiring fine-grained trace collection become more prevalent, and as the number of on-chip cores grows in comparison to trace port bandwidth capabilities.

2. *Fine-grained trace generation*

   A preponderance of trace generation tools that are currently available are focused on generating traces of coarse-grained events, such as system calls. Rather than allowing a range of traceable data structures and observation frequencies, many of the tools are primarily intended to augment the use of execution-control debugging methods with only some tracing capabilities. On hardware-based software development platforms, that can partially be attributed to the trace storage limitations
of small on-chip buffers, and the problems associated with excessive data volume, as described above. The ability to detect even some types of events may be adequate for debugging some types of complex software faults, and can be considered an improvement over traditional debugging methods. However, the ability to perform fine-grained instruction-level trace generation allows the full capabilities of trace generation to be leveraged, and all types of software faults to be observed. The need for fine-grained trace collection is expected to become more important as the usefulness of event-based trace generation is exhausted by growing software complexity, and the consequent rise in complex software faults. For that reason, the enhancements introduced in Chapters 3–5 focus on instruction-level trace generation.

3. Unobtrusive trace generation

A number of widely used trace generation tools collect trace data through software instrumentation, by injecting observation routines into executing software. They attempt to satisfy a middle-ground between obtrusive execution-control methods and passive trace generation, by adding instrumentation that introduces only subtle changes to software timing. While such methods allow some complex software faults to be observed, they can also mask the manifestation of some timing-related faults that are the intended subject of the trace experiment. A parallel can be drawn to the observer effect [14], which holds that the mere act of observing something will alter the subject of the observation. Alternatively, hardware-instrumented trace generation can provide passive observations of the necessary data structures without altering software timing. Such passive methods are needed to definitively observe all types of software faults, especially as timing-related faults become more prevalent with the increasing development of multi-threaded software. As a result, the scheme introduced in Chapter 4 enables unobtrusive trace generation on physical development platforms, such as those targeting Application-Specific Integrated Circuits (ASICs). The scheme presented in Chapter 5 enables the same for hardware-emulated development platforms, specifically ones that use Field-Programmable Gate Arrays (FPGAs) to emulate microprocessor hardware. The infrastructure presented in Chapter 3 also takes the approach of emulating the presence of hardware instrumentation, allowing traces to be generated much faster than would otherwise be possible using software instrumentation.
4. **Low-level contextual awareness**

Performing software-instrumented trace generation requires that a debugging tool possess an awareness of the software environment in which it is being executed. To trace software that executes under a task-switching Operating System (OS), a debugging tool that consists of both a user-mode component and an OS kernel module is typically required. Their combined use allows a specific OS process or kernel event pertinent to the debugging effort to be instrumented. However, software-instrumented trace generation alters software timing and potentially masks complex software faults. On the other hand, the use of hardware-instrumented trace generation can preserve software timing but typically lacks contextual awareness. That can result in traces of low-level data structures that cannot necessarily be attributed to any particular OS process, including the process-under-debug. The debugging of all types of faults under a variety of operating systems requires an open approach to combining hardware-instrumented trace generation with contextual-awareness. The infrastructure presented in Chapter 3 provides a means of achieving that with the Linux OS, and its methodology can potentially be applied to any open-source operating system.

5. **Run-time trace qualification**

Selecting only those traces that are relevant to the debugging effort is known as trace qualification, which can refer to both specific execution periods as well as data structures. Carefully selecting the traces that are needed to perform software debugging can reduce the computational intensity of post-hoc trace analysis, which can reduce debugging time and cost. Performing trace qualification at run-time can also reduce trace data volume, and by extension, reduce the cost of providing trace storage. On some platforms, run-time trace qualification also enables traces to be transferred through a low-bandwidth trace port. However, some existing trace generation tools only allow a developer to define static trace qualification criteria before a trace experiment is run. In order to exploit the full capabilities of trace qualification, the run-time ability to change the type of trace data collected in the midst of a trace experiment is needed. The infrastructure presented in Chapter 3 leverages the capabilities of an open-source debugging tool to create trace experiments with run-time trace qualification criteria. While contextual-awareness serves the purposes of trace qualification by allowing only relevant OS processes to be traced, the methodology
can be extended to other relevant data types. The implementation also offers the options of automating trace experimentation through scripting, allowing collected data types to be changed over time based upon predefined triggers or events.

6. Lack of standardized and open implementations

Many of the trace generation-based debugging solutions currently available are implementations that rely upon both commercial IP blocks and proprietary software interfaces. Even though open-source software tools are available to control trace experiments, few processors currently support the use of such tools. The widespread use of trace generation will require the adoption of standardized interfaces to create, define, control, and execute trace experiments in much the same way as that the open-source GNU Debugger (GDB) interface currently serves as a de facto standard for execution-control debugging. The infrastructure presented in Chapter 3 harnesses the trace experimentation capabilities of the GDB debugger. The ability to dynamically control trace generation is added to the open-source Quick EMUlator (QEMU), a software emulator of multiple Instruction-Set Architectures (ISAs) that is widely used as a software-emulated development platform. The contributions can serve as a basis for implementing transparent trace experimentation control on other platforms that meet or even exceed the capabilities of proprietary tools. They also show how the functionality of open-source tools can be extended to offer additional features that are not natively supported, such as the control of run-time trace qualification.

1.2 Summary of Thesis Contributions

This thesis presents contributions in several areas of trace-based software debugging infrastructure, including supporting hardware, software tools, and interfaces which bridge the gap between the two. A detailed list of contributions can be found under the “Contributions of this Chapter” subsection in each of Chapters 3–5, while a summary of contributions is listed below.

Chapter 3

Since software is increasingly being both developed and deployed on software-emulated platforms, there is a need for the same advanced software debugging mechanisms that
are increasingly becoming available on hardware platforms. This work creates an instruction-level trace generation infrastructure for an open-source software emulator that integrates with its existing execution-control debugging capabilities. The infrastructure emulates the presence of hardware instrumentation, allowing trace experiments to be run through a standardized software debugger client without modifying executing code. The flexible control mechanism even allows the scripting of trace experiments to change collected data types over time. To debug a variety of complex software faults, the ability to generate traces of any registers, memory addresses, and symbolic data types such as variables, is included. Low-level contextual awareness of Linux OS processes is also provided, allowing run-time trace qualification to be exercised in any of five scopes, down to the level of a single Linux process.

When fine-grained trace generation is performed unobtrusively through hardware instrumentation, the resulting traces have the potential to accurately detect all types of complex software faults. However, continuous and real-time trace generation on hardware development platforms has the potential to create excessive trace data volumes that can neither be transferred off-chip through low-bandwidth links, nor efficiently stored. In an effort to satisfy current and future debugging needs, two different trace compression schemes are presented in Chapters 4 and 5. Both schemes target execution traces, which are arguably the most commonly collected type of trace.

Chapter 4

The execution trace compression scheme presented in this chapter targets physical development platforms with ARM ISA processors, recognizing the current dominance of that architecture in several computing domains. The scheme identifies architecture-specific redundancies within such execution traces, and uses several different predictors, transforms, and encodings to achieve maximum compression performance. The scheme repurposes modern microprocessor Instruction-Level Parallelism (ILP) hardware, including a Branch Predictor (BP) and Branch Target Buffer (BTB), to predict software-under-debug behaviour. Most significantly, it exploits the linked branches that are integral to many instruction sets, as well as the compiler-driven movement of branch return addresses between link-register, stack, and program counter. The statistical nature of execution traces is also analyzed to select a combination of transforms and encodings that result in maximum compression performance. The two
variants of the resulting real-time instruction-level trace compression scheme satisfy different needs on the spectrum of tradeoffs between performance and hardware logic utilization. Both variants perform better than other schemes in the literature, resulting in less trace data volume and requiring less trace port bandwidth in transferring execution traces off-chip.

Chapter 5

This chapter introduces an execution trace compression scheme suited to hardware-emulated development platforms, which are widely used in prototyping, and as part of the process of hardware-software co-design. The scheme takes an algorithmic approach to trace compression, ensuring both compression and decompression compatibility with any fixed instruction-width ISA. That allows it to be rapidly included on a FPGA along with a variety of different soft-processor designs, allowing the debugging of complex software faults to begin earlier than would otherwise be possible. The parameterizable scheme offers five variants which cater to differing amounts of hardware resources that typically remain on FPGA devices after the configuration of a soft-processor design. The variants are distinguished by their logic resource usage and maximum clock speeds, making it possible to reach a number of points in the spectrum between those tradeoffs.

1.3 Self-Citations

This thesis is comprised of a body of work that has either been previously published, or is at various points in the process of being considered for future publication. Each paper comprising the thesis has been primarily authored by Bojan Mihajlović, who is listed as the first author of each of the publications seen below.

1. Providing infrastructure support to assist NoC software development [15]. This paper identifies a wide scope of software development and debugging challenges in future multi-core systems, specifically those adopting Network-on-Chip (NoC) architectures. It is widely accepted that such NoCs have the potential to displace traditional bus-based processor architectures as the number of on-chip cores rise. The paper identifies three sets of problems related to software debugging on NoC systems, including debug
methodology, performance monitoring, and topological considerations. A decentralized software debugging and monitoring infrastructure is proposed for such systems. The primary author, Bojan Mihajlović, suggests generalized solutions to the problems of debug system organization, trace order preservation, and trace compression. Some of those topics are addressed in Chapter 2 of the thesis, while others are explored in greater detail by other publications described below.

2. *Software debugging infrastructure for multi-core systems-on-chip* [16]. This book chapter provides an overview of different types of software faults, and the strategies that can be used to debug each type. Several of the most prominent debugging methods are contrasted and compared, including details of their hardware and software interfaces, control methods, and limitations. The problems surrounding the use of traditional debugging methods to resolve complex and multi-threaded software faults are detailed. The environment under which trace generation-based schemes can be used in the debugging of current and future software are described, including a generalized debugging procedure, and the impediments to widespread adoption. The chapter also describes efforts to mitigate overwhelming trace volumes through on-chip trace compression. Many of those topics are covered in Chapter 2 of the thesis. This publication concludes with a description of a trace compression scheme that can be applied to hardware-emulated platforms, which is used as part of Chapter 5 of the thesis.

3. *Dynamically instrumenting the QEMU emulator for Linux process trace generation with the GDB debugger* [17]. This journal publication describes the addition of software debugging infrastructure to the open-source QEMU for performing instruction-level trace generation. Enhancements made to ARM instruction emulation allow trace experiments to be created, controlled, and triggered through a standard GDB client. The option to add a small Linux kernel patch to the QEMU guest operating system gives the debugging infrastructure an awareness of the executing Linux Process ID (PID) and context switching behaviour. Each trace experiment allows up to 16 registers and 16 memory locations to be dynamically instrumented in the midst of execution, for instructions belonging to any of five scopes: from all executed instructions, to those belonging to a single Linux PID. The work is described as part of Chapter 3 of the thesis.
4. *Real-time address trace compression for emulated and real system-on-chip processor core debugging* [18]. This conference paper proposes a trace compression scheme suited to the tracing of software executing on hardware-emulated processors, which are typically deployed on FPGAs. The method consists of five pipelined stages of data transforms and encodings, but especially leverages the Static Random-Access Memory (SRAM) embedded in FPGAs that is unused by an emulated processor. Five variants of the compression scheme are presented that satisfy different points on the spectrum of tradeoffs between SRAM usage and trace compression performance. The scheme is directly reusable on a number of fixed instruction-width ISAs, allowing rapid deployment without the need to customize the scheme to a target architecture. The compression scheme is presented as part of Chapter 5 of the thesis.

5. *Architecture-aware real-time compression of execution traces* [19]. This journal paper presents a real-time execution trace compression scheme for ARM processors, targeted towards SoC implementation. The statistical nature of execution trace data is examined to select a combination of prediction schemes, transforms, and encodings that are best suited to high-performance, real-time usage. As part of the proposed scheme, customized ILP hardware, including a BP and BTB, are harnessed to predict the execution trace stream to a high degree of accuracy. The paper is the first to show that compression performance can be significantly improved by also exploiting micro-architectural details related to the ARM ISA. A variety of transforms and encodings are also explored to give a final degree of compression to the trace stream. The resulting two trace compression scheme variants are described in Chapter 4 of the thesis, and are suited to different points on the spectrum of tradeoffs between hardware area and compression performance.

### 1.3.1 Related work

The following work has also been primarily authored by Bojan Mihajlović, but is not directly represented in the content of this thesis. Rather, the presented ideas indirectly led to work that would ultimately form the focus of the thesis.

1. *Infrastructure for testing nodes of a wireless sensor network* [20]. This book chapter describes an infrastructure for testing the functionality of wireless sensor network
nodes while they are deployed in the field, ones with strict limits on the amount of energy and time they can devote to self-testing purposes. In contrast to traditional hardware verification strategies, the work leverages software self-test programs to verify the functionality of hardware sub-systems. In that sense, the self-testing can be viewed as a form of software instrumentation, not unlike the type used in software debugging. The chapter also presents a scheme to distribute self-test programs to field-deployed nodes, including the application of a compression scheme that leverages the self-test program ISA to achieve good compression ratios. That work can be seen as a precursor to a core element of the thesis, that of exploiting the ARM ISA and micro-architectural details of a processor to compress execution traces.

2. On transparently exploiting data-level parallelism on soft-processors [21]. This paper describes the creation of a toolchain and simulator supporting Single-Instruction Multiple-Data (SIMD) instruction extensions to the ISA of a custom soft-processor. The work enabled a GNU Compiler Collection (GCC) compiler to gain an awareness of the enhanced instruction-set, and to leverage its existing auto-vectorization abilities to automatically speed up sequential code. As part of the toolchain, the GDB debugger and its internal Instruction-Set Simulator (ISS) also received an awareness of the expanded ISA. Such familiarization with GDB internals revealed the mechanisms through which advanced debugging features, such as trace experiments, can be controlled. That knowledge served as a basis for one of the main contributions of this thesis: the ability to create and control trace experiments on an emulated processor platform through the GDB debugger.

1.4 Thesis Organization

In Chapter 2 the thesis reviews software development and debugging practices, and how they are applied to various software development platforms. Different types of software faults are contrasted, and a basis is established for the detection and resolution of each fault type. A generalized debugging procedure gives the reader an idea of how complex bugs can be resolved through trace generation. It also contains definitions of relevant terms, as well as an overview of work related to the topics covered in subsequent chapters. Chapter 3 discusses the addition of advanced software debugging capabilities to the QEMU
software emulator, one that is increasingly being used for software development purposes. The work focuses on the interaction between a software debugger, guest machine OS, and a software-emulated development platform. An understanding of the interplay between those different components within the process of debugging software is also given. Chapters 4 and 5 focus on the importance of execution traces to software debugging, and present different methods for compressing those traces on hardware-emulated and physical platforms, respectively. Chapter 4 describes a high-performance execution trace compression scheme that is specifically catered to ARM ISA processors, recognizing the current dominance of that architecture in several computing domains. The chapter discusses the leveraging of prediction hardware, micro-architectural details, and encoding methods in order to achieve exceptional trace compression performance. The final method described in Chapter 5 is intended to bring execution trace compression to hardware-emulated development platforms. The parameterizable scheme is capable of being rapidly deployed to allow soft-processor execution traces of various ISAs to be compressed.
Chapter 2

Background and Related Work

This chapter covers software development methodologies, including development models, tools, and practices that are commonly employed in the creation of software. It also explores the way in which software debugging is performed on both physical and emulated development platforms. Different types of software faults are described, along with the circumstances under which they manifest. The supporting tools that are needed to perform debugging are also presented, for both traditional debugging methods and in trace generation-based debugging. Considerations surrounding the use of trace generation are explored in detail, while highlighting the problem of excessive trace data volume. Different approaches to practical trace generation are also reviewed, including the use of trace compression. Finally, a generalized software debugging procedure gives an example of the steps that can be followed to resolve a complex software fault with trace generation.

2.1 Software Development

The development, debugging, and deployment of software has undergone large changes over the years. Decades ago it was common practice to develop software directly for a specific physical machine, such as the large mainframes that were ubiquitous at one time. With knowledge of architectural details, a developer could find an optimized way of mapping an algorithm to an architecture to maximize software performance. Software of relatively low complexity allowed ad-hoc software development and debugging methods to be used, while manual optimizations featured prominently in the development process.

Modern software development is more formalized due to the need to manage rising
software complexity and large codebases. A variety of software development models and platforms can be used to decrease development and debugging time, modularize the development process, and increase code re-usability. Many projects choose standardized development and deployment platforms to reduce the complexity of retargeting software to different machines. Those platforms are increasingly virtualized, abstracting away the architectural details of the physical hardware.

2.1.1 Models

Early efforts to formalize the process of software development encouraged following a development model that would not have been out of place in other branches of engineering. The waterfall model [22] is composed of several core phases of development: requirements gathering, design, implementation, verification, and maintenance. Since each phase of development is to be completed before the next can start, long lead times can result before working software is available, and any change in requirements necessitates that the development process be restarted.

Complex designs sometimes require changes in product requirements to suit a budget or time frame after development is already underway. The adoption of iterative development models, such as the spiral model [23], offers the flexibility to make those changes. Under such a model, a gradually increasing set of requirements is repeatedly cycled through the phases of development. Each iteration results in a working application with a growing set of implemented requirements, even allowing the feature set of a project to be determined by a budget or time frame.

Modern software development is based upon the use of iterative methods, which includes methods centered around Agile [24] principles, such as Scrum [25] and Extreme Programming (XP) [26]. While based upon similar principles, many methods can be differentiated by their prescription of specific practices and methodologies. There are a variety of methods in active use, each of which aims to increase the overall efficiency of a team of software developers.

2.1.2 Practices

Decades ago, a developer could have been expected to create a piece of software in the assembly language of the deployment machine. The application may have even executed
on “bare-metal” if the machine could not support the overhead of executing an OS. In that case, recreating common functions on each customized platform could require extensive developer time and effort.

In contrast, modern code is predominantly written in high-level languages that achieve comparable performance to hand-optimized code, in part due to decades of advances in compiler optimizations. That high-level approach, and the increased reliance on automation, allows developers to be more productive while ensuring that software is portable to many different architectures. Software targeted at all but the smallest Microcontrollers (MCUs) typically executes within an established OS, which also obviates the need to recreate common supporting subroutines. The tools that developers use have advanced considerably, where it is common for an Integrated Development Environment (IDE) to automate rote development activities. They commonly include features such as auto-completion of source code, automated building, and integrated debugging that allow developers to be more efficient.

The use of Source Code Management (SCM) in software projects also enables a number of developers to collaborate on a single project effectively. Tools such as Git [27] work together with iterative development models to allow source code changes to be tracked, allowing the best revisions to be incorporated into a project. Individual developers are often responsible for verifying the functionality of their contributions before “checking in” their additions to a larger software project. System-level functionality is typically verified using a set of regression tests that verify the software as a whole. For large software projects, tests may be automated to execute overnight, ensuring that new code contributions have not affected functionality in other portions of the software project.

2.1.3 Platforms

In software development, different points on the spectrum of tradeoffs between development cost, time-to-market, and deployment cost can be achieved by utilizing different software development models, development platforms, and deployment platforms. The most straightforward platform for software development is one in which development, debugging, and deployment of software is performed natively on the same physical hardware, once that hardware has been developed. Such platforms can achieve reduced hardware cost but can extend development time, since software development can only begin after hard-
Background and Related Work

Hardware has been developed, verified, and tested. To reduce overall development time, hardware and software can be developed in tandem through a process known as hardware-software co-design. In such a process, the final hardware design can be approximated with a model that acts as a placeholder. That allows software development to begin early in the co-design process, reducing overall development time and cost.

There are different types of alternative development platforms that can be harnessed for both co-design purposes, and to meet other development goals. Among such platforms, the ones that reproduce system timing accurately are known as timed, while the ones producing correct outputs regardless of timing are known as untimed. Typical co-design practices involve developing hardware models at progressively finer levels of timing accuracy, where an untimed platform is eventually supplanted by a timed platform. All the while, iterative software development can proceed concurrently, with developers continually resolving software faults that are uncovered due to hardware model refinements.

For the purposes of software debugging, a Central Processing Unit (CPU) can be modeled at several levels of granularity. Software emulation allows the machine code of one ISA to be translated into the machine code of a different ISA. When instruction translation is untimed, both software execution and functional verification can proceed relatively quickly. Checking software functionality is restricted to ensuring that correct program outputs are generated, without regard to the timing-related faults that may eventually be introduced when the software is executed on other platforms. To enable software timing to be preserved, timed software emulation platforms such as SystemC with Transaction-Level Modeling (TLM) can be used. The cost of such timing accuracy is a performance penalty of several orders of magnitude over untimed platforms, which may render timed platforms impractical for many co-design projects.

Later steps of the co-design process typically involve creating an Register-Transfer Level (RTL) model of the hardware-under-development. The design can then be hardware-emulated by an FPGA, which preserves relative hardware timing, but can typically be clocked at slower speeds than an ASIC. From the perspective of software development, hardware emulation allows further software faults to be uncovered while hardware development progresses toward an ASIC. In some cases ASICs serve as a final deployment platform, while designs that would not benefit from their economies of scale may opt to develop, debug, and deploy software on software-emulated or hardware-emulated platforms instead.
2.1.4 Parallelism

To fully harness the growing number of processor cores available on a typical SoC, software development is in the midst of a fundamental shift. Maximizing software performance increasingly means creating programs consisting of multiple concurrently-executing threads. While there are many frameworks that allow such software to be developed, the process is inherently more complex than developing single-threaded programs. Multi-threaded software is especially sensitive to timing differences between threads, and as such cannot be reliably modeled on untimed platforms. On timed platforms, traditional software debugging techniques are also unsuited to debugging such programs. As a result, multi-threaded software development can be slow and complex, and is often performed only when performance that exceeds single-threaded capabilities is required. Even then, a single-threaded version of the software project is typically developed first, before being scaled to a more complex multi-threaded version.

2.2 Software Debugging

Software faults (also known as bugs) are unintended software behaviours which can lead to errors in the intended functionality of software. In formal terms, that means that the software that has been developed is not conforming to its requirements or specifications. The process by which such faults are identified and resolved is called software debugging.

In general, the process of software debugging aims to identify bugs and trace their manifestations back to an offending segment of source code. Tracking the bug to a certain point within a program’s execution requires that the program’s internal data structures be observed in the midst of execution, which is known as observability. Observable structures can include variables, registers, memory addresses, or other internal data that may help to identify the fault, both in terms of location and time. Once a fault is identified, its cause can be determined if the program’s symbol table can be used to link its executable machine code back to its source code.

Once a fault has been isolated, resolving it typically involves modifying and recompiling source code and re-executing the program. However, if either execution or recompilation is expected to take a long time, it can also mean utilizing other methods. Certain debugging methods allow controllability to be exercised on the program’s data structures once a
program has been halted at some point in its execution. Variables, registers, or memory locations can then be changed to their expected or known-good values, and execution can be resumed to verify that the changes have resulted in fault resolution.

2.2.1 Debugger Programs

A program known as a software debugger is most often used to centralize and facilitate the debugging process, including observing faults and controlling data structures. A software debugger and software-under-debug can either run locally on the same machine, or remotely on different machines. In a remote debug scenario, an example of which is seen in Figure 2.1, the machine running the debugger is known as the host, which tends to be a performance-oriented workstation that software developers use to run various development tools. The processor running the software-under-debug is known as the target, which can range from a high-performance CPU to a small embedded MCU. Local debugging is commonly used when the debugger and software-under-debug can be executed on the same machine, such as for native application development. Remote debugging is sometimes the only choice when the target system lacks the resources or interfaces to directly interact with a software developer. That is typically the case in embedded and mobile software development, but can also be useful on a variety of targets for which software is being cross-compiled.

The most prominent multi-target software debuggers include the GDB debugger [28], which has seen widespread across all domains of computing for decades, and the relatively recent LLDB Debugger (LLDB) [29] project. While the LLDB debugger offers the promise of future scalability and expanded features, it currently supports a limited number of ISAs, and lacks support for remote debugging under the Linux OS [29]. As such, the following discussion is primarily focused upon the operation of the GDB debugger, whose functionality is exploited by the work of Chapter 3.

Using the GDB debugger in a remote debugging scenario involves executing a small piece of interface software on the target system, known as a GDB server or GDB stub. Either interface enables access to the target from a host system running the full-featured GDB debugger client. That allows the complexity and memory footprint of the target software to remain relatively small, while allowing the same debugging capabilities as local debugging. Debug communication between host and target machines occurs over the GDB Remote Serial Protocol (RSP) protocol, which is typically routed through a network or
Universal Asynchronous Receiver/Transmitter (UART) interface. If the target machine contains hardware that is specifically intended to support software debugging, commands issued by the GDB client are interpreted by a driver executing on the host, which completes them by configuring the appropriate supporting hardware on the target.

Many software debuggers support debugging both with and without an underlying OS, and some have support for multiple threads. When debugging a single-threaded program under an OS, the thread will effectively be executed under full control of the debugger. The OS will schedule the debugger thread as it would any other, allowing the debugger to remain oblivious to any OS thread scheduling or control mechanisms. However, this is not the case when debugging multi-threaded programs. Since the software-under-debug may instantiate additional threads at any given time, the software debugger must be aware of the Application Programming Interface (API) under which threads are instantiated and terminated in order to take control of them.

2.2.2 Fault types

Software faults may take several different forms. From the standpoint of the software developer, it may be of benefit to differentiate between bugs based upon their apparent manifestations, as it allows the different approaches to uncovering and resolving them to also be differentiated on that same basis. Three of the main types of software faults are summarized below, using a categorization first made by Gray [30]:

1. Bohrbugs: As in the Bohr model of the atom, such bugs are readily detectable and
“manifest consistently under a well-defined set of conditions” [31]. These easily-reproducible bugs can be defined as Bohrbugs.

2. Mandelbugs: As in the Mandelbrot set, “a fault whose activation and/or error propagation are complex” [31], where the complexity is derived from many dependencies, or is related to the time required for the bug to manifest. These bugs can be much more difficult to reproduce.

3. Heisenbugs: As in the Heisenberg Uncertainty Principle, refers to “elusive faults” [31] whose “observation affected the phenomena you were trying to see” [32]. The manifestation of these bugs is typically sensitive to changes in software timing, making their reproduction especially difficult under certain circumstances.

The ease of reproducibility is a primary distinction between the three types of bugs. Using a software debugger, Bohrbugs may be identified and resolved by reproducing the conditions under which they appear, while Mandelbugs and Heisenbugs first require the developer to identify the conditions under which fault manifestation occurs. In a complex system with ever-changing sets of conditions, elusive Mandelbugs may appear only infrequently over long periods of execution. Heisenbugs can generally only be observed on timed platforms, as they are frequently intolerant of changes to software timing. Even small timing changes may block their manifestation, precluding the use of some debugging methods to observe them. In the case of the latter two types of bugs, periodically recording the conditions of the system may help determine the combination of factors that lead to the manifestation of the software fault. In some cases, controllability can be exercised to recreate the same conditions that led to the fault. When a software fault can be consistently reproduced, it can considerably aid the uncovering of its underlying cause.

2.2.3 Multi-threaded Software

There are unique challenges to debugging some multi-threaded software. In single-threaded software, instructions are retired in the same order as they are encountered in the instruction stream. This property of determinism allows the results of multiple executions to be compared with each other, aiding in the identification of software faults. If multiple software threads are time-scheduled to share a single core, the execution order of threads can change depending on the thread-scheduling algorithm and the presence of other active threads. In
that case, OS contextual awareness is needed to determine which executed instructions belong to which thread. The same is true of multi-core SoCs, when debugging multiple software threads that are executing on multiple cores simultaneously.

The debugging of concurrently-executing multi-threaded software presents an even more complex challenge. In an effort to extract maximum performance from a chip, the different cores of a multi-core SoC often operate in different clock domains. This means that each core executes a software thread asynchronously with respect to the others found on-chip. While the order of executed instructions is maintained within each thread, execution order with respect to other threads is non-deterministic. In fact, the chronological order of executed instructions between cores is likely to change with each execution of the program. That can result in a different overall flow of execution each time a concurrently-executing multi-threaded program is run. Such non-determinism can result in decreased reproducibility for all types of software faults, even those whose manifestation is otherwise predictable in single-threaded software. Both Bohrbugs and Mandelbugs can thus appear to have timing-related causes, making them resemble Heisenbugs.

The problem of non-determinism increases the effort needed to debug multi-threaded software above that of a single-threaded version of the same software. For that reason, multi-threaded software development typically begins as a single-threaded version, where most bugs are resolved before scaling to a multi-threaded application.

### 2.3 Traditional Software Debugging Methods for SoC Platforms

Over the years, a multitude of debugging methods have been employed to identify and eliminate software faults. All methods require some amount of supporting hardware, software, or some combination thereof. If the debugging method changes the software behaviour or timing in any way, it is known as *intrusive*. Intrusive debugging methods have enjoyed wide popularity for decades because they are effective in identifying most bugs in many types of programs. If single-threaded software does not have real-time scheduling requirements, it can have its timing altered without major consequence to its functionality. Unobtrusive debugging methods have traditionally been used only when intrusive methods could not identify bugs adequately. That is the case with real-time software, where altered timing has the potential to cause additional faults in the system, such as by causing deadlines to be missed. That also applies to modern software comprised of interdependent
and concurrently-executing threads. In the following discussion an examination is made of traditional debugging methods and the implications of using them with multi-threaded programs.

2.3.1 Software Instrumentation

It can be said that the simplest type of debugging uses a purely software-based approach. Most software developers are familiar with an informal method of adding print statements within a program in order to observe a variable at an opportunistic point during execution. This form of software instrumentation allows a limited degree of observability to be achieved at the expense of much manual effort, including a re-compilation and re-execution of code for every change that is made. Software instrumentation can also be done with a monitor program, which is an environment under which the program being debugged can be executed. Monitors can instrument software by modifying it so that, for example, execution proceeds until a predetermined memory address is reached, known as a software breakpoint. Data structures can then be observed or changed within the monitor environment, and execution resumed. Monitor programs are useful when a processor lacks debug support hardware, or access to it is limited. While monitor programs have mostly been superseded by software debuggers in recent years, if debug support hardware is unavailable, some software debuggers will also fall back on software instrumentation as a means of performing debugging tasks.

One of the drawbacks of using software instrumentation is that a limited number of locations can be instrumented, which exclude protected (or privileged) memory locations such as within Interrupt Service Routines (ISRs). In all cases, the instrumentation instructions are added to those of the software-under-debug. The resulting software-instrumented version of the software-under-debug requires additional execution time to achieve the desired observability, and as a result has the potential to mask certain types of complex software faults. Software instrumentation methods can therefore be categorized as intrusive.

2.3.2 Scan-chain Methods

Unlike software instrumentation methods, the addition of on-chip debug support hardware, known as hardware instrumentation, allows unaltered software to be executed on-chip. That alleviates some of the problems associated with software-instrumented debugging
approaches, namely the changes between debug and release versions of the same software.

In designing SoCs, some Design-for-Test (DFT) hardware is often added to aid in hardware verification and manufacturer testing of chips. The IEEE 1149.1 standard [33], also known as Joint Test Action Group (JTAG), describes a Test Access Port (TAP) that links the flip-flops of a chip together in one or more scan-chains. That provides data Input/Output (I/O) to all flip-flops connected to the scan-chain through the TAP. The feature is useful for hardware testing purposes, but is also useful for software debugging. Many processor cores include a "debug" mode which allows data, such as registers, to be output to the host by using a scan-chain. In the simplest scenario, the mode enables the target processor clock to be externally supplied and controlled by the host, which is able to read and write internal data structures in between executed instructions. From a software perspective, remote debugging commands are issued by the host GDB client and interpreted by a TAP driver on the host, an example of which is shown in Figure 2.2. The TAP driver then creates the signals that command the target to start and stop execution, or transfer target registers or memory addresses to/from the host. The host is then said to be exercising execution control over the target in order to achieve the desired controllability and observability.

In a multi-core SoC, the scan-chains of each core could either be linked serially, as seen in Figure 2.3, or provided in parallel as several individual scan-chains. Implementing either of those approaches poses several additional problems. Multiple scan-chains require additional external pins, which are a scarce commodity in many SoCs. On the other hand, a long serial scan-chain requires the least number of external pins, but incurs a read/write latency proportional to its length. There have been some efforts to reduce the I/O latency of such long scan-chains by bypassing cores that are not under observation, that being one of the improvements of the IEEE 1149.7 [34] standard.

While debugging, the use of a synchronous, externally-supplied clock can change the behaviour of software that normally executes on cores in asynchronous clock domains. In such a scenario, timing bugs may not manifest when they are related to data synchronization between clock domains. That causes a significant change in software behaviour for some applications, and as a result scan-chain methods can be considered intrusive.

One of the biggest impediments to the use of scan-chain methods, even for single-threaded debugging, is that the maximum externally-supplied clock speed can be an order of magnitude slower than its on-chip counterpart. Many modern SoC cores are clocked
above 2 GHz, while the maximum scan-chains clock speeds are reportedly in the 100 MHz range [35]. Using an external scan-chain clock to drive a processor core can then result in a dramatic increase in execution time, and consequently, increases in the time spent both debugging and waiting for faults to manifest. Since the cost of debugging software is related to the amount of time spent debugging, scan-chain methods would incur an increasing cost if SoC clock frequencies continue to rise in the future.

### 2.3.3 In-Circuit Emulation

Using DFT hardware to exercise fine-grained processor control may be an advantage during hardware testing, but can be excessively slow for software debugging. For that reason, some commonly-used software debugging control logic can be moved from the host to the target in order to enhance overall debugging speed and functionality. Such approaches can roughly be grouped into a category known as In-Circuit Emulation (ICE). Many of those methods aim to accelerate the functionality that is already possible using scan-chain methods, at the expense of additional on-chip hardware instrumentation. The enhancements are often complementary to scan-chain methods, and augment the execution control provided by them. For example, in Figure 2.4 the addition of hardware breakpoint registers and their corresponding hardware comparators (not shown) allow software to execute in real-time by
2.3 Traditional Software Debugging Methods for SoC Platforms

![Diagram](image)

**Fig. 2.3:** Execution-control software debugging of a physical SoC with multiple processor cores through an IEEE 1149.1 (JTAG) TAP interface

utilizing an on-chip clock while waiting for the desired point of execution to arrive. Once a hardware breakpoint register has been configured with an address and enabled through the TAP interface, that register is continuously compared to the Program Counter (PC) register. When a comparator determines that the PC and breakpoint addresses are equal, an interrupt is triggered which halts execution and returns control to the host.

Systems with expanded ICE capabilities also allow hardware breakpoints to be loaded with data values in addition to addresses, which can then be compared to general-purpose registers or memory locations. They are then known as *hardware watchpoints*, and allow execution to be stopped when a desired data value has been observed. In either case, after execution is stopped and control transferred to the host, a scan-chain can then be used to enable controllability and observability at precisely the desired point of execution.

Applying execution control through ICE can help avoid the problems of using a slow external clock in scan-chain methods. For single-threaded programs, it allows the execution of individual cores to be paused in order to control or observe them. That can be especially useful for quickly resolving Bohrbugs, since the *trigger* criteria for exercising execution control can often be clearly established. On the other hand, Mandelbugs may have a delayed manifestation after their root cause, which may make it difficult to pinpoint the conditions that gave rise to a fault using execution-control methods alone. Intrusive debugging
methods such as ICE can also mask the manifestation of Heisenbugs entirely, since halting execution once a breakpoint has been reached causes changes to software timing.

There are also problems with the use of execution-control methods such as ICE in the debugging of multi-threaded software. Applying execution-control methods to one thread in a multi-threaded program can lead to timing-faults such as missed deadlines due to the overhead of exercising controllability and observability. Attempting to control an entire group of concurrently-executing threads can also lead to non-deterministic behaviour. For example, a breakpoint on one core could be \textit{cross-triggered} to stop the execution of all other cores, but the delay in propagating such a signal to asynchronous clock domains leads the cores to stop at non-deterministic points (unpredictable locations). Some SoCs with homogeneous cores that use this imprecise approach \cite{36} report hundreds of cycles of “skid” before all cores can be stopped or started. When debugging heterogeneous cores, some of which can lack cross-triggering hardware, the time needed for execution control to be performed through software is reported to be on the order of seconds \cite{37}. This method clearly does not allow for the desired points of execution to be observed on a multi-core SoC.

As the complexity and number of cores per chip rise, the latency of performing centralized execution control will also continue to grow. Although execution control debugging per-
formed through ICE is expected to remain essential to the debugging of various software, it is not seen as a solution to the debugging of increasingly complex and multi-threaded software.

2.4 Software Debugging Methods for Alternative Platforms

Software development is increasingly being performed on alternative platforms that act to mimic the physical hardware platforms that have conventionally been used for both software development and deployment. The goal of using such platforms is often to reduce total development time and cost, as discussed in Section 2.1.3. Alternative platforms can be either timed or untimed, and can include simulators, software emulators, hardware emulators, or any combination thereof. The decision to incorporate any alternative platform within the process of development or co-design hinges on the extent to which development can progress under such a platform. Under a typical iterative development model, rapid cycles of debugging are an essential part of the development process. Since the majority of development time and cost is reportedly spent on debugging [11], the amount of support a platform provides for software debugging can be a determinant of overall development time and cost.

The following sections discuss two of the most prominent alternative development platforms and the type of support that is commonly provided for software debugging.

2.4.1 Software-Emulated Platforms

Entire hardware platforms can be modeled in software as part of the hardware-software co-design process [38], where there is typically an inherent tradeoff between the accuracy of modeling hardware, and the computational intensity of doing so. Many models exist to satisfy various points on the continuum between tradeoffs. At one extreme of that scale, an untimed emulator performs binary translation to quickly translate executable code between a host ISA and guest ISA without regard to timing. At the other end of the scale, a timed simulator can precisely model the cycle-accurate behaviour of the machine components it is mimicking. In software development targeted at the ARM ISA, an ISS such as ARMulator [39] satisfies a middle ground that enables the simulation of some micro-architectural effects.

When applied to software debugging, modeling a platform at several different levels can play a role in the co-design process. Different models can be used to target different types
of software faults. For example, untimed software-emulated platforms can be useful for quick ly establishing software correctness, correcting obvious bugs, and for executing long sequences of code to observe infrequently-occurring bugs. Once the debugging capabilities of an untimed platform have been exhausted, timed simulation platforms can be used to execute code in an environment which accurately preserves timing. A given period of execution can then require up to two orders of magnitude more time to model [40], but potentially allows timing-related bugs to be targeted.

The open-source QEMU [41] is an untimed emulator used as a software-emulated development platform by projects such as the Android Software Development Kit (SDK) [42] and others. It emphasizes code execution speed over the reproduction of accurate guest machine timing, and as such tracks only the guest machine data structures pertinent to producing correct results from guest instruction execution. Consequently, any potential micro-architectural properties of a guest machine, such as pipelining and caching, are ignored by such an emulator for the sake of emulation speed. Untimed emulators such as QEMU also keep accurate time through the use of timing interrupts tied to host hardware. However, the number of guest instructions executed in any unit of time can vary depending upon the efficiency of their translation to host instructions. As such, they are suited to acting as fast software debugging platforms for the resolution of software faults that do not have timing-related causes.

Software debugging methods whose operation is based upon the direct control of processor hardware, such as methods based upon scan-chains or ICE, are generally unadapted to software-emulated platforms. Instead, a typical software debugging scenario involves the execution of a GDB server within the guest machine OS to communicate with a GDB client on the host machine. That can be used to retain the needed contextual awareness of executing OS processes in order to isolate the process representing the software-under-debug.

In the case of bare-metal development without an OS, or during the bootstrapping of an OS, the use of a GDB server is not possible on the guest machine. For debugging such software from the first emulated instruction onward, Figure 2.5 shows that emulators such as QEMU include a built-in GDB stub that executes independently of the guest processor. The built-in stub effectively software-emulates the low-level debugging capabilities found in ICE, since it allows instruction execution on the guest machine to remain unchanged. That same low-level functionality also means that it cannot be contextually aware of the guest
OS or its processes, and is limited to the subset of debugging commands supported by the stub itself. However, the use of the GDB stub is potentially faster than executing a GDB server within the emulated guest machine, since stub routines are executed natively on the host ISA without translation. Communication between a GDB client and the built-in stub is typically directed through a virtual network interface, since both are typically executing on the same host workstation.

2.4.2 Hardware-Emulated Platforms

In the process of co-design, hardware-emulated platforms are commonly used to both verify the functionality of hardware designs, and to allow software development to proceed concurrently with that of hardware. Such platforms can provide a measure of timing accuracy the cannot be achieved with untimed software-emulated platforms, at speeds that far exceed those that can be attained by timed software-emulated platforms. They are also commonly used in prototyping, as well as in hardware deployments that do not warrant the cost of ASIC development and manufacturing.

Depending upon the size of a hardware design, one or more FPGAs can be configured to replicate its behaviour. Designs may include multiple IP cores, typically operating in separate clock domains to achieve maximum performance. When one of those IP cores is that of a processor, it is known as a soft-core processor or soft processor. Such processors
typically operate at speeds in the hundreds of megahertz [43], while the equivalent hardware on a physical platform may be clocked in the gigahertz range. Despite the fact that FPGA-configured designs can be an order of magnitude slower than an equivalent physical SoC, their hardware outputs can be produced concurrently, and they are able to preserve the relative timing of hardware-emulated logic.

Traditional execution-control debugging methods, such as scan-chains and ICE, can be used for software debugging on hardware-emulated platforms. An emulated SoC can be controlled through a standard TAP interface the same way a physical SoC would be, as the scenario in Figure 2.6 shows. Using such an interface to perform software debugging on soft-core processors can potentially detect a greater proportion of software faults than would manifest on untimed software-emulated platforms. However, some faults can depend upon external inputs whose timing cannot be synchronized with hardware-emulated platforms. As a result, some timing-related bugs that do not manifest on hardware-emulated platforms may be revealed when software is deployed on physical platforms.
2.5 Debugging with Trace Generation

The development of increasingly complex software has given rise to complex bugs once con-
sidered infrequent, while the limitations of traditional execution-control debugging methods
have been shown to be ill-suited to the resolution of such bugs. These factors have con-
spired to create an environment where software debugging is especially difficult, with the
time needed to resolve complex software faults reaching several weeks in some cases [12].
From the discussion in Sections 2.3 and 2.4, it should be evident that non-intrusive de-
bugging methods are required for a variety of development platforms in order to aid the
resolution of complex bugs that are intolerant of traditional execution-control debugging
methods.

Modern software is not unique in its intolerance of intrusive debugging. The domain
of real-time software requires that software respect deadlines and perform data processing
within a specified period of time, at the risk of a fault or failure. Real-time software
developers have used a method known as trace generation, or tracing, to resolve bugs
since the 1980s [44]. In that same decade, developers of High-Performance Computing
(HPC) software also faced challenges in debugging what were then uncommon and highly-
specialized concurrently-executing multi-threaded programs. At that time, the use of trace
generation with HPC software aided the resolution of faults that could not be observed
through execution-control methods [45]. In the intervening time, the development of multi-
threaded and complex programs has increased to the point that it now comprises a much
larger portion of modern software development. As such, trace generation methods are
currently considered a viable alternative for the debugging of modern software that is
unsuited to execution-control methods [46, 47, 48]. They are considered especially useful
for the observation of complex, intermittent, and multi-threaded faults [49, 15] that are
becoming increasingly common.

Trace generation functions by leveraging instrumentation in either software or hard-
ware to collect and log data structures pertinent to the debugging effort. The resulting
log (or trace) of data structures such as registers, variables, or memory locations is created
transparently while executing the software-under-debug. In doing so, the precise conditions
leading to a fault can be captured for post-hoc analysis without the need to interrupt soft-
ware execution. One of the most commonly collected trace types is known as an execution
trace, which lists the addresses of instructions which have been executed, indicating the
path of execution taken through the software-under-debug. If needed, a data trace may be generated alongside an execution trace to allow the observation of data structures such as registers or memory locations that aid the debugging effort.

Software-instrumented trace generation can be considered both a flexible and and intrusive debugging method due to the injection of trace collection code into the execution of the software-under-debug. It can potentially be less intrusive than execution-control methods because software is allowed to execute at full-speed without interruption during trace generation. Hardware-instrumented trace generation is intended to be passive and transparent to the software-under-debug, leaving software timing unaltered and allowing all types of faults to be observed. Both types of instrumentation can potentially be used to observe those Mandelbugs that do not have timing-related causes, by forming a record of the complex sequence of events leading up to a fault. Since timing-sensitive Heisenbugs are potentially masked by changes to software behaviour, they are typically targeted only by unobtrusive hardware-instrumented trace generation.

The following subsections focus on various aspects of trace generation-based debugging, and allow the contributions in Chapters 3–5 to be put into context.
2.5 Debugging with Trace Generation

2.5.1 Software Instrumentation

Traces generated through software instrumentation generally do not require hardware support, though some may utilize on-chip ICE structures such as hardware counters to implement some of their features. Many software instrumentation tools can observe operating system events such as system calls that are otherwise opaque to a software developer. Such event-based trace generation requires that a corresponding Linux kernel module or kernel patch be present. A few tools, such as SystemTAP [50] and LTTng [51], extend support to user-space events, but only when fixed instrumentation points are compiled into the software-under-debug. These static instrumentation points are generally limited to the number of software tracepoints that can manually be added by a developer, and their locations cannot be changed without recompiling the software-under-debug.

The limitations of both static and event-based instrumentation can restrict the types of data structures that can be collected, how often they can practically be collected, and can slow down the debugging process by requiring periodic recompilation. Some software faults, such as Mandelbugs, may require finer-grained trace collection. To observe such bugs, a developer can generate a trace after each executed instruction, known as instruction-level tracing. Only a limited number of software instrumentation tools allow such fine-grained trace collection. Of those, the Performance Inspector [52] tool Itrace offers static instruction-level instrumentation, but lacks support for the ARM ISA which is widely used in embedded and mobile computing. Only Dtrace [53] offers the ability to dynamically instrument software without the need for source-code changes, though its Linux variant is currently not feature-complete for the ARM ISA.

2.5.2 Hardware Instrumentation

The addition of software instrumentation to software-under-debug results in changes in software timing and is considered an intrusive debugging method. Such timing changes can be small due to occasional event-based instrumentation, or much larger when instruction-level instrumentation is required. In the latter case, instrumenting every instruction executed by the software-under-debug can result in profound timing changes which are likely to mask the manifestation of timing-related bugs. Instead, hardware-instrumented trace generation is seen as a solution to the transparent and unobtrusive observation of executing software. In fact, hardware support for trace generation is being integrated into many modern SoC
designs, and is an integral part of many commercial debugging strategies, including ARM
Coresight [13] and Infineon Multi-Core Debug Solution (MCDS) [54].

Trace generation performed by on-chip hardware requires that a hardware trigger be
user-configured prior to the execution of the software-under-debug. Triggers define when
tracing should occur, but also control the type of debug data that should be collected.
Useful trace data typically includes processor registers or memory values whose observation
is expected to aid the debugging effort. When the trigger is “hit” during execution, the
trace data can be stored in an on-chip trace buffer (also known as a trace memory) as seen
in Figure 2.7. Alternatively, trace data can also be streamed directly off-chip in real-time
to the host machine.

2.5.3 Emulated Hardware Instrumentation

Software development performed on software-emulated or simulated platforms can also
benefit from trace generation capabilities. The commercial ARM Fast Models [55] include a
trace interface known as Model Trace Interface (MTI). That is used as a basis for simulators
such as Synopsys Virtualizer [56], which additionally provides software debugging tools for
contextually-aware trace generation and analysis. Trace generation capabilities can also
be found in non-commercial simulators based upon SystemC/TLM, some of which use the
QEMU emulator as a basis for modeling a processor [40]. Many such simulators have the
ability to model entire hardware platforms at several different levels of accuracy depending
upon need. That allows a developer to progressively resolve bugs using increasingly precise
simulation models.

While some software-emulated development platforms provide trace generation capa-
blities, platforms not explicitly providing such support can still typically make use of
software-instrumented trace generation. However, the execution of instrumentation in-
structions represents an overhead to the software emulation of the software-under-debug.
Unlike hardware instrumentation that generally supports storing generated traces into a
dedicated buffer, or streaming them through a trace port, a software instrumentation tool
may instead be forced to write traces to the guest machine filesystem. Those overheads
are compounded by the inherent inefficiency of performing software emulation, since the
instrumentation code and filesystem I/O must themselves be software-emulated within the
guest machine. That can result in prohibitively long execution times, especially when
2.5 Debugging with Trace Generation

fine-grained instruction-level trace generation is required.

There is an alternative to performing trace generation through software instrumentation on software-emulated platforms, where the emulation of software instrumentation code itself presents a significant overhead. Just as the presence of a hardware device can be emulated on such platforms, so can the presence of hardware support for trace generation targeting the emulated guest machine. By software-emulating the presence of trace generation hardware, the guest machine can instead be instrumented through code that executes natively on the host machine. Known as *emulated hardware instrumentation*, it is potentially much more efficient than using software instrumentation on software-emulated platforms. It allows generated traces to be written directly onto the host filesystem instead of a software-emulated guest filesystem, which also increases the efficiency of trace generation.

There are only a limited number of emulated hardware instrumentation options available for software-emulated platforms. Some platforms themselves offer primitive support for instruction-level trace generation. Such features are more often suited to debugging OS kernels rather than user-space applications, while observable data structures are either static, or controlled through a proprietary interface. The QEMU emulator includes the ability to emulate hardware instrumentation of the PC register to generate complete execution traces. However, QEMU must be executed under a *debug mode* to enable tracing, and lacks the necessary control to instrument other potentially useful data structures. A specialized version of QEMU has also been developed by Bordin et al. [57] for collecting execution flow information, but is unsuited to software debugging. Instead, it targets code coverage analysis in the domain of software testing. The QEMU emulator also offers an event-based tracing framework that instruments a number of predetermined machine events. The events include hardware-level information, such as memory events, that may not necessarily aid software debugging.

2.5.4 Deterministic Replay

Trace generation can capture the conditions leading to a fault, so that its manifestation can be connected with its cause during post-hoc analysis. In an attempt to observe faults that are known to occur only infrequently, a developer may not be able to perform fine-grained trace generation for an extended period of execution due to the problems described
in Section 2.5.10. Instead, a trace of processor data structures may only be captured periodically until one of more faults occur. The trace data can then be used to configure processor data structures into a state that is known to precede a fault. In a process known as deterministic replay, the software-under-debug can then be replayed from a given intermediate point of execution. Faults can then be reproduced deterministically while under observation, such as with a finer granularity of trace generation. Configured data structures can also be modified in order to evaluate the manifestation of a fault under different conditions.

Several deterministic replay schemes have been proposed for both hardware-instrumented [58, 59] and software-emulated platforms [60]. Since such schemes rely on captured trace data to replay execution, they do not reproduce timing-related behaviour. As such, deterministic replay can be effective in replaying Mandelbugs that do not have timing-related causes, on both timed and untimed platforms. Since Heisenbugs manifest nondeterministically and are highly sensitive to software timing changes, deterministic replay may not effectively reproduce such faults.

2.5.5 Triggers

Triggers used in hardware-instrumented trace generation can be compared to the hardware breakpoints and watchpoints commonly employed by ICE methods. In fact, the hardware structure of a trigger can be similar to that of a breakpoint, consisting of comparators and/or counters. Like breakpoints, triggers can be used to compare instruction addresses or data values with a preset trigger value, or may use counters to follow events such as loop iterations. When trigger conditions are met, it is said that the trigger has “hit”. Rather than breaking the execution of the program, a trace of some desired data is output instead.

In the example of Listing 2.1, the trigger of tracepoint 1 is loaded with the address of the first instruction of function my_function, creating an unconditional trigger that will hit when the program counter reaches that address. Assuming the trigger hardware supports the outputting of three trace values simultaneously, it can be configured to output the pc and r1 registers, as well as the variable var. For unobtrusive tracing to occur, var must be located on-chip at the time of the hit, and the software debugger must use the symbol table of the program-under-debug to link it to a register location. If any of those conditions are not satisfied, the debugger may create a software-instrumented tracepoint instead. Doing
so would typically interrupt the execution of the program-under-debug to issue a memory read on the variable, in effect acting as a breakpoint with automated data collection. While some processors have hardware support for on-the-fly memory I/O that potentially does not require halting software execution, instructions that alter software timing must still be injected to issue the reads. A large number of such operations could also lead to a more appreciable change in software behaviour. For that reason, any type of software tracepoint is considered intrusive for the purposes of this discussion.

```
$ gdb program_under_debug // executable with included debug symbols
Reading symbols from program_under_debug...done.
(gdb) trace my_function // first instruction of my_function()
Tracepoint 1 at 0x8080: file my_function.c, line 4.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
> collect $pc // program counter
> collect $r1 // data register
> collect var // variable (symbolic memory address)
> collect 0xdeadbeef // absolute memory address
end
```

Listing 2.1: Example of Tracepoint Creation in the GDB Debugger

Multiple triggers can also be compounded or chained to allow for conditional trace generation using compound boolean conditions. That can allow more data to be output than any one trigger unit may support (though in practice, that may be difficult for the reasons discussed in Section 2.5.10). The inclusion of cross-triggers, where the trigger action on one core acts as the trigger condition on another core, can be useful for multi-threaded debugging. Since processor cores are typically located in separate clock domains, the crossing of clock domains by cross-trigger signals can present problems in achieving deterministic cross-triggering times. However, cross-triggering functionality still provides more options for debugging multi-threaded software than would be otherwise be possible.
2.5.6 Debugging Interfaces

Execution-control software debugging on physical development platforms has predominantly been performed over TAP hardware interfaces over the years, including the IEEE 1149.1 (JTAG) [33] and IEEE 1149.7 [34] standards. However, there have been efforts to standardize the hardware interfaces and protocols that are required for both trace generation and other future software debugging. The most prominent successor is the Nexus 5001 standard [61], which is backward-compatible with both of the older IEEE standards [62]. In terms of trace generation, one of the main advantages of Nexus is an optional auxiliary port of up to 16-bits which can be used to output trace data. The standard also includes a mechanism to address the problem of large trace data volumes (discussed in Section 2.5.10) that is robust, yet lacking in performance compared to current research methods. It is expected that widespread adoption of the Nexus standard could ensure interoperability between SoCs, debug adapters, and software debuggers. That could decrease the cost of providing debugging infrastructure, and in turn, streamline software debugging itself.

The software used to perform trace experiments can also determine the overall difficulty of performing software debugging. While many modern SoCs contain some type of trace generation hardware, few have the capability of being controlled through standardized software interfaces. While execution-control debugging can usually be performed through a GDB client, both academic and commercial trace generation tools are predominantly controlled through proprietary software interfaces [60]. Even though the open-source GDB debugger has supported the dynamic control of trace experiments for some time, relatively few target processors have so far adopted its standardized interface [63]. That could be the result of some companies restricting access to proprietary software debugging interfaces in order to distinguish between similar products of different price. For example, VMWare’s Workstation and Player virtualization tools can be considered similar, but Workstation includes access to a debugging interface intended for use in virtualized software development, while the free Player lacks those features and is intended mainly for software deployment.

2.5.7 Trace Ordering

Since the main goal of trace generation is the accurate observation of a sequence of events, the relative order in which traces are generated must be preserved. That is especially important in an environment where traces are generated concurrently by multiple cores,
where the global order of events that lead to a fault should be recorded. Since such traces would typically be generated in different clock domains, an implicit or explicit mechanism is needed to aggregate, label, and reorder them chronologically. Analyzing an ordered trace offline with the help of a debugging tool can then help determine the sequence of events that led to a fault.

One of the most constrained SoC resources is the number of pins available to interface its various functions with the outside world, including its debugging functionality. For that reason, debug interfaces are usually restricted to a bare minimum number of pins, including pins needed for transferring trace data. There must then be an on-chip mechanism for aggregating the traces of multiple cores into a single chronological trace stream before transferring those traces to the host in real-time, or storing them in a trace buffer.

In a simple multi-core architecture where the trace data path from each processor core to a hardware aggregator is the same length, trace ordering can be accomplished by implicitly processing the traces that arrive first. However, many interconnection schemes can be more complex, especially those employed by NoCs. In a typical NoC mesh interconnect, network congestion and routing delays may lead to out-of-order traces being collected by a hardware aggregator. One solution may be to arrange for accurate and synchronized time-keeping on each core, and to time-stamp each trace before it is transmitted [15]. That would allow traces to be reordered accurately either by the on-chip aggregator or by offline debugging tools.

### 2.5.8 Trace Qualification

The collection and recording of only those traces that are relevant to the debugging effort is known as trace qualification. It can be useful to limit the amount of trace data collected by a trace experiment for several reasons. For software-instrumented trace generation, limiting trace collection to relevant portions of code can reduce the overhead of executing instrumentation code, and limit its intrusive effect on software timing. In hardware-instrumented trace generation, large trace data volumes can be difficult to transfer or store. For platforms that transfer trace data off-chip in real-time, Section 2.5.10 explains that both high-bandwidth trace interfaces and provisioning trace storage can be prohibitively expensive. On platforms that store trace data within on-chip buffers, large trace data volumes can quickly fill a buffer and limit the observable execution time of the software-under-debug.
In the case of software-instrumented trace generation, a GDB server or other debugging tool executing under an OS is typically responsible for executing the software-under-debug as a process under its full control. The flexible nature of software instrumentation allows the debugging tool to perform trace qualification by injecting instrumentation code to target only relevant portions of the OS process representing the program-under-debug.

Performing run-time trace qualification on hardware-instrumented traces can pose additional challenges. When the software-under-debug executes under a context-switching operating system, the trace generation hardware must be able to differentiate between OS processes and must be made aware of context switching activity. That can be difficult to achieve from a hardware perspective, due to each process having access to its own virtual memory address space, and the masking of its physical location in memory. Operating system security techniques such as Address Space Layout Randomization (ASLR) only further complicate the matter. To provide meaningful trace qualification, hardware-instrumented trace generation methods thus require an explicit bridge across that hardware-software gap. Only then can the traces of the OS process representing the program-under-debug be isolated from those of other executing processes, as well as those of the OS kernel.

For software development under the QEMU emulator, there has been an effort to bridge the gap between its built-in GDB stub and the FreeBSD operating system. The method described by Giovanazzi [64] is capable of extracting traces from a single OS process, but it remains unclear whether other scopes of trace collection are offered, or how easily trace experiments are controlled.

2.5.9 Trace Storage

By performing software debugging through either traditional debugging methods or trace generation-based methods, a developer attempts to locate and observe the manifestation of software faults. Execution-control methods achieve that goal by periodically halting software execution at opportune points, or by using on-chip ICE triggers to accelerate the process. In doing so, observations may reveal that a suspected fault has not yet manifest, in which case software execution is resumed until the fault is eventually observed. Such “on-line” analysis of debugging information allows unnecessary intermediate observations to be discarded. Trace generation-based methods operate in a fundamentally different manner, by recording all of the selected debugging information for post-hoc analysis. Since
the determination of what constitutes a useful observation is made offline, the collection of significantly larger volumes of debugging information may be required.

Depending upon the capabilities of a development platform, the traces resulting from a trace experiment can either be recorded to an on-chip trace buffer, or can be streamed off-chip in real-time. Many commercial SoCs use dedicated trace buffers, whose size is typically limited by the amount of SRAM that can be included on-chip for such purposes. In the case of NXP LPC1800-series processors, an ARM Embedded Trace Buffer (ETB) of only 16 KB is included [65]. While certain techniques have been proposed for repurposing on-chip caches to act as trace buffers [66], and improving the efficiency of trace buffer utilization [67], trace buffers can still be quickly filled if captured trace data is not carefully managed. When a buffer is full, it can force a trace experiment to stop prematurely, or force previously stored trace data to be overwritten. As such, small trace buffers can be useful for infrequent storage of event-based traces, but their size limitations can all but preclude the practical use of instruction-level trace generation.

Instruction-level trace generation involves regularly capturing traces over a period of execution that enables a fault to be observed. Small trace buffers allow the traces of only a small window of execution to be recorded, and can be especially unsuited to observing infrequently-occurring faults. To enable the observation of those and other types of faults over any length of execution time, traces can be transferred off-chip as they are generated. Observable execution time is then only limited by the amount of off-chip trace storage that can be dedicated to the debugging effort. Commercial solutions for off-chip streaming of trace data include external trace capture hardware such as ARM DSTREAM [68], which includes 4 GB of trace storage capacity. Compared to on-chip trace buffers, the increased capacity allows much more trace data to be captured, but can also fill after some window of execution time. Off-chip streaming of trace data also requires the use of a hardware trace port in addition to the TAP interface typically used for execution-control debugging. To prevent the trace port from acting as a bottleneck between trace generation and trace storage, it must support sufficient bandwidth to keep pace with the rate of trace generation. For instruction-level trace generation, trace streaming must typically be performed in real-time, while less frequent event-based traces can potentially flow through slower links. The scenario in Example 2.1 quantifies the limitations of using both on-chip trace buffers and typical external trace capture hardware.
Example 2.1. The observation of a complex fault on a physical platform requires instruction-level trace generation of the PC register over an execution time of one minute. The 32-bit processor core is clocked at 1 GHz, offers a performance of 1 Cycle per Instruction (CPI), and is capable of streaming traces off-chip through a trace port of sufficient bandwidth. Using an internal 16 KB trace buffer, a maximum of 4096 instructions can be stored, which represents an observable execution time of $4\mu s$. Using an external trace capture device with a 4 GB storage capacity, approximately 1 billion instructions can be captured, extending observable execution time to just over 1 second.

Since external trace capture devices such as the ARM DSTREAM act as an intermediary between the host machine and target platform, traces must be transferred from such devices to the host after the completion of the trace experiment. The cost of the devices themselves can also reach into the thousands of dollars. Those factors can contribute to increased debugging time, cost, and complexity.

Alternative debugging scenarios have also been proposed, particularly one wherein the host machine itself acts to receive and store traces [18]. If an appropriate interface and sufficient trace port bandwidth can be provided, potential cost savings can be realized by eliminating the need for intermediary trace storage devices.

2.5.10 Trace Data Volume

Depending upon the type of trace generation support extended by a development platform, trace experiments may be configured to collect coarse-grained event-based traces, or fine-grained instruction-level traces. For both types of trace collection, observing intermittent faults over long periods of execution can result in difficulties storing the large volumes of generated trace data. Example 2.2 quantifies the trace storage requirements of observing a lengthy period of execution, and presents a case in which the observation of a fault can exceed available trace storage.
Example 2.2. To observe an intermittently-occurring software fault on a hardware-emulated processor core, a developer determines that an instruction-level trace of one data register as well as the PC register is required over a two-hour period of execution. For a 32-bit soft-core processor core executing at 150 MHz, with a performance of 1 CPI, the volume of raw trace data generated in two hours will total 7.86 TB. The trace data volume greatly exceeds the size of buffers that can practically be included on-chip, and must instead be streamed off-chip in real-time. Even then, the data volume exceeds the size of the largest hard disks, forcing an array of multiple disks to be used for trace storage and analysis. If trace data from longer periods of execution is needed, or any additional trace experiments are to be stored, even greater amounts of storage would be required.

Even if multiple hard disk arrays are available for storing the traces of Example 2.2, scaling the collection of trace data to longer periods of execution, wider registers, or the collection of more data structures could become prohibitively expensive. The limited bandwidth capabilities of a typical trace port can also pose a problem to the continuous streaming of traces off-chip. When physical platforms are used for trace generation, the observation of timing-related faults requires that software execution and trace generation occur in real-time. However, Example 2.3 demonstrates that a considerable amount of bandwidth could be needed to accommodate the streaming of traces generated at modern SoC clock speeds.

Example 2.3. To observe a timing-related software fault on a 32-bit physical processor core clocked at 2 GHz, a real-time instruction-level trace of the PC register is needed over an indeterminate period of time. The processor core is clocked at 2 GHz and achieves a performance of 1 CPI, which generates 7.45 GB/s of raw trace data. To stream traces off-chip in real-time, a trace port of such high bandwidth would be required that it would rule out all but the most expensive communication links.

While high-speed communication links could presumably support the trace port bandwidth needed in Example 2.3, the instrumentation of additional registers or the tracing of processor cores with faster clock speeds could easily overwhelm even those links.

Potential solutions to excessive trace data volumes involve performing run-time or post-hoc trace qualification prior to storage, as well as the instrumentation of fewer data struc-
tures. While some of those solutions can also reduce the bandwidth requirements of streaming traces off-chip, they would still not necessarily allow for real-time streaming on many platforms. To enable real-time traces to be streamed off-chip, a physical or hardware-emulated platform can be clocked at a slower speed to restrict the rate of trace generation. That approach reportedly adds a 34x overhead to uninstrumented execution time on the hardware-emulated platform employed by Chung and Hoe [69], while a similar approach has reported at least an order of magnitude increase in execution time [70]. Since modifying system timing may mask the manifestation of timing-related faults, restricting the rate of trace generation can be seen as intrusive. It can also prolong debugging time, and consequently increase debugging cost, especially when infrequently-occurring faults are to be observed. As the number of on-chip processor cores continues to grow, the approach is also not considered a scalable solution to future debugging needs.

Some commercial platforms depend upon generating event-based traces of variable length in order to manage trace data volume [71]. As a solution to both excessive trace data volume and large trace bandwidth requirements, many other platforms offer some type of compression of trace data. The use of trace compression allows compressed traces to be stored more efficiently on disk, and if an on-chip trace compression scheme is used, also reduces the need to provide large amounts of trace port bandwidth.

2.6 Trace Compression

Trace compression is considered a solution to several problems that are hampering the widespread adoption of trace generation for the debugging of all types of software faults. Reducing trace data volume through compression increases the efficiency of utilizing a trace storage medium, which can allow traces from more trace experiments, or longer trace experiments, to be retained. However, the benefits of trace compression are most notable on physical platforms and hardware-emulated platforms. To ensure that all types of software faults can be observed under such platforms, trace data must be streamed off-chip in real-time. Providing sufficient trace port bandwidth to do so can prohibitive in terms of both cost and the number of pins available for the task. Reducing trace data volume through on-chip trace compression can allow traces to be streamed in real-time through lower-bandwidth interfaces, including those potentially requiring fewer physical pins. It can also be considered a solution to some of the problems facing software debugging on
future platforms. Those platforms are expected to produce increasing trace data volumes as processor clock speeds rise, and especially as a growing number of processor cores are placed on a typical SoC or NoC.

The goal of a compression algorithm is to represent input data as a series of symbols, which represent the original data but consume less data volume. Many compression algorithms are composed of a combination of data transforms and encodings which aim to minimize extraneous information contained within input data. The algorithms identify redundancies within an input dataset, which can manifest as patterns, regularity, or predictability. If the algorithm is successful at performing compression, its output dataset can be represented in fewer bits than its input dataset, allowing the data to be stored or transferred more efficiently. In lossless data compression, a decompression algorithm can reverse the data transforms and encodings applied by the compression algorithm to reproduce the original dataset.

In general, there is a tradeoff between the performance of a compression algorithm and its algorithmic complexity, computational intensity, and memory usage. As such, the effort required to perform compression and decompression must be weighed against the benefit of allowing fewer bits to represent a dataset. General-purpose compression algorithms such as lzma2 [72], bzip2 [73], and gzip [74] aim to maximize compression performance over a wide variety of input datasets, by making extensive use of system resources. Those algorithms are most suited to software implementation, and are ideal for inputs of varying or unknown characteristics.

Purpose-built compression algorithms operate by using models of known dataset redundancies and structure to extract maximum redundancy from particular data types. The redundancies within traced data types can vary widely, and depend upon the nature of the observed data structures. A data trace whose value changes seemingly non-deterministically is inherently more difficult to model, and hence to compress, than an execution trace whose flow can be more predictably modeled by various means. A data trace can also exhibit vastly different characteristics depending upon which data structures are observed, and the type of data stored within them by the software-under-debug. Some data types may contain little or no redundancy, or contain redundancy that is difficult to model and extract. As a result, data trace compression typically lags in performance compared to execution trace compression, and if implemented in software, can require much longer compression time [75, 76].
Software-based trace compression algorithms aim to maximize compression performance, but do so by modeling a narrower range of input data types than general-purpose algorithms. Prominent methods include the set of VPC algorithms [75, 77], which implement several multi-pass algorithms and transforms in tandem in order to have options from which to select the smallest representation to output. Such methods are computationally intense, make considerable use of system memory, and consequently require relatively long compression and decompression time. As such, they are mainly useful for reducing the trace storage requirements of traces that have already been collected, or potentially for online compression of traces generated by untimed platforms. However, software-based compression methods are unsuitable for reducing trace port bandwidth on physical platforms, which requires the compression of traces in real-time.

2.6.1 Real-time Trace Compression

Hardware-based trace compression algorithms can serve the dual purpose of reducing both trace storage requirements and trace port bandwidth. By including on-chip hardware to compress traces in real-time, trace data can be streamed off-chip continuously through a low-bandwidth trace port. However, hardware-based methods face far greater constraints than do software-based methods. Ensuring real-time operation requires setting hard bounds on the time allotted to compression tasks, which restricts the complexity of algorithms that can be employed. Constrained hardware area budgets also limit the on-chip hardware resources that can be dedicated to supporting software debugging, which includes any trace compression hardware. The lack of sizable on-chip memory also limits the types of algorithms that can be used. The above constraints have a disproportionately adverse effect on the compression of data traces than they do on the compression of execution traces. Due to the difficulty of modeling data traces, the task of compressing them can be likened to the same problem faced by general-purpose algorithms. Unlike general-purpose algorithms, the constraints under which real-time hardware-based compression of data traces is performed can drastically limit their performance. In fact, in some scenarios only relatively modest compression ratios of 1x–4x can be achieved [76].

Trace experiments typically collect an execution trace to track the flow of execution of the software-under-debug, either with or without also capturing data traces. Since execution traces generally form the basis of every trace experiment, reducing their inherent
redundancies can result in a significant decrease in trace port bandwidth and trace storage requirements. The original schemes that performed real-time on-chip compression of execution traces involved simple techniques with a minimal hardware area footprint. One such method encodes only the difference between consecutive PC addresses [49], which are unlikely to change significantly from one instruction to the next during typical software execution. Such techniques form the basis of some commercial trace generation schemes such as ARM Embedded Trace Macrocell (ETM) v1 [78].

Improvements to simple differential encodings include the filtering of consecutive PC addresses [79, 80], effectively allowing only changes in program flow to be recorded. Such filtering schemes record the addresses of lead instructions in basic blocks, each of which is composed of a continuous instruction sequence after a branch. Determining the intermediate instructions that have been executed requires that a decompression algorithm have access to additional information, such as to the machine code of the executing program, also known as the program binary. For execution traces generated on processors that use a fixed instruction-width ISA, the decompression algorithm may instead interpolate missing instruction addresses based upon the known stride of the ISA. During decompression, such addresses can be reproduced by inspecting the Branch Target Address (BTA) of each preceding branch, and in the case of conditional branches, knowing whether or not the branch was taken.

Other common improvements include the addition of transforms that dynamically adapt to changing source data, such as the general-purpose Lempel-Ziv (LZ) [81] algorithm that is known to be suitable for hardware implementation. Such schemes record encountered input data within a buffer of encoding history, and exploit the temporal locality of recurring inputs by representing them in terms of prior inputs. Kao et al. [79] introduce a three-stage technique which filters non-sequential addresses, encodes the remaining addresses as differences, and then applies the LZ algorithm. Later methods recognized that the encoding of commonly-encountered addresses was particularly well-served by the adaptive Move-to-Front (MTF) transform [18, 82]. Uzelac and Milenkovic [82] first reduce consecutive addresses to a combination of starting address and length, where the Least Significant Bits (LSBits) of the address are encoded as a difference, then apply a two-level MTF scheme to both the length and the Most Significant Bits (MSBits) of the address. Although not geared specifically toward software debugging, encoders based upon both MTF and LZ methods are proposed by Anis and Nicolici [76]. Yet another approach caches basic
block identifiers, while implementing prediction hardware to minimize the coding length of commonly-executed sequences [83]. Trace output is avoided if a software-emulated predictor can generate the same predictions by following the program binary at decompression time. That work, as well as the work presented in Chapter 4, draw a parallel between the goals of execution trace compression and that of the ILP hardware used by many processors. Both attempt to predict which basic block will next be executed, although ILP techniques do so to reduce pipeline stalling, while trace compressors do so to avoid generating output.

Some methods [84, 85], including the one described in Chapter 4, explicitly implement a BP and BTB to predict branch outcomes (taken or not taken) and BTAs, respectively. The units rely on previous branching history to generate predictions of future behaviour. As branches are encountered in the instruction flow, the number of correct predictions that have been made by each unit is counted. When an incorrect prediction occurs, the count is output and then reset. Where the actual BTA cannot be determined from the program binary, it is also output. During decompression, the instruction flow is reproduced by following the program binary while shadowing the state of each predictor in software. The count values are then used to determine the locations within the instruction flow where the shadow predictor will make an incorrect prediction, allowing the actual BTA to be input from the stream if it cannot otherwise be inferred.

2.7 Generalized Software Debugging Procedure

To understand how different aspects of trace generation-based software debugging relate to each other in a typical debugging scenario, an example is provided of the generalized debugging procedure that can be followed to resolve a bug. Using a software debugger that supports trace generation, a typical debug session would take the following form:

1. **Identify that the software is not behaving as expected.**

   Unintended software behaviour can manifest in a number of ways. The spectrum can range from a catastrophic failure causing a fatal exception in hardware, to subtle behavioural changes that do not necessarily cause errors.

2. **Attempt to replicate the faulty behaviour.**

   With the aid of a software debugger, executing the application under controlled conditions will allow Bohrbugs to predictably manifest, while Mandelbugs and Heisenbugs
will appear to manifest in non-deterministic ways, and perhaps after longer periods of time. It is also possible that faulty hardware or transient environmental factors, like temperature or radiation, may be the cause of the fault. These external causes can be identified by executing the software on a development platform that is known to be operating correctly, and by placing the system in a controlled environment, respectively.

3. *Isolate the location of the bug in source code.*

The approach to isolating a bug depends upon the nature of the software-under-debug. Software that is tolerant to intrusive debugging can be debugged by using an execution-control method such as ICE. A faulty code segment can quickly be identified by placing breakpoints at various points of execution, which allow internal data structures to be observed when execution is halted.

Software that is intolerant of intrusive debugging can instead be the subject of a trace experiment. Assuming trace data can be transferred off-chip in real-time, instead of stored in a trace buffer, the following steps can be followed:

(a) Determine the points of execution where traces should be collected.
(b) Configure triggers to collect relevant data structures that will aid the debugging effort.
(c) Enable trace generation and execute the software-under-debug.
(d) Collect trace data until the trace experiment ends.
(e) Decompress the traces (*if need be*) and forward them to a trace analysis tool.

Similar to execution control methods, trace collection occurs at pre-determined instruction addresses known as *tracepoints*. At these addresses, triggers select the data structures to observe, often based upon one or more boolean conditions. Once traces have been collected, they are forwarded to a host machine, where they are decompressed and forwarded to a software tool for analysis. Such tools can be integrated with software debuggers to maintain a link with the program’s symbol table. They are responsible for navigating through large amounts of trace data in order to connect the manifestation of a bug with its cause.
Since more than a single trace experiment may be needed to find a bug, there are efficient ways to perform consecutive trace experiments. An initial experiment may trace a small amount of data over a long time period to identify the approximate location of errors. In each subsequent trace experiment, the time window being traced can be narrowed while the amount of data being traced can be increased. This iterative refinement finally allows a full picture of the software’s data structures to be observed closest to the location of the fault.

4. Correct the faulty segment of code and recompile the software-under-debug.

5. Re-execute the software to observe whether the bug has been resolved.

The software can be executed with a range of inputs to verify that the fault has been corrected. Regression testing can also be done to ensure that no new bugs have been introduced into the software-under-debug.
Chapter 3

Dynamically Instrumenting the QEMU Emulator

3.1 Introduction

There are many benefits to the use of software-emulated development platforms. They allow software development to begin early in the process of hardware-software co-design, and can also act as flexible, low-cost, and standardized development platforms for software developers. As ARM processors have gained traction in the domains of embedded and mobile computing, untimed binary-translation emulators such as QEMU are increasingly featured as part of standardized development platforms for building software geared toward those domains.

The need to reliably debug the types of software faults that are increasingly found in complex software has driven the adoption of instruction-level trace generation on a variety of development platforms. There are a growing number of tools that enable trace generation through software instrumentation, which are suited to observing faults that continue to manifest despite the software changes introduced by instrumentation code. While software instrumentation tools can be useful, many lack support for the ARM ISA, and can also be inefficient under software-emulated platforms. Since those platforms execute software instrumentation code inside a mimicked guest machine, an overhead must be incurred to emulate or simulate the additional instructions.

Some software-emulated platforms such as the QEMU emulator offer specific trace
generation capabilities that effectively emulate the presence of hardware instrumentation. While that allows instrumentation code to be executed directly on the host ISA without the need for binary translation, the existing capabilities of many platforms can be described as rudimentary. The QEMU emulator offers only static instrumentation that is enabled at guest machine boot-up time, and lacks the ability to instrument data structures other than the PC register. The option to perform trace qualification is also absent, as is the contextual awareness to enable debugging under a context-switching OS.

This chapter describes the addition of instruction-level trace generation support to the ARM instruction-set emulation of the open-source QEMU emulator. Trace experiment control is exercised through a standard open-source GDB debugger client, and interpreted by enhancements to QEMU’s internal GDB stub. That allows for the creation and triggering of tracepoints that continuously collect registers and memory addresses. The added instrumentation is inserted during the code translation process, allowing unaltered code to execute on the guest processor. The process of adding dynamic instrumentation is described as part of five operating modes. The proposed method is also made aware of context switching within the Linux kernel of the guest machine, allowing traces belonging to a specific Linux PID to be qualified directly from the GDB client. The combination of features allows for significant reductions in debugging time, and allows for large savings in trace storage space.

3.1.1 Contributions of this Chapter

This chapter introduces a dynamic trace generation infrastructure for the QEMU emulator which contributes to the state of the art in several ways:

- Demonstrating a minimally-invasive technique of achieving hardware-level contextual awareness of Linux PIDs, by requiring only a small change to a standard OS kernel.
- Presenting a method to prevent memory abort exceptions while performing run-time trace collection of memory addresses that lack a valid virtual-to-physical address translation in all contexts.
- Identifying the changing bottlenecks that dominate a software-emulated platform when trace generation is applied, depending upon the proportion of Floating-Point (FP) instructions present in the emulated software-under-debug.
3.2 The QEMU Emulator

3.2.1 Instruction Translation

Recent versions of QEMU have employed an extensible interface that enables the emulation of a variety of guest ISAs on a number of host ISAs. All guest code is first translated into an intermediate form known as Tiny Code Generator (TCG) bytecode, before being translated into the required host ISA. This design obviates the need to maintain separate codebases for translating code between all supported guest and host ISAs.

In the example seen in Figure 3.1, a set of three ARM guest instructions are dynamically translated to TCG instructions, before those instructions are themselves translated to x86 host instructions. In the example, each of the guest instructions require multiple TCG

---

Fig. 3.1: Example of QEMU Dynamic Binary Instruction Translation from ARM to x86 ISAs

- Quantifying the decreases in both trace volume and execution time that are possible when run-time trace qualification is enabled on software-emulated platforms.

The above contributions provide insights which can generally inform software debugging and the development of trace generation infrastructure, on both software-emulated and other platforms. To the author’s knowledge the contributions have been applied for the first time to the ARM ISA emulation of the QEMU emulator, though the described techniques can be considered portable to other software-emulated platforms and ISAs.
instructions to reproduce their functionality. The TCG instruction `mov_i32` is used to copy register operands to/from TCG temporary variables such as `tmp` or `tmp2`, while `movi_i32` does the same for immediate values. That allows arithmetic instructions such as `add_i32` to both operate on, and to store their results to, only TCG variables. The example set of TCG instructions constitute a data structure known as a QEMU Translation Block (TB), which ends in the TCG branch instruction `goto_tb`. Each TB has a corresponding translation to a set of host instructions. Once a TB has been translated, it is typically placed into a translation cache maintained in host memory to allow for faster code re-execution.

The emulation of a typical register-register guest instruction involves loading a guest register from a data structure in host memory, operating on it, and storing it back into the same data structure. In the example of Figure 3.1, the x86 register `ebx` is used to store the base address of the guest register file, starting with `r0`. To emulate the guest instruction `mov r5,r0`, the guest register `r0` is first loaded from the address pointed to by `ebx`, and placed into the host register `ecx`. It is then stored into the address pointed to by `ebx+0x14`, which is the memory location of the guest register `r5`.

### 3.2.2 Dynamic Binary Instrumentation

To emulate the presence of instruction-level hardware instrumentation within QEMU, the tracing infrastructure must allow unmodified guest instructions to execute on the guest. Any instrumentation should then be added during the binary instruction translation process. The proposed method achieves this by adding custom TCG instrumentation instructions to the TB as each guest instruction is translated. In the example of Figure 3.2, a trace of the PC register is collected by inserting a call to the helper function `gen_helper_collect_pc` immediately following the translated guest instruction. The helper function inserts TCG instructions that collect the guest PC register, whose value is then passed to an `fprintf` library call that writes the PC to the host filesystem.

An important consideration of building an instrumentation scheme is the ability to exercise flexible control over where and when that instrumentation will be active. Some emulators offer limited static instrumentation, where either the instrumented data structures are fixed, or the ability to enable trace collection at opportune execution points is absent. In the latter case, the inability to select a `trigger` address at which to begin trace collection means that traces are also collected during periods of execution that are useless.
to the debugging effort. This drawback can drastically increase debugging time and trace data volume. Instead of static instrumentation, the proposed method offers the ability to dynamically add and remove instrumentation-related TCG code. Instrumentation control is exercised through enhancements made to the QEMU GDB stub, which are explained in Section 3.5.

The proposed scheme allows uninstrumented code to execute at full speed until a *tracepoint* is enabled through a GDB client. Once enabled, TCG instrumentation is added to only the guest PC register. That allows a user-configurable trigger address to be checked against the current PC address during the translation of each instruction. To ensure that TBs within the host memory translation cache are updated to the new PC-instrumented instructions, the translation cache is flushed when starting a trace experiment. When a
trigger address is reached, instruction-level trace generation of registers and memory addresses begins. Additional TCG instrumentation is added to observe the data structures that have been specified using the tracepoint actions keyword. The translation cache is again flushed to ensure that fully-instrumented code replaces the earlier PC-instrumented version. Instruction execution continues until the desired code portion has been traced. The tracepoint can then be disabled through the GDB client, which causes a final flushing of the translation cache to ensure that faster uninstrumented code replaces its fully-instrumented predecessor.

This dynamic scheme effectively uses two tiers of instrumentation: a lightweight PC trace to establish whether a trigger address has been reached, and a full instruction-level trace of the desired data structures. Since adding instrumentation code reduces execution speed in a guest machine, the proposed scheme allows for a minimal performance impact when executing portions of code that need not be traced. The proposed method also allows tracepoints to be dynamically inserted and removed, which can even be scripted to maximize flexibility. As shown in Section 3.6, all of these features enable significant reductions in trace data volume and trace collection time compared to static instrumentation methods.

### 3.3 Overview of Proposed Method

The presented method adds support for dynamically adding instruction-level instrumentation to the QEMU 1.4.0 emulator. The presented prototype supports guest machines with the ARM instruction set, and allows for the collection of guest registers and guest memory contents. As seen in Figure 3.2, instrumentation control is exercised through a standard GDB client executing natively on a host workstation. Trace experiments involve creating a tracepoint at any guest program counter trigger address, and defining a set of associated actions that are carried out when the tracepoint has triggered. Supported actions include the collection of up to 16 registers and 16 memory addresses, which can be defined in symbolic, relative, or absolute notation. Any one of 5 user-configurable modes can be selected for a tracepoint. Depending upon the mode, the user-defined actions will be repeated on either every subsequent instruction, all user-mode instructions, or with the application of a small Linux kernel patch, only when executing a specified PID. Trace collection continues until a tracepoint is disabled or deleted. Collected traces are written in ASCII to the host filesystem for inspection after trace experiment completion.
In order to achieve the described functionality, several additions are made to the QEMU GDB stub and ARM instruction translation routines. The GDB stub is given the ability to decode RSP tracepoint packets sent by any GDB client, and to perform high-level control over trace generation. Likewise, modifications to the ARM instruction translation routines of QEMU act to perform low-level trace generation control. That includes the injection of instrumentation instructions into translation blocks composed of TCG intermediate bytecode, checking trigger addresses, and performing trace output.

3.4 Modes of Operation

When debugging software faults using trace generation, care must be taken to select only traces that benefit the debugging effort. Increasing the number of instrumented data structures, length of time a trace is collected, or the scope of the collection will lead to increases in trace volume. In the case of emulated hardware instrumentation, execution speed can also be negatively impacted.

The proposed method allows the above factors to be controlled through the tracepoint features of the GDB debugger. However, varying the scope of trace generation is especially difficult for emulated hardware instrumentation schemes. For this reason, the proposed trace generation infrastructure offers 5 modes of operation that can be set directly from a GDB client. Table 3.1 shows that the modes correspond to different scopes of trace collection under the Linux operating system. In ascending order, the given modes collect traces for progressively smaller subsets of executing code:

Mode 1:
Allows the capturing of traces for every executed instruction, regardless of the underlying operating system or application. While a similar feature is already offered by some emulators, the methods are either static or controlled the proprietary interfaces. The presented implementation is both dynamic and controlled through a standard GDB client.

Modes 2 and 3:
All software executing in user-mode is instrumented. For each instruction, the tracing infrastructure determines whether the emulated ARM processor is in user-mode
Dynamically Instrumenting the QEMU Emulator

by checking that the 4 least-significant bits of the Current Program Status Register (CPSR) are zero. While mode 2 does not differentiate between running Linux processes, mode 3 additionally reports the process PID when a process-switch event occurs. The latter mode is useful for the debugging of faults that manifest when multiple processes interact, though requires a small two-line Linux kernel patch. This is needed in order to memory-map the PID of upcoming Linux processes to a fixed memory address known to the tracing infrastructure (seen as ReservedAddr in Figure 3.3). The patch is applied to the Linux kernel’s process scheduler, at a point when the upcoming process PID is known, but before the process-switch event has taken place. Figure 3.3 also shows that other events requiring a switch to kernel-mode, such as system calls, are unaffected by the change. After a tracepoint is triggered in mode 3, the tracing infrastructure issues a guest memory read of the PID from ReservedAddr on every instruction. A change in the value indicates that a process switch has occurred, resulting in the outputting of the new PID to the trace storage file.

Modes 4 and 5:

Enables the filtering of traces corresponding to the execution of a single PID. As with mode 3, the tracing infrastructure first determines whether the processor is in user-mode. If so, ReservedAddr is read from guest memory to determine if the PID of the executing process matches the PID whose traces are desired. In mode 4, the desired PID number can be explicitly set through a GDB client. The functionality of mode 5 is similar, though the PID does not need to be known in advance. A trace experiment can instead be created to capture the traces of a process that has not yet launched. In mode 5, the PID present in ReservedAddr at the time the trigger address is hit serves as the PID filter for the remainder of the trace experiment. This mode can be advantageous for a process-under-debug whose launch time is nondeterministic. Since the operating system will assign the process-under-debug a PID number according to its execution order relative to other processes, it can be difficult to determine a given PID number in advance.
### Table 3.1: Modes of Operation

<table>
<thead>
<tr>
<th>Trace Mode</th>
<th>Scope of Instrumentation</th>
<th>Kernel patch required?</th>
<th>Trace state variable $$pid$ setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All code</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>User-mode code</td>
<td>No</td>
<td>65534</td>
</tr>
<tr>
<td></td>
<td><em>No PID listed with trace output</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>User-mode code</td>
<td>Yes</td>
<td>65534</td>
</tr>
<tr>
<td></td>
<td><em>List PID with trace output</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Single user-mode PID</td>
<td>Yes</td>
<td>0–32768</td>
</tr>
<tr>
<td></td>
<td><em>Filter by PID number</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Single user-mode PID</td>
<td>Yes</td>
<td>65535</td>
</tr>
<tr>
<td></td>
<td><em>Filter by PID of triggering process</em></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 3.5 Trace experiment control

The QEMU GDB stub presently includes support for basic debugging features such as breakpoints, guest register inspection, and guest memory inspection. Even though standard GDB clients have long supported a framework for tracepoint control on supported target machines, neither the QEMU GDB stub, nor any of its emulated targets currently support those features. To provide a standardized high-level control interface for the trace generation infrastructure, the proposed method adds tracepoint support to the existing execution-control debugging features of the QEMU GDB stub.

#### 3.5.1 RSP Trace Packets

Changes made to the QEMU GDB stub allow a standard GDB client to run trace experiments on a QEMU-emulated ARM processor. The GDB stub packet handler has been modified to support a range of RSP trace packets [86], including those for initializing ($T_{\text{init}}$), starting ($T_{\text{start}}$), and stopping ($T_{\text{stop}}$) of trace experiments. Packets for the enabling ($T_{\text{enable}}$) and disabling ($T_{\text{disable}}$) of individual tracepoints are also supported, though the most significant additions add support for the $TDP$ range of packets. Those are the packets that define the creation of tracepoints, their trigger addresses, and the actions taken when such addresses are hit.
The described implementation allows a number of actions to be associated with a trigger address. The collection of 16 registers and 16 memory locations is supported by TDP packet “R” and “M” actions, respectively. However, some actions may also be encoded by a GDB client as agent expressions [86], which are entire micro-programs encoded in a special GDB-specific bytecode. When interpreted by a target machine, agent expressions enable runtime trace collection, manipulation, and qualification without user intervention. For the purposes of this chapter, the decoding of agent expressions is limited to those opcodes that enable register and memory trace collection, as well as the manipulation of some trace state variables. Such variables are used within trace experiments to perform simple runtime data processing during trace collection. One such variable is used by the trace infrastructure to pass tracepoint conditions from GDB client to stub. As shown in Table 3.1, the trace state variable $pid can be used to select the mode of operation for the trace infrastructure. While the default mode 1 is assumed when no trace state variable is
defined, setting $\text{pid}$ to a value between 0–32768 filters only traces from that PID. *Modes 2, 3 and 5* can be selected by setting the value $\text{pid}$ to special values outside of the normal Linux PID range of 0–32768. When $\text{pid}$ takes the value 65534, the tracing infrastructure detects whether a kernel patch is present to select between *modes 2 and 3*. Mode 5 is similarly selected by setting $\text{pid}$ to 65535.

### 3.5.2 Example Trace Experiment

An example of a trace experiment initiated from a GDB client to collect a variety of data structures is given in Listing 3.1, with its corresponding trace output seen in Listing 3.2. The example shows *mode 4* trace collection of a single Linux PID executing on a QEMU-emulated ARM core.

The creation of a trace experiment in Listing 3.1 begins with the processor stopped at a breakpoint. The trace state variable $\text{pid}$ is initially created, which is used to create a *mode 4* conditional tracepoint to collect traces from PID 1234. That generates agent expression bytecode notifying the GDB stub to record only traces belonging to that PID value. Assuming that the debug symbols for the process-under-debug have already been loaded into the GDB client, the relative trigger address *main+8* is resolved directly by the client, and is forwarded to the GDB stub as an absolute address.

Every tracepoint must have a set of associated actions to perform during software execution. While GDB specifies numerous actions, the *collect* keyword is currently supported for observing registers, and for observing memory addresses expressed in symbolic, relative, or absolute terms. In Listing 3.1, the pc and r2 registers are collected for every instruction including, and following, the trigger address. The contents of two memory addresses are similarly collected. The address corresponding to the symbol *variable* is resolved by the GDB client before being sent to the stub, which then collects its memory word for each executed instruction. The absolute memory address *0xffff0020* is also collected, even though the memory region it occupies is protected. That is possible because the entirety of the memory space is accessible to the emulated hardware instrumentation.

The ASCII trace output from the Listing 3.1 trace experiment can be seen in Listing 3.2. The first line of output shows the PID of the process from which traces are collected. The trigger address *main+8* appears in its resolved form as the first traced instruction address *0x8988*. For each executed instruction belonging to PID 1234, the unindented instruction
```sh
$ arm-none-eabi-gdb // GDB for ARM EABI targets
(gdb) target remote localhost:1234 // connect to QEMU GDB stub
(gdb) file program_under_debug // import debug symbols
Reading symbols from program_under_debug...done.
(gdb) tvariable $pid // trace variable containing PID
Trace state variable $pid created.
(gdb) trace *(main+8) if ($pid == 1234) // conditional tracepoint
Tracepoint 1 at 0x8988.
(gdb) actions // tracepoint actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
> collect $pc // program counter
> collect $r2 // data register
> collect variable // variable (symbolic memory address)
> collect 0xffff0020 // absolute memory address
> end
(gdb) tstart // begin trace experiment
(gdb) continue // resume software execution
Continuing.
```

Listing 3.1: Example of GDB Trace Experiment Configuration with Mode 4 PID Filter

address corresponds to the action `collect $pc`, followed by any remaining `collect` actions. Among them is the resolved form of the symbol `variable`, which appears as the memory address 0x84904.

### 3.5.3 Avoiding “Memory Abort” Exceptions

In a context-switching environment, the collection of absolute memory address across virtual address spaces may result in a situation where a virtual-to-physical address translation does not exist. This means that care should be taken to collect valid memory addresses in `mode 1`, or a `memory abort` exception may result.
3.5 Trace experiment control

```plaintext
<pid: 01234>       // below traces belong to PID 1234
00008988          // $pc -- trigger address
r2=be93adbc       // register r2
mem[00084904]=00000000  // symbolic variable memory address
mem[fff0020]=000004d2  // 0xffff0020 memory address
0000898c          // $pc -- next instruction
r2=be93adbc
mem[00084904]=00000000
mem[fff0020]=000004d2
00008990          // $pc -- next instruction
r2=be93adbc
mem[00084904]=00068604  // variable contents changed
mem[fff0020]=000004d2
00008994          // $pc -- next instruction
r2=8f3d785a       // $r2 contents changed
mem[00084904]=00068604
mem[fff0020]=000004d2
00008998          // $pc -- next instruction
r2=8f3d785a
mem[00084904]=00068604
mem[fff0020]=0000a6d4  // 0xffff0020 contents changed
00054e04          // $pc -- branch target
r2=8f3d785a
mem[00084904]=00068604
mem[fff0020]=0000a6d4
00054e08          // $pc -- next instruction
r2=8f3d785a
mem[00084904]=00068604
mem[fff0020]=0000a6d4
...
```

Listing 3.2: Example of ASCII Output of GDB Trace Experiment from Listing 3.1
For modes 2–5, the proposed method instruments all code, while selectively filtering traces based upon the runtime value of the CPSR and/or PID filter. For those modes, a special workaround is provided to avoid reading invalid addresses while in kernel-mode. Since modes 2–5 do not generate traces of kernel-mode code, TCG code is generated to decide at runtime whether to read a given user-space memory address, or to deflect the memory read to a dummy kernel address. A memory abort exception is thus avoided while in kernel-mode by reading a valid kernel address, instead of an invalid user-space address. Such deflected memory reads are then not output as part of the trace. To the author’s knowledge, this is the first reported use of such a solution within any similar trace generation infrastructure.

3.6 Experimental Results

To evaluate the proposed tracing infrastructure, a software debugging scenario that exploits trace generation is used to present the results of two experiments. In such a scenario, a developer seeks to collect an instruction-level trace from a single Linux process, to the exclusion of other processes executing under the Linux operating system. The experiments evaluate the time and storage space needed to collect the execution trace of a single Linux process. While execution traces can be used in code coverage analysis, they also represent the minimum amount of instrumentation needed to generate a trace useful for software debugging. Both experiments collect the execution traces of 15 applications of the MiBench Benchmark Suite [87], an established benchmark in related trace infrastructure research [18]. All data is collected with QEMU 1.4.0 emulating an ARM926 processor, executing on a 2.4 GHz AMD Opteron 8327 host. In both cases, the guest machine is running Linux kernel 2.6.32.27, which includes the aforementioned kernel patch.

3.6.1 Execution Time

In the first experiment, the execution time overhead of using several instrumentation schemes is compared in Table 3.2. Baseline uninstrumented execution times are first established through normal application execution within the emulated guest machine. To collect only the execution traces of individual applications, the use of the proposed method in mode 5 is compared to the use of both software instrumentation and emulated hardware instrumentation.
Among established software instrumentation tools there is currently little support for ARM instruction-level tracing of Linux processes (see Section 2.5.1), making a direct comparison rather difficult. Instead, a script is created to execute on the guest machine for capturing the execution trace of an individual application. The script loads an executable into a GDB 6.8-3 client on the guest machine, and uses a looped \texttt{si} command to single-step through program code. Setting the PC register contents to auto-display with the command \texttt{disp/i $pc} allows the PC address and its disassembly to be output on each step. By redirecting standard output to a file on the guest filesystem, an execution trace can be captured. It should be noted that this differs from the Figure 2.5 scenario in which a GDB client executing on the \textit{host} machine is otherwise used to control emulated hardware instrumentation. Due to the long runtimes involved for some applications, trace collection times for the script are approximated at the peak collection rate of 160 instructions per second. Slow performance can be attributed to the massive overhead added by the GDB client between executed instructions, and is to be expected from a tool unintended primarily for trace collection. On every step, such overheads include setting a breakpoint on the next instruction, transferring control to/from GDB, and mandatory instruction disassembly. The script then cannot be considered a practical alternative to the established software-instrumented tracing methods discussed in Section 2.5.1, which have yet to offer ARM instruction-level tracing of Linux processes. Nevertheless, Table 3.2 shows that the proposed method offers on average between 3–4 orders of magnitude faster trace collection than the software-instrumented script.

The proposed tracing infrastructure can more reasonably be compared to other emulated hardware instrumentation methods. For that reason, the result of collecting execution traces with the QEMU debug mode is also presented in Table 3.2. The debug mode is a static instrumentation method that must be enabled with a command-line switch when the emulator is launched. While the debug mode can capture the address of every instruction executed on the guest, it lacks contextual awareness of guest operating systems, and as such cannot differentiate between Linux processes. This means that execution traces belonging to all other code must be collaterally captured in addition to those of the desired Linux process. Traces resulting from guest machine boot-up are discarded by zeroing the trace output file prior to MiBench application execution. Execution traces produced by the QEMU debug mode also store the corresponding location of the guest instruction in host memory. The time required to write that additional information to disk also contributes
to increases in execution time.

The benchmarks in Table 3.2 are separated into two categories based upon their use of FP instructions: *Integer benchmarks* and *Mixed Int/FP benchmarks*. The results show that the uninstrumented benchmarks containing FP instructions have far longer execution times, despite the fact that they contains fewer instruction than many of the other benchmarks. The difference can be attributed to the relatively inefficient translation of FP instructions from guest to host ISAs, whose emulation has been shown to be many times slower than other types of instructions [41].

The benchmarks containing FP instructions provide insight into the nature of the execution time overhead experienced by the proposed method, since their execution speed is most likely bound by host CPU performance than by any other factor. When instrumented using the proposed method, the *Mixed Int/FP benchmarks* experience only an average 2.9x execution time overhead compared to their uninstrumented execution time. In contrast, the *Integer benchmarks* experience an average 47x overhead. It can be reasoned that *Integer benchmarks* are subject to another type of bottleneck, most likely that of host disk I/O bandwidth.

The proposed method collects the traces of *Integer benchmarks* and *Mixed Int/FP benchmarks* at 9.6x and 0.7x the speed of the QEMU debug mode, respectively. The slowdown experienced by the latter group of benchmarks can be attributed to the overhead of performing run-time trace qualification in an environment bound by host CPU performance. Compared to static instrumentation methods, the additional overhead of instrumenting the CPSR register and memory address containing the PID contribute to longer execution times. However, the developer may find it desirable to incur such an overhead at run-time in order to significantly reduce the volume of generated trace data, by preventing the collection of traces that are unhelpful to the debugging effort. That ability can be particularly useful when debugging intermittently-occurring, or infrequently-occurring bugs. Even with the added overhead, applications with a low to moderate proportion of FP instructions may still execute faster when instrumented with the proposed method rather than with the QEMU debug mode, as demonstrated by the *fft* benchmark.
3.6 Experimental Results

Table 3.2: Execution Trace Collection Time for the MiBench Benchmark Suite on a QEMU ARM Guest Machine

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Proportion of Floating-Point Instructions</th>
<th>Instrumented Execution Time (s)</th>
<th>Baseline Uninstr. Execution Time (s)</th>
<th>Software Emulated Instruction Script</th>
<th>Emulated HW Instr. QEMU Debug Mode</th>
<th>Proposed Method (Mode 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm_c</td>
<td>0%</td>
<td>0.2</td>
<td>265504</td>
<td>109.7</td>
<td>10.9</td>
<td></td>
</tr>
<tr>
<td>bf_e</td>
<td>0%</td>
<td>0.7</td>
<td>679719</td>
<td>280.9</td>
<td>30.5</td>
<td></td>
</tr>
<tr>
<td>cjjpeg</td>
<td>0%</td>
<td>0.5</td>
<td>737226</td>
<td>304.6</td>
<td>30.4</td>
<td></td>
</tr>
<tr>
<td>djjpeg</td>
<td>0%</td>
<td>0.3</td>
<td>186550</td>
<td>77.1</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>gsm_d</td>
<td>0%</td>
<td>0.1</td>
<td>89951</td>
<td>37.2</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>mad</td>
<td>0%</td>
<td>0.3</td>
<td>150360</td>
<td>62.1</td>
<td>7.0</td>
<td></td>
</tr>
<tr>
<td>rijndael_e</td>
<td>0%</td>
<td>0.4</td>
<td>282251</td>
<td>116.6</td>
<td>12.9</td>
<td></td>
</tr>
<tr>
<td>sha</td>
<td>0%</td>
<td>0.3</td>
<td>780282</td>
<td>322.4</td>
<td>33.1</td>
<td></td>
</tr>
<tr>
<td>stringsearch</td>
<td>0%</td>
<td>0.2</td>
<td>24683</td>
<td>10.2</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>tiff2bw</td>
<td>0%</td>
<td>0.4</td>
<td>196692</td>
<td>81.3</td>
<td>8.8</td>
<td></td>
</tr>
<tr>
<td>tiff2rgba</td>
<td>0%</td>
<td>0.9</td>
<td>150433</td>
<td>62.2</td>
<td>7.9</td>
<td></td>
</tr>
<tr>
<td>tiffdither</td>
<td>0%</td>
<td>1.1</td>
<td>2205468</td>
<td>911.4</td>
<td>95.0</td>
<td></td>
</tr>
<tr>
<td>tiffmedian</td>
<td>0%</td>
<td>0.7</td>
<td>889169</td>
<td>367.4</td>
<td>36.6</td>
<td></td>
</tr>
<tr>
<td>Total (Integer benchmarks)</td>
<td></td>
<td>6.1</td>
<td>6638288</td>
<td>2743.1</td>
<td>286.6</td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td>6.7%</td>
<td>7.7</td>
<td>260909</td>
<td>107.8</td>
<td>32.2</td>
<td></td>
</tr>
<tr>
<td>lame</td>
<td>31.2%</td>
<td>144.4</td>
<td>472633</td>
<td>195.3</td>
<td>406.4</td>
<td></td>
</tr>
<tr>
<td>Total (Mixed Int/FP benchmarks)</td>
<td></td>
<td>152.1</td>
<td>733542</td>
<td>303.1</td>
<td>438.6</td>
<td></td>
</tr>
</tbody>
</table>

3.6.2 Trace Volume

In addition to the time savings realized with the use of the presented trace collection infrastructure, the ability to filter unwanted traces before they are recorded to disk can be just as valuable. To quantify the effect of such trace qualification on trace volume, the
results of a second experiment are shown in Figure 3.4. The number of collected instructions is shown for 3 trace qualification modes: single-PID code, user-mode code, and all code.

The execution traces of the same applications from the prior experiment are collected. However, this experiment attempts to reproduce a software debugging scenario in which the MiBench application contends for CPU time with another active process whose traces are not desired. To achieve such a scenario, the MiBench application is executed while a stress process executes on the guest machine with 50% CPU utilization. That allows the overhead of collecting unrelated traces to be quantified in terms of trace volume. It should be noted that there are efficiency differences in translating different types of guest instructions to host instructions. In any fixed-length period of time between context-switch events, there is a difference between the number of stress instructions that may be executed compared to MiBench application instructions. This property may be observed in Figure 3.4 as the small difference in executed instructions between single-PID code and user-mode code for most of the benchmarks.

The results show that the aggregate ASCII-encoded MiBench execution traces from single-PID code, user-mode code, and all code utilize 9.89 GB, 15.46 GB, and 299.54 GB of trace storage, respectively. That corresponds to a trace volume reduction of 96.7% and 94.8% from the qualification of single-PID and user-mode traces, respectively. The ability of the presented method to qualify traces at the single-PID and user-mode levels can thus result in drastically lower trace volumes compared to methods that collect traces of all executing code.

3.7 Chapter Summary

This chapter recognizes the increasing use of emulated platforms for software development and debugging. It presents a novel software debugging infrastructure for the QEMU emulator, suited to observing complex Mandelbugs in emulated ARM software. The instruction-level trace generation infrastructure allows trace experiments to be dynamically controlled through a standard GDB debugger client. The proposed method emulates the presence of hardware instrumentation, allowing unobtrusive logging of registers and memory addresses during software execution. The unique ability to qualify traces in five different modes allows the scope of trace generation to be narrowed as needed, down to the level of a single Linux process. That allows the presented method to collect the execution traces of a
Fig. 3.4: Number of Instructions Captured by Qualification Level while Recording Execution Traces of the MiBench Benchmark Suite
Linux process on average between 9.6x–0.7x the speed of an existing static instrumentation scheme, with 96.7% less trace data volume. Compared with execution traces extracted using a software-instrumented script, the presented method performs on average between 3–4 orders of magnitude faster.
Chapter 4

Architecture-Aware Real-Time Compression of ARM Execution Traces

4.1 Introduction

The development of increasingly complex software has given rise to software faults that were once considered rare, ones that can be difficult or even impossible to resolve using traditional debugging methods. Many of those faults can appear to manifest non-deterministically, such as when a complex set of conditions is encountered, or a seldomly-used execution path is followed. Such faults may even arise after long intervals of seemingly correct behaviour, making them especially difficult to resolve using execution-control debugging.

The need to reliably debug complex and timing-related faults has driven the need for unobtrusive debugging methods that enable software execution to be observed in detail. Many physical SoC platforms are increasingly including hardware support for instruction-level trace generation, which allows a developer to potentially observe all types of software faults during real-time software execution. However, the volume of trace data generated on-chip by instruction-level instrumentation can easily reach several gigabytes per second, as shown by Example 2.3. That volume is expected to increase as a growing number of on-chip processor cores are instrumented concurrently to provide multi-threaded software debugging.
Storing large volumes of trace data within on-chip trace buffers can be problematic, since even the largest of such buffers are quickly filled during real-time software execution. They can be especially impractical for debugging intermittent software faults that can take hours or days to manifest. To allow more than a small window of execution to be observed, an SoC must be capable of continuously transferring traces off-chip as they are generated. However, implementing a trace port supporting such high bandwidth can be cost-prohibitive or even impossible for many designs.

One of the most promising solutions to the problem involves employing on-chip trace compression techniques to reduce the volume of trace data that must be streamed off-chip. In particular, execution traces have been shown to contain a significant amount of redundancy that can be exploited with the addition of on-chip trace compression hardware. Since those traces generally form the basis of any trace experiment, compressing them can have a large impact on the total trace data volume resulting from a trace experiment.

In general, data compression is the result of transforms applied to source data in order to reduce its size. A compression algorithm can typically be viewed as a combination of Source Modeling and Encoding phases, as depicted by Figure 4.1. Modeling source data involves identifying patterns and redundancies in its structure, allowing the data to be represented in a more compact form. The resulting abstracted representations can then be encoded to achieve further reductions in size. The compression scheme introduced in this chapter targets the redundancies inherent within the execution traces of the ARM ISA, which is an architecture that is especially prominent in embedded and mobile computing.

The proposed scheme leverages both novel and existing sources of redundancy reduction in order to model execution traces. The augmented model allows a decompressor to better predict the content of an execution trace, allowing it to be represented by fewer symbols. The encoding of those symbols then exploits any residual redundancy remaining in the trace due to spatial and temporal redundancy.

This chapter presents an on-chip trace compression scheme, suited to real-time streaming of execution traces off-chip through a low-bandwidth trace port. The proposed method achieves a compression performance of 0.0177 or 0.0208 bits per instruction, depending upon available hardware area. To the author’s knowledge, the described method offers the highest performance of any such scheme in the literature. While potentially saving both development time and cost, high-performance trace compression is also expected to become more important in the future. As a growing number of multi-threaded and other traces are
forced to share the same trace port, a scalable solution to reducing trace data volume will be vital in enabling software debugging on future development platforms.

4.1.1 Contributions of this chapter

The on-chip execution trace compression scheme introduced in this chapter improves upon other techniques in several ways:

- Identifying expanded sources of redundancy within typical execution traces due to:
  - The widespread use of linked branches within compiler-generated code, which are defined within many Instruction-Set Architectures (ISAs).
  - The predictable, three-way compiler-driven movement of function return-addresses between a program counter, link-register, and stack.

- Leveraging identified sources of redundancy by achieving compression through a novel definition of synchronization rules between a compressor and decompressor.
Creating a desirably asynchronous method in which a compressor need only track the validity of on-chip link-register and stack contents, while requiring a decompressor to partially emulate those hardware structures to reconstruct their contents.

- Optimizing the way in which BP and BTB events have been counted by past works to produce smaller encodings.
- Analyzing spatial and temporal redundancies within symbol streams, and identifying encodings and transformations to systematically extract those redundancies.
- Showing that a significant temporal redundancy exists in some symbol types, which can be exploited to increase compression performance at the cost of hardware area.
- Considering alternative mappings of integers to codewords, and showing that a bijective mapping for differential symbol types results in compression gains.
- Demonstrating that stream separation of disparate symbol types allows the redundancies of each to be extracted separately, resulting in smaller encodings.

The contributions are represented within this chapter and in its proposed compression scheme, which to the author’s knowledge has the highest overall performance of all comparable schemes in the literature.

### 4.2 Architecture-Aware Trace Compression Overview

The proposed Architecture-Aware Trace Compression (AATC) scheme consists of both Tracking and Encoding Modules, which can be equated to the Source Modeling and Encoding stages in Figure 4.1, respectively. The Tracking Module harnesses all five sources of redundancy reduction seen in Figure 4.1 to create a small, abstracted representation of the execution trace dataset. Its functionality depends upon using knowledge of the structure of a processor core and its software to reduce the size of the trace. The abstracted dataset is subsequently encoded into a smaller representation by the Encoding Module, and ultimately streamed off-chip, where a software decompressor can reproduce the execution trace offline.
4.2 Architecture-Aware Trace Compression Overview

4.2.1 Usage Scenario

This chapter considers the scenario in Figure 4.2, wherein compressed execution traces are streamed off-chip in real-time. A developer working on a host workstation within a software debugger, such as GDB, initiates a trace experiment by issuing commands to a software driver. The driver controls a combined debugging and tracing interface, such
as one based upon the Nexus-5001 standard [88], which includes a trace port. Issued commands include trigger information, such as conditions for starting and ending the trace experiment, as well as a command to start remote execution of the program on the target processor core. A stream of retired instructions is input into the included BP and BTB units, which are independent from any such units included within the target processor. The Tracking Module takes as input the most recent retired instruction, its address, and its effect on the BP and BTB. The resultant outputs are further compressed by the Encoding Module, before being streamed in real-time to the host through the trace port. Compressed traces are collected by the host until the trace experiment is complete. They can then be decompressed offline with software that has access to the original binary of the software-under-debug. The resulting raw traces can subsequently be inspected within the software debugger to determine the execution path of the program.

4.2.2 Tracking Stage

It is recognized that most software debugging is being performed on user-level code, written in high-level languages, and executed under established operating systems. In this typical software development scenario, high-level code is compiled to adhere to a specific ISA and Application Binary Interface (ABI), while making use of performance-enhancing ISA features or extensions offered by the architecture. The inspection of standard program binaries reveals that generated machine-code follows a regular form, especially when branching to subroutines. Code compiled for many ISAs, including those of ARM and MIPS, makes wide use of linked branches that concurrently branch while writing the branch return address to a specific register. If further linked branches are invoked, the compiler will move older return addresses to the stack. The Tracking Module exploits the use of such ISA features and compiler conventions to track the movement of return addresses between registers and the stack, without the need to replicate the data stored within them. When branching behaviour conforms to expected norms, the Tracking Module can avoid generating symbols. Alone, the tracking is found to reduce the generation of BTA symbols by 52%. At decompression time, a shadow link-register and partial stack (containing only return addresses) are maintained in software to enable branches to return to the correct addresses.

The Tracking Module also takes advantage of the branching history maintained by the included BP and BTB units. While other schemes have used those same structures,
enhancements to the counting of BP and BTB events are identified and implemented in the Tracking Module. Under equivalent conditions, those enhancements generate a smaller abstracted dataset than other schemes.

4.2.3 Encoding Stage

To encode the symbols generated by the Tracking Module, an analysis of the spatial and temporal redundancy of each symbol type is performed. Two variants of an Encoding Module are presented, denoted fAATC and xAATC. They are differentiated by a tradeoff between encoding performance and hardware area. Both variants encode output data as three separate streams to exploit the redundancies of individual data types, allowing smaller encodings to be achieved [75]. Based upon a comparison of 7 encoding schemes, Fibonacci encoders are chosen for the compression of BP and BTB count symbols due to their performance and small hardware area. The encoder variants differ in their handling of BTA addresses, which are shown to contain both spatial and temporal locality.

The fAATC encoder requires a small hardware area, by exploiting only the spatial locality of BTA symbols through a series of transformations. A differential encoding of the BTA stream is first shown to produce a symbol distribution of smaller values. A comparison of encoding schemes and integer mappings then reveals the best-performing combination. For the BTA stream, a Variable-Length (VarLen) Encoding scheme [84] is found to achieve the greatest compression in conjunction with the proposed bijective integer mapping.

The xAATC encoder adds a higher-performance compression stage to the fAATC design, which additionally exploits the temporal locality of the BTA symbol stream. An MTF transform implements the on-chip encoding history depicted in Figure 4.1, which has been shown to be particularly effective at extracting temporal redundancy from BTA addresses [18, 82].

4.2.4 Limitations

While the Tracking Module is targeted at the tracing of user-level code, extensions can be added for the purposes of detecting and tracing exception-related events. Exceptions such as system calls to an operating system can be traced by providing the decompressor with a copy of the system call handler and its associated program binaries. For unpredictable exceptions such as external interrupts, the Tracking Module can be made aware of processor
mode changes in order to output additional signals representing such events.

It is recognized that most software is written in high-level languages whose machine code is generated by compilers. The Tracking Module is catered to such scenarios, and made to follow a set of instruction tracking rules that are sufficient in maintaining data consistency between compressor and decompressor. While it is feasible to circumvent the tracking rules by executing hand-crafted assembly code, the inclusion of additional invalidation rules should be able to eliminate that possibility, albeit at the cost of increased hardware area.

4.3 Branch Counters

The problem of recording an execution trace can be reduced to that of recording the lead instructions of basic blocks in the execution flow, each of which is a BTA of the preceding
4.3 Branch Counters

branch instruction. While that provides for a reduction in trace data volume, further reductions can be achieved by differentiating between the types of branches that precede lead instructions. During compression, the proposed method inspects retired PC register values to differentiate between direct and indirect branches, as well as between conditional and unconditional branches. That process is depicted by the Partial Instruction Decode block in Figure 4.3. Each of the four branch types can be processed differently to reduce its footprint on the resulting execution trace, the details of which are discussed in the following section. Example instructions referenced in the remainder of this chapter are from the ARM ISA, whose format generally appears as: \texttt{inst<cond> dest,src1,src2}, where the \texttt{inst} instruction may have a condition \texttt{cond}, its input is from one or more \texttt{src} registers, and its result is written to register \texttt{dest}. The format \texttt{inst<cond> addr} is also used for instructions using a direct addressing mode, such as branches to the address \texttt{addr}.

4.3.1 Direct Unconditional Branches

The BTA of a direct unconditional branch, such as \texttt{b 0xa5d0}, is determined at compile-time and is encoded in the program binary with the instruction. Since the BTA of such a branch can be determined by inspecting the software-under-debug binary at decompression time, the BTA need not be recorded.

4.3.2 Direct Conditional Branches

Unlike direct unconditional branches, the results of conditional branches are not known at compile-time. In the case of \texttt{beq 0xa238}, the branch occurs only if the zero-bit is set in the status register. The actual outcome (\textit{taken} or \textit{not taken}) of the branch then needs to be known by the decompressor to reproduce the correct execution flow. While it would suffice to output the result of each conditional branch, the proposed method makes use of BP hardware. The BP acts to predict the outcome of all conditional branches, and is generally accurate due to the loop-heavy nature of software execution. The decompressor implements a software version of the BP that precisely mimics the hardware version, generating the exact same predictions when given the same inputs. The conditional branch outcomes that are correctly predicted by the BP need not be recorded, as they can be reproduced independently during decompression. When an incorrect prediction is made, the software decompressor is notified of where and when it occurred, and modifies its prediction table to
match the execution reality. It suffices to record only the locations of incorrect predictions in order to synchronize the compressor and decompressor. The proposed method counts the number of branches encountered by the BP between incorrect predictions by incrementing \( bCnt \), as seen in Table 4.2. An incorrect prediction will result in the compressor outputting \( bCnt \) before resetting the count to zero, as shown in Table 4.1.

### 4.3.3 Indirect Unconditional Branches

When branches are used with indirect addressing modes, such as `mov pc, r5`, the BTA cannot be determined from the software-under-debug binary, since it sits in a register or memory location. It would seem that the resulting BTA needs to be recorded in order to be preserved for the decompressor. Fortunately, a cache known as a BTB can associate the instruction address of an indirect branch with its resulting BTA. That allows a prediction to be made based upon the BTA of prior branches made from the same instruction address. Just as BP predictions achieve high accuracy because of the execution of software loops, BTB predictions of the BTA are accurate for the same reason. Similar to the way a decompressor mimics the BP, it can also shadow the operation of a BTB in software, allowing it to record only the locations where BTA mispredictions are made.

The proposed method counts the number of correct BTB predictions that have been encountered by incrementing \( iCnt \), as shown in Table 4.2. When an incorrect BTB prediction is made, Table 4.1 shows that \( iCnt \) is output and reset to zero. The described method improves upon other methods [85] by not incrementing \( bCnt \) when indirect unconditional branches are encountered. Since the BP has no bearing on those branches, the proposed method results in smaller \( bCnt \) values for a given execution trace, potentially resulting in smaller encodings on output.

### 4.3.4 Indirect Conditional Branches

When indirect conditional branches are encountered, such as `moveq pc, r5`, the output depends upon the predictions of both the BP and BTB. As above, a correct BP prediction will result in the incrementing of \( bCnt \), but a correct BTB prediction will only increment \( iCnt \) if the branch is determined to have been taken, as shown in Table 4.2. While \( iCnt \) is incremented by Uzelac et al. [85] upon encountering every indirect instruction, that is redundant because the position of instructions that are not taken can already be determined
4.4 Link-Register and Stack Tracking

Table 4.1: BP and BTB event handling

<table>
<thead>
<tr>
<th>Module</th>
<th>Event</th>
<th>Output</th>
<th>Counter Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>Correct Prediction</td>
<td>(None)</td>
<td>(See Table 4.2)</td>
</tr>
<tr>
<td></td>
<td>Misprediction</td>
<td>bCnt</td>
<td>Reset bCnt</td>
</tr>
<tr>
<td>BTB</td>
<td>Hit</td>
<td>(None)</td>
<td>(See Table 4.2)</td>
</tr>
<tr>
<td></td>
<td>Miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Invalidation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† if BTA is inferred by the Link-Register Tracking or Stack Tracking of Section 4.4.

Table 4.2: BP prediction correct or BTB hit

<table>
<thead>
<tr>
<th>Branch Type</th>
<th>Conditional–Taken</th>
<th>Conditional–Not Taken</th>
<th>Unconditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Increment bCnt</td>
<td>Increment bCnt</td>
<td>(None) ‡</td>
</tr>
<tr>
<td>Indirect</td>
<td>Increment bCnt &amp; iCnt</td>
<td>Increment bCnt</td>
<td>Increment iCnt</td>
</tr>
</tbody>
</table>

‡ BTA extracted from program binary.

by the decompressor through bCnt. The proposed improvement can achieve smaller iCnt values for a given execution trace, translating into potentially smaller encodings on output.

4.4 Link-Register and Stack Tracking

When an indirect branch is encountered in the execution flow, a portion of its instruction address is used to index the BTB. Depending upon the associativity of the BTB, one or more tags are stored at each BTB index, each with its respective BTA entry. A matching tag whose entry correctly predicts the BTA is deemed a “hit”, and results in the incrementing of counters as dictated by Table 4.2. A matching tag that incorrectly predicts the BTA will generate an “invalidation”, while an unmatched tag generates a “miss”. The decompressor determines the location of BTB misses by examining its emulated BTB, while invalidation events are pointed to by the iCnt signal. However, locating miss and invalidation events is
not sufficient to synchronize the decompressor with the execution flow, as the actual BTA must also be communicated. Without a way to mitigate the transmission of the BTA each time a miss or invalidation occurs, the BTA signal would constitute a disproportionately large portion of the execution trace.

In this section two schemes are introduced which add a layer of BTA prediction when a BTB miss or invalidation occurs. The proposed scheme exploits the fact that the BTA can often be predicted by the decompressor through other means. The first scheme involves tracking return addresses that have been placed in the Link Register (LR) by linked branches, while the second involves tracking branch return addresses that have been pushed to the stack. In tracking the movement of return addresses between those structures, the compressor is made aware of when a BTA need not be explicitly communicated. Neither scheme involves recreating structures on-chip, but instead provides on-chip tracking of whether or not they are populated with valid return addresses. Both schemes can be seen as part of the AATC Tracking Module in Figure 4.3.

4.4.1 Link-Register Tracking

Linked branching is a performance-enhancing feature of many ISAs, including ARM and MIPS. A linked branch such as `bl 0x2e30` can be used to atomically branch to a subroutine while storing the branch return address into a predetermined link register. When the subroutine is complete, a `mov pc,lr` instruction can be issued to resume execution at the address following the subroutine call. This means that a branch-and-return can be completed in two instructions, instead of the three instructions required without the use of linked branches.

Without Link-Register Tracking, the tracing of a `mov pc,lr` instruction would involve a BTB lookup, and the output of the BTA if a `miss` or `invalidation` resulted. To prevent BTA output in such cases, the proposed method is made aware of linked branches. Reads and writes to the LR register are tracked by partially decoding executed instructions in the Tracking Module. After a linked branch, a “valid” bit in the Link-Register Tracking block is set. That allows the proposed scheme to track whether there is a viable address for a decompressor to restore, without the need for on-chip storage of the BTA itself. In the typical case where a BTA has been stored in the LR register, and then restored back to the PC, the compressor can avoid outputting the BTA. The decompressor need only
4.4 Link-Register and Stack Tracking

Table 4.3: Link-Register Tracking Block: Instruction Tracking Rules for the ARM ISA

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moving PC to LR</td>
<td><code>mov lr, pc</code></td>
<td>unconditional branch</td>
</tr>
<tr>
<td></td>
<td><code>bl &lt;bta&gt;</code></td>
<td>direct conditional branch</td>
</tr>
<tr>
<td></td>
<td><code>bl&lt;cond&gt; &lt;bta&gt;</code></td>
<td>direct conditional branch (if taken)</td>
</tr>
<tr>
<td>Moving LR to PC</td>
<td><code>mov pc, lr</code></td>
<td>unconditional branch</td>
</tr>
<tr>
<td>LR Invalidation</td>
<td><code>mov&lt;cond&gt; lr, pc</code></td>
<td>conditional branch: cannot determine outcome during decompression offline</td>
</tr>
<tr>
<td></td>
<td><code>inst lr, !pc</code></td>
<td>any instruction where LR is destination register and source is not PC register: cannot determine LR contents during decompression offline</td>
</tr>
</tbody>
</table>

inspect the software-under-debug binary as software execution progresses, and populate a software-emulated LR register whenever a linked branch is executed. When a branch return instruction such as `mov pc,lr` is encountered, the BTA can then be restored from the software-emulated LR register. The scheme is then desirably asymmetrical, as it requires little hardware area to track the validity of the LR register, while instead requiring a marginal amount of offline effort to emulate the register in order to decompress the trace.

To ensure that the LR register contents do not change between a linked branch and the restoration of the BTA, any instructions that independently write to the LR register will cause an invalidation of the Link-Register Tracking block’s “valid” bit. By setting the same invalidation rules in both compressor and decompressor, synchronization between the two can be maintained. For the implementation in this chapter, the Link-Register Tracking block tracks the instructions and arguments seen in Table 4.3 for the ARM ISA. The tracking of that set is found to be sufficient to cover all linked-branch scenarios encountered in the user-level executable code of the MiBench benchmark suite used in Section 4.6.

The impact of Link-Register Tracking on the volume of data generated during Source Modeling is evaluated in Figure 4.4. Its use results in the generation of 33.4% fewer BTA symbols when applied to an unweighted execution trace of the MiBench suite, one in which individual benchmarks are given equal weight.
88 Architecture-Aware Real-Time Compression of ARM Execution Traces

Fig. 4.4: Branch target address volume reduction when applying Link-Register Tracking and Stack Tracking to an unweighted execution trace of the MiBench benchmark suite.

Table 4.4: Stack Tracking Block: Instruction Tracking Rules for the ARM ISA

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack <em>push</em></td>
<td>push &lt;pc or lr&gt;</td>
<td>single return address pushed to stack</td>
</tr>
<tr>
<td></td>
<td>push {..., lr, pc}</td>
<td>double return addresses pushed to stack</td>
</tr>
<tr>
<td>Stack <em>pop</em></td>
<td>pop &lt;pc or lr&gt;</td>
<td>single return address popped from stack</td>
</tr>
<tr>
<td></td>
<td>pop {..., &lt;pc or lr&gt;}</td>
<td>single return address popped from stack as part of multi-register pop</td>
</tr>
<tr>
<td>LR Invalidation</td>
<td>pop lr</td>
<td>when tracked <em>stack frame depth</em> is zero</td>
</tr>
<tr>
<td></td>
<td>pop {..., lr}</td>
<td>when tracked <em>stack frame depth</em> is zero</td>
</tr>
<tr>
<td>Stack Invalidation</td>
<td>inst sp</td>
<td>any instruction modifies stack pointer</td>
</tr>
</tbody>
</table>

4.4.2 Tracking Return Address Movement to/from the Stack

Link-Register Tracking is useful for preventing BTA output when a single linked branch returns to its calling function. However, most software contains nested branches that run several levels deep. Relying on Link-Register Tracking alone in such cases would be impractical, as every successive linked branch would overwrite the LR register, causing the invalidation of the Link-Register Tracking block’s “valid” bit. In that case, only the BTA output of the last branch in a hierarchy would be prevented.

To address that issue, the Stack Tracking block takes advantage of the way register data is preserved by a compiler when a nested branch is called. Before the execution of a nested
branch, modified registers (including the PC and LR registers) are typically moved to the stack, and are restored after the branch returns. At any given time, the stack can contain many such stack frames, including return addresses several levels deep.

By tracking the movement of only PC and LR registers to/from the stack in the Stack Tracking block, the Link-Register Tracking method is also extended to nested branches. Partially decoding push and pop instructions reveals whether those instructions’ arguments involve the PC and/or LR registers. If those registers are written to the stack, an on-chip Stack Frame Depth (SFD) counter is incremented, while the reverse transfer results in it being decremented. When BTAs are moved back from the stack to the PC or LR registers, BTA output is prevented if the SFD is greater than zero. By following the software-under-debug binary and maintaining a partial stack of only return addresses, the decompressor can read the omitted BTAs from its own emulated stack, which contains only return addresses.

Like Link-Register Tracking, this technique is desirably asymmetrical in that the stack emulation occurs during decompression in software, while the on-chip hardware need only maintain a running SFD count. Since the proposed method is unconcerned with transfers of other registers to/from the stack, the SFD is only related to the number of nested branches executed. The proposed method has several advantages to the Return Address Stack (RAS) used by some other schemes, which requires that return addresses be stored and compared in hardware. That can be both costly for a large SFD, and unnecessary due to the way a compiler moves return addresses between a program counter, link-register, and stack. Instead, the proposed method recognizes that the hardware need only track the existence of valid stack frames, enabling large SFD depths without the accompanying increase in hardware area.

In MiBench code compiled for the ARM ISA, the return address of a singular branch was generally stored in the LR register. Subsequent branches could result in either the LR, PC, or both registers being transferred to the stack simultaneously. The presented implementation satisfies all such instances by tracking the instructions and arguments seen in Table 4.4. As with Link-Register Tracking, invalidation rules serve to maintain data consistency. The SFD counter is reset to zero if any instruction is found to have changed the stack pointer, which implies that the previous stack has been abandoned either temporarily or permanently. In the case when the LR is overwritten by a value from the stack, and its new contents cannot be determined because the SFD is zero, an LR invalidation signal is propagated to the Link-Register Tracking block.
The efficacy of using Stack Tracking combined with Link-Register Tracking can be seen in Figure 4.4. When both techniques are applied to an unweighted execution trace of the MiBench suite, 52.0% fewer BTA symbols are generated during Source Modeling.

### 4.4.3 Example: Link-Register Tracking and Stack Tracking

To demonstrate the role of both the Link-Register Tracking and Stack Tracking techniques in preventing unnecessary BTA output, an instruction execution example is presented in Table 4.5. It lists the PC addresses that together constitute an execution trace, their corresponding assembly instructions, and the data movement that results from their execution. After the execution of each instruction, the action taken by the compressor’s Tracking Module is listed, and its effect on the LR “valid” bit and SFD counter are shown. A column reflecting whether the execution of the instruction results in a BTA output is also included. The final two columns represent the contents of a decompressor-emulated LR register and stack. The example starts by assuming that both the LR “valid” bit and SFD are initially zero, and that all indirect branches result in a BTB miss. A description of actions taken by the compressor and decompressor when encountering relevant instructions can be seen below:

**Instruction 2**: The first linked branch `bl r3` atomically moves its return address `0xe158` into the LR while also overwriting the PC with the value in `r3`. Since the value of `r3` cannot be determined by inspecting the software-under-debug binary, the decompressor must be provided the BTA explicitly by the compressor. As such, a BTA output of `0xb4e4` is issued once the compressor has been passed the address of the next instruction. The compressor’s LR Tracking block identifies the instruction as a linked branch, and updates its LR “valid” bit to `yes`. During decompression, the linked branch is similarly detected, causing the emulated LR to be populated with the return address `0xe158`.

**Instruction 3**: The instruction `push lr` copies the contents of the LR onto the Top of the Stack (TOS). The compressor’s Stack Tracking block identifies the action, and increments its SFD to “1”. Upon processing the same instruction, the decompressor would instead copy the contents of its emulated LR to the top of its emulated stack.

**Instruction 4**: A `mov lr,r2` instruction is identified by both compressor and decompressor as modifying the LR with the contents of an unknown register, triggering an LR invalidation event by setting the LR “valid” bit to zero.
### Table 4.5: Link-Register Tracking and Stack Tracking Example

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
<th>Data</th>
<th>Compressor</th>
<th>Decompressor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Action</td>
<td>LR Valid</td>
</tr>
<tr>
<td>1</td>
<td>e150</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>e154</td>
<td>bl r3</td>
<td>lr ⇔ pc+4;</td>
<td>[LR valid]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pc ⇔ r3;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b4e4</td>
<td>push lr</td>
<td>tos ⇔ lr;</td>
<td>[Inc. SFD]</td>
</tr>
<tr>
<td>4</td>
<td>b4e8</td>
<td>mov lr, r2</td>
<td>lr ⇔ r2;</td>
<td>[Inv. LR]</td>
</tr>
<tr>
<td>5</td>
<td>b4ec</td>
<td>pop pc</td>
<td>pc ⇔ tos;</td>
<td>[Dec. SFD]</td>
</tr>
<tr>
<td>6</td>
<td>e158</td>
<td>add</td>
<td>–</td>
<td>[LR valid]</td>
</tr>
<tr>
<td>7</td>
<td>e15c</td>
<td>bl r5</td>
<td>lr ⇔ pc+4;</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pc ⇔ r5;</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f010</td>
<td>xor</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>f014</td>
<td>push pc</td>
<td>tos ⇔ pc+4;</td>
<td>[Inc. SFD]</td>
</tr>
<tr>
<td>10</td>
<td>f018</td>
<td>mov sp, r6</td>
<td>sp ⇔ r6;</td>
<td>[Inv. SFD]</td>
</tr>
<tr>
<td>11</td>
<td>f01c</td>
<td>mov pc, lr</td>
<td>pc ⇔ lr;</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>e160</td>
<td>sub</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

† BTA inferred by emulated stack, * BTA inferred by emulated LR.

**Instruction 5**: The TOS is moved to the PC through a `pop pc` instruction. As this is an indirect branch, the new BTA `0xe158` must be communicated to the decompressor. Since the SFD is still “1”, the compressor avoids a BTA output knowing the decompressor can find the new BTA in its emulated stack. Once that occurs, both compressor and decompressor decrement SFD back to zero.

**Instruction 7**: When the compressor encounters the linked branch `bl r5`, it recognizes that the decompressor would populate its emulated LR with the return address `0xe160` upon encountering that same instruction. Accordingly, it sets its LR valid bit to `yes`. Once the new BTA of `0xf010` is established during execution of the next instruction, the compressor generates a BTA output of that address.

**Instruction 9**: The `push pc` instruction causes the compressor to set its SFD to “1”, and the decompressor to place the return address `0xf018` into its emulated stack.
**Instruction 10**: The instruction `mov sp,r6` modifies the stack pointer, causing a stack invalidation event for both compressor and decompressor. The compressor’s SFD is again zeroed, while the decompressor’s emulated stack is emptied.

**Instruction 11**: The final instruction `mov pc,lr` is a branch return that copies the contents of the LR to the PC. Since the compressor’s LR valid bit is still set to `yes`, the decompressor can find the BTA in its emulated LR. As such, the compressor does not output the new BTA.

The preceding example is not meant to be exhaustive, but rather it demonstrates how BTA output is prevented through the use of Link-Register Tracking and Stack Tracking. That feature is central to the performance of the AATC method introduced in this chapter. Although nested branches are not shown, they are an extension of the tracking, copying, and invalidation of return addresses that are demonstrated in tandem on both the compressor and decompressor.

### 4.5 Encoding Modules

In the AATC Tracking Module, the volume of execution trace data is reduced by representing it as a combination of `BTA`, `bCnt`, and `iCnt` signals. While that reduction can be significant, the resulting binary-encoded signals still contain residual redundancy that can be exploited to attain further compression. To achieve precisely that, two variants of an
AATC Encoding Module are presented in Figures 4.5 and 4.6, respectively. Each variant is tailored to a different tradeoff in terms of performance vs. hardware area footprint.

The fAATC Encoding Module processes the three signals produced by the AATC Tracking Module, encoding \( bCnt \) and \( iCnt \) signals with Fibonacci encoders, and the differential BTA signal with a VarLen encoder. As Figure 4.7 reveals, \( bCnt \) and \( iCnt \) symbols exhibit spatial redundancy when tracing the execution of the MiBench benchmark suite. That is demonstrated by the symbol values falling within the lower end of their ranges the vast majority of the time. In a binary encoding, the predominantly small values would be padded with leading zeros. In general, BTA addresses are also known for containing significant spatial locality, which is a property that is readily exploited by ILP hardware. That property can be observed in Figure 4.8, which shows that a differential encoding of the BTA signal results in a symbol distribution of shorter codewords. In the case of all 3 signals, an asymmetric symbol distribution skewed toward smaller codewords is desirable. It suggests that an alternative encoding can be used to reduce the overall size of the execution trace. To achieve that effect, the fAATC Encoding Module applies encoding schemes to shrink the majority of small values at the expense of expanding seldom-encountered large values. By focusing on exploiting only the spatial redundancy of each signal, fAATC also offers a small hardware area footprint.

The xAATC Encoding Module builds upon fAATC by also exploiting the temporal
locality of the BTA signal. Figure 4.9 shows that the BTA symbol stream exhibits a relatively high autocorrelation, with the probability of BTA symbols being repeated over time being much higher than that of $bCnt$ and $iCnt$ symbols. The sinusoidal autocorrelation is thought to be caused by the loop-driven nature of software execution. The xAATC Encoding Module adds a dictionary-based MTF compression stage to exploit the high BTA autocorrelation seen at small lag values, which results in superior performance. Both the fAATC and xAATC encoders process input signals independently in order to retain separate output streams. That allows the encoding of each data type to be optimized, resulting in a performance advantage [75]. Outputs are presented as separate signals, such that they are ready to be output to separate chip pins. Designs with constrained pin budgets may opt to introduce additional hardware to multiplex output signals, with some likely bandwidth overhead. In the following subsections, the design and operation of the encoders’ components is described.

### 4.5.1 Fibonacci Encoder

To exploit the asymmetric symbol frequency of $bCnt$ and $iCnt$ signals seen in Figure 4.7 to achieve data compression, a variety of coding schemes are evaluated for the task. As with Figure 4.4, symbols are derived from an unweighted execution trace of the MiBench benchmark suite. Since the symbol distribution of both signals is sufficiently similar, their aggregate data is used to compare the performance of seven schemes in Figure 4.10. Six of the schemes are universal codes known as Fibonacci [89], VarLen [84], Elias-Delta [90],
Elias-Omega [90], Rice [91] (with divisor parameter of 16), and Exponential-Golomb [92]. Such codes process symbols to generate codewords of varying length, where the length of a codeword corresponds to an implied probability of occurrence of the symbol within a set of symbols. Frequently-occurring symbols are assigned codewords with the shortest bit-length, while seldomly-occurring symbols would be represented by the longest bit-length codewords. For universal codes, a stream of codewords can also be uniquely decoded into their original symbols.

Among the seven schemes compared in Figure 4.10, it can be seen that Fibonacci encoding achieves the best compression of aggregated \( bCnt \) and \( iCnt \) symbols. In that
The sequence of Fibonacci numbers $F_2$ to $F_{25}$, seen in (4.2), is found to be sufficient to represent the full range of $bCnt$ and $iCnt$ symbols. An encoded symbol is represented as an array of bits, with individual bits corresponding to the presence or absence of a Fibonacci number as part of the sum representing the symbol. Table 4.6 shows an example of the encoding of the symbol 130, in decimal notation, into a Fibonacci-encoded sequence of bits. Fibonacci codewords also have the property that no adjacent “1” bits will be present in the encoded bit sequence. That allows a single-bit “1” delimiter to be appended to the codeword to represent its end. Codewords can then be unambiguously differentiated from each other in a stream, allowing a decompressor to decode them back into their constituent
Fig. 4.10: Comparison of encoding schemes on the size of \( b\text{Cnt} \) and \( i\text{Cnt} \) symbols for an unweighted execution trace of the MiBench benchmark suite.

The presented hardware implementation of the Fibonacci encoder includes the array of the precomputed Fibonacci numbers shown in (4.2), where the relatively small array size lends itself to an area-efficient encoder implementation. In both Figures 4.5 and 4.6, \( b\text{Cnt} \) and \( i\text{Cnt} \) signals are denoted \( b\text{Cnt}_{\text{enc}} \) and \( i\text{Cnt}_{\text{enc}} \) after having been encoded, respec-


Table 4.6: Example of the Fibonacci encoding of decimal symbol 130

<table>
<thead>
<tr>
<th>Base</th>
<th>Symbol</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>$F_4$</th>
<th>$F_5$</th>
<th>$F_6$</th>
<th>$F_7$</th>
<th>$F_8$</th>
<th>$F_9$</th>
<th>Delimiter-Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>130</td>
<td>=</td>
<td>2</td>
<td>+</td>
<td>5</td>
<td>+</td>
<td>34</td>
<td>+</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>130 (Fibonacci)</td>
<td>=</td>
<td>{1 0 1 0 0 0 1 0 1 1}_binary</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It should be noted that an attempt to extend this encoding scheme to large symbols is likely not scalable, due to the larger hardware structures needed to store and process longer Fibonacci sequences. In the case of larger symbol distributions, an alternative encoding should be considered to achieve an acceptable result on the spectrum of tradeoffs between performance and hardware area.

4.5.2 Differential BTA

Both AATC Encoding Modules include VarLen Bijection Encoder blocks for the encoding of BTA addresses. In fAATC, the difference ($\Delta$) between the current BTA address and the previously-encoded BTA address (Old BTA) is calculated using an Exclusive-OR (XOR) operation. Figure 4.8 shows that successive BTA outputs are likely to exhibit spatial locality, such that a “differential” encoding of the BTA is more likely to contain leading zeros. Just as Fibonacci encoding of $bCnt$ and $iCnt$ symbols with leading zeros reduce their overall sizes, passing the differential BTA through a VarLen Bijection Encoder allows a similar size reduction to be achieved.

The xAATC block offers two ways of encoding the BTA signal. If a BTA is found in the MTF dictionary, it will be encoded as an index to that dictionary entry. In cases where the BTA is not in the dictionary, it is sent to the module’s fAATC constituent block and encoded with a VarLen Bijection Encoder. Since the encoded size an MTF index is likely to be smaller than a VarLen-encoded value, only “misses” of the MTF dictionary will result in a VarLen Bijection encoding of the differential BTA.

4.5.3 VarLen Bijection Encoder

Transforming BTA addresses through differential encoding can be useful for exploiting spatial locality and representing the resulting signal using fewer bits. Similar to the encoding of $bCnt$ and $iCnt$ symbols, the differential BTA signal can itself be encoded to achieve com-
For that purpose, Figure 4.11 compares the compression performance of three encoding schemes that potentially offer a small hardware area footprint. Since the differential BTA can also take on a negative value, two different methods of representing negative numbers are explored for each encoding scheme. The example of Table 4.7 shows the extension of negative numbers to an non-negative integer code. Codewords that ordinarily represent positive numbers can either be prefixed with a “negative bit”, or a bijection can be used to change the code mapping to alternate between positive and negative symbols. Both methods are compatible with small hardware area encoders, though their performance varies depending upon the encoded data distribution. A “negative bit” offers a fixed 1-bit overhead, so datasets with symbols concentrated in the low end of the range will incur a disproportionate size penalty. The same datasets encoded with bijection would have no such penalties, but codeword length will grow faster than negative-bit encoding for large symbols.

The VarLen encoding scheme [85] is included as part of both Encoding Modules due to its performance, and ability to generate codes in real-time with a relatively small hardware area footprint. Improvements over other implementations include the use of bijection map-
Table 4.7: Example of negative-bit and bijection representations of negative numbers

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Non-negative Integer Code</th>
<th>Negative-bit Representation</th>
<th>Bijection Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>codeword 3</td>
<td>0 + codeword 3</td>
<td>codeword 5</td>
</tr>
<tr>
<td>2</td>
<td>codeword 2</td>
<td>0 + codeword 2</td>
<td>codeword 3</td>
</tr>
<tr>
<td>1</td>
<td>codeword 1</td>
<td>0 + codeword 1</td>
<td>codeword 1</td>
</tr>
<tr>
<td>0</td>
<td>codeword 0</td>
<td>0 + codeword 0</td>
<td>codeword 0</td>
</tr>
<tr>
<td>-1</td>
<td></td>
<td>1 + codeword 1</td>
<td>codeword 2</td>
</tr>
<tr>
<td>-2</td>
<td></td>
<td>1 + codeword 2</td>
<td>codeword 4</td>
</tr>
<tr>
<td>-3</td>
<td></td>
<td>1 + codeword 3</td>
<td>codeword 6</td>
</tr>
</tbody>
</table>

ping to achieve greater compression ratios. The encoding scheme works by determining the smallest of a set of variable-length “bins” into which a binary-encoded value can be placed. If necessary, the value is padded with zeros to take up the entire bit-width of the bin, and a prefix representing the length of the bin is appended. The prefix consists of a sequence of ‘1’ bits in unary notation followed by a divisor bit ‘0’ between unary and binary segments of a codeword. The length of the prefix \((pLen)\) is related to the length of the binary portion of the codeword which follows it, and relationship between the two can be varied using a combination of parameters \(start\) and \(step\). Optimizing the parameters to the most frequently-encountered input values results in the minimum size of encoded data. Such an encoding scheme is uniquely decodable, meaning that codes in a stream can be deterministically distinguished from one another. That property ensures that encoded values can always be decoded back into their original form. In this case, a decoder would count the number of bits between the last processed codeword and the divisor bit of the current codeword to determine the length of the prefix, \(pLen\). The total length of a codeword \((L)\), including the binary portion, is given by \(L = start + (pLen - 1) \times step\).

The optimum \(start\) and \(step\) parameters are determined in Section 4.6.2 by static profiling across the benchmark suite. Although an adaptive scheme could have been implemented, it would have been at the expense of increased complexity and hardware area. After VarLen Bijection encoding, the differential BTA signal is shown in Figures 4.5 and 4.6 as \(BTA_{enc}\).
4.6 Experimental Results

4.5.4 Move-to-Front Transform

Previous work [18, 82] has shown that applying an MTF transform to execution traces is effective in exploiting the temporal locality of an execution trace. Though Figure 4.9 confirms that the BTA signal can benefit from such a transform, the implementation of an on-chip MTF dictionary can quickly increase the encoder hardware area footprint. For that reason, an MTF transform is only included as part of the higher-performance xAATC Encoding Module. An MTF array of the 31 most-recently-encountered unique BTA addresses is found to strike a balance between hardware area footprint and compression performance. As shown in Figures 4.5 and 4.6, all BTA inputs are passed through the MTF transform, where they are compared to an array of addresses. If a match is found, a 5-bit $MTF_{idx}$ is output that represents the array index of the matching address, and the array element is moved to the top of the array. If there is no match, a reserved $MTF_{idx}$ value is output that represents an array miss. The array element is moved to the top of the array, discarding the least-recently-encountered element. In the case of a miss, the xAATC method encodes the differential BTA with the VarLen Bijection encoder contained within its constituent fAATC block.

To decode MTF indices offline, the decompressor must maintain an identical MTF array in software as it follows the execution flow using the software-under-debug binary. Just as the decompressor shadows the actions of the BP and BTB blocks, it must also shadow the actions of the MTF transform in order to decompress BTA addresses encoded by the xAATC Module.

4.6 Experimental Results

To evaluate the AATC execution trace compression method, four groups of experiments are presented in this section. Since compression performance is correlated with the prediction accuracy of the on-chip BP and BTB units, a comparison against other schemes relying on those units must also take into account their prediction accuracies. The first experiment determines the BP and BTB structures that result in comparable prediction rates to other methods while executing a uniform benchmark suite. That allows a fair performance comparison of compression methods, abstracted away from the prediction accuracy of ILP hardware.
The second experiment determines the VarLen Bijection Encoder parameters that result in the minimum encoded size of the BTA signal for a given execution trace, while the third experiment compares the compression performance of the proposed method against others in the literature. For the first three experiments, the aggregate execution traces of 15 applications of the MiBench benchmark suite [87] are used. The same suite is used by several other schemes [84, 18, 83, 85, 82, 93], allowing a direct comparison of compression performance.

Execution traces were extracted for the execution of a virtualized ARM9 processor using QEMU 0.14.1 [41]. The processor supports an ARM ISA, which includes the linked branches exploited by the proposed AATC method. The MiBench suite was compiled from source code using the GCC v4.3.2 [94] running under the Debian 5.0 flavour of the Linux OS. Individual benchmarks were executed under GDB [28] to isolate only the execution traces of the required user-level code. A GDB script was used to automate single-stepping through entire executions of benchmarks, and the contents of the PC were recorded into a log file after each instruction execution. The logs of captured execution traces provide inputs into functional models of a BP [95], BTB, and AATC Tracking and Encoding modules implemented in MATLAB r2010a. Compressed outputs are recorded to files, and verified to expand back to their original forms with a decompressor also implemented in MATLAB.

The final experiment evaluates the hardware area requirements of both AATC variants. A VHSIC hardware description language (VHDL) implementation of the Tracking Module and both variants of the Encoding Module were synthesized to RTL with Synopsys Design Compiler D-2010.03-SP4. Functional verification was performed in Mentor Graphics Modelsim 6.5c, where the output of the RTL model was matched to the output of the functional model of the compressor implemented in MATLAB.

4.6.1 BP and BTB Performance

Evaluating the performance of self-contained execution trace compression methods generally involves comparing their input and output data volumes. In methods where compression performance is directly correlated to the rate of BTB hits and correct BP predictions, a fair comparison must involve equalizing the performance of on-chip BP and BTB units under similar execution scenarios. The first experiment explores different structures for BP and BTB models while executing the MiBench benchmark suite. That allows BP and
### Table 4.8: Rates of Correct BP Predictions and BTB Hits

<table>
<thead>
<tr>
<th>MiBench Benchmark</th>
<th>Instructions (millions)</th>
<th>Branch Predictor</th>
<th>Branch Target Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm_c</td>
<td>825.8</td>
<td>0.999</td>
<td>0.999</td>
</tr>
<tr>
<td>bf_e</td>
<td>1131.0</td>
<td>0.984</td>
<td>0.984</td>
</tr>
<tr>
<td>jpeg</td>
<td>118.0</td>
<td>0.921</td>
<td>0.928</td>
</tr>
<tr>
<td>djpeg</td>
<td>29.8</td>
<td>0.950</td>
<td>0.954</td>
</tr>
<tr>
<td>fft</td>
<td>389.1</td>
<td>0.906</td>
<td>0.937</td>
</tr>
<tr>
<td>gsm_d</td>
<td>783.6</td>
<td>0.973</td>
<td>0.976</td>
</tr>
<tr>
<td>lame</td>
<td>1322.4</td>
<td>0.871</td>
<td>0.879</td>
</tr>
<tr>
<td>mad</td>
<td>308.7</td>
<td>0.908</td>
<td>0.926</td>
</tr>
<tr>
<td>rijndael_e</td>
<td>470.2</td>
<td>0.951</td>
<td>0.967</td>
</tr>
<tr>
<td>sha</td>
<td>124.8</td>
<td>0.951</td>
<td>0.956</td>
</tr>
<tr>
<td>stringsearch</td>
<td>3.9</td>
<td>0.916</td>
<td>0.931</td>
</tr>
<tr>
<td>tiff2bw</td>
<td>132.6</td>
<td>0.996</td>
<td>0.997</td>
</tr>
<tr>
<td>tiff2rgba</td>
<td>98.6</td>
<td>0.993</td>
<td>0.994</td>
</tr>
<tr>
<td>tiffdither</td>
<td>1161.7</td>
<td>0.909</td>
<td>0.918</td>
</tr>
<tr>
<td>tiffmedian</td>
<td>517.3</td>
<td>0.980</td>
<td>0.982</td>
</tr>
</tbody>
</table>

**Unweighted Average**

|                | 0.947 | 0.955 | 0.959 | 0.923 | 0.939 | 0.909 |

BTB configurations to be selected which achieve similar prediction rates to the work of Uzelac et al. [84, 85]. Once equalized, any difference in compression performance between methods can then be attributed to the compression method itself. The execution traces of 15 MiBench benchmarks were applied to the functional model of the AATC Tracking Module, and the rates of BTB hits and correct BP predictions recorded in Table 4.8.

For exploring BP prediction rates, a model of the perceptron BP originally proposed by Jimenez [95] was developed. That class of neural network-based predictors is known to outperform other types, and is has been shown to be capable of prediction rates similar to that of commercial BPs [96]. A 512x32 configuration of the BP model was chosen due to its similarity to the *gshare* predictor [97] employed by Uzelac et al. [84, 85]. Since the perceptron predictor generally outperforms *gshare*, various entry “weights” were explored...
to determine a variant that closely matches the prediction rates reported by Uzelac et al. [84, 85]. In Table 4.8, a version with 8-bit weights can be seen to predict within a relatively small 1.2% and 0.4% margin of the rates achieved by bTMBP [85] and B4 [84], respectfully. The small differences allow for a reasonable comparison between those methods and the work presented in this chapter.

From the modeling of various BTB configurations, a 64-entry, 2-way (64x2) BTB yielded results most similar to bTMBP [85] and B4 [84], to within an average of 1.4% and 3.0% of those schemes, respectively. The 64x2 BTB is consequently used as a basis for performing other experiments, and to ultimately compare compression performance with the other methods in Table 4.10.

### 4.6.2 VarLen Bijection Encoding Parameters

The VarLen Bijection Encoding scheme used to produce the $BTA_{enc}$ signal can be optimized using the parameters $start$ and $step$ to attain a minimal encoded size. Using the same MiBench execution traces from the previous experiment, this experiment statically determines the optimal parameters for each encoded signal within the ranges $start = [1, 14]$ and $step = [1, 6]$. The parameters resulting in the smallest encoded data size were found to be $(start, step) = \{12, 4\}$, and are consistent with the optimum values found in another work [85] for some variants of their encoding scheme. Those same parameters were used for obtaining the encoded size of differential BTA symbols in Figure 4.11 for both representations of negative numbers.

### 4.6.3 Compression Performance

In this experiment, the execution trace compression performance of fAATC and xAATC is compared with other methods from the literature. The BP and BTB models discussed in Section 4.6.1 are used to drive the AATC Tracking Module, ensuring a fair basis for comparing results against other schemes. The optimum VarLen parameters found in Section 4.6.2 are also applied to both variants of the AATC Encoding Module to minimize the size of the BTA outputs.

The result of compressing the execution traces of the MiBench benchmark suite can be seen in Tables 4.9 and 4.10 for both AATC variants. Two groups of compression algorithms are offered for comparison: three software-based general-purpose techniques in
Table 4.9: Software-based Execution-Trace Compression Scheme Performance Comparison with AATC (bits/instruction)

<table>
<thead>
<tr>
<th>MiBench Benchmark</th>
<th>gzip [74]</th>
<th>bzip2 [73]</th>
<th>lzma2 [72]</th>
<th>fAATC</th>
<th>xAATC</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm_c</td>
<td>0.0425</td>
<td>0.0061</td>
<td>0.0015</td>
<td>0.0004</td>
<td>0.0004</td>
</tr>
<tr>
<td>bf_e</td>
<td>0.7663</td>
<td>0.0998</td>
<td>0.0050</td>
<td>0.0063</td>
<td>0.0056</td>
</tr>
<tr>
<td>cjpeg</td>
<td>0.2767</td>
<td>0.0931</td>
<td>0.0612</td>
<td>0.0249</td>
<td>0.0244</td>
</tr>
<tr>
<td>djpeg</td>
<td>0.5108</td>
<td>0.1406</td>
<td>0.0262</td>
<td>0.0128</td>
<td>0.0121</td>
</tr>
<tr>
<td>fft</td>
<td>0.3796</td>
<td>0.2253</td>
<td>0.0888</td>
<td>0.0660</td>
<td>0.0511</td>
</tr>
<tr>
<td>gsm_d</td>
<td>1.1594</td>
<td>0.5283</td>
<td>0.0125</td>
<td>0.0052</td>
<td>0.0046</td>
</tr>
<tr>
<td>lame</td>
<td>0.8210</td>
<td>0.4429</td>
<td>0.0259</td>
<td>0.0161</td>
<td>0.0153</td>
</tr>
<tr>
<td>mad</td>
<td>0.7721</td>
<td>0.4370</td>
<td>0.0694</td>
<td>0.0236</td>
<td>0.0226</td>
</tr>
<tr>
<td>rijndael_e</td>
<td>0.2257</td>
<td>0.2885</td>
<td>0.0055</td>
<td>0.0081</td>
<td>0.0081</td>
</tr>
<tr>
<td>sha</td>
<td>0.1629</td>
<td>0.0728</td>
<td>0.0049</td>
<td>0.0046</td>
<td>0.0046</td>
</tr>
<tr>
<td>stringsearch</td>
<td>0.4003</td>
<td>0.1692</td>
<td>0.0778</td>
<td>0.1058</td>
<td>0.0825</td>
</tr>
<tr>
<td>tiff2bw</td>
<td>0.4569</td>
<td>0.0664</td>
<td>0.0067</td>
<td>0.0021</td>
<td>0.0016</td>
</tr>
<tr>
<td>tiff2rgba</td>
<td>0.5781</td>
<td>0.0627</td>
<td>0.0076</td>
<td>0.0055</td>
<td>0.0029</td>
</tr>
<tr>
<td>tiffdither</td>
<td>0.4003</td>
<td>0.1840</td>
<td>0.1229</td>
<td>0.0257</td>
<td>0.0252</td>
</tr>
<tr>
<td>tiffmedian</td>
<td>0.2801</td>
<td>0.0565</td>
<td>0.0276</td>
<td>0.0052</td>
<td>0.0047</td>
</tr>
<tr>
<td>Unweighted Average</td>
<td>0.4822</td>
<td>0.1915</td>
<td>0.0362</td>
<td>0.0208</td>
<td>0.0177</td>
</tr>
</tbody>
</table>

Table 4.9, and six real-time hardware-based techniques in Table 4.10. The general-purpose techniques presented use vast amounts of memory and execution time to compress traces in software, and are thus not directly comparable to hardware-based techniques. General-purpose compression results are nevertheless given, as the algorithms gzip [74] and bzip2 [73] are ubiquitous in general computing, while lzma2 [72] often offers superior performance. A fairer comparison of the proposed method is given in Table 4.10 against six real-time hardware-based technique: SCI-Ntup [93], eSDC-LSP [83], eDMTF [82], v0 [18], bTMBP [85], and B4 [84]. For both groups of compression algorithms, overall compression performance is given as the unweighted average performance of benchmarks comprising the MiBench suite.

While an attempt was made to collect traces from the superset of MiBench applications used by comparative schemes, issues were encountered in compiling and/or executing 2 of
Table 4.10: Hardware-based Execution-Trace Compression Scheme Performance Comparison with AATC (bits/instruction)

| MiBench Benchmark | SCI-Ntup [93] | eSDC-LSP [83] | eDMTF [82] | v0 [18] | bTMBP [85] | B4 xAATC adpcm c 0.0202 0.001 0.001 N/A 0.0013 0.0003 0.0004 0.0004 | BF e 0.3250 0.345 0.284 0.222 0.0093 0.0093 0.0063 0.0056 | Cjpeg 0.2169 0.088 0.091 0.1116 0.0420 0.0382 0.0249 0.0244 | Djpeg 0.1697 0.053 0.052 0.0714 0.0205 0.0199 0.0128 0.0121 | FFT N/A 0.542 0.201 0.1871 0.0909 0.0711 0.0660 0.0511 | GSM d 0.1833 0.051 0.040 N/A 0.0129 0.0122 0.0052 0.0046 | LAME 0.2024 0.090 0.113 N/A 0.0288 0.0267 0.0161 0.0153 | Mad 0.1799 0.116 0.147 N/A 0.0331 0.0273 0.0236 0.0226 | Rijndael e 0.0704 0.183 0.096 N/A 0.0158 0.0111 0.0081 0.0081 | SHA 0.0727 0.074 0.049 0.0444 0.0218 0.0204 0.0046 0.0046 | Stringsearch 1.3212 0.412 0.387 0.2680 0.1644 0.1644 0.1058 0.0825 | Tiff2bw 0.1361 0.030 0.011 0.0067 0.0065 0.0042 0.0021 0.0016 | Tiff2rgba 0.0786 0.012 0.006 0.0042 0.0075 0.0056 0.0055 0.0029 | Tiffdither 0.4887 0.158 0.166 N/A 0.0618 0.0572 0.0257 0.0252 | Tiffmedian 0.0772 0.027 0.012 0.0137 0.0070 0.0067 0.0052 0.0047 |
| Unweighted Average | 0.2530 | 0.1455 | 0.1104 | 0.1033 | 0.0349 | 0.0316 | 0.0208 | 0.0177 |

17 programs. Compression results are therefore given for the remaining 15 applications. The number of instructions comprising each trace is given in Table 4.8, which generally correspond to those used by other methods. The use of different ISAs and compilers between methods makes it difficult to achieve an exact match.

It can be seen that both of AATC variants achieve greater average performance than the best general-purpose techniques, even with the additional constraints that the proposed real-time algorithm must operate under. A performance comparison of hardware-based techniques also reveals that both variants of the AATC method offer better performance than any of the other schemes, for all but a single application. Compared to B4, which is the highest-performing method in the literature, fAATC and xAATC achieve an average
4.6 Experimental Results

Table 4.11: Comparison of Logic Utilization and Average Compression Performance

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Performance (instructions/bit)</th>
<th>Logic Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>xAATC</td>
<td>56.50</td>
<td>24133</td>
</tr>
<tr>
<td>fAATC</td>
<td>48.08</td>
<td>9854</td>
</tr>
<tr>
<td>B4 [84]</td>
<td>31.65</td>
<td>6382</td>
</tr>
<tr>
<td>M4 [84]</td>
<td>28.82</td>
<td>4846</td>
</tr>
<tr>
<td>bTMBP [85]</td>
<td>28.65</td>
<td>5200</td>
</tr>
<tr>
<td>LZ-based [79]</td>
<td>14.20</td>
<td>51678</td>
</tr>
<tr>
<td>eDMTF [82]</td>
<td>9.06</td>
<td>25000</td>
</tr>
<tr>
<td>ARM ETM v1 [98]</td>
<td>1</td>
<td>7000</td>
</tr>
</tbody>
</table>

51.9% and 78.5% better performance, respectively. Performance gains can be attributed to a number of factors, including the novel use of Link-Register and Stack Tracking, improvements to the counting of $bCnt$ and $iCnt$ events, and the systematic application of encoding schemes to extract residual redundancies.

4.6.4 Hardware Area

Hardware area footprint is one of the design constraints of a real-time execution trace compression scheme. The AATC method employs several tactics which avoid performing on-chip data manipulation in order to reduce hardware area footprint, and the following experiment quantifies that effort in terms of hardware logic gate utilization. An RTL model of both fAATC and xAATC schemes was synthesized using Synopsys Design Compiler with the lsi_10k library, and their respective gate counts added to those of BP and BTB structures. The logic utilization of known BP and BTB units [84] are incorporated, ones that have been established to perform near-equivalently to the simulation models presented in Section 4.6.1. Overall logic utilization for both AATC variants can be seen in Table 4.11 alongside several other methods. The inverse of the average compression performance metric used in Table 4.10 is also presented for each scheme, with higher values denoting better compression performance.

The fAATC method offers a significant performance improvement over other recent work, with only a modest increase in hardware area. However, the additional performance
gains offered by the xAATC method require approximately 2.5 times the logic resources of fAATC. The difference in resources between the two variants can be attributed to the size and complexity of the MTF stage in the xAATC design.

Even though a single design has been presented for each AATC encoding module, the design space presents additional opportunities for decreasing hardware area, or increasing design robustness. The presented design implements dual sequential Fibonacci encoders and a sequential VarLen encoder in both AATC variants, as well as a sequential MTF transform in xAATC. That allows simultaneous data inputs to be processed, each requiring multiple clock cycles to complete. Some designs may opt to buffer encoder inputs to handle worst-case execution scenarios, or share a single encoder between signals to reduce hardware area footprint. Those approaches would require making an estimate of the frequency of inputs, and setting safety margins on the buffer size to avoid buffer overflows.

4.7 Chapter Summary

There is a growing consensus that real-time trace generation and post-hoc trace analysis are the answer to future software debugging needs. However, the difficulty of moving large volumes of data off-chip can drastically increase debugging time and cost. This chapter presents a method to perform real-time on-chip compression of execution traces, one of the most useful types of traces in software debugging.

This chapter shows that an awareness of the linked branches that are typical of many ISAs can be exploited for trace compression purposes. In recognizing the methods by which compilers generate code, the proposed compression scheme also exploits patterns in the movement of return-addresses between link-register, stack, and program counter. Through the on-chip tracking of a small set of counters and invalidation rules, the combined techniques allow the traced volume of branch target addresses to be reduced by 52%, while algorithmic improvements also reduce the volumes of two other data types. The addition of an encoding stage systemically exploits the residual spatial and temporal redundancies that are shown to remain in the trace stream, resulting in significant improvements in compression performance over comparable methods.

The two variants of the proposed Architecture-Aware Trace Compression (AATC) scheme satisfy different needs on the spectrum of tradeoffs between performance and hardware logic utilization. The fAATC variant encodes 48.1 instructions per bit using 9854
gates, while xAATC encodes 56.5 instructions per bit using 24133 gates. Compared to the highest-performing method in the literature, fAATC and xAATC achieve 51.9% and 78.5% better performance, respectively.
Chapter 5

Multi-Architectural Soft-Processor Execution Trace Compression

5.1 Introduction

Hardware-emulated development platforms play an important role in hardware-software co-design, allowing software development and debugging to proceed concurrently alongside hardware development and verification. They are also commonly used in rapid prototyping of designs, and can serve as platforms for small-scale deployments that would otherwise not meet the needed *economies of scale* if manufactured as physical SoCs.

Traditional debugging interfaces are typically built into hardware-emulated platforms, enabling both hardware verification and execution-control software debugging through a unified TAP interface. While such a dual-purpose interface is sufficient for detecting deterministic software faults, it is unsuitable for detecting increasingly common complexity-related and timing-related software faults. Since such bugs can take a long time to resolve [12], identifying them early in the co-design process can reduce overall development time and cost. That can be achieved by including support for hardware-instrumented real-time trace generation, which enables all types of software faults to be observed.

The clock speeds of soft-core processors are largely limited by the underlying FPGA device used to emulate them, resulting in a wide disparity between their clock speeds and those of equivalent *hard-core processor* designs implemented on physical platforms. Modern FPGA devices allow processor speeds on the order of hundreds of megahertz to be reached,
which can still result in both excessive trace data volume (as shown by Example 2.2), and
large trace port bandwidth requirements for streaming traces off-chip in real-time. Just
as on-chip trace compression can be beneficial in reducing both trace data volume and
bandwidth on physical platforms, it can be similarly valuable when applied to hardware-
emulated platforms. During hardware-software co-design, the speed at which traces can be
streamed off-chip has been shown to be a bottleneck in performing software debugging on
hardware-emulated platforms [69, 70]. On-chip trace compression can serve to reduce or
eliminate that bottleneck, allowing hardware emulation to potentially proceed at the limits
of the configured design.

Trace compression schemes suited for use on hardware-emulated platforms have dif-
ferent requirements than those catered to physical platforms. The unique characteristics
of hardware-emulated platforms must be considered, as must the scenario under which
software development occurs on such platforms. A trace compression scheme created for
implementation on a physical platform typically attempts to minimize its hardware re-
source usage. However, a scheme created for hardware-emulated platforms can potentially
harness any unused resources that will remain on the FPGA after configuring the hardware-
emulated design. The proportion of unused resources can differ depending upon the nature
of the hardware design, choice of FPGA device, and the efficiency of mapping the design
onto the device. With so many variables to consider, trace compression on hardware-
emulated platforms could be well-served by a scheme that offers multiple options on the
spectrum of tradeoffs between compression performance and hardware resource usage.

The hardware-software co-design scenario in which a hardware-emulated trace compres-
sion scheme is to be used should also be considered. The ability to reuse an IP block that
represents the compression scheme, without requiring considerable developer effort to make
hardware changes, can be considered important. The compression scheme should be flex-
ible, allowing direct reuse on a variety of hardware architectures and ISAs across a set of
projects. A developer should ideally also have the ability to cater the compression scheme,
with minimal effort, to satisfy a point on the spectrum between compression performance
and hardware resource usage.

This chapter presents a real-time execution trace compression scheme that enables soft-
ware debugging of complexity-related and timing-related faults on hardware-emulated plat-
forms. The proposed Multi-Architectural Trace Compression (MATC) scheme is suited to
compressing the execution traces of a soft-core processor that is configured on an FPGA
device. When considering the hardware-software co-design scenario in which such a scheme is likely to be used, the MATC method offers distinct advantages to other schemes. The scheme is compatible with execution traces of all fixed instruction-width ISAs, allowing the MATC IP block to be rapidly deployed to a co-design project without requiring ISA-specific customization. Five parameterizable variants of the MATC scheme are also presented to satisfy different points on the spectrum of tradeoffs between compression performance and FPGA resource usage. The variants are differentiated by their utilization of spare FPGA SRAM and Lookup Tables (LUTs), an increase of which is shown to correlate with improved compression performance. Variants are enabled by a high-level VHDL abstraction that allows spare FPGA resources to be rapidly claimed for trace compression purposes.

The resulting scheme aims to reduce software development time and cost by enabling complexity-related and timing-related software fault debugging to begin as early in the development process, and to be deployed quickly to hardware-emulated platforms. It allows compressed traces to be streamed off-chip in real-time, or at speeds limited only by the spare resources of the underlying FPGA device.

5.1.1 Contributions of this Chapter

This chapter proposes an execution trace compression scheme suited to hardware-emulated development platforms that makes several improvements over existing methods:

- Creating portable hardware suited to compressing execution traces of any fixed instruction-width ISA.

- Leveraging the unused hardware resources remaining on an FPGA after its configuration into a hardware-emulated development platform for trace compression purposes.

- Demonstrating self-contained trace decompression software that can be applied to execution traces of any fixed instruction-width ISA, without access to software-under-debug binaries.

- Parameterizable exploitation of FPGA LUTs and on-chip SRAM, allowing trace compression performance to be catered to available hardware resources.

- Showing that there is a directly proportional relationship between execution trace compression performance and Finite Context Method (FCM) prediction table size.
The contributions allow trace compression hardware and trace decompression software to be rapidly reused by various hardware-emulated architectures. That enables a software developer to employ instruction-level trace generation earlier in the hardware-software co-design process, and with less effort, than would otherwise be possible.

5.2 Multi-Architectural Trace Compression Overview

An overview of the proposed Multi-Architectural Trace Compression (MATC) scheme can be seen in Figure 5.1, where the PC register of a 32-bit fixed instruction-width ISA acts as the input to the first compression stage. The hardware-based scheme contains six data transforms which progressively remove redundancy from an execution trace as part of five pipeline stages. The figure shows that the first three pipeline stages apply a single transform each, after which the execution trace is represented as a combination of two separate streams: a prefix stream and a data stream. Transforms are then applied concurrently to each stream during the fourth pipeline stage, while the fifth stage operates only on the data stream.

The hardware-based execution trace compression scheme meets several constraints that allow it to be used with hardware-emulated development platforms. The pipelined scheme is designed to work in real-time, at the clock speed of either the configured soft-processor, or of the MATC module itself. The scheme is well-suited to implementation on FPGA devices, using only a small amount of hardware resources to achieve good compression performance. Using a small amount of unused FPGA block ram to implement a compression dictionary ensures FPGA LUT resources are available to configure other logic.

The MATC scheme offers parameterization options which allow any of five variants of the scheme to be implemented. The variants are represented in Figure 5.1 by the \((s, m)\) ordered pair, where \(2^s\) is the size of the FCM address space, and \(m\) represents the number of elements in the MTF array. This chapter considers MATC variants \(v0\) to \(v4\), which correspond to \((s, m) = (16, 256), (16, 128), (14, 128), (14, 64),\) and \((12, 64)\), respectively. Allowing the FCM transform address space to be adjusted enables the size of its hash table to be catered to the amount of unused FPGA block ram. Similarly allowing the MTF array size to be adjusted impacts the overall FPGA resource utilization of the MATC scheme. Since operations on the MTF array also represent the critical path in the MATC scheme, variants with differing array sizes also attain different maximum clock frequencies, which...
allows the MATC scheme to be further catered to the hardware-emulated platform.

5.2.1 Usage Scenario

The usage of the presented MATC scheme within a hardware-emulated development platform can be seen in Figure 5.2. In such a scenario, both a soft-core processor and MATC module are configured to the same FPGA device, where both harness embedded block ram found on the FPGA to implement their own memories. Similar to the AATC scheme usage scenario described in Section 4.2.1, a developer working on a host workstation initiates a trace experiment by issuing commands to a software driver. The driver controls a combined debugging and tracing interface that operates within the speed limitations of
hardware-emulated devices, and includes a trace port. Issued commands include *trigger* information, such as conditions for starting and ending the trace experiment, as well as a command to start remote execution of the program on the *target* soft-processor. A stream
of retired instructions is input in real-time into the MATC module, whose pipeline stages progressively compress the execution trace into prefix and data streams, both of which are output through the trace port. Compressed traces are collected by the host until the trace experiment is complete, and can then be decompressed offline by ISA-independent decompression software. The resulting raw execution traces can subsequently be inspected to determine the execution path of the software-under-debug.

5.3 MATC Compression Stages

5.3.1 Consecutive Address Elimination

The nature of instruction execution typically involves executing a consecutive sequence of instructions followed by a branch to a BTA, which together are known as a basic block. The BTA at the end of every basic block acts as the starting address for the next basic block executed. For processors that use a fixed instruction-width ISAs, an execution trace of a basic block takes the form of a sequence of fixed-stride instruction addresses followed by a BTA. The redundancy within the predictably sequential portion of such a basic block can be exploited by representing the basic block in a more compact form.

The first MATC compression stage aims to eliminate the redundancy of recording sequential portions of basic blocks by performing Consecutive Address Elimination (CAE). The CAE stage represents each basic block as a combination of a BTA and a length term, the latter of which represents the number of consecutive instructions executed after a BTA. The example in Table 5.1 demonstrates the operation of the CAE stage, along with the necessity of including the length term. Since the goals of the MATC scheme include flexibility and portability, a decompressor is not expected to have access to the software-under-debug binary. As such, the length term serves to instruct the decompressor as to how many fixed-width instruction addresses should be interpolated between basic blocks.

Even if the decompressor were to hypothetically inspect the software-under-debug binary, it would only be able to reproduce basic blocks ending in branches whose BTAs could be determined offline, such as unconditional jmp branches. However, the second basic block of Table 5.1 can end in either a conditional bnz branch or with an unconditional jmp branch. Since the outcome of the bnz branch cannot be determined by inspecting the software-under-debug binary, the branch outcome must be provided at trace generation.
**Table 5.1**: Example of Consecutive Address Elimination with 16-bit Addresses

<table>
<thead>
<tr>
<th>Basic Block 1</th>
<th>Pre-CAE</th>
<th>Post-CAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Address</td>
<td>Instruction</td>
</tr>
<tr>
<td>0xa120</td>
<td>add</td>
<td>0xa120</td>
</tr>
<tr>
<td>0xa122</td>
<td>sub</td>
<td></td>
</tr>
<tr>
<td>0xa124</td>
<td>jmp 0xb144</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basic Block 2a</th>
<th>Pre-CAE</th>
<th>Post-CAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Address</td>
<td>Instruction</td>
</tr>
<tr>
<td>0xb144</td>
<td>load</td>
<td>0xb144</td>
</tr>
<tr>
<td>0xb146</td>
<td>sub</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basic Block 2b</th>
<th>Pre-CAE</th>
<th>Post-CAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Address</td>
<td>Instruction</td>
</tr>
<tr>
<td>0xb148</td>
<td>bnz 0xc298</td>
<td></td>
</tr>
<tr>
<td>0xb14c</td>
<td>xor</td>
<td></td>
</tr>
<tr>
<td>0xb150</td>
<td>jmp 0xc298</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Basic Block 3</th>
<th>Pre-CAE</th>
<th>Post-CAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Address</td>
<td>Instruction</td>
</tr>
<tr>
<td>0xc298</td>
<td>jmp 0xd422</td>
<td></td>
</tr>
</tbody>
</table>

time along with the BTA. In this case, the *length* term serves to provide that information, and additionally allows decompression without access to the software-under-debug binary.

Although the input to this stage is a 32-bit PC address, the two LSBits of a 32-bit instruction address are redundant on fixed instruction-width processors. As such, the output of the CAE stage includes an *address* consisting of the 30 MSBits of the BTA, and an 8-bit *length*, which are collectively referred to as an Address-Length (AL) combination. Both the work of Uzelac and Milenkovic [82], and the experiment results of Section 5.4 confirm that an 8-bit length is adequate to process the maximum number of instructions encountered within any basic block.

### 5.3.2 Finite Context Method

While the CAE stage reduces basic blocks into a sequence of AL combinations, the same ALs are likely to be repeatedly encountered within an execution trace. That redundancy is known as *temporal locality*, and originates from the inherent characteristics of executing software over time. The execution of loop iterations or iterative functions can generate significant amounts of temporal locality that may also be exploited by on-chip ILP hardware,
such as a BP.

The FCM aims to exploit the temporal locality of repeatedly-encountered sequences of instructions. While originally applied to software-based trace compression [99], the technique is used as the second stage of the MATC compression pipeline. The FCM hardware design in Figure 5.3 depicts an adapted algorithm that is suited to implementation on FPGA devices, using the sequence of AL combinations provided by the CAE stage as input.

The basis of the algorithm is to keep a history of previously-encountered sequences of AL combinations, and to remember the AL combination that followed a given sequence. The method operates similarly to a BTB, which provides a prediction of a BTA based upon some historical execution path information. The presented FCM implementation uses a search key that is created based upon execution history to address a line of a hash table in memory. The contents of a hash table line provide a prediction of the next AL combination that is likely to be encountered.

In constructing the search key, the aim is to obtain a unique value for each set of AL combinations encountered in a particular order. By using a portion of the length term from the input AL to construct the key, different keys will be generated for basic blocks with identical address terms but with differing length terms. That means basic blocks containing conditional branches that are taken can be differentiated from ones that are untaken. That concept is demonstrated in the example of Table 5.1, where unique search keys would be generated for basic blocks 2a and 2b. Despite their identical BTAs, the differences in their length terms are sufficient to uniquely identify each path of execution for the purposes of the FCM algorithm.

The FCM hash table resides in on-chip FPGA memory, which is commonly known as block ram. Such on-chip memory consists of a collection of SRAM banks that are commonly included on FPGA devices, and can be harnessed by hardware-emulated designs. While a soft-core processor may use several banks of block ram to implement an instruction cache or data cache, it is unlikely to require all available SRAM banks. Different variants of the MATC scheme harness different amounts of block ram to implement the hash table, allowing a variant which uses the most unused SRAM to be implemented.

The search key is used to address a hash table line that stores an AL combination, one which followed the four previously-encountered ALs. If the AL stored in the hash table line matches the input AL, it means that the same sequence of basic blocks has been previously
encountered. That property can be exploited in the Move-to-Front & Address Encoding (MTF/AE) compression stage to avoid storing the full input AL as part of the execution trace. To achieve that, a decompressor must shadow the same FCM operations during trace decompression in order to produce an identical hash table within its own memory. When the compressor finds a match between an input AL and hash table AL, a reference to the hash table can then be stored as part of the execution trace, instead of requiring that the full input AL be communicated to the decompressor. The FCM method then allows the temporal redundancy of the execution trace to be reduced significantly.

The FCM implementation of MATC variants $v0$ and $v1$ can be seen in Figure 5.3. Those variants construct a 16-bit search key by applying an XOR between staggered 16-bit portions of the last four instruction addresses (h1–h4) and the four LSBits of the length term (L) of the input AL. Since the search key acts as a pseudo-unique execution path vector for the most recently-encountered AL combinations, the use of staggered inputs is intended to reduce the likelihood of aliasing. Variants $v2$ and $v3$ similarly construct a search key from 14 bits of each address, while $v4$ uses 12 bits. In MATC variants $v0$ and $v1$, the search key is used to address a 304 kB hash table in SRAM which stores predicted AL combinations, each consisting of the 30 MSBits of the address term concatenated with an 8-bit length term. Variants $v2$ and $v3$ address 76 kB tables, while $v4$ addresses a 19 kB table. On every cycle, a new prediction is written to the table based upon the four previously-encountered ALs, while a new prediction is simultaneously read for the input AL. The storage of the prediction table in a dual-ported SRAM is essential to the operation. The FCM stage forwards both the predicted AL and input AL to the MTF/AE stage for comparison.

As with many cache-like structures, the hash table does not initially contain valid predictions. The FCM method therefore produces an incorrect prediction the first time a set of AL combinations is encountered, similar to a cache miss. Only when the hash table line is addressed with the same search key more than once can an AL prediction be made. If the prediction matches the input AL, it can be considered similar to a cache hit. The FCM algorithm can also experience something similar to cache aliasing, since there is some possibility that the same search key can be created from entirely different sequences of ALs. While the probability of aliasing increases as smaller hash tables are used, the only consequence would be reduced compression performance.
5.3.3 Move-to-Front & Address Encoding

This MTF/AE compression stage can be seen as an combination of two separate operations. The first implements the MTF transform for AL combinations which are not correctly
The transform has been shown to be effective at exploiting the temporal locality of execution traces [18, 82], and is used as part of both the AATC compression scheme described in Chapter 4, and as a compression stage of the MATC scheme described here. The transform operates by maintaining a chronological history of the unique symbols it has encountered. In the case of the MTF/AE compression stage, the symbols consist of a set of 38-bit AL combinations which are stored in a hardware array. Also known as an MTF dictionary, it contains a history of AL combinations which have not matched a predicted AL from the FCM stage. The dictionary consists of 256 elements for MATC variants v0 and v1, 128 elements for v2 and v3, and 64 elements for v3.

If the predicted AL does not match the input AL from the FCM stage, the MTF array is searched for a match to the input AL. Using parallel comparators, each element of the MTF array is compared with the input AL in the same clock cycle. Since the array contains only unique ALs, a single array index representing the matching element is placed in the
data stream if there is a match. The element is then moved to the top of the array, with the remaining elements shifted down to assume its previous position in the array. The process of matching an array element is depicted in the example of Figure 5.4. If an MTF array match cannot be found, the input AL is again placed in the element at the top of the array. However, the remainder of the array is shifted down by one element, eliminating the element containing the least-recently encountered AL combination.

The second operation of the MTF/AE stage involves encoding ALs according to Table 5.2. Encoding generates two separate streams: a byte representing the prefix and 1–5 bytes representing the data stream. If the input AL matches the predicted AL, the output is a single 0x00 prefix byte, which represents a correct prediction. Otherwise, if the AL is found to be located in the MTF array, the output is a 0x05 prefix byte along with the array index as the data stream byte. Failing both the previous possibilities, the size of the mispredicted and unmatched AL is minimized. To achieve that, a differential address encoding similar to the one demonstrated in Table 5.3 is employed. It should be noted that the table reflects the use of 16-bit addresses for clarity, although in practice the MTF/AE stage operates on 32-bit addresses.

Differential address encoding takes advantage of the fact that an instruction address of a 32-bit word-aligned, fixed instruction-width processor has two redundant LSBits. The example in Table 5.3 shows that a processor with 16-bit addresses would similarly have a single redundant LSBit. After ignoring redundant bits, the differential encoding scheme makes a comparison between the address of the most-recently encountered AL and the address of the input AL. The scheme determines the minimum number of whole bytes needed to represent the difference between the two addresses, and places those bytes in the data stream followed by the single-byte length of the input AL. In the case of differential encoding, the prefix represents the number of bytes a decompressor should read from the data stream in order to reproduce the input AL address, after which the input AL length byte should additionally be read from the same stream.

5.3.4 Data Stream Serializer

The MTF/AE stage places a variable-length output of between 1–5 bytes onto the data stream in a single cycle. The data stream can be further compressed with the LZ compression stage, but an efficient implementation of the LZ77 algorithm requires that it operate
Table 5.2: MTF/AE Compression Stage – 32-bit Address Encoding into Prefix and Data Streams

<table>
<thead>
<tr>
<th>Prefix Byte</th>
<th>Data Stream Byte(s)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>–</td>
<td>Correct AL prediction by FCM stage</td>
</tr>
<tr>
<td>0x01</td>
<td>address[9:2] + length[7:0]</td>
<td>1 byte Difference</td>
</tr>
<tr>
<td>0x02</td>
<td>address[17:2] + length[7:0]</td>
<td>2 bytes from the address of the</td>
</tr>
<tr>
<td>0x03</td>
<td>address[25:2] + length[7:0]</td>
<td>3 bytes previously-encountered</td>
</tr>
<tr>
<td>0x04</td>
<td>address[31:2] + length[7:0]</td>
<td>4 bytes AL combination</td>
</tr>
<tr>
<td>0x05</td>
<td>mtf_index[7:0]</td>
<td>Index of AL which is in the MTF array</td>
</tr>
</tbody>
</table>

on fixed-width inputs. For that reason, the data stream must be serialized into byte-width inputs before being handed to the LZ stage.

One possible solution is to include a small on-chip buffer to store data stream outputs from the MTF/AE stage. Buffering could take advantage of the fact that MTF/AE stage data stream outputs greater than one byte are relatively infrequent, which would allow serialization of buffered inputs to take place during periods of inactivity. The buffer size could be experimentally adjusted to ensure that it does not overflow under normal execution scenarios, such as when a rapid series of multi-byte inputs is encountered.

Another solution is to operate the serializer stage at 5x the clock frequency of the MTF/AE stage, which allows any series of worst-case 5-byte inputs to be serialized successfully. The MATC scheme presented in this chapter takes that approach for the sake of simplicity and robustness, though a buffering scheme presents opportunities for future work.

5.3.5 Run-length and Prefix Encoding

Like the MTF/AE stage, the Run-Length & Prefix Encoding (RL/PE) stage can also be considered a combination to two separate operations: a Prefix Encoding which exploits the spatial and temporal redundancy within the prefix stream, and a Run-Length Encoding (RLE) transform which represents long sequences of correct FCM predictions within the prefix stream in a more compact way.

The MTF/AE stage allows an AL combination to be represented by any of three different
5.3 MATC Compression Stages

Table 5.3: Example of Differential Address Encoding – 16-bit Addresses

<table>
<thead>
<tr>
<th>Input Address</th>
<th>Input Length</th>
<th>Predicted?</th>
<th>Bytes diff.</th>
<th>Output Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>Binary</td>
<td></td>
<td>Prefix</td>
<td>Data (diff. + length)</td>
</tr>
<tr>
<td>0x5090</td>
<td>101 0000 1001 0000</td>
<td>0x12</td>
<td>Yes</td>
<td>–</td>
</tr>
<tr>
<td>0x50a2</td>
<td>101 0000 1010 0010</td>
<td>0x32</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>0x614c</td>
<td>110 0001 0100 1100</td>
<td>0x1a</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>0x6199</td>
<td>110 0001 1001 1001</td>
<td>0x3f</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>0x63a5</td>
<td>110 0011 1010 0101</td>
<td>0x07</td>
<td>Yes</td>
<td>–</td>
</tr>
</tbody>
</table>

†after ignoring redundant least significant bit of original 16-bit address

*boxed area denotes least significant byte(s) that differ from previous input

transforms, depending upon which can represent an AL in the most efficient way. As one of two streams generated by the MTF/AE stage, bytes of the prefix stream inform a decompressor as to which transform should be used to recover compressed AL combinations. However, the prefix bytes 0x00–0x05 themselves contain spatial redundancy, which can be seen from the leading zero bits contained within every possible prefix. It can also be experimentally determined that different prefix bytes occur in different frequencies when a typical execution trace in compressed. That frequency variation constitutes an additional temporal redundancy that can also be exploited.

To exploit the spatial and temporal redundancy of the prefix stream, an encoding scheme based upon the Huffman Tree of Figure 5.5 is used. The tree assigns the shortest codes to the prefixes which are found to occur most frequently in experimental characterization of typical execution traces. The shortest single-bit '0' code is assigned to a correct FCM prediction, which is overwhelmingly the most commonly-encountered prefix. A longer '11' code represents a matching AL in the MTF array, which is the second most-commonly encountered prefix. Longer codes are assigned to differential address prefixes, with the shortest of those representing the smallest address differences. The codes of the Huffman Tree also possess what is known as the prefix property, meaning no code constitutes the start of any other code. That ensures unambiguous decoding when codes are stored sequentially in a bitstream, without incurring the overhead of adding delimiters between codewords. While the presented static Huffman Tree presumes a fixed set of frequencies over time, a
dynamic encoding scheme could also have been used to adapt to changing prefix frequencies. However, it would have done so at the cost of additional hardware area, complexity, and memory usage.

While Prefix Encoding suffices to reduce some redundancies within the prefix stream, additional redundancies can be exploited through the use of the RLE transform. Experimental observations reveal that correct FCM predictions generally occur in bursts, when long sequences of basic blocks are repeatedly executed. While Prefix Encoding represents each correct prediction as a single '0' bit, there is an inherent redundancy in encoding bursts of correct predictions as long sequences of '0' bits. Instead, the RLE transform is applied to represent the sequences in a more compact form. It identifies sequences of between 3–256 correct predictions ('0' bits) and replaces them with a single byte. While there is no change to Prefix Encoding output if sequences of three or fewer '0' bits are encountered, the RLE transform is activated when a fourth consecutive '0' bit is observed. In that case, the prefix stream output is paused in order to count the number of '0'-bit inputs that follow. Counting continues until either the first non-'0' bit is encountered, or the maximum sequence of 256 '0'-bits is reached. A byte representing the count is then output as part of the prefix stream.

The described RLE transform allows a sequence of 260 correct FCM predictions to be represented by only 12 bits: a sequence of four '0' bits, followed by a 1-byte count. There is also a possibility that the scheme results in a data expansion, such as when representing a sequence of only four correct FCM predictions by the same 12 bits. However, in practice it is found that such expansions seldomly occur, and the net effect of the RLE transform is a positive contribution to execution trace compression performance.

5.3.6 Lempel-Ziv Encoding of the Data Stream

The previous compression stages focus on exploiting temporal and spatial redundancies at the granularity of an AL combination. The resulting data stream output of the MTF/AE stage is comprised of single-byte MTF indices and differential ALs of between 2–5 bytes. However, experimental observation of the data stream reveals that certain sequences of AL combinations still generate repeating output patterns. The LZ compression stage aims to extract any non-obvious or vestigial redundancy that remains in the data stream. That is achieved by using a hardware implementation of the original Lempel-Ziv algorithm [81],
also known as LZ77, to apply a byte-level transform which identifies and exploits repeating combinations of data stream bytes. The LZ77 algorithm is used partly due to its efficient hardware implementation, which has been shown to be effective when used for trace compression purposes [76].

An LZ compression dictionary of 256 elements is implemented to balance FPGA resource usage, latency, and compression performance needs. Each dictionary element is composed of a byte which can be compared to one from the input data stream. In processing an input byte, a parallel comparator searches the array for matching elements in a single clock cycle. Matching elements are flagged in a tag array that contains one bit per dictionary element. The dictionary array is then shifted by a single element, eliminating the least-recently encountered byte, and adding the input byte to the top of the array. In processing subsequent input bytes, only elements flagged in the tag array are compared to the input byte. The process allows an arbitrary sequence of inputs bytes to be matched to a sequence already within the dictionary. A major advantage of the LZ77 scheme is the incremental matching of input elements and dictionary elements, allowing one element to be matched in every clock cycle in order to reduce algorithmic and hardware complexity. When an input byte no longer matches a previously-matched sequence, or if no match is
initially found, a 3-byte output is issued. The output includes the dictionary array index of the original matching element, the number of elements successfully matched (up to and including the dictionary size), and the value of the non-matching input byte. Decompression can be performed by maintaining an identical LZ dictionary array in memory, which allows entire sequences of repeating bytes to be read from the decompression dictionary instead of requiring it to be explicitly stored as part of the execution trace.

The LZ compression stage achieves compression rates ranging from 0.9x–5x in compressing the data stream output of the MTF/AE stage, when applied to the execution traces of individual benchmarks from the MiBench Suite used in Section 5.4.

5.4 Experimental Results

The performance of the proposed MATC execution trace compression scheme is evaluated with three sets of experiments. The first experiment compares the performance of three software-based general-purpose compression algorithms in compressing the execution traces of nine benchmarks of the MiBench Benchmark Suite [87]. Those schemes indicate the performance that can be achieved using complex algorithms that employ vast amounts of memory and computational power. The results serve as a basis for comparing the performance of purpose-built execution trace compression schemes in the second experiment. In that experiment, the compression performance of the five variants of the proposed MATC scheme is compared with three hardware-based execution trace compression schemes from the literature. In the final experiment, the FPGA resource usage of all MATC variants is evaluated using an FPGA device that could typically serve as a hardware-emulated development platform.

5.4.1 Compression Performance

The execution traces of nine benchmarks of the MiBench Benchmark Suite [87] are used to evaluate the compression performance of the proposed MATC scheme. A comparison of the scheme is made with both software-based general-purpose compression algorithms, as well as other real-time hardware-based execution trace compression schemes.

The benchmark suite includes programs that approximate typical workloads, and is used by several real-time trace compression schemes for performance comparison purposes [83, 82, 93], including the AATC scheme introduced in Chapter 4. The benchmarks are
Table 5.4: Performance of Software-based General-Purpose Compression Schemes in Compressing Execution Traces (bits/instruction)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Instructions (millions)</th>
<th>Software-based Schemes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>gzip</td>
</tr>
<tr>
<td>bf_e</td>
<td>781.3</td>
<td>0.3392</td>
</tr>
<tr>
<td>jpeg</td>
<td>93.5</td>
<td>0.3801</td>
</tr>
<tr>
<td>djpeg</td>
<td>22.6</td>
<td>0.6764</td>
</tr>
<tr>
<td>fft</td>
<td>36.5</td>
<td>0.4409</td>
</tr>
<tr>
<td>sha</td>
<td>118.5</td>
<td>0.2002</td>
</tr>
<tr>
<td>stringsearch</td>
<td>4.5</td>
<td>0.5315</td>
</tr>
<tr>
<td>tiff2bw</td>
<td>157.6</td>
<td>0.3457</td>
</tr>
<tr>
<td>tiff2rgba</td>
<td>190.9</td>
<td>0.4294</td>
</tr>
<tr>
<td>tiffmedian</td>
<td>595.1</td>
<td>0.3108</td>
</tr>
<tr>
<td>Unweighted Average</td>
<td></td>
<td>0.4060</td>
</tr>
</tbody>
</table>

compiled for a PowerPC fixed instruction-width ISA using the GCC compiler under the Linux OS. To collect execution traces resulting from benchmark execution on the target architecture, a physical, hardware-emulated, or software-emulated platform is needed.

For the purposes of this chapter, a hard-core 1.25 GHz 32-bit PowerPC 7455 processor is used for executing the benchmarks in order to collect their corresponding execution traces. The same software instrumentation script described in Section 3.6.1 is used to extract the execution traces of the MiBench benchmarks. The script loads a benchmark executable into a GDB 7.1-1 client under the Linux OS running kernel 2.6.32-24, and uses a looped `si` command to single-step through program code. Setting the PC register contents to auto-display with the command `disp/i $pc` allows the PC address and its disassembly to be output on each step. By redirecting standard output to a file, an execution trace can be captured.

The software-instrumented script is found in Section 3.6.1 to add an overwhelming overhead to the execution time of benchmarks used in that chapter, and consequently to the time required to collect their execution traces. In contrast to Chapter 3, the experiments of this section rely on execution traces collected through the more efficient execution of a
Table 5.5: Hardware-based Execution-Trace Compression Scheme Performance Comparison with MATC (bits/instruction)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Hardware-based Schemes</th>
<th>MATC Scheme Variants</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SCI-Ntup</td>
<td>eSDC-LSP</td>
</tr>
<tr>
<td>bf_e</td>
<td>0.3250</td>
<td>0.345</td>
</tr>
<tr>
<td>cjpeg</td>
<td>0.2169</td>
<td>0.088</td>
</tr>
<tr>
<td>djpeg</td>
<td>0.1697</td>
<td>0.053</td>
</tr>
<tr>
<td>fft</td>
<td>N/A</td>
<td>0.542</td>
</tr>
<tr>
<td>sha</td>
<td>0.0727</td>
<td>0.074</td>
</tr>
<tr>
<td>stringsearch</td>
<td>1.3212</td>
<td>0.412</td>
</tr>
<tr>
<td>tiff2bw</td>
<td>0.1361</td>
<td>0.030</td>
</tr>
<tr>
<td>tiff2rgba</td>
<td>0.0786</td>
<td>0.012</td>
</tr>
<tr>
<td>tiffmedian</td>
<td>0.0772</td>
<td>0.027</td>
</tr>
<tr>
<td>Unweighted Average</td>
<td>0.2997</td>
<td>0.176</td>
</tr>
</tbody>
</table>

The result of compressing execution traces with three software-based general-purpose algorithms can be seen in Table 5.4. The Linux programs gzip v1.3.12 [74] and bzip2 v1.0.5 [73] are used to provide the results of their underlying algorithms, while p7zip v9.04 is used to provide results of the lzma2 algorithm [72]. All programs are executed with the default compression settings, and operate on execution traces of individual benchmarks stored in binary format. The results show that execution traces contain significant amounts of redundancy that can be exploited through various compression techniques. Each of gzip, bzip2, and lzma2 algorithms offer progressively increased performance at the cost of progressively more memory usage and compression time. In particular, the lzma2 algorithm is currently similar script on a hard-core processor, instead of a software-emulated one. Nevertheless, the significant overhead added by the software-instrumented script remarkably limits execution trace collection speed. The potentially excessive trace collection times for some benchmarks, along with a difficulty in compiling other benchmarks, limit the evaluation of execution trace compression performance to a comparison of nine benchmarks of the MiBench suite.
considered one of the highest-performing general-purpose compression algorithms, and its results represent the performance achievable at the cost of vast amounts of computational intensity and memory usage.

In Table 5.5, a comparison of the five variants of the presented MATC scheme is made with three real-time hardware-based execution trace compression schemes from the literature: SCI-Ntup [93], eSDC-LSP [83], and eDMTF [82]. An unweighted average performance is calculated across benchmarks for each compression scheme, representing an execution scenario where each benchmark is given an equal execution time on the target processor. The MATC scheme is implemented as a parameterizable VHDL description, each variant of which is functionally simulated with ModelSim SE-64 v6.5c. The previously-collected benchmark execution traces are supplied as simulation inputs, and the resulting compressed outputs recorded. To verify that compressed outputs expand back into their original forms, software decompression is performed with a MATLAB R2010a program created for the task.

The results of Table 5.5 show that MATC variants $v0$–$v3$ outperform all other schemes on average, while variant $v4$ outperforms both SCI-Ntup and eSDC-LSP. When considering the inverse performance metric of instructions per bit used in Table 4.11 of Chapter 4, the best-performing MATC variant $v0$ encodes 18%, 70%, and 190% more instructions per stored bit than SCI-Ntup, eSDC-LSP, and eDMTF schemes, respectively. All variants also outperform the commercial tools ETM v1 and MCDS, which are reported to achieve execution trace compression performance of 1 bit/instruction [98] and 1.6 bits/instruction [54] on unspecified workloads, respectively.

Even though the MATC scheme operates under the constraints of real-time operation and with limited hardware resources, all MATC variants perform better than both gzip and bzip2 software-based general-purpose compression schemes. Only the most computationally-intense lzma2 algorithm can be observed to outperform the MATC scheme.

5.4.2 FPGA Resource Utilization

The third experiment evaluates the hardware resource usage of the MATC scheme for an FPGA device that could potentially serve as a hardware-emulated development platform. Hardware synthesis of all MATC variants is performed for an Altera Stratix III EP3SL200F1152C2 FPGA device using Altera Quartus II v9.1. Logic utilization for every MATC variant can be seen in Table 5.6, including a per-stage breakdown for variant $v0$. 
Table 5.6: MATC Compression Scheme Logic Utilization on an Altera Stratix III FPGA

<table>
<thead>
<tr>
<th>MATC Variant ((s, m))</th>
<th>ALUTs</th>
<th>Registers</th>
<th>Memory (kB)</th>
<th>Speed (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0 ((16, 256))</td>
<td>7859</td>
<td>12597</td>
<td>304</td>
<td>68</td>
</tr>
<tr>
<td>CAE</td>
<td>65</td>
<td>108</td>
<td>–</td>
<td>261</td>
</tr>
<tr>
<td>FCM</td>
<td>63</td>
<td>146</td>
<td>304</td>
<td>325</td>
</tr>
<tr>
<td>MTF/AE</td>
<td>5305</td>
<td>9808</td>
<td>–</td>
<td>75</td>
</tr>
<tr>
<td>Serializer</td>
<td>155</td>
<td>113</td>
<td>–</td>
<td>275</td>
</tr>
<tr>
<td>Run-Length</td>
<td>112</td>
<td>86</td>
<td>–</td>
<td>277</td>
</tr>
<tr>
<td>LZ77</td>
<td>2183</td>
<td>2369</td>
<td>–</td>
<td>119</td>
</tr>
<tr>
<td>v1 ((16, 128))</td>
<td>5210</td>
<td>7735</td>
<td>304</td>
<td>99</td>
</tr>
<tr>
<td>v2 ((14, 128))</td>
<td>5170</td>
<td>7724</td>
<td>76</td>
<td>102</td>
</tr>
<tr>
<td>v3 ((14, 64))</td>
<td>3830</td>
<td>5285</td>
<td>76</td>
<td>131</td>
</tr>
<tr>
<td>v4 ((12, 64))</td>
<td>3839</td>
<td>5289</td>
<td>19</td>
<td>139</td>
</tr>
</tbody>
</table>

The MATC variants represent between 2–5% of available Adaptive Lookup Table (ALUT) resources for the FPGA device, and between 3–8% of available logic registers.

It is difficult to perform a direct comparison of resource utilization between the MATC scheme and the others listed in Table 5.5. The schemes SCI-Ntup [93] and eSDC-LSP [83] report only complexity estimates or normalized ASIC area information, which is difficult to compare with a scheme such as MATC that is intended for FPGA implementation. While the eDMTF [82] scheme claims a logic utilization of 25000 gates, it is unclear how such a scheme would map to FPGA ALUTs, or what speeds it could achieve on FPGA devices.

Due to the design decision possibilities detailed in Section 5.3.4, the effective speed of the MATC scheme would match reported clock rates with the use of a small input buffer in the Serializer stage. If a buffer is omitted, the Serializer and LZ stages would instead need to be clocked at 5x the speed of the other stages. The MTF/AE stage can be observed to be the limiting factor in the overall performance of the pipeline. In particular, there is a trade-off between the speed of that stage and MTF array size, due to the fact that MTF elements reside in a long shift register.
5.5 Chapter Summary

This chapter presents a parameterizable execution trace compression scheme suited to hardware-emulated development platforms. The pipelined FPGA implementation can be catered to meet various points on the spectrum of tradeoffs between compression performance and embedded SRAM utilization, logic resource usage, and maximum clock frequency. That allows the presented MATC scheme to potentially harness any unused logic and memory resources that will remain on the FPGA after the configuration of a hardware-emulated soft-processor.

The presented hardware design is portable to various fixed instruction-width ISA, while the corresponding execution trace decompression software can similarly decompress the traces of any fixed instruction-width ISAs. The combination enables the scheme to be rapid deployed on hardware-emulated designs in the midst of hardware-software co-design, allowing software debugging of complexity-related and timing-related faults to begin earlier in the co-design process than would otherwise be possible. If that can be achieved by leveraging unused FPGA resources, both development time and cost can be reduced.

Of the variants of the MATC presented in this chapter, the best-performing variant \( v_0 \) encodes 18% more instructions per stored bit than a similar scheme, and requires only 0.1033 bits of storage per instruction. The fastest variant \( v_4 \) operates at 139 MHz, while requiring only 2% of the ALUTs, and 3% of the logic registers available on an FPGA device that can be used as a basis for a hardware-emulated development platform.
Chapter 6

Conclusions and Future Work

As the computational capabilities of modern microprocessors and SoCs continue to grow, so too will the complexity of the software that is created to take advantage of them. Increasingly complex software is composed of many interdependent parts, the interaction of which can cause unintended behaviour. Such software has been shown to be inherently susceptible to a growing number of complexity-related and timing-related faults that were once considered rare. Traditional execution-control software debugging methods are based upon principles that introduce changes to software timing, potentially masking the manifestation of such faults. The fact that established debugging methods are unsuitable for resolving complex faults has contributed to an environment wherein software debugging now reportedly constitutes the majority of software development time and cost. There is a consensus that the debugging of modern software should instead be performed by unobtrusively observing its execution in real-time, and generating traces that can be analyzed post-hoc to identify software faults. This thesis presents contributions to trace generation infrastructure that enable or enhance its use across different types of software development platforms.

Software-emulated development platforms enable standardized and low-cost development environments to be employed by developers, and are especially being adopted in the domains of embedded and mobile computing. With rising software complexity in those domains, there is a growing need to use trace generation-based software debugging methods to resolve complex software faults. In Chapter 3, an infrastructure for performing trace generation-based debugging is provided for the ARM ISA emulation of the open-source
QEMU emulator. It enables instruction-level traces of registers and memory addresses to be generated for emulated software-under-debug, allowing complex faults to be observed at a fine-grained level. The execution-control debugging capabilities of the QEMU emulator are extended, allowing the control of trace experiments to be dynamically exercised through a standardized interface with a GDB debugger client. To ensure that only the traces of relevant segments of software are recorded, the infrastructure allows traces to be qualified in any of five different scopes. The most discerning scope allows only the traces of a single Linux process to be captured when the trace generation infrastructure is made aware of Linux context-switching activity through a small Linux kernel patch. The resulting trace generation is orders of magnitude faster than a software instrumentation scheme, while allowing far less trace data volume to be captured than by existing QEMU static instrumentation capabilities.

The use of instruction-level trace generation is considered essential to observing complexity-related and timing-related software faults. On-chip trace generation hardware that enables such faults to be observed is increasingly found on development platforms with one or more processor cores. The observation of long periods of software execution, or of multiple processor cores for shorter periods, requires transferring large volumes of generated trace data off-chip in real-time for post-hoc analysis. However, doing so can generate excessive trace data volume and require impractically large amounts of trace port bandwidth. As a result, on-chip trace compression is widely used to reduce the inherent redundancies contained within some types of traces. The use of high-performance on-chip trace compression schemes is expected to become essential to enabling the debugging of future software. A growing number of on-chip processor cores will serve to further drive the development of multi-threaded software. The leading solution to adequately debugging such software is the use of multi-core trace generation, which will require advanced compression techniques to manage the vast quantities of generated trace data.

The Architecture-Aware Trace Compression scheme introduced in Chapter 4 is suited to the compression of ARM ISA execution traces on physical development platforms. The scheme repurposes modern Instruction-Level Parallelism hardware, including a Branch Predictor and Branch Target Buffer, and leverages the accuracy of their predictions for compression purposes. Enhanced performance is achieved through an awareness of the linked branches used by many ISAs. Recognizing that most software-under-debug is typically comprised of machine code generated by compilers, the method also exploits the predictable
compiler-driven movement of return-addresses between a link-register, stack, and program counter. An analysis of the nature of execution trace data reveals structural patterns that inform the use of additional data transforms and encodings. The resulting scheme offers two variants which satisfy different tradeoffs between performance and hardware logic utilization, both of which encode considerably more instructions per stored bit than any other scheme in the literature.

The Multi-Architectural Trace Compression scheme presented in Chapter 5 targets the compression of execution traces on hardware-emulated development platforms. Such platforms typically contain one or more soft-core processors configured on an FPGA device, and are used in prototyping or as part of the hardware-software co-design process. In the latter scenario, the ability to debug complex software faults early in the co-design process can be considered essential in reducing overall software development time and cost. The pipelined scheme is compatible with any fixed instruction-width ISA, and can be rapidly deployed alongside a soft-processor to compress execution traces without requiring knowledge of the underlying hardware or executing software. A parameterizable design allows five different variants of the scheme to be instantiated, depending upon the amount of unused logic and memory resources that will remain on the FPGA after the configuration of a hardware-emulated design. The resulting scheme offers rapid deployment, flexibility, and better performance than comparable compression schemes when used with hardware-emulated platforms.

In summary, this thesis presents a range of contributions to the domain of trace generation-based software debugging, including advanced on-chip supporting hardware, complementary software tools, and interfaces to bridge the gap between them. The contributions enable or enhance the use of trace generation across a variety of software development platforms. For software-emulated development platforms that depend upon the QEMU emulator, a contextually-aware and standardized interface for trace generation is provided. For physical and hardware-emulated development platforms, solutions are presented to the excessive trace data volumes that result when tracing the flow of software execution.

6.1 Future Work

There are many opportunities for future research in trace generation-based software debugging infrastructure. While the compression schemes presented in both Chapters 4 and
Conclusions and Future Work

5 can be applied to execution traces, a scheme that applies to data traces can also be considered. Even though data traces have been shown to contain far less redundancy than their counterparts, an investigation of redundancy reduction strategies is needed that extends beyond what have mainly been general-purpose methods employed in the past. The rise in the number of on-chip processor cores also presents an opportunity to explore trace compression schemes that exploit the redundancies between trace streams that are generated concurrently. There is potentially significant spatial and temporal redundancy between the traces of concurrently-executing threads which execute similar program code, but happen to operate on different data sets. There is also a need for modularized decompression of traces, which allows execution time slices to be individually analyzed without requiring the trace storage to decompress long execution sequences. That can be challenging due to the fact that many compression schemes dynamically build a compression dictionary, or generate predictions, based upon a history of symbols that have already been processed. A method would need to be developed that preserves compression performance while limiting the history of symbols that is needed to perform decompression.

Extensions of the trace generation infrastructure introduced in Chapter 3 include the completion of the GDB trace experiment feature set. The implementation of an interpreter for the full set of GDB agent expressions could deliver powerful debugging capabilities to a software developer. Entire micro-programs of GDB bytecode could be generated to perform trace collection, qualification, manipulation, and certain decision-making at runtime. While such micro-programs represent an execution overhead, untimed platforms such as the QEMU emulator can benefit from their capabilities when debugging software faults about which little or nothing is known. When trace data can be “groomed” at collection time based upon some criteria, the potential problems associated with excessive trace data volume and excessive post-hoc analysis time can also be minimized. The QEMU emulator also supports a long list of target processor ISAs, and has the ability to software-emulate a multi-core processor in a round-robin fashion within a single host processor thread. The trace generation infrastructure can also be extended to support those full capabilities of the QEMU emulator.

In general, realizing the benefits of reduced software development time and cost will require standardized, modularized, and integrated software debugging infrastructure. The adoption of a standardized tracing interface, hardware trace generation structures, software debugging tools, and trace output formats can ensure cross-compatibility between existing
and future development tools. Such standardization can allow interoperable hardware structures and software tools to be implemented and incrementally improved.

A growing number of on-chip processor cores will also require a scalable mechanism to configure triggers and cross-triggers, exercise trace control, and output traces in chronological order. The ability to maintain a deterministic order between traces generated in different clock domains can be considered especially important to future multi-threaded software debugging.

The analysis and interpretation of trace data is also an important consideration to a software developer. Unlike execution-control software debugging, additional software tools are needed to aggregate, analyze, and visualize traces in a post-hoc debugging scenario. While passive software profiling tools currently have similar abilities, future debugging tools will need to exercise dynamic control over trace experiments, organize analyzed trace data, and find relevant differences between trace experiments. The efficacy of those tools, and their ability to interact with a software developer, can ultimately determine software debugging time and cost.

Finally, a parallel can be drawn between modern approaches to both hardware and software debugging. Similar to trace generation-based software debugging, assertion-based hardware debugging [100] allows triggers to be activated in real-time to signal erroneous conditions, which enables post-hoc hardware debugging to take place. There is an opportunity to investigate the overlap between both types of debugging within combined co-debug platforms which have been proposed in recent years [101]. In hardware-software co-design, unified control mechanisms for configuring and controlling both interfaces can also potentially save co-development time and cost.
Bibliography


List of Acronyms


**ABI** Application Binary Interface. 80

**AL** Address-Length. 118–126

**ALUT** Adaptive Lookup Table. 132, 133

**API** Application Programming Interface. 23

**ASIC** Application-Specific Integrated Circuit. 7, 20, 33, 132

**ASLR** Address Space Layout Randomization. 44

**BP** Branch Predictor. 10, 13, 52, 80, 81, 83, 84, 101–104, 107, 119, 136

**BTA** Branch Target Address. 51, 52, 80–94, 98–102, 104, 117–119

**BTB** Branch Target Buffer. 10, 13, 52, 80, 81, 84–86, 90, 101–104, 107, 119, 136

**CAE** Consecutive Address Elimination. 117–119

**CPI** Cycle per Instruction. 46, 47

**CPSR** Current Program Status Register. 62, 68, 70

**CPU** Central Processing Unit. 20, 22, 70, 72

**DFT** Design-for-Test. 27, 28
ECU  Engine Control Unit. 1
ETB  Embedded Trace Buffer. 45
ETM  Embedded Trace Macrocell. 51, 131
FCM  Finite Context Method. 113, 114, 119, 120, 122, 124–126
FP   Floating-Point. 56, 70
FPGA Field-Programmable Gate Array. v, 7, 11, 13, 20, 33, 111–115, 119, 127, 128, 131–133, 137
GCC GNU Compiler Collection. 14, 102, 129
GDB GNU Debugger. 9, 12, 14, 22, 23, 27, 32, 33, 42, 44, 56, 59–65, 69, 72, 79, 102, 129, 136, 138
HPC High-Performance Computing. 35
I/O  Input/Output. 27, 38, 41
ICE In-Circuit Emulation. 28, 29, 31, 32, 34, 37, 40, 44, 53
IDE Integrated Development Environment. 19
ILP Instruction-Level Parallelism. 10, 13, 52, 101, 118, 136
IP   Intellectual Property. 5, 9, 33, 112, 113
ISR Interrupt Service Routine. 26
ISS Instruction-Set Simulator. 14, 31
JTAG Joint Test Action Group. 27, 42
LLDB LLDB Debugger. 22
List of Acronyms

**LR** Link Register. 86–92

**LSBit** Least Significant Bit. 51, 118, 120, 123

**LUT** Lookup Table. 113, 114

**LZ** Lempel-Ziv. 51, 123, 124, 126–128, 132

**MATC** Multi-Architectural Trace Compression. v, 112–115, 117, 119, 120, 122, 124, 128, 131–133, 137

**MCDS** Multi-Core Debug Solution. 38, 131

**MCU** Microcontroller. 19, 22

**MSBit** Most Significant Bit. 51, 118, 120

**MTF** Move-to-Front. 51, 81, 94, 98, 101, 108, 114, 121–123, 125, 126, 132

**MTF/AE** Move-to-Front & Address Encoding. 120–126, 128, 132

**MTI** Model Trace Interface. 38

**NoC** Network-on-Chip. 11, 43, 49

**OS** Operating System. 8, 10, 15, 19, 22, 23, 25, 32, 33, 39, 44, 56, 102, 129

**PC** Program Counter. 29, 39, 46, 47, 51, 56, 58–60, 69, 83, 86, 89–92, 102, 114, 118, 129

**PID** Process ID. 12, 56, 60, 62, 65, 68, 70, 72

**QEMU** Quick EMUlator. v, 9, 12, 14, 32, 39, 44, 55–61, 63, 65, 68–70, 72, 102, 136–138

**RAS** Return Address Stack. 89

**RL/PE** Run-Length & Prefix Encoding. 124

**RLE** Run-Length Encoding. 124, 126
RSP  Remote Serial Protocol.  22
RTL  Register-Transfer Level.  20, 102, 107
SCM  Source Code Management.  19
SDK  Software Development Kit.  32
SFD  Stack Frame Depth.  89–92
SIMD  Single-Instruction Multiple-Data.  14
SoC  System-on-Chip.  5, 13, 21, 25, 27, 28, 30, 34, 37, 42, 43, 45, 47, 49, 75, 76, 111, 135
SRAM  Static Random-Access Memory.  13, 45, 113, 119, 120, 133
TAP  Test Access Port.  27, 29, 34, 42, 45, 111
TB  Translation Block.  58, 59
TCG  Tiny Code Generator.  57–61, 68
TLM  Transaction-Level Modeling.  20, 38
TOS  Top of the Stack.  90, 91
UART  Universal Asynchronous Receiver/Transmitter.  23
VarLen  Variable-Length.  81, 93, 94, 98–102, 104, 108
VHDL  VHSIC hardware description language.  102, 113, 131
XOR  Exclusive-OR.  98, 120
XP  Extreme Programming.  18