

Reid, Alastair David (2019) *Defining interfaces between hardware and software: Quality and performance*. PhD thesis awarded by published work.

<https://theses.gla.ac.uk/41068/>

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten: Theses

<https://theses.gla.ac.uk/>
research-enlighten@glasgow.ac.uk

Defining interfaces between hardware and software: Quality and performance

Alastair David Reid

M.Sc. (University of Glasgow) 1994

B.Sc. (University of Strathclyde) 1988

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School of Computing Science
College of Science and Engineering
University of Glasgow



University
of Glasgow

March 2019

To Edye Hoffmann
Wife, best friend and tireless cheerleader.

© Copyright
Alastair David Reid
All rights reserved
2019

Abstract

One of the most important interfaces in a computer system is the interface between hardware and software. This interface is the contract between the hardware designer and the programmer that defines the functional behaviour of the hardware. This thesis examines two critical aspects of defining the hardware-software interface: quality and performance.

The first aspect is creating a *high quality* specification of the interface as conventionally defined in an instruction set architecture. The majority of this thesis is concerned with creating a specification that covers the full *scope* of the interface; that is *applicable* to all current implementations of the architecture; and that can be *trusted* to accurately describe the behaviour of implementations of the architecture. We describe the development of a formal specification of the two major types of Arm processors: A-class (for mobile devices such as phones and tablets) and M-class (for micro-controllers). These specifications are unparalleled in their scope, applicability and trustworthiness. This thesis identifies and illustrates what we consider the key ingredient in achieving this goal: creating a specification that is used by many different user groups. Supporting many different groups leads to improved quality as each group finds different problems in the specification; and, by providing value to each different group, it helps justify the considerable effort required to create a high quality specification of a major processor architecture. The work described in this thesis led to a step change in Arm's ability to use formal verification techniques to detect errors in their processors; enabled extensive testing of the specification against Arm's official architecture conformance suite; improved the quality of Arm's architecture conformance suite based on measuring the architectural coverage of the tests; supported earlier, faster development of architecture extensions by enabling animation of changes as they are being made; and enabled early detection of problems created from architecture extensions by performing formal validation of the specification against semi-structured natural language specifications. As far as we are aware, no other mainstream processor architecture has this capability. The formal specifications are included in Arm's publicly released architecture reference manuals and the A-class specification is also released in machine-readable form.

The second aspect is creating a *high performance* interface by defining the hardware-software interface of a software-defined radio subsystem using a programming language. That is, an interface that allows software to exploit the potential *performance* of the underlying hardware. While the hardware-software interface is normally defined in terms of machine code, periph-

eral control registers and memory maps, we define it using a programming language instead. This higher level interface provides the opportunity for compilers to hide some of the low-level differences between different systems from the programmer: a potentially very efficient way of providing a stable, portable interface without having to add hardware to provide portability between different hardware platforms. We describe the design and implementation of a set of extensions to the C programming language to support programming high performance, energy efficient, software defined radio systems. The language extensions enable the programmer to exploit the pipeline parallelism typically present in digital signal processing applications and to make efficient use of the asymmetric multiprocessor systems designed to support such applications. The extensions consist primarily of annotations that can be checked for consistency and that support annotation inference in order to reduce the number of annotations required. Reducing the number of annotations does not just save programmer effort, it also improves portability by reducing the number of annotations that need to be changed when porting an application from one platform to another. This work formed part of a project that developed a high-performance, energy-efficient, software defined radio capable of implementing the physical layers of the 4G cellphone standard (LTE), 802.11a WiFi and Digital Video Broadcast (DVB) with a power and silicon area budget that was competitive with a conventional custom ASIC solution.

The Arm architecture is the largest computer architecture by volume in the world. It behooves us to ensure that the interface it describes is appropriately defined.

Contents

Abstract	iii
Contents	v
List of Figures	vii
Acknowledgements	ix
Preface and Declaration	xi
1 Defining interfaces between hardware and software	1
1.1 Introduction	1
1.1.1 Organisation	2
1.2 Creating high quality definitions of hardware-software interfaces	2
1.2.1 Literature survey	6
1.2.2 Overview of published work	10
1.2.3 Contributions	12
1.2.4 Limitations and further work	13
1.2.5 Conclusions	15
1.3 Defining high performance hardware-software interfaces	15
1.3.1 Literature survey	18
1.3.2 Overview of published work	19
1.3.3 Contributions	22
1.3.4 Limitations and further work	22
1.3.5 Conclusions	23
1.4 Conclusions	23
1.4.1 Limitations and further work	24

I	Creating high quality definitions of the hardware-software interface	25
2	Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture (Paper I)	29
3	End-to-End Verification of ARM Processors with ISA-Formal (Paper II)	39
4	Who guards the guards? Formal Validation of the ARM v8-M Architecture Specification (Paper III)	59
II	Creating high performance hardware-software interfaces	85
5	SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip (Paper IV)	89
6	Reducing inter-task latency in a multiprocessor system (Patent I)	101
	Appendices	131
	Bibliography	131

List of Figures

1.1	Virtuous cycle created by multiple users sharing a common specification	4
1.2	Overview of specifications, tools, verification IP and testing. Adapted from Figure 1 of Paper I with addition of more recent flows. The contents of the dashed box is the result of a collaboration with Cambridge University on their Sail ISA specification language and tools [8]. A machine-readable version of the Architecture Specification has been publicly released by Arm Limited on the Arm website [5] and the translation of that specification to Sail has been publicly released by the REMS group at Cambridge University [104].	5
1.3	The architecture of a communication-processing subsystem. This system consists of one RISC processor, five processing elements and six DMA engines (highlighted in grey), five private memories and one shared memory. Reproduced from Paper IV.	17
1.4	Illustration of the effect of optimization of task invocations. The control processor (CP) and each data engine (DE0, P, Q and R) are on the vertical axis and time is on the horizontal axis. These figures are reproduced from Figures 5a, 5c and 7 in Patent I.	20

Acknowledgements

This thesis was written in the middle of my research career and I have benefited enormously from people at different institutions along the way. At the University of Glasgow, Muffy Calder gave me a solid grounding in the techniques of research. At Yale University, Paul Hudak and John Peterson taught me the importance of publishing and of applying my research to other fields. At the University of Utah, Jay Lepreau gave me the confidence to start forming my own research agenda while Matthew Flatt and John Regehr taught me how to find interesting problems and how to explain my work. At Arm, Kris Flautner had the insight that my work on stream computation and accelerators would form a critical part of the “Ardbeg” Software Defined Radio project; his talent for forming strong teams and of providing a clear vision, both internally and externally, has inspired a lot of how I try to work today. At the University of Cambridge, Peter Sewell has encouraged me to raise my game and to make my work more open. Finally, Simon Gay at the University of Glasgow helped me to navigate the process of submitting a PhD by published work.

I would like to thank my managers and coworkers at Arm Limited for their help and support. On the Ardbeg project, I would like to thank Mladen Wilder and Kate Kneebone for their support and their friendship during a project with many highs and lows. My managers Stephen Hill and Eric Van Hensbergen provided strong support during the early days of developing Architecture Explorer when others could not see the value of executable specifications. The processor teams persuaded me that the complexity of modern processors was beyond the capability of traditional test-based verification: we had to find a way to use formal verification. David Seal created the original pseudocode specification of the Arm architecture. If I had not had David’s specification as a starting point, I would probably not have started working on Arm specifications. Sadly, David left Arm before the Architecture Explorer project started so he never saw his specification run. While I was busy formalizing the Arm specification, David Gilday was finding ways to use formal verification tools and techniques to verify processors: David was a constant source of enthusiasm, encouragement and ideas. I also benefited enormously from working with Will Keen, Anastasios Deligiannis, Lewis Russell, Erin Shepherd and the Teal team while developing ISA-Formal — it was an honour to work with the entire ISA-Formal team.

I would like to thank my fellow “Sailors” in the REMS group at the University of Cambridge and especially Peter Sewell, Alasdair Armstrong, and Kathy Gray. They have taught me many

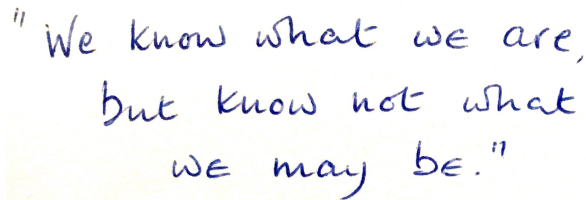
things but I especially value their commitment to creating formal specifications of major, real-world artifacts (e.g., ISAs, TCP/IP, etc.) and the many things I learned by reflecting on Sail's design and implementation choices.

Users are of tremendous importance in systems research. I would like to thank Danny Kershaw and the Nokia LTE Research group for their feedback on the SoC-C language. I would especially like to thank Tom Grocutt and François Botman at Arm for their enthusiastic adoption of the Architecture Explorer tool and their patience with its limitations.

I would like to thank my friends Nathan Chong and John Regehr who helped me with advice, suggestions and encouragement and gave feedback on early drafts of this thesis.

I wish that my dear friend John Peterson was here to mock me for engaging in this *foolish quest*. He was *probably* kidding about carrying his climbing pack and the threat of cannibalism if benighted near the summit; but I am glad that I now qualify for an exemption.

I will be forever grateful to my parents John and Anne Reid for their love and for helping me to get the education that has opened so many doors for me. I love you both very much and wish you were here to see what you helped me to achieve and how the words you wrote when I left Glasgow turned out.

A photograph of a handwritten quote in blue ink on a piece of lined paper. The text is written in a cursive, slightly slanted script. The quote is: "We know what we are, but know not what we may be." The paper has horizontal lines and a small yellowed area at the bottom left.

"We know what we are,
but know not what
we may be."

— William Shakespeare

Preface and Declaration

The portfolio of publications in this submission represents a selection of the applicant's academic papers and patents since 2004 while employed in the Research Department of Arm Limited. Arm has two primary businesses: the design of high-performance, low-energy microprocessors; and the creation of the Arm architecture specification that is licensed to more than 15 other companies who create compatible implementations of the same architecture specification. In the year 2017, 21.3 billion chips containing Arm compatible processors were manufactured [6] making Arm the largest processor architecture on the planet by a significant margin.

This thesis presents four peer-reviewed papers and one granted US patent. The papers are published in the high impact international journal PACMPL and in the proceedings of high impact international conferences CAV, FMCAD and CASES.

Alastair David Reid is the lead author and corresponding author on all four of the papers.

Alastair David Reid is the sole inventor of the patented invention. The patented invention was initially considered by Arm's internal Patent Review Committee to determine whether the invention was patentable and of sufficient value to Arm. As is customary, the patent filing was prepared by a patent attorney based on the inventor's description of the invention and detailed discussion with the inventor; the final filing was reviewed by the inventor; and the inventor worked with the patent attorney to respond to any objections raised by the patent examiner. Before being granted, the patent submission was examined by an examiner in the United States Patent and Trademark Office (USPTO) and found to meet the requirements of being novel, useful and non-obvious.

All work in this thesis was carried out by the author unless otherwise explicitly stated.

Submission

The work is presented in two parts that examine key aspects of defining the hardware-software interface.

Creating trustworthy definitions of the hardware-software interface

This part is concerned with the creation of trustworthy formal specifications of the Arm v8-A and v8-M processor architectures and consists of three papers.

[Paper I](#) “Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture” [98] describes the process of creating a specification with a particular focus on scope, applicability and trustworthiness.

[Paper II](#) “End-to-End Verification of ARM Processors with ISA-Formal” [101] describes a collaboration with formal verification engineers in Arm’s Processor Design Division to find a way to formally validate the pipeline of Arm processors against the formal specification.

[Paper III](#) “Who guards the guards? Formal Validation of the ARM v8-M Architecture Specification” [100] describes the use of formal verification tools to formally validate the formal specification described in Papers I and II.

I am the sole author of [Paper I](#) and [Paper III](#). [Paper II](#) is the result of a collaboration with a group of nine very talented and committed hardware verification engineers working in Arm’s processor division: Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Erin Shepherd, Peter Vrabel, and Ali Zaidi. I am the sole writer of the paper although the other authors provided useful feedback about whether the paper accurately described the hardware aspects of the project and they provided several suggestions for improving the writing. When I started working with these verification engineers, a manual version of the ISA-Formal method of using model checkers to detect a challenging class of bugs had already been developed and demonstrated on two of Arm’s micro-controllers. The challenge was that the method would require the creation and maintenance of a specification of Arm’s entire instruction set written in Verilog. My contribution to the project was the formal Arm specification that I had been creating; the development of a series of transformations that convert the executable specification into readable, synthesizable Verilog that would be accepted by the bounded model checker being used; and the initial version of the Arm Formal Verification Interface that processors must support to enable the ISA-Formal process. During the course of the project, I worked with the hardware engineers to design extensions to this verification interface such as support for verifying memory operations and to generalise it for use with other processors. The verification engineers worked on multiple processor projects and created verification code (not included in the final chips) that would extract information about each processor’s state, inputs and outputs — a task requiring considerable skill and ingenuity on their part. The verification engineers also had the task of running the model checker each week, sifting through failures, generating bug reports for designers to deal with, checking whether proposed bug fixes eliminated problems and setting up for the next week’s verification run.

Defining high performance hardware-software interfaces

This part is concerned with defining the hardware-software interface in a way that allows the software to exploit the potential performance of the underlying hardware.

This part consists of one paper and one patent.

[Paper IV](#) “SoC-C: efficient programming abstractions for heterogeneous multi-core systems on chip” [102] describes the design and implementation of a set of extensions to the C programming language that direct the mapping of that program to different parts of a heterogeneous multiprocessor system.

[Patent I](#) “Reducing inter-task latency in a multiprocessor system” [103] describes an extension of the ideas in [Paper IV](#) that is able to exploit a broad range of different kinds of task triggering/sequencing hardware to reduce the latency between tasks running on the different processors within the system.

I am the sole inventor of [Patent I](#). [Paper IV](#) has four authors: myself (Arm), Krisztián Flautner (Arm), Edmund Grimley-Evans (Arm) and Yuan Lin (University of Michigan). I am the sole writer of the paper although Krisztián provided some useful suggestions for improving the opening section. Krisztián Flautner lead the Ardbeg project and identified that the transformation I had developed was related to the decoupling transform used with decoupled access/execute computer architectures [112]. Edmund Grimley-Evans joined the SoC-C team once most of the compiler was in place and was responsible for extending and maintaining the compiler. Yuan Lin was working on the sister project “SODA” [120] at the University of Michigan under the supervision of Professor Trevor Mudge. Yuan identified the challenge of performance portable programming but we proposed completely different approaches to programming the system: Yuan Lin proposed the SPEX language [74] that is based on stream programming and I proposed the SoC-C language described in [Paper IV](#) that is based on imperative programming. We each listed the other as an author on our papers about our different languages to acknowledge the role of the other in forming our ideas.

Chapter 1

Defining interfaces between hardware and software

1.1 Introduction

Computer systems consist of large stacks of different kinds of technology: from chemical and lithographic processes used to manufacture silicon chips at the bottom of the stack; through pipelining and out-of-order execution in microprocessors; through programming languages and operating systems; all the way up to applications and user-interfaces at the top of the stack. Between each of these layers is an interface that abstracts away some of the details of the layer below to insulate the layer above from changes in the layer below.

This thesis is concerned with one of the most fundamental of these interfaces: the interface between hardware and software. This interface is important because it is used by many different groups: processor designers; tool creators writing assemblers, compilers, JITs and debuggers; operating system writers; creators of processor simulators; verification engineers creating test suites for processors and simulators; library writers; etc. Given its importance, it is essential that this interface is well defined and that it is designed in a way that enables efficient implementation of the interface.

This thesis explores two key aspects of this interface that are of critical importance to industry:

Creating high quality definitions of the hardware-software interface The first and primary aspect is creating high quality specifications of microprocessors that are complete enough and accurate enough that they can be used for formal verification. Section 1.2 describes the development of formal, executable specifications of the Arm v8-A and v8-M processor architectures; steps taken to test the specifications; use of the specifications to formally validate commercial Arm processors against the specification; and formal validation of the specifications.

This specification is now part of Arm’s official specification and it has been released publicly in Arm’s Architecture Reference Manuals [3,4] and in machine readable form [5,99]. As far as we are aware, the resulting specification is the most complete and most trustworthy specification of any mainstream processor architecture. The formal specification and the validation methodology we developed led to a step change in Arm’s ability to formally validate the processors they design. The methodology has been deployed on five commercial processors developed by Arm and is being rolled out onto the next generation of Arm processors and to Arm’s other design centres in France and the USA.

Defining high performance hardware-software interfaces The second aspect of the hardware software interface considered in this thesis is defining an interface that allows software to exploit the potential performance of the underlying hardware. Section 1.3 describes the use of programming language extensions and compiler technology that allow complex parallel hardware to be programmed in a simple, portable way without sacrificing performance.

We demonstrate how raising the boundary between software and hardware can expose the potential performance of the hardware to the programmer while allowing the hardware designer to use a broad variety of techniques to provide an efficient, high-performance system.

The SoC-C language and compiler were significant components of Arm’s Ardbeg project: a project that developed hardware, software and tools for creating software defined radios [120].

1.1.1 Organisation

The remainder of this thesis is structured as follows: Sections 1.2 and 1.3 describe each aspect of defining the hardware-software interface: the related work, contributions, limitations and potential future work. Section 1.4 concludes and considers limitations that apply to the entire body of work. Parts I and Part II contain the published work consisting of four peer-reviewed papers and one granted US patent.

1.2 Creating high quality definitions of hardware-software interfaces

The main aspect of defining the hardware-software interface considered in this thesis is the creation of trustworthy formal specifications of processor architectures. The three publications exploring this aspect develop, extend and apply techniques to create, test, use and formally validate formal specifications of complex, real-world artefacts. Besides these technical contributions, the

papers also describe and form part of the social process of building trust and consensus around the specification.

Three key properties of a processor specification are its *scope*, its *applicability* and its *trustworthiness*.

The *scope* of a specification is the set of features that one can reason about. For example, a certified compiler such as CompCert [72] only requires a specification of those instructions that the compiler could generate. But in order to reason about arbitrary user-mode binaries, one would need a specification of the entire instruction set. And to reason about Operating System code, the scope of the specification is dramatically increased and includes a specification of instructions for changing execution mode (e.g., entering/leaving supervisor mode), interrupt handling mechanisms, page faults, mechanisms for changing memory protection, etc. Prior to our work, all formal specifications of the Arm architecture were targeted at reasoning about the behaviour of instructions with no support for system-level features such as memory protection or taking an exception in response to a page fault.

The *applicability* of a processor specification is whether the specification applies to the target processor. Most changes to architecture specifications are backward compatible extensions and so most proofs about code for one architecture version are valid when executing that code on a processor implementing a later architecture version. But architecture revisions also remove instructions, add restrictions or change functionality so proofs based on the ARMv6 specification (1996) or the ARMv7-A specification (2007) are not necessarily sound for ARMv8-A (2013). This is especially true for Arm’s micro-controller architecture that has a completely different exception model from Arm’s mainstream architecture.

The *trustworthiness* of a processor specification is whether the specification can be trusted to reflect the behaviour of all processors implementing the specification. The Arm HOL specification of Fox and Myreen [46] is noteworthy for the degree of testing performed: systematically testing all user-mode, integer instructions against three actual processors. This is a critical step and must be repeated against as many expressions of the architecture as possible (processors, implementations, test suites, etc.) and must go beyond simply testing user-mode instructions and test the full scope of the specification: floating point and vector instructions, exceptions, interrupts, privilege checks, virtual memory, etc.

The effort required to create a specification increases with the desired scope, applicability and trustworthiness of the specification. Worse, since Arm regularly releases extensions and corrections to the architecture, the challenge of retaining applicability to current processors is more of a continuous process than a one-off sprint. Our solution to this problem has been to change Arm’s existing architecture specification process so that machine-readable, executable specifications can be automatically generated from the same materials used to generate conventional documentation. These changes required not just technical solutions but also the development of a process for agreeing on and building trust in the specifications.

A key aspect of this process was the decision to create a single specification that is shared between multiple groups. We found that doing this creates a *virtuous cycle* where each time one group finds and fixes a bug in the specification, it improves the utility of the specification for other groups and, over time, enables more demanding groups to make use of the specification (Figure 1.1). For example, the “ISA-Formal” team (Paper II) required that each instruction specification that they used be correct and required clarifications about the intended priority of different forms of underspecification while those creating testsuites required that the exception model be correct since many architectural tests revolve around checking that the correct exception is generated at the right time and is accompanied by the correct exception syndrome information. In addition, the sharing of a specification created an understanding across the different user groups of which parts of the specification were trustworthy and which required further work before they could be trusted.

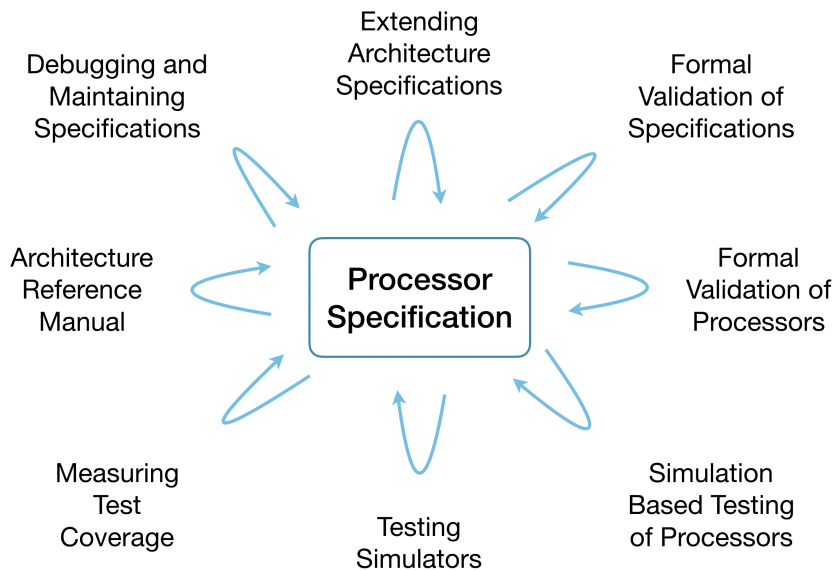


Figure 1.1: Virtuous cycle created by multiple users sharing a common specification

Supporting multiple groups is also essential to build the business case for the considerable cost of creating, debugging and maintaining a large specification. Figure 1.2 provides an overview of the variety of different uses that our specification currently supports.

These benefits are not without a cost: the specification language is a compromise between all the different user groups. Some groups would find it more convenient if the specification was written in some other language and they require tools to translate the specification into the form their tools require but the advantages of having a single specification outweigh the costs.

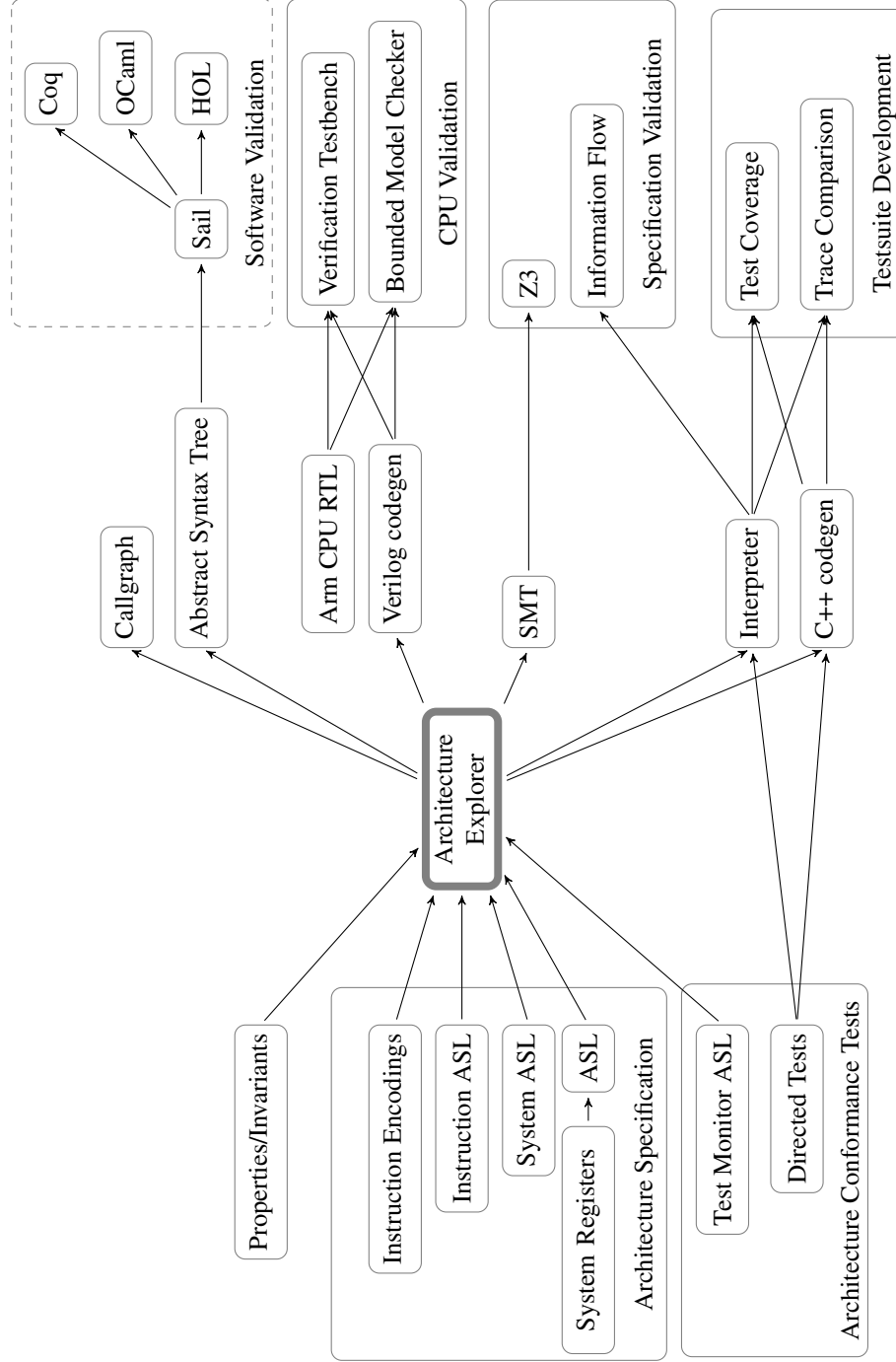


Figure 1.2: Overview of specifications, tools, verification IP and testing. Adapted from Figure 1 of [Paper I](#) with addition of more recent flows. The contents of the dashed box is the result of a collaboration with Cambridge University on their Sail ISA specification language and tools [8]. A machine-readable version of the Architecture Specification has been publicly released by Arm Limited on the Arm website [5] and the translation of that specification to Sail has been publicly released by the REMS group at Cambridge University [104].

1.2.1 Literature survey

Some of the earliest formal specifications of computer architecture were Falkoff et al.’s use of APL to describe the IBM System/360 [37] and Bell and Newell’s “Instruction Set Processor” (ISP) notation [14] that was used to write specifications for 14 systems including the PDP-8, PDP-11 and CDC 6600 and that later gave rise to Barbacci’s machine readable “Instruction Set Processor Semantics” (ISPS) [11] that targets compiler-related uses. ISP followed in the Algol language tradition and is similar to the less formal pseudocode notations typically used in ISA definitions in the present day. ISP was used during design of the PDP-11 and included in the manufacturer’s processor handbook [28]. Some of the earliest uses of formal semantics were for automatic reasoning about programs such as Samet’s development of Translation Validation [106, 107] (later reinvented and refined by Pnuelli [94] and Necula [87]) and automatic derivation of compiler code generators by Fraser [48] (based on ISP specifications of the IBM-360 and PDP-10) and Cattell [24] (based on ISPS specifications of the IBM-360, PDP-8, PDP-10, PDP-11, Intel 8080, and Motorola 6800).

Since those early days, many other uses of formal specifications have been found including verifying compilers [70, 72, 86]; verifying assembly language functions against a specification [80]; verifying operating systems [31, 53, 68, 110]; discovery, verification and synthesis of peephole optimisations [9, 66, 78]; automatic generation of binary translations between architectures [10]; verifying processor pipelines [17, 42, 64, 101]; automatic generation of test cases [51, 79]; decompilation of binaries [85, 89]; and abstract interpretation of binaries [95, 96].

Creating a specification that can support this broad range of potential uses affects how the specification is written. If a formal specification is being used to mechanically verify hardware or software in a theorem prover, it is common to write the specification using the language of the theorem prover using either a deep or shallow embedding [21]. If a shallow embedding is used, it is also necessary to overcome mismatches between the semantic features of the specification using the theorem prover’s language. For example Goel et al. [54, 55] use abstract stobjs to encode state and constrained functions to encode undefined behaviour, Fox and Myreen [46] use monads to encode state and exceptions.

A significant limitation of embedding a specification in a theorem prover is that it limits reuse of the specification: it is hard to use the same specification with other theorem provers or for non-proof purposes. This is a significant problem because it fragments efforts at creating specifications between the user communities of the major interactive theorem provers (e.g., ACL2 [67], Coq [27], HOL [57], Isabelle-HOL [88]), model checkers (e.g., JasperGold® [23] and SymbiYosys [121]) and SMT solvers (e.g., Z3 [33] and CVC4 [13]). Embedding a specification in a theorem prover also has the problem of impacting readability: the specification is only really usable to those familiar with a particular theorem prover. This almost certainly necessitates the creation of multiple specifications: one for each tool one wishes to use plus another for humans to read. Having multiple separate specifications leads to a further problem of

trust: how can one trust that the different versions of the specification are consistent with each other? To address these limitations, we follow the common approach of writing the specification in an external Domain Specific Language (DSL) specifically designed for writing processor specifications [11, 25, 38, 45, 124] and mechanically translate the specification into the languages of different formal verification tools. In Mishra and Dutt’s taxonomy [83], the DSL described in this thesis would be classified as a “Behavioural Architecture Description Language.” That is, it describes the externally visible behaviour of a processor without necessarily reflecting the hardware structures that might implement it. In contrast, a “Structural Architecture Description Language” closely reflects the hardware structure they describe and are primarily used to generate and verify hardware designs. Structural specifications reveal implementations details making them suited to specifying microarchitecture while behavioural specifications abstract away inessential differences between different implementations making them best suited to specifying processor architecture.

A major challenge tackled in the papers in this section is how to create a trustworthy formal specification of the *entire* architecture that includes all instructions, address translation, memory protection, interrupts, exceptions, etc. Most processor specifications in existence are for simpler architectures such as the IBM/360 [37] or are for a subset of the architecture such as the instruction set [40, 46] or are only for outdated (and smaller) architectures such as the ARMv6 architecture [111].

Fonseca et al.’s empirical study of the correctness of formally verified systems found bugs in specifications [41]. One way of establishing trust in a specification is by testing specifications against existing implementations [8, 40, 46, 56, 98, 111]. The quality of testing depends on the accuracy of the test oracle [12] and on the completeness of the tests used. Formal verification of a processor against a specification [42, 64, 101] has the desirable side-effect of detecting bugs in the specification and ensuring compatibility.

Creating a fully verified stack of hardware and software eliminates the need to trust the processor specification. Notable steps in this direction are: the “CLI stack” of Computational Logic Inc. [16] that consisted of FM8502 processor, the Piton assembler and a code generator for the micro-Gypsy language (a Pascal derivative); and the Verisoft project that produced the VAMP out-of-order processor implementing the DLX ISA [17] and PikeOS hypervisor [109]. More recently, the CakeML project succeeded in creating a formally verified processor “Silver” [77] and a formally verified compiler for a dialect of ML [70] that can compile itself and so the compiler can both generate code for Silver and the compiler can run on Silver itself.

There are two main classes of formal ISA specifications: specifications of the system-level features such as address translation or taking an exception and specifications of the instruction set.

System-Level Specifications

A major milestone for an executable processor specification is demonstrating that it can boot an operating system up to the first command prompt: demonstrating the ability to take interrupts, implement virtual memory, etc. Fox’s MIPS specification [45] written in L3 has been shown to boot FreeBSD. Fox’s Arm specifications [45] contains some system features but lack memory protection and address translation or the ability to take an exception in response to a failed permission check. Our specification is more complete (the v8-A architecture specification includes all 1280 instructions, all four privilege levels, both secure and non-secure modes, address translation, reset, interrupts and exceptions) and has been more thoroughly tested (we test all four levels and both security modes and we test that privilege checks deny access when appropriate). Armstrong et al. [8] develop formal specifications of Arm v8-A, RISC-V, MIPS and CHERI-MIPS written in Sail that have been shown to boot Linux, seL4 or FreeBSD. The Arm v8-A model is a translation of our ASL specification implying that our specification can also boot Linux.¹

Goel’s x86-64 specification [56] is embedded in ACL2 and includes key parts of the system architecture including paging, segmentation and both user/supervisor levels. The specification has been used to verify both user-mode code and system-mode code: a “zero-copy” program that duplicates memory by manipulating the page table to create aliases in the virtual address space. Instead of embedding the specification inside any particular theorem prover (such as ACL2), our specification is written in a simple, imperative specification language and different backends translate it into the languages required by different users: C++, Verilog, SMT2 and Sail [8, 59]. We believe that using a DSL and actively pursuing and supporting different user groups is critical to creating a trustworthy specification: each different use stresses the specifications in different ways and finds different classes of bugs in the specification.

The Verisoft-XT project developed a substantial specification of x86 processors encompassing the concurrency model, 140 general-purpose and system programming instructions and security mechanisms such as memory protection and hypervisor support [34]. Their goal was to enable the formal verification of low level software (hypervisors). Like our work, the Verisoft-XT specification is written in a domain specific language in order to keep the specification readable and compact. The specification differs in having a different *scope*: it adds a concurrency model and parts of the platform architecture such as the APIC interrupt controller but it supports only a fraction of the full x86 instruction set. The more significant difference lies in *trustworthiness*. While Degenbaev [34, Chapter 19] describes techniques that they could potentially apply to test their specification; we developed and applied multiple approaches including testing using Arm’s architecture conformance suite, formal validation of processors against the specification and formal validation of the specification itself. We consider this investment in trustworthiness to be critical when creating a specification that will be used for formal verification purposes.

¹This has since been confirmed within Arm with assistance from Armstrong.

We are not aware of any x86 processor specification that can boot an operating system. Running an operating system on an x86 processor specification would require not only the x86-64 features needed to run an OS but, since x86 processors power up in a legacy mode, one would also require support for legacy modes to support the boot process.

Instruction Set Specifications

Over a period of more than 18 years, Fox and others at the University of Cambridge have developed and used specifications of multiple versions of the Arm processor architecture starting with an ARMv3 specification written in HOL [47] that was used to formally verify the ARM6 processor microarchitecture [42, 43], later updated to support the ARMv4–ARMv7 architecture versions [46] and tested against actual hardware. This specification was then converted to a domain specific language L3 [44, 45] that can be translated to HOL and a specification of the integer AArch64 instructions from the ARMv8 architecture was added. For a long time, this was the most complete and highest quality formal specification of the Arm architecture and this series of specifications has been used in the formal verification of microkernels [110], hypervisors [31] and compilers [70, 86]. Our specification is more complete (the v8-A architecture specification includes both AArch32 and AArch64 modes and all instructions); has been more thoroughly tested; and, like the original Fox specification, has been further validated through its use in the formal validation of Arm processors.

There are a number of notable specifications of the x86 instruction set. RockSalt [84] is a formally verified implementation of Software Fault Isolation that relies on the ability to perform an analysis of (a subset of) x86 machine code. A key part of RockSalt is an x86 instruction specification that covers 130 instruction encodings with semantics for 70 instructions. This specification is written in an embedded DSL in Coq and has been validated by extensive testing against executions of (compiled) randomly generated C programs and constrained random sequences of the implemented instructions. An important difference from our work is the reduced *scope*: RockSalt only describes a small subset of the x86 instruction set and does not describe system architecture features such as address translation.

Goel’s x86-64 specification [56] embedded in ACL2 includes 413 instructions (as well as the system architecture features mentioned above) and has been verified against real processors using the Pin binary instrumentation tool. Both the ACL2 and RockSalt specifications are embedding within a theorem prover enabling them to formally verify their reasoning about the specification including reasoning about any transformations/analysis they perform on the specification. This is a powerful capability that our Arm specifications lack.

Heule [62] used synthesis techniques to generate specifications of 1,795 variants out of 3,684 instruction variants in the x86-64 Haswell ISA. This specification formed the basis for two recent specifications that describe the complete x86-64 user-space instruction set: Roessle et al.’s Chum specification [105] to support decompilation of binaries using HOL; and Dasgupta

and Adve’s specification written using the K-framework [32]. Both specifications stand out for the completeness of their support of the instruction set and the degree of testing performed on their specifications.

Recently, Huang et al. have generalized the notion of Instruction Set Architecture (ISA) to an “Instruction-Level Abstraction” (ILA) [63] that extends the familiar instruction model found in programmable processors to semi-programmable accelerators. They have demonstrated that this framework can be used to formally specify and verify both accelerators (for image processing, machine learning and cryptography) and general-purpose processors (RISC-V). Zhang et al. have further extended the ILA framework with a memory consistency model and associated verification tools [123].

An, apparently promising, alternative approach to creating a processor specification is to generate it automatically based on experiments run on particular processors [52, 62]. This approach can lead to overfitting: Godefroid identified a number of instructions whose functional behaviour differed across the range of processors used in their experiments [52].

1.2.2 Overview of published work

The specifications described in this thesis are written using a Domain Specific Language called the Arm Architecture Specification Language (ASL).² ASL is an executable, strongly-typed, imperative language with support for dependent types and for throwing and catching exceptions. For the first 20–25 years of Arm’s history, specifications were created *after* the corresponding implementation as documentation of what had been built. This pattern has changed as a result of the work described in this thesis and specifications are now written and tested before implementation starts.

The ASL language was created by reverse engineering a specification language from the pseudocode notation used in Arm’s existing documentation, fixing the pseudocode in the documentation and evolving that pseudocode into a formal specification. This choice of *evolution*, *not revolution* enabled a smooth transition from the official, informal pseudocode to a formal specification with trust in the specification gradually increasing as new user groups successfully used the specification. An earlier, more revolutionary, attempt to replace Arm’s architecture specification with a more formal specification had failed to gain traction and had been abandoned. We believe that a major factor was that replacing the specification required buy in from multiple groups at once whereas the slower, incremental process of submitting bugfixes and focussing on providing value to one group at a time deferred the need for Arm architects to make a decision until after support had been built across the company. An early example of building cross-company support is that a member of the validation team that writes architectural testsuites asked if it was possible to measure coverage of their tests against the architecture. Measuring coverage was not a high priority for me at the time but it helped people who had helped me, it

²There is no connection to the ASL specification language that was the subject of the author’s M.Sc. thesis [97].

took little effort to implement and it led both to the validation team championing the work and to practical help from the validation team. Thereafter, I sought to repeat this experience with other groups in order to establish the virtuous cycle of Figure 1.1.

This section presents three papers that tackle different aspects of creating, using and checking this specification.

Paper I “Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture” [98] describes the first 5 years of the project to change Arm’s existing architecture specification process so that machine-readable, executable specifications can be automatically generated from the same materials used to generate Arm’s conventional architecture documentation.

The focus of this paper is on the different methods (summarised in Figure 1.2) used to improve our trust in the specification and the various uses of the specification.

A key contribution of this work was a scalable, sustainable methodology for creating high quality formal specifications. To achieve the broad scope required, we developed methods for defining and testing the system architecture by extending the specification with a programmable monitor and stimulus generator. To the best of our knowledge, such testing has not previously been applied to architecture specifications. A major output of this work was the creation of formal specifications of Arm’s v8-A and v8-M processor architectures — a critical prerequisite for the formal verification of Arm processors.³ The formal specification now forms part of Arm’s official architecture specification and has been publicly released in machine readable form [5].

Paper II “End-to-End Verification of ARM Processors with ISA-Formal” [101] describes a collaboration with formal verification engineers in Arm’s Processor Design Division to develop a repeatable method of formally validating the instruction pipelines of Arm processors against the Arm v8-A and v8-M specifications described in **Paper I**.

The ISA-Formal method described in the paper uses a bounded model checker that uses symbolic techniques to validate sequences of instructions up to some bound. The approach was tested on a small scale on three processors using hand-written System Verilog assertions that implemented partial specifications of a few dozen instructions. To turn the approach from a promising idea into a viable verification flow that could be broadly applied, two problems had to be overcome:

Scaling The Arm v8-A specification has over 1,280 instruction encodings; the Arm v8-M specification has over 380 instruction encodings. Each instruction has many potential effects: writing registers, writing flags, accessing memory, raising exceptions, etc. Manually writing properties to check each instruction for each possible effect would be a monumental task and the result would inevitably contain bugs and divergences from Arm’s official specification. The solution was to find a way of automatically translating the specification from ASL to a language that commercial model checkers would accept (Verilog).

³Adaptations to the formal v8-A specification to include the v8-R architecture are underway.

Defining a standard verification interface Arm designs large numbers of processors to target different niches with radically different pipeline structures. Defining a standard verification interface was critical to creating a verification flow that can be applied across the entire range. This was created by refactoring the manual interface used with the System Verilog assertions into a higher level interface that the machine-generated Verilog could interface with.

Paper III “Who guards the guards? Formal Validation of the ARM v8-M Architecture Specification” [100] tackles a fundamental problem with executable architecture specifications: they contain little or no redundancy. This lack of redundancy is especially problematic when extending the architecture because existing test suites do not cover any new extensions so testing is not able to detect when an extension breaks cross-cutting properties involving security, the ability to return from exceptions, etc. The paper describes the creation of a meta-specification of the architecture and tools to verify that the architecture specification satisfies the meta-specification. Despite the high level of testing already performed on the specification, this found a further 12 confirmed bugs.

1.2.3 Contributions

The contributions of the published work in this section consist of both technical developments and a significant impact on the design and verification practices of the world’s largest processor designer that affects both the creation of processor architecture specifications and the verification of processors.

- I created a methodology to develop high quality processor architecture specifications with a particular focus on features of the system architecture (e.g., page table walks, memory protection, privilege checks and system registers) [98, 100]. This methodology involves reverse-engineering a language “hidden within” the pseudocode previously used, filling gaps in the specification, building tool chains to read and execute the specification, building monitors and stimulus generators to support testing of system architecture, testing the specification, measuring the coverage of processor test suites, automatically translating the specification to Verilog and to the SMT-Lib2 format and formal validation of the specification. In addition to these technical developments, I developed a process to build consensus around the specification and to create a virtuous cycle where multiple groups share a common specification such that bug reports and fixes from one group benefit all other groups.

I applied the methodology to both the Arm A-class and M-class specifications and the methodology has been adopted by the teams responsible for developing new architecture extensions including TrustZone for M-class (TZM) [4] that added new privilege levels to the micro-controller architecture; and the Scalable Vector Extension (SVE) [113] that

added 488 new instructions to the v8-A architecture. The executable specification is used within Arm as a “golden reference”. The A-class specification has been publicly released in machine-readable form under a liberal license.

- I created a highly effective, repeatable, formal validation methodology [101] that could be applied in an industrial setting to formally validate processors against the formal specification of the architecture. The methodology was applied to five commercial processors (Arm’s Cortex A32, A35, A55, R52 and M33). The methodology has proved to be very effective at finding complex bugs that other techniques might have missed and also at finding simple bugs earlier in development than traditional test-based validation techniques. This led to a step change in Arm’s use of formal verification techniques on its processors. The methodology is being rolled out onto the next generation of processor development projects. Beyond Arm, the ISA-Formal validation approach has already been adapted for use on RISC-V processors in the “riscv-formal” flow [122] and a company “Symbiotic EDA” has been created to provide formal validation services to RISC-V implementers.
- There have been two significant public releases of Arm’s specification. In 2017, Arm publicly released around 90% of the v8.2-A specification and subsequent public releases included the v8.3, v8.4 and v8.5 extensions [5]. In addition, Arm has released the entire v8.5-A specification to the REMS group at Cambridge University. With my assistance, they have created tools to (mostly) automatically translate the specification from ASL to their “Sail” ISA specification language and they have created translators to convert Sail to C, Isabelle and Coq [7, 8, 59]. The process of performing this translation lead to several Sail language extensions that were necessary to handle the size and complexity of a commercial processor architecture. The C code generated from their tools has been tested using Arm’s internal architecture conformance test suite; is able to boot the Linux kernel; and the translated code has been used to prove properties about the virtual memory system using Isabelle. Cambridge University has publicly released both the translated specification and their tools [104]. We hope that this will lead to many new uses of the specification. (This work is described in a recent POPL paper [8] that is not included in this thesis.)

To our knowledge, no other major processor company has these capabilities or has chosen to release a formal specification of its architecture.

1.2.4 Limitations and further work

This work has transformed the use of formal specification and verification within Arm and has created a single reference that those outside Arm can freely use.

Future work that is already underway includes: extending the range of uses by, for example, using the specification as a reference for traditional test-based verification of processors; formally specifying the ASL language used in the specification; continuing the work reported in [Paper III](#) to check more of the v8-M architecture and to start checking the v8-A architecture.

Future work that is still in the proposal stages includes improving the performance of simulators perhaps using Pydgin [76].

The most significant limitations of this work are associated with underspecification: where the specification allows for a range of allowed behaviours. Forms of underspecification found in Arm specifications includes implementation defined behaviour; unknown values⁴; unpredictable behaviour; and non-determinism in exception raising. This is the subject of ongoing discussion with Arm architects. The main challenge in clarifying underspecification is in creating a definition that is simultaneously useful to programmers; applicable to existing implementations; and provides the desired degree of implementation freedom to hardware designers. For example, the notion of unknown values is used both to describe the non-determinism of register values when hardware is powered up and also to describe situations where different microarchitectural states can lead to different values. Teasing apart these different notions of “unknown” apart would enable each notion to have a narrower, more useful definition.

The interaction of underspecification with security is especially important. Arm’s Architecture Reference Manual [3] says “An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege ...” but such statements are hard to test. In practice, individual processor implementations tend to choose a deterministic value such as a zero value but that does not help those verifying the security of software that must run on any implementation.

Some other limitations of the specifications in this work include: they do not capture the semantics of weak memory [1, 40, 108]; they do not specify the assembly language syntax; limited testing of multiprocessor, debug and performance monitoring features; they lack specifications for interrupt controllers, I/O MMUs, Arm’s “TrustZone” support in interconnect, and other features needed to reason about low-level system software.

Architecture specifications have traditionally limited themselves to the functional behaviour of processors. This traditional view excludes timing information and speculation induced effects since details such as the execution time of an instruction is viewed as a detail of a specific implementation, not an architectural requirement. The growing awareness of microarchitectural side-channel attacks [50] has lead to the notion of “data independent timing” in Arm v8.4-A where the execution time of certain instructions is independent of the data being processed by the instruction. The discovery of speculative side-channel attacks such as “Meltdown” [75], and “Spectre” [69] lead to the further addition of speculation barrier instructions that control speculative memory accesses. A critical next step will be finding a way to formally specify

⁴Other ISA specifications use the term “undefined values” for what Arm calls “unknown” values.

the semantics of these extensions (and of existing features such as flushing caches, branch predictors, etc.) in order to support reasoning about side channels in software. Cock et al. [26] measured the impact of mitigations such as deterministic scheduling, scheduled delivery, cache colouring and TLB flushing on the channel capacity of known side channels. In addition to measuring and mitigating high-level microarchitecture details, they found new channels caused by the low-level implementation details such as an impact of branch mispredictions on the cycle counter reinforcing the need for thorough verification of processor implementations against such a semantics. McIlroy et al. [81] is an early step towards such a semantics but their emphasis on determinism limits them to describing just one processor instead of the full envelope of behaviour allowed by the architecture.

Finally, when we started this work, Arm relied on the fact that the definition of the architecture was, in effect, spread across several artefacts within Arm: the Architecture Reference Manual, a “golden simulator,” and a test suite. Our goal is to turn the formal specification into a “perfect” specification against which these other artefacts are measured. This would automatically guarantee consistency but, by reducing redundancy, it brings the risk of creating a single point of failure. Our efforts in creating a meta-specification in [Paper III](#) and in recruiting many different user groups are intended to reduce this risk.

1.2.5 Conclusions

This section is concerned with creating complete specifications of commercially important processor architectures and in establishing trust in those specifications. The central idea is of creating a single specification that is used by many different teams in order to create a *virtuous cycle*. This approach has proved effective in transforming the approach of the largest designer of microprocessors in the world to formal specification and verification and has resulted in the creation of a formal specification of unparalleled scope, applicability and trustworthiness that can be used both inside and outside the company.

1.3 Defining high performance hardware-software interfaces

The previous section focuses on functional aspects of the interface but does not mention non-functional aspects such as the timing or energy requirements of an instruction or the silicon area required to implement an instruction.

In general purpose microprocessors, portability between different hardware designs is usually achieved by adding hardware such as pipeline interlocks to hide pipelining hazards and register renaming to hide out of order execution from the programmer. This portability comes at the cost of extra hardware to give the illusion of sequential execution and a significant fraction of the potential parallelism of the hardware is lost. One way to exploit the potential performance

is to expose all the hardware features to the programmer but that decreases portability because programs will only work on systems providing the same features. Some Very Large Instruction Word (VLIW) processors [39] tackle this problem by requiring programmers to program them in a high level language such as LISP or C instead of using assembly level: tackling performance portability with programming language extensions and compiler technology. This section applies a similar approach to the problem of programming asymmetric multiprocessing (AMP) systems designed to handle Digital Signal Processing (DSP) pipelines.

The physical layer of the cell radio interface that lies at the heart of modern cellphones is traditionally implemented using a combination of hardware blocks that provide little or no configurability or programmability and are therefore limited to implementing a small number of protocols and require modifications to the chip if a bug is found or the protocol specification is changed. Software Defined Radio (SDR) aims to disrupt this model by replacing most of the fixed function blocks with programmable processors. SDR applications can have extremely high processing requirements (in excess of 10Gops per second) and, to compete with conventional fixed-function hardware designs, they must operate on tight power budgets (a fraction of a Watt). Achieving such extreme performance on such a small power budget requires that the hardware platform exploits numerous techniques to save energy and boost performance: processors provide a high degree of data and instruction level parallelism; each processor is provided with private data and instruction memories to which they have faster, more efficient access; DMA engines are provided to copy data from one processor's private memory to another's; some fixed function accelerators are provided for functionality that is required by a large number of different protocols (e.g., an accelerator for error correction such as a Turbo decoder); and a simple RISC control processor is charged with loading programs into each processor's instruction memories, with sequencing different parts of the algorithm across the platform and with handling small, low performance parts of the protocol. Figure 1.3 illustrates a communication-processing subsystem that might be used in a *Software Defined Radio* (SDR) system.

The approach explored in this section is to support porting software from a conventional computer (e.g., a desktop computer) to such specialised architectures by adding a small number of annotations to the program. These annotations guide the compiler to adapt the program to the particular compute system. The goal of this work was not to completely automate the mapping of software to such hardware but, rather, to provide the programmer with an interface that allows them to use their insight into the trade-offs to unlock extremely high levels of performance by making small changes in the annotations.

The language and compiler were developed as part of Arm's "Ardbeg" research project to develop a Software Defined Radio (SDR) platform. This project developed five co-designed components: a DSP engine based on a 512-bit SIMD vector unit that could sustain over 10 billion multiplies per second at less than 300mW in 65nm technology [120]; a programming model and tools for the DSP engine; a system architecture combining multiple DSP engines,

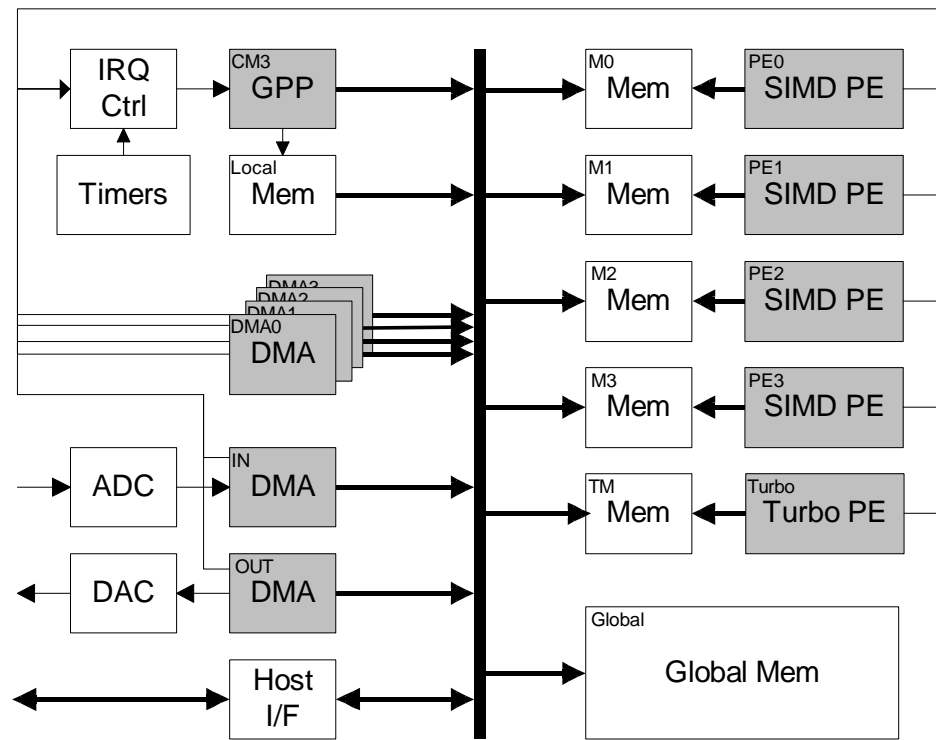


Figure 1.3: The architecture of a communication-processing subsystem. This system consists of one RISC processor, five processing elements and six DMA engines (highlighted in grey), five private memories and one shared memory. Reproduced from [Paper IV](#).

accelerators, DMA engines, private memories and a control processor (see Figure 1.3); a programming model and tools for the system architecture; and software to test and demonstrate the capabilities of these components including implementations of the physical layer processing of 4G cellphones, 802.11a WiFi and DVB digital video broadcast. This section is concerned with the system programming model and tools.

1.3.1 Literature survey

Programming DSP and SDR systems is conventionally performed in a very low-level way: code uses numerous techniques such as double-buffering, DMA-transfers, interrupt handlers, etc. to make efficient use of the available hardware and achieve maximal parallelism. Programming in this fashion results in software whose structure is determined by the hardware platform and by the decision of which tasks to map to each part of the system. This need to match software structure to hardware structure means that porting the application to different hardware or even experimenting with a different mapping is a significant and error prone effort.

Our solution to these problems is to draw on a number of techniques familiar in the programming language, computer architecture and systems communities:

- The (synchronous) remote procedure call (RPC) [19] model simplifies triggering code execution on remote processors by making it look like a function call. Greater parallelism can be achieved by sacrificing some of this simplicity and using asynchronous RPCs [2].
- Software distributed shared memory [15, 65, 73] simplifies a potentially complex memory topology to look like a single shared address space by introducing data copies at the right places to implement cache coherency.
- Providing efficient compiler support for domain specific languages and library-specific optimisations by annotating libraries with information to guide optimisation of applications using the library [61, 93, 118].
- “Decoupling” transformations to introduce pipeline parallelism into sequential programs by splitting programs into independent threads communicating via FIFO queues [22, 30, 35, 90, 92, 112, 114, 114, 116]. The techniques used range from entirely manual, through requiring some annotation to fully automatic.
- Double-buffering and other zero-copy interfaces efficiently implement FIFO queues by queuing buffer pointers instead of copying data [117].
- There has been a long-running dispute over the relative merits of thread-based and event-based systems [29, 71, 91, 119] based on their impact on programming, ease of program analysis, and efficiency. Decoupling transformations introduces threads but, on resource-constrained systems, event-based systems have compelling performance advantages [49].

Cooperatively threaded code can be transformed to event-based code using variations on Duff’s device [36, 115] or equivalent compiler transforms.

- Type inference mechanisms such as that found in the Hindley-Milner type system [82] provides a way of inferring annotations without sacrificing significant expressiveness.

1.3.2 Overview of published work

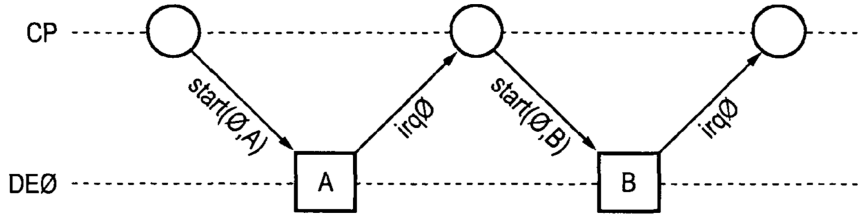
[Paper IV](#) draws on all these influences to provide a way of mapping software defined radio applications onto heterogeneous multiprocessor systems. The paper describes the design and implementation of a set of extensions to the C programming language that tackle the performance portability problem by allowing the programmer to lightly annotate their program to indicate the desired mapping of tasks and variables onto the system. The SoC-C model is of a program running on a single control processor and performing remote procedure calls to accelerators, DSP engines and DMA engines. The SoC-C compiler takes care of (radically) restructuring the program to run efficiently.

SoC-C was a reaction against the dataflow-based stream programming model exemplified by StreamIt [58]. We chose a sequential communication language instead of a dataflow language because we found it hard to express global control (i.e., conditionals that span multiple pipeline stages) over pipeline stages that execute asynchronously with respect to each other. Using decoupling to introduce parallelism, gives the ease of expression of global control that imperative languages provide combined with the pipeline parallelism that stream languages provide.

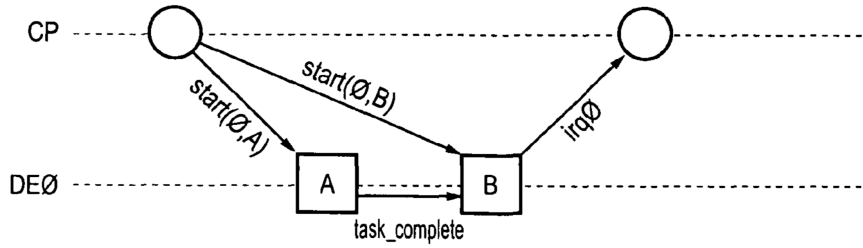
Several alternative approaches to decoupling programs exist [22, 30, 35, 90, 114, 116]. A distinctive feature of our approach is that the annotations can indicate that a non-FIFO communication mechanism should be used — we found that this was helpful in avoiding “loss of decoupling” [18] where parallelism collapses because the start of the pipeline requires results from the end of the pipeline.

[Patent I](#) [103] tackles a problem we identified after some experience of using SoC-C. We realised that the control processor was becoming a bottleneck: when a task finished on one accelerator, there was an unavoidable delay of 50-60 cycles before the next task in the sequence was started. [Figure 1.4a](#) illustrates this with an example task invocation pattern where the completion of a first task A triggers an interrupt that causes the control processor to configure and start a second task B.

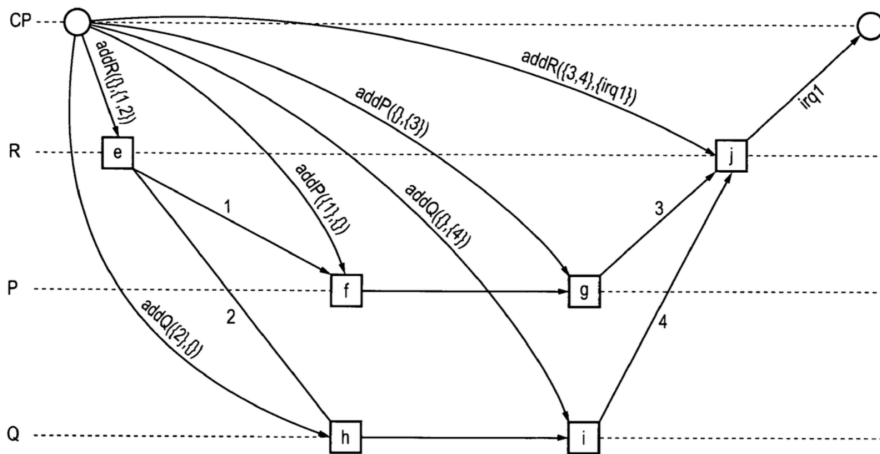
[Patent I](#) describes an extension of [Paper IV](#) that is able to exploit a variety of different simple task triggering hardware mechanisms to reduce this delay between tasks to a few cycles. [Figure 1.4b](#) illustrates this with an improved invocation pattern where the completion of task A directly triggers the start of a second task B that had been configured while task A was executing. The “description of embodiments” section in columns 11–28 of [Patent I](#) describes how accelerator interfaces can often be viewed as a FIFO task queue (typically with a modest capacity of



(a) Unoptimised execution: The Control Processor CP initiates tasks on the data engine DE0 and receives interrupts when each task completes.



(b) Optimized execution: The Control Processor CP performs setup of task B while task A is running on data engine DE0. The completion of task A directly triggers the start of task B without the intervention of CP.



(c) Highly optimized execution of a multiprocessor example using inter-engine event triggering mechanism. Edges between data engines are labelled with the event number that is sent when the source task completes. The Control Processor CP performs setup of tasks (e)-(j) to run on data engines P, Q and R; data engines use the event triggering mechanism to allow completion of tasks to directly trigger the start of new tasks on other engines. The CP only receives one interrupt from the six tasks and the latency between tasks is reduced to a few cycles (the latency of event signalling and of task startup/shutdown).

Figure 1.4: Illustration of the effect of optimization of task invocations. The control processor (CP) and each data engine (DE0, P, Q and R) are on the vertical axis and time is on the horizontal axis. These figures are reproduced from Figures 5a, 5c and 7 in [Patent I](#).

0–3 pending tasks). Modeling these queues explicitly enables the data queue optimizations in [Paper IV](#) to be adapted to the optimization of task queues.

A synchronous RPC can be viewed as putting an RPC request into a task queue and then waiting for an RPC request from the accelerator (column 13). This model enables the compiler to optimize a sequence of synchronous RPCs

```
RPC(P,f);
RPC(P,g);
```

by first splitting the RPCs into a sequence of RPC-put/get pairs

```
RPC_put(P,f);
RPC_get(P,f);
RPC_put(P,g);
RPC_get(P,g);
```

then reordering the puts and gets to eliminate the inter-task latency

```
RPC_put(P,f);
RPC_put(P,g);
RPC_get(P,f);
RPC_get(P,g);
```

obtaining the benefits of asynchronous RPCs without the usual programmer burden.

Even with these optimizations, the cost of creating a task queue entry can be significant when it occurs on the critical path. A further set of optimizations (columns 21–23) describe how the zero-copy optimizations applied to data queues can also be applied to task queues by further splitting each of the RPC-put/get operations into two parts.

```
RPC(P,f);
split into →
RPC_put(P,f);
RPC_get(P,f);
split into→
RPC_acquireRoom(P,f);
RPC_releaseData(P,f);
RPC_acquireData(P,f);
RPC_releaseRoom(P,f);
```

This exposes the expensive task creation step (`RPC_acquireRoom`) providing opportunities for the compiler to reschedule sequences of RPC operations to remove task creation from the critical path. A key part of these transformations is tracking the data dependencies and the

number of pending tasks to determine which reorderings are safe to perform and to allow the RPC sub-operations to be reordered with respect to non-RPC operations.

If accelerators are able to send events to each other when a task completes and to delay task start until an event arrives, then it is possible to optimize more complex patterns of task invocation involving multiple accelerators as illustrated in Figure 1.4c (columns 16–19).

The key to implementing such transforms was the realization that many different hardware mechanisms for task configuration and sequencing can be modelled as FIFOs; that this uniform model allows extensive optimisation of task creation and task triggering; and that this enables the use of a broad range of optimizations more normally used for instruction scheduling. One of the main technical challenges is that the FIFOs are finite (and usually quite small) so it is essential to reason about how full the FIFOs are relative to their individual capacity in order to avoid introducing potential deadlock and other problems.

1.3.3 Contributions

The technical contributions of this section include a set of annotations and inference techniques to define the mapping of an application onto a high-performance, energy-efficient parallel system, a set of compiler techniques to implement the annotations efficiently and a unified model of hardware interfaces that enabled these techniques to be applied to a broad range of different accelerators in such systems.

In addition to these technical contributions, the work played a key role as part of a commercial system development. From an early stage in the Ardbeg project, the team recognised that long-term success depended not just on the high performance DSP accelerators but also on the ease with which the overall system could be programmed. The SoC-C compiler described in this section was a key part of the overall project. The Ardbeg project was spun out of Arm in 2009 to create “Cognovo Ltd” that specialised in the creation of Software Defined Modems although SoC-C was not part of that spinout. Cognovo successfully manufactured a chip capable of implementing the 4G LTE standard based on the Ardbeg reference design. Cognovo was acquired by the Swiss wireless modem company “u-blox” for \$16.5M in 2012.

1.3.4 Limitations and further work

One of the major limitations of our work is that SoC-C maintains a clear separation of the coordination and control language (SoC-C) from the language used to program the DSP engines. In a software defined radio application, each DSP task typically contains considerable data parallelism and may contain some potential pipeline parallelism but the only way for the programmer to exploit this pipeline parallelism would be to split the tasks into subtasks to expose this potential to SoC-C. A more unified description of the control language with the DSP language would expose more optimisation opportunities.

General Purpose Graphics Processing Units (GPGPUs) have similar characteristics to SDR systems: they have complex memory hierarchies, multiple processors and are often programmed using the same techniques of pipeline parallelism and double-buffering. At the time that we performed the work (2006–2008), users of such systems often struggled with the quantity and complexity of “boilerplate” code that had to be written to copy data from the main processor over to the GPGPU, to invoke tasks on the GPU and to wait for the GPGPU to finish. *In theory*, this is the problem that SoC-C is designed to handle although, *in practice*, we expect that SoC-C would, at least, need extensions to express data parallelism to be useful. In the intervening 10 years, OpenCL has been extended with explicit task queues and the SYCL embedded domain-specific language (EDSL) [60] that enables the programmer to dynamically specify the dependencies between tasks. Use of SYCL does not enable the rich set of compile-time transformations performed by SoC-C but it has the advantage of not requiring a custom compiler. Similarly, OpenMP has been extended with better support for expressing task-parallelism such as explicit tasks and task synchronization [20]. This provides OpenMP with a lot of the expressive power of SoC-C (in addition to the data-parallelism support it already provided) although it would be useful to add the annotation checking and inference of SoC-C to OpenMP.

Finally, SoC-C is not a fully automatic solution: the programmer must choose the mapping of tasks and data to the hardware platform. It would be interesting to add a profile-driven mapping algorithm on top of SoC-C to automatically discover efficient mappings; this would benefit from SoC-C’s ability to detect invalid mappings.

1.3.5 Conclusions

This section explores the definition of high performance hardware-software interfaces by raising the level of the interface. The paper and patent in this section of the published work describe programming language extensions and compiler technology that allow software to exploit the potential performance of the hardware but, at the same time, provide a high degree of portability.

1.4 Conclusions

This thesis is concerned with defining the hardware-software interface in modern microprocessors and makes contributions in two key aspects:

- Creating high quality definitions of the interface resulting in: a methodology for creating a formal specification of unparalleled scope, applicability and trustworthiness; a methodology for formally validating commercial processor pipelines against the formal specification; and the public release of the formal specification of a major commercial architecture.
- Defining high performance hardware-software interfaces resulting in a set of language extensions and compiler techniques for tackling the performance portability problem in

high performance, energy efficient asymmetric multiprocessor systems used for software defined radio.

1.4.1 Limitations and further work

The two individual parts of this thesis identified individual limitations. One common limitation is that all this work was performed at a single company and that company inevitably has certain preconceptions about the hardware-software interface based on its particular role in the computer industry and on its business model. For example, a software team with no ambitions for formal verification might place more value on other aspects of the architecture specification such as the clarity of the English prose; and performance portability may be more important to the hardware vendor wishing to sell an upgraded system than it is to a programmer who may be perfectly content with the previous system.

There are also many other facets of the hardware-software interface that have not been explored in this thesis. Two key aspects for the future are security and supporting significantly more parallelism in the hardware-software interface.

Part I

Creating high quality definitions of the hardware-software interface

Three papers are presented in this section.

Paper I “Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture” [98] describes the process of creating a specification and of thoroughly testing the specification.

Paper II “End-to-End Verification of ARM Processors with ISA-Formal” [101] describes a collaboration with formal verification engineers in Arm’s Processor Design Division to find a way to formally verify the pipeline of Arm processors against the formal specification.

Paper III “Who guards the guards? Formal Validation of the ARM v8-M Architecture Specification” [100] describes the use of formal verification tools to formally validate the formal specification described in Papers I and II.

Chapter 2

Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture (Paper I)

Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. *In Proceedings of Formal Methods in Computer-Aided Design, (FMCAD 2016), Mountain View, CA, USA*, pages 161–168, October 2016.

© 2016 Alastair David Reid and FMCAD, Inc.

doi: <https://dx.doi.org/10.1109/FMCAD.2016.7886675>

Trustworthy Specifications of ARM[®] v8-A and v8-M System Level Architecture

Alastair Reid
Research, ARM Ltd.
first.last@arm.com

Abstract—Processor specifications are of critical importance for verifying programs, compilers, operating systems/hypervisors, and, of course, for verifying microprocessors themselves. But to be useful, the *scope* of these specifications must be sufficient for the task, the specification must be *applicable* to processors of interest and the specification must be *trustworthy*.

This paper describes a 5 year project to change ARM's existing architecture specification process so that machine-readable, executable specifications can be automatically generated from the same materials used to generate ARM's conventional architecture documentation. We have developed executable specifications of both ARM's A-class and M-class processor architectures that are complete enough and trustworthy enough that we have used them to formally verify ARM processors using bounded model checking. In particular, our specifications include the semantics of the most security sensitive parts of the processor: the memory and register protection mechanisms and the exception mechanisms that trigger transitions between different modes. Most importantly, we have applied a diverse set of methods including ARM's internal processor test suites to improve our trust in the specification using many other expressions of the architectural specification such as ARM's simulators, test suites and processors to defend against common-mode failure. In the process, we have also found bugs in all those artifacts: testing specifications is very much a two-way street.

While there have been previous specifications of ARM processors, their *scope* has excluded the system architecture, their *applicability* has excluded newer processors and M-class, and their *trustworthiness* has not been established as thoroughly.

Our focus has been on enabling the formal verification of ARM processors but, recognising the value of this specification for verifying software, we are currently preparing a public release of the machine-readable specification.

I. INTRODUCTION

Recent years have seen an increasing focus on verification of machine-code programs [1], compilers [2], operating system kernels [3], hypervisors [4] and processors [5]. These activities rely on having correct specifications of the meaning of machine-code and one of the first steps in such verification efforts is creating a specification of the computer architecture of interest.

Three key properties of a processor specification are its *scope*, its *applicability* and its *trustworthiness*.

The *scope* of a specification is the set of features that one can reason about. For example, a certified compiler such as CompCert [2] only requires a specification of those instructions that the compiler could generate. But in order to reason about arbitrary user-mode binaries, one would need a specification of the entire instruction set. And to reason

about Operating System code, the scope of the specification is dramatically increased and includes a specification of instructions for changing execution mode (e.g., entering/leaving supervisor mode), interrupt handling mechanisms, page faults, mechanisms for changing memory protection, etc. To date, all formal specifications of the ARM architecture have been targetted at reasoning about user-mode programs and have not included a specification of these system-level features.

The *applicability* of a processor specification is whether the specification applies to the target processor. Most changes to architecture specifications are backward compatible extensions and so most proofs about code for one architecture version are valid when executing that code on a processor implementing a later architecture version. But architecture revisions also remove instructions, add restrictions or change functionality so proofs based on the ARMv6 specification (1996) or the ARMv7-A specification (2007) are not necessarily sound for ARMv8-A (2013). This is especially true for ARM's Microcontroller architecture which has a completely different exception model from ARM's mainstream architecture.

The *trustworthiness* of a processor specification is whether the specification can be trusted to reflect the behaviour of all processors implementing the specification. The ARMv7 HOL specification of Fox and Myreen [1] is noteworthy for the degree of testing performed: systematically testing all user-mode, integer instructions against three actual processors. This is a critical step and must be repeated against as many expressions of the architecture as possible (processors, implementations, test suites, etc.) and must be used to test the full scope of the specification.

The effort required to create a specification increases with the desired scope, applicability and trustworthiness of the specification. Worse, since ARM regularly releases extensions and corrections to the architecture, the challenge of retaining applicability to current processors is more of a continuous process rather than a one-off sprint. Our solution to this problem has been to change ARM's existing architecture specification process so that machine-readable, executable specifications can be automatically generated from the same materials used to generate conventional documentation.

This paper describes our work over the last 5 years on transforming the ARM processor specifications from documents intended for human consumption into trustworthy machine-readable specifications.

Creating this specification required understanding and cod-

ifying the precise meaning of various notations used in the documentation; inferring the lexical, syntax, type rules and semantics from examples in the documentation; making the specification conform to these rules; filling gaps in the original specification; and creating a frontend and several backends to allow the specification to be executed.

Using ARM's specifications directly addresses the issues of *scope* and *applicability* but the resulting formal part of the specification is just one part of the whole specification and, like any large specification, may contain bugs wrt the informal parts of the specification or with the architects' informal intent. To address the issue of *trust*, we have used a diverse set of testing methodologies to compare against as many different expressions of the specification as possible: testsuites, simulators and processors. We have simulated billions of instructions and used bounded model checking to compare the RTL of five ARM processors currently in development against the specification [6]. Bugs found in the process have been fixed in the master copy of the specification from which ARM's architecture specification documents are generated. This process has the effect of distilling more of the architectural intent into the formal part of ARM's official specification.

The structure of this paper is summarized in Figure 1 which gives an overview of the specifications, tools, verification IP, and testing we created or used in the process of this project. Section II gives a brief overview of the structure and content of the different ARM Architectures. Sections III and IV describe the steps we took to convert ARM's existing informal documentation into machine-readable, executable, trustworthy specifications of the ARM-v8A and ARM-v8-M architectures; Section V discusses related work; and Section VI concludes.

This paper deals with the Instruction Set Architecture (ISA), Exceptions, Memory Protection/Translation and Security. It does not deal with multiprocessor features and, in particular, the Memory Ordering Model [3], [7], [8]. And it does not deal with debug or performance monitoring features.

II. ARM SPECIFICATIONS

ARM Architecture specifications have two main sections: Application Level Architecture and System Level Architecture.

The Application Level Architecture (aka the Instruction Set Architecture or ISA) consists of all instructions and all user-mode registers (the integer and floating point register files, condition flags, stack pointer and program counter). ISA specifications consist of instruction encodings, matching rules to match encodings to opcodes and the semantics of instruction execution.

The System Level Architecture defines Memory Translation and Protection, Synchronous Exceptions (e.g., page faults and system traps), Asynchronous Exceptions (e.g., interrupts), Security (e.g., register banking and access protection of registers), and System Registers and System Operations (which are used to control and read the status of all the system-level features). In other words, the facilities needed to support Operating Systems, Hypervisors and Secure Monitors.

The ARM architecture comprises three main processor classes: "A-class" processors support Applications (characterized by having an operating system that uses address translation to provide virtual memory); "R-class" processors support Real-Time systems that cannot handle the timing variability associated with virtual memory and use memory protection instead; and "M-class" microcontrollers are optimized for programming interrupt-driven systems in the C language. The A-class specification consists of two parts: *AArch32* supports 32-bit programs and is generally backward compatible with ARM's traditional architecture; and *AArch64* which supports 64-bit programs.

The A- and R-class architecture [9] share the same ISA and exception model but have different memory protection/translation models. The M-class architecture [10] has a subset of the A-class ISA but has significant differences from A-class at both the Application Level and System Level.

A. ISA Differences between A/R- and M-class

The M-class architecture only supports the Thumb® (aka "T32") variable-length instruction encodings whereas the A/R-class architecture also supports the A32 and A64 encodings.

Much more significantly though, the specifications identify certain instruction encodings as UNPREDICTABLE for which a processor is free to do anything that can be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and that does not halt or hang the processor or parts of the system.

In the M-class architecture, many of the instruction encodings which access the stack pointer (R13) or the program counter (R15) are UNPREDICTABLE but the same encodings are well defined in the A/R-class architecture. This is a significant difference — it would be unsound to use the A-class specification to reason about Thumb machine code intended for an M-class processor.

More broadly, when performing formal verification, it is essential to ensure that the specification version being used matches the architecture version supported on the target processor because later specifications are *almost* but *not entirely* backward compatible. This is obvious but easily overlooked.

B. System Differences between A-, R- and M-class

The R/M-class architectures support memory protection based on setting attributes and protection for a small number of contiguous memory regions whereas the A-class architecture supports both address translation and memory protection for a large number of memory pages.

M-class processors automatically save the callee-save registers on the stack on taking an exception whereas A/R processors require registers to be saved in software. This allows M-class processors to respond more quickly to interrupts and also allows exception handlers to be written in plain C with no assembly language or special calling conventions. This has a large impact on the architecture specification since it introduces many corner cases associated with the effect of triggering memory faults while saving or restoring registers.

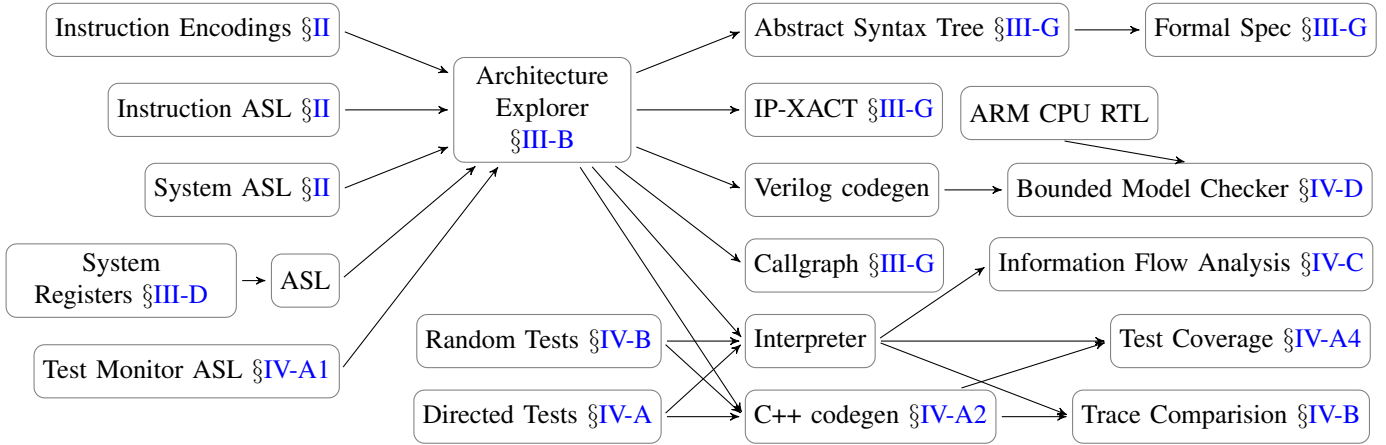


Fig. 1: Overview of specifications, tools, verification IP and testing. This flow was applied separately to the v8-A specification and to the v8-M specification. Section numbers indicate which section primarily discusses each aspect.

M-class processors have an orthogonal set of 8 execution states composed of combinations of three properties: privileged/unprivileged, secure/non-secure and handler/thread. A/R-class processors have a more traditional set of nested execution states EL0, EL1 (supervisor), EL2 (virtualization) and EL3 (secure monitor) with increasing levels of privilege at each level.

A consequence of these differences is that the M-class system specification is completely different from the A/R-class system specification.

III. EXECUTABLE SPECIFICATIONS

We faced five major challenges in turning ARM’s documentation-based specification into an executable specification: (1) Scale: ARM specifications are very large; (2) Informality: ARM specifications are written in “pseudocode”; (3) Gaps: key parts of the specification only existed in natural language specification; (4) System Register Specifications; and (5) Implementation Defined Behaviour.

A. ARM Specifications Are Large

One of the main challenges in creating machine-readable specifications of the ARM Architecture is the scale of the problem. The A and M-class architectures together consist of over 6,000 pages of documentation, 1,570 instruction encodings, over 50,000 lines of pseudocode, over 4,500 system register fields grouped into 772 system register, and 112 system operations. To this specification that ARM publishes, we added an additional 8,190 lines of support pseudocode which were required to make the execution executable. (A more detailed breakdown of the size of the specification is given in table 2a and table 2b.)

B. Pseudocode

A secondary challenge in creating a machine readable specification was that the bulk of the specification is written in what the ARM documentation refers to as “pseudocode”. For example, the T32 CMP instruction is specified with the

following encoding diagram and pseudocode in the v8-A architecture. (The same instruction is UNPREDICTABLE in v8-M if “m == 13”.)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	0	1	1	Rn				(0)	imm3			1	1	1	1	imm2		type	Rm				

CONDITIONAL

```

n = UInt(Rn); m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m == 15 then UNPREDICTABLE;
shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
(result, nzc) = AddWithCarry(R[n], NOT(shifted), '1');
PSTATE.<N,Z,C,V> = nzc;
```

Fortunately for us, this “pseudocode” was fairly complete and it appeared possible to implement a conventional parser, typechecker and interpreter for pseudocode (a tool we call “Architecture Explorer”). Through a process of experimentation, discussion and negotiation with the architecture designers, we were able to infer consistent indentation rules, precedence rules, a type-system and semantics and to clean up the specifications to use the resulting simpler, more consistent language that is now internally referred to as ARM Specification Language (ASL).

At a high level, ASL is an indentation-sensitive, imperative, strongly typed, first-order language with dependent types (to reason about length of bit vectors), type inference, exceptions, enumerations, arrays, records, no pointers. Unusually for an otherwise simple language, ASL allows overloading of array syntax for function calls: the use of “R[m]” and “R[n]” on lines 4 and 5 of the example above are both function calls. This syntactic sugar provides an initial impression that registers (and memory) are simple arrays, while allowing one to dig deeper and understand register banking, virtual memory, etc. We refer readers to Fox and Myreen [1] or to ARM’s specification [9, Appendix G] for a more detailed description of ASL.

The initial cleanup of syntax and type errors resulted in changes to approximately 12% of the lines of code but,

	ARMv8-A				ARMv8-M	
	AArch32	AArch64	Shared	Support	Spec	Support
Instrs.	18318	5757			4998	
Integer	23		352		246	
Float Point			1179		953	76
Exceptions	1474	1611	235		781	
Registers	310	446	398		2011	461
Memory	1584	1169	393		369	481
Debug	675	537	1103			
Instr. Fetch				199	367	128
Test Monitor	-	-	-	1323	-	1893
Misc.	1647	1137	2984	1678	415	1434
Total	24315	10657	5489	3200	9898	4990

(a) Size of ASL specification (lines of code)

	v8-A	v8-M
Registers	586	186
Fields	3951	622
Constant	985	177
Reserved	940	208
Impl. Defined	70	10
Passive	1888	165
Active	68	62
Operations	112	10

(b) Size of System Register specification

Fig. 2: Size of ARM Specifications

since ARM specifications are extensively reviewed before release, these were all fairly low-grade errors: they confused automatic tools but few were likely to confuse a human reader. The process of cleaning up the specification also uncovered a number of instances of “implement by comment” where comments were used instead of pseudocode: these parts had to be rewritten before the code could be executed. These simple comments often turned out to be surprisingly complicated and the process of writing code would identify corner cases or the need to modify other parts of the specification.

C. Gaps in the specification

Some parts of the architecture were only defined in English and the information to implement them was typically scattered throughout the documentation. An example is the specification of the “top-level” step of fetching an instruction, decoding and executing it, and incrementing the program counter was not written in ASL and the description was scattered across the specification document. The exact specification of this step took some time to develop as it includes details like dealing with page faults that occur during instruction fetch, not incrementing the PC after a branch instruction or exception, conditional execution of instructions and its interaction with UNDEFINED encodings, and testing for pending interrupts.

D. System Register Specification

The major negative surprise of this project was how hard it was to specify something as apparently simple as a register.

The A-class architecture specification comprises 586 system registers which are used to read the status of and to control the behaviour of the processor (such as whether the MMU or cache is turned on) and to perform operations such as flushing the cache or invalidating the TLB. The main properties of these registers are captured in the architecture specification by tables specifying the opcode to access each register, its name, size (32/64-bits) whether it is read-only and the reset value of the register. For each register, there is a description consisting of a register diagram which identifies the name and extent of any

used bits in the register. And each such field of contiguous bits has a natural language specification.

The challenge in creating a machine-readable specification for system registers is that different fields within the register can behave in several different ways. After some experimentation we settled on identifying five major types of field.

i) *Constant fields* have an architecture defined value and cannot be changed.

ii) *Reserved fields* are not used in the current version of the architecture but could be assigned a meaning in future versions of the architecture. These are like constant fields but, to maintain forward compatibility, software should not assume that the field is constant and should avoid changing the value of that field.

iii) *Implementation Defined fields* have an implementation defined value that programs may read to determine whether the processor has some ISA or system level feature.

iv) *Passive fields* behave like a global variable and simply store the value last written to the field. The value written often has a significant effect such as enabling address translation but this effect is completely captured by the ASL functions implementing the affected behaviour.

v) *Active fields* do not behave like a global variable: reading the field may not see the last value written to the field; writing to the field may be disabled by the value of some other register; etc. These are used for everything from system timer registers (which decrement every cycle) to allowing a hypervisor to intercept interrupts targetted at the guest operating system.

Fields that are Constant, Reserved, Implementation Defined or Passive are easy to describe completely and are described in a simple table-based format but 68 of the fields of system registers are Active fields whose behaviour can only be captured by writing ASL getter and setter functions to implement the natural language specification. The process of implementing registers with active fields proved to be quite error prone as the behaviour of the fields was rather subtle.

It was also hard to find the correct design point. We chose to identify just 5 classes of field but we could have identified further common patterns within the Active class. For example,

there are some pairs of registers that have complementary effects such as enabling and disabling exceptions. If this pattern is a one-off, it is probably best described as an Active register but if the pattern occurs in several pairs of registers, then the argument for recognizing it as a new class of field becomes stronger. As the number of tools using the system register specification grows, we expect that we will identify a number of patterns that are useful to recognise explicitly because that enables tools to make more use of the specification without having to embed the ASL parser/interpreter.

One significant aspect of system registers not yet captured in the executable specification is what Lustig et al. [8] call a *memory transistency model* which captures places where the specification allows reordering of writes to system registers with respect to other instructions and requires insertion of instruction barrier instructions (ISB) to restrict.

E. Implementation Defined Behaviour

The specification allows for some implementation defined behaviour such as whether a particular feature is implemented or the number of memory protection regions supported. This behaviour is often specified by “stub functions” returning booleans or an enumerated value and with a natural language definition. We had to implement these stub functions before we could execute the specification. In most cases, these feature test functions could be implemented by testing a corresponding implementation defined field.

F. Executable Specification

After creating all the tooling, bugfixes, etc. described above, there were some further steps required to make the specification executable so that it could be tested. We had to add additional infrastructure such as generating decode trees for a set of encodings to identify which instruction to execute; ELF readers to load test programs into memory; a physical memory implementation which allocates pages of memory on demand. and breakpoint and trace facilities to use when debugging.

We also introduced a continuous integration flow where every specification change runs regression tests. This was critical for confining new code to the ASL subset of pseudocode.

G. Machine Readable Specifications

Our primary goal in doing the above was not to make the specification executable but, rather, to improve its quality so that the specification is useful to many potential users. To support these uses, we generate a variety of machine-readable outputs.

- i) *IP-XACT* is a standard XML-based format for describing registers in a chip [11]. It is used by debuggers needing to view or change the value of a register.
- ii) *Callgraph summaries* are convenient summaries of the function calls and variable accesses performed by each instruction and function in the specification. One use of these summaries is in generating a summary of the list of exceptions that an instruction can raise — for inclusion in documentation.
- iii) *Abstract Syntax Trees* are a complete dump of Architecture Explorer’s internal representation after typechecking. We have

provided these to the University of Cambridge REMS group who are in the process of transforming them into a form suitable for formal verification of machine-code programs.

IV. TRUSTWORTHY SPECIFICATIONS

ARM spends considerable effort on reviewing specifications. It also benefits from feedback from users of the specifications: processor designers, verification engineers, implementers of simulators, compiler writers, etc. Nevertheless, the sheer size of the specification made it unlikely that the specifications are bug-free. This was especially true of the relatively fresh v8-M specification since it had not yet had the benefit of feedback from users of the specification.

This Section describes the steps we have taken to test the v8-A and v8-M specifications using testsuites, random instruction sequences, information flow analysis and using bounded model checking to compare against the Verilog implementation of processors. One of the recurring themes of this project was that this testing process improves the specification and our trust in the specification — but it also improves the tools, verification IP, etc. that is being used to test the specification which creates a virtuous cycle of improving any other uses of those tools and artifacts.

A. Using ARM Processor testsuites

ARM performs extensive testing of its processors and simulators (it is estimated that more than 80% of the engineering effort of designing a new processor is spent on testing the processor). One part of this testing process is use of ARM’s Architecture Validation Suite (AVS) which consists of programs that test the architectural conformance of individual instructions, memory protection, exception handling and all other aspects of the architecture. Excluding multiprocessor and debug tests, the AArch64 AVS consists of over 11,000 test programs with a combined runtime of over 2.5 billion instructions; the M-class AVS consists of over 3,500 test programs with a combined runtime of over 250 million instructions. Almost all of these tests were considered to be free of assumptions about instruction timing or implementation defined behaviour. (ARM has a large number of other tests which were less appropriate to run because they are aimed at testing micro-architectural performance optimizations in particular processors.)

Using ARM’s official Architecture Validation Suite has some significant advantages: the suite is very thorough, checks many corner cases, and has good control and data coverage of the architecture; the suite is self-checking: each test prints “PASSED” or “FAILED” when it runs; and, since the purpose of the tests is to test processors, it was possible to compare the behaviour against actual processors for additional confidence. The primary disadvantage of using the AVS was that the tests are “bare metal” tests that exercise the System Level Architecture and require a large test harness to run.

As we started using Architecture Explorer to develop new architecture extensions (such as the new security features of v8-M), we encountered a chicken-and-egg problem: the AVS

is extended with new tests only once the architecture specification is available but we were still writing the specification. Worse, v8-M is not entirely backward compatible with the previous architecture version so we could not even run the old tests. This led us to use a hybrid approach: we temporarily created a modified specification supporting the old memory protection design so that we could use the old tests; and we created a temporary test suite to test the new security features of v8-M (see Section IV-C) before the official test suite was developed. Once updated AVS tests became available, we switched to using the official test suite.

1) *Programmable Monitor and Stimulus Generator*: Part of the development of every ARM processor is creating a test harness which allows the AVS to be run. This test harness consists of a programmable monitor and stimulus generator that allows programs to monitor their own behaviour at a very low-level. The test monitor design dates back to the earliest days of ARM and each successive architecture extension typically adds new test features.

The monitor consists of 177 memory mapped registers of which 45 are Active. The main features of the test monitor are

- (i) *Console FIFO* for writing ASCII text to log file.
- (ii) *Memory attribute monitors* which record the attributes of memory accesses in a given range of addresses. This allows test programs to verify that the MMU/MPU is correctly associating attributes such as cacheability of an access with each address. These checkers are repeated for each bus interface.
- (iii) *Memory abort generators* to trigger a bus fault response if the processor accesses a specified range of addresses.
- (iv) *Interrupt generators* to test triggering, prioritization and nesting of interrupts.
- (v) *Reset generators* to schedule resets.

2) *Optimizing the simulator*: During this testing process, we slowly built our capability from being able to execute one instruction to being able to execute most usermode instructions, to being able to execute entire tests and then entire test suites. As we did so, we were increasingly limited by the performance of our interpreter which initially ran at a few hundred instructions per second. Over time, we have optimized this in a variety of ways increasing performance to 5kHz (v8-A) and 50kHz (v8-M). The main optimizations applied are: (i) Memoizing a few critical functions associated with the current configuration or execution state (this has not been yet been applied to v8-A); (ii) Implementing a few critical arithmetic functions as builtin primitives even if they can be defined in ASL; (iii) Creating a C++ code generator and runtime (including ELF reader, etc.).

3) *Testing the specification*: One of the issues found while testing the specification initially manifested as a failing AVS test. On closer inspection, we found a mismatch between the English text and the pseudocode and that the test had originally followed the pseudocode and ARM's reference simulator followed the English text. This mismatch had been "fixed" by changing the test to match the simulator. Consulting the architects, we learned that the pseudocode was correct and

the English text was wrong and so the English text, the test and the simulator were fixed to match the architects' intent.

The pass rate of our specifications on the AVS is summarized in Table I. We have achieved a 100% pass rate for the v8-A and v8-M ISA tests and for the v8-M System tests. For the v8-A System tests, there remain some failing tests in areas related to interprocessing (switching between 32-bit and 64-bit modes) and prioritization of multiple exceptions within the same instruction. These results omit debug and multiprocessor tests which are just under 50% of the total number of tests.

	ARMv8-A	ARMv8-M
ISA		
Integer	100%	100%
Floating Point	100%	100%
SIMD	100%	100%
System		
Exceptions	100%	100%
Memory	99%	100%
Interprocessing	98%	-

TABLE I: Pass rate for AVS testsuite

4) *Testing the testsuite*: Testing the specification with a testsuite has the side-effect of testing the testsuite. We found two classes of problems in the process of diagnosing test failures. The first is that a test may depend on some property not guaranteed by the architecture but which had been true in every tested processor. For example, a test might check that a reserved field of a register is always zero and will then fail on later versions of the architecture. Secondly, many of the M-class AVS tests depended on UNPREDICTABLE behaviour but this had not been observed before because, in practice, UNPREDICTABLE behaviour can depend on the particular pipeline state when an instruction runs.

To improve testing of the AVS, we extended the interpreter to collect line coverage information as it executes. A rare example of a coverage hole we found was in a floating point test which tested with inputs that produced the result +0.0 but did not test with inputs that produced the result -0.0 — with the result that one of the branches associated with rounding was not being exercised. The AVS development team now routinely measure the architectural coverage of test suites.

B. Random Instruction Sequence Testing

Random Instruction Sequence (RIS) testing is a complementary technique to the directed testing of using hand-written tests based on generating random sequences of instructions. ARM's RIS tool [12] uses templates that specify the desired distribution of instructions, the likelihood of reuse of a given register, etc. Automatically generating random tests is different from hand-writing tests because it requires an accurate simulator to define the correct behaviour of a test. Also, because RIS generates random sequences of instructions, it is necessary to run the same test on multiple systems (processors, simulators or the specification) and compare execution traces. So at least two models are needed to develop RIS tests.

We were able to use the executable specification as part of the process for testing new RIS tests by extending the simulator to generate a trace and extending the existing trace comparison script to accept those traces. This process was especially useful for the v8-M specification because the v8-M support in ARM's reference simulator was new and had not been fully debugged. Using RIS to test the simulator against the executable specification was an effective way of testing the RIS tests, the simulator and the specification.

This process was able to uncover subtle errors in the specification. For example, v8-M's new security features splits some of the system registers into two banked registers –a non-secure register and a secure register– and the appropriate register is automatically accessed depending on the current security mode. But instructions that switch between secure and non-secure registers start in one mode and end in a different mode and the normally convenient automatic banking mechanism obscures exactly which of the two registers is being accessed. RIS testing found an error in the specification of the Test Target (TT) instruction which queries the security state and access permissions of a memory location.

C. Information Flow Analysis for v8-M

The most significant new feature of the v8-M microcontroller specification is a set of security extensions to enable secure Internet of Things applications.

To improve confidence in both the extensions and in the way they were expressed in the ASL specification, we modified the interpreter to generate dynamic dataflow graphs on which we could perform information flow analyses. Most of the analyses performed can be characterized as a non-interference property: ensuring that non-secure modes cannot see secure data and that non-secure data can only influence secure code in safe ways.

An example scenario tested in this process involved information leaks via interrupts. Interrupts automatically save integer registers on the stack of the interrupted code and zero the integer registers but, in order to keep interrupt latency low, floating point registers are lazily saved on the stack only when/if the interrupt handler uses a floating point instruction. We wanted to ensure that lazy FP state preservation did not introduce security holes. We wrote tests that iterated over all combinations of initial mode, final mode, whether FP registers had been modified and scanned the dynamic dataflow graph for information leaks.

This form of testing caught two classes of bugs. First, it caught bugs in how the architecture specification implemented the architectural intent — resulting in fixes to how the specification was written. Second, and more importantly, it caught bugs in the architectural intent by identifying potential security attacks that had not been considered before.

D. Bounded Model Checking of Processors

We have been using both the v8-A and the v8-M architecture specifications to perform bounded model checking of pipelines for processors currently under development at ARM [6]. This has primarily focused on verifying the ISA-implementation

parts of the processor, not the memory system, security mechanisms or exception support. This process has been very effective at detecting bugs in various stages of processor development. But, besides verifying processors, it has another important side-effect of performing a very thorough check that the architecture specification and our tooling agrees with how the processor implementors interpret the specification. We found no errors in the published part of the specification in this process but we did find a rather subtle bug in our understanding of conditional UNDEFINED encodings and UNPREDICTABLE encodings.

The M-class specification requires that conditional execution of an UNDEFINED instruction behaves as a no-op if the condition does not hold and we had assumed that the same was true for UNPREDICTABLE instructions. During verification of a processor, the model checker detected an apparent bug that involved a conditional UNPREDICTABLE encoding but, through discussion between the processor designers and the architects, we learned that there had been a recent clarification of the architecture which said that conditional UNPREDICTABLE encodings are UNPREDICTABLE even if the condition does not hold.

This error in our interpretation of the specification had not been detected by testing because it is very, very hard to construct useful tests of the UNPREDICTABLE instructions because they are almost entirely unconstrained and can branch, change registers, trigger exceptions, etc.

E. Summary

Large specifications are as likely to contain errors as large programs so we have used many different approaches to test the specifications. In the process, we realized that although ARM publishes an official specification, the full requirements are really distributed around many different places in the company: the AVS suite, the reference simulator ARM uses for processor verification, and the processor implementations. The act of testing all these different instantiations of the specification against each other has the effect of centralizing this specification in a single location.

V. RELATED WORK

The most closely related work is that of Goel et al. [13] who have created an executable specification of many key parts of the x86-64 ISA and system architecture including paging, segmentation and both user/supervisor levels. Their model has been verified against real processors using the Pin binary instrumentation tool and they have added a syscall emulation layer to let them run real programs including (amusingly) a SAT solver. This is a monumental piece of work that sets the standard against which other architecture specifications should be judged. Despite the similarities, our different project priorities have led to many differences: (1) They have a specification of user and supervisor levels, we also have a specification of hypervisor and secure monitor levels. (2) They have used their specification to formally verify *software* using theorem proving, we have used our

specification to formally verify *hardware* using bounded model checking. (3) They have implemented syscall emulation to let them use user-level programs as tests, we have implemented a test monitor and debugged the EL2/EL3 levels to allow us to run ARM's Architecture Conformance Suite which explores the dark corners of the architecture by running bare-metal programs. (4) They have focussed on modelling the x86-64 64-bit ISA, we have modelled the A64, A32 and T32 ISAs. (5) They have consulted processor designers to understand Intel's architecture specification document, we have had all our bugfixes and clarifications reviewed by ARM's architects and incorporated into ARM's official architecture specification document.

The most closely related ARM specifications are the Fox/Myreen ARM v7-A ISA specification in HOL [1] and Flur et al.'s ISA and concurrency specification in Sail [3] both of which were tested against actual processors using random and directed tests (8400 tests in Flur et al., 281,307 tests in Fox/Myreen). In addition to user-mode instructions, our specification covers both the ARMv8-M architecture and the larger ARMv8-A architecture, includes floating point, Advanced-SIMD and the System Level Architecture. We have tested the entire specification in multiple ways and with a larger range of values and simulated more than 2.5 billion instructions in the process. And we have used a model checker to compare the ISA specification against actual implementations for all instructions, all execution modes, all integer inputs and a subset of floating point inputs [6].

Shi [14] extracted the ISA pseudocode from ARM's v6 Architecture Reference Manual, automatically translated the code to Coq and used that to verify that the ARM model in the SimSoC simulator written in C faithfully implemented the Coq specification. This is an impressive piece of work, and it would be interesting to repeat their work using our new, more trustworthy specification or to extend their proof to cover the system level architecture.

The other major ARM ISA specification that we are aware of is embedded in the CompCert compiler and is used in the proof that the compiler faithfully translates the input C program to ARM assembly code. This specification is limited to a subset of the user-mode ARMv6 specification and there is no published statement of how it was validated.

Hunt created a specification of the FM8501 processor [5] and used it to formally verify the processor. The process of formal verification greatly increases the trust we can place in the corresponding parts of the specification because it ensures that all the corner cases in both the processor and the specification have been explored.

More broadly, anyone wrestling with a large specification is obligated to find ways to verify that the formal specification captures the (informal) requirements.

VI. CONCLUSIONS

Historically, ARM's specification efforts have focused on a single set of products: the ARM Architecture Reference Manuals [9], [10]. However, there are many more potential uses of

the specification if the specification is delivered in a flexible, machine-readable format – for example, formal verification of hardware and software, tools that manipulate instruction encodings, debug tools, creating hardware verification tests. Traditionally, all these other users manually transcribe parts of the specification into some other notation: HOL, C, Verilog, spreadsheets, etc. This process is laborious and error-prone but, worse, it is fragmented: bugfixes or clarification found by one group are not necessarily propagated to other groups or to the master specification. Our primary goal in this project was to enable formal verification of ARM processors against the specification. But, by supporting as many of these uses as possible, we created a virtuous cycle where bugfixes or improvements were incorporated into the central specification so that all users benefit from bugfixes as well as to amortize the development effort across many uses.

This paper describes the steps required to create trustworthy specifications of the full v8-M and v8-A architectures including the instruction set architecture, memory protection and translation, exceptions and system registers. While checking that a formal specification captures the architects' informal intent is an unending process, we believe that our specification is the most trustworthy and complete system specification of any mainstream processor architecture.

We are currently working with Cambridge University on a public release of our specification suited to verification of machine code programs.

REFERENCES

- [1] A. C. J. Fox and M. O. Myreen, "A trustworthy monadic formalization of the ARMv7 instruction set architecture," in *Proc. Interactive Theorem Proving ITP 2010*, ser. LNCS, vol. 6172. Springer, 2010, pp. 243–258.
- [2] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.
- [3] S. Flur et al., "Modelling the ARMv8 architecture, operationally: concurrency and ISA," in *Proc. Principles of Programming Languages, POPL 2016*, 2016, pp. 608–621.
- [4] M. Dam, R. Guanciale, and H. Nemati, "Machine code verification of a tiny ARM hypervisor," in *Proc. Workshop on Trustworthy Embedded Devices*, ser. Trusted '13. ACM, 2013, pp. 3–12.
- [5] W. A. Hunt, "FM8501: A verified microprocessor," ser. LNCS, vol. 795. Springer, 1994.
- [6] A. Reid et al., "End-to-end verification of ARM[®] processors with ISA-Formal," in *Proc. Computer Aided Verification (CAV)*, ser. LNCS, vol. 9780. Springer-Verlag, 2016, pp. 42–58.
- [7] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 7:1–7:74, 2014.
- [8] D. Lustig et al., "Coatcheck: Verifying memory ordering at the hardware-OS interface," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS*. ACM, 2016, pp. 233–247.
- [9] ARM Ltd, *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. ARM Ltd, 2013.
- [10] —, *ARM v7-M Architecture Reference Manual*. ARM Ltd, 2006.
- [11] IEEE, "IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows," *IEEE Standard 1685-2014*, 2014.
- [12] B. Greene and M. McDaniel, *The Cortex-A15 Verification Story*. <http://www.testandverification.com/downloads/DVClub-Jan-2012/Cortex-A15-Verification-Story-DVclub-final.pdf>, 2011.
- [13] S. Goel et al., "Simulation and formal verification of x86 machine-code programs that make system calls," in *Formal Methods in Computer-Aided Design, FMCAD*, 2014, pp. 91–98.
- [14] X. Shi, "Certification of an instruction set simulator," Ph.D. dissertation, University of Grenoble, July 2013.

Chapter 3

End-to-End Verification of ARM Processors with ISA-Formal (**Paper II**)

Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Erin Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of ARM processors with ISA-Formal. In *S. Chaudhuri and A. Farzan, editors, Proceedings of the 2016 International Conference on Computer Aided Verification (CAV'16), volume 9780 of LNCS*, pages 42–58. Springer Verlag, July 2016.

© 2016 Springer Verlag. Reprinted by permission. License number: 4296060618361.

doi: https://dx.doi.org/10.1007/978-3-319-41540-6_3.

End-to-End Verification of ARM[®] Processors with ISA-Formal

Alastair Reid^(✉), Rick Chen, Anastasios Deligiannis, David Gilday,
David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel,
and Ali Zaidi

ARM Limited, 110 Fulbourn Road, Cambridge, UK

{alastair.reid,rick.chen,anastasios.deligiannis,david.gilday,david.hoyes,
will.keen,ashan.pathirane,owen.shepherd,peter.vrabel,ali.zaidi}@arm.com

Abstract. Despite 20+ years of research on processor verification, it remains hard to use formal verification techniques in commercial processor development. There are two significant factors: scaling issues and return on investment. The *scaling issues* include the size of modern processor specifications, the size/complexity of processor designs, the size of design/verification teams and the (non)availability of enough formal verification experts. The *return on investment* issues include the need to start catching bugs early in development, the need to continue catching bugs throughout development, and the need to be able to reuse verification IP, tools and techniques across a wide range of design styles.

This paper describes how ARM has overcome these issues in our Instruction Set Architecture Formal Verification framework “ISA-Formal.” This is an end-to-end framework to detect bugs in the datapath, pipeline control and forwarding/stall logic of processors. A key part of making the approach scale is use of a mechanical translation of ARM’s Architecture Reference Manuals to Verilog allowing the use of commercial model-checkers. ISA-Formal has proven especially effective at finding micro-architecture specific bugs involving complex sequences of instructions.

An essential feature of our work is that it is able to scale all the way from simple 3-stage microcontrollers, through superscalar in-order processors up to out-of-order processors. We have applied this method to 8 different ARM processors spanning all stages of development up to release. In all processors, this has found bugs that would have been hard for conventional simulation-based verification to find and ISA-Formal is now a key part of ARM’s formal verification strategy.

To the best of our knowledge, this is the most broadly applicable formal verification technique for verifying processor pipeline control in mainstream commercial use.

1 Introduction

Modern microprocessor designs apply many optimizations to improve performance: pipelining, forwarding, issuing multiple instructions per cycle, multiple

independent pipelines, out-of-order instruction completion, out-of-order instruction issue, etc. All of these optimizations are supposed to be invisible to the programmer in a uniprocessor context: the overall effect should be the same as executing instructions one at a time in program order. But each of these optimizations introduces corner cases that potentially change the behaviour and the different optimizations interact with each other in complex ways.

For example, in a pre-release version of one of ARM’s dual-issue processors, there was a defect in the inter-pipeline forwarding control logic that resulted in an instruction reading its input value from the wrong place if the instruction was preceded by a conditional instruction whose condition did not hold (and whose results should therefore not be used as inputs). The shortest instruction sequence which could demonstrate this defect was 5 instructions long. The particular set of instructions that could trigger the defect was fairly narrow because it was necessary that the instructions used particular parts of the pipeline, and the instruction sequence had to be aligned such that the first of these instructions executed in pipeline 0.

For traditional simulation-based verification to detect this defect you would need a detailed understanding of the micro-architecture of that particular processor, of the corner cases caused by the forwarding paths and of the kinds of errors one is likely to make in implementing forwarding control logic. Creating such tests is not only hard and unreliable, but it is also expensive because the tests would be specific to the particular micro-architectural choices in a processor and different tests must be created for each processor.

This paper describes the “ISA-Formal” verification technique that we have developed at ARM for verifying that processors correctly implement the Instruction Set Architecture (ISA) part of the architecture specification. Our method uses bounded model checking to explore different sequences of instructions and was able to detect the above defect prior to release of the RTL to manufacturers.

The *effectiveness* of ISA-Formal is important to its adoption within ARM but it is not the most important requirement we had to satisfy in order to make formal verification a useful part of ARM’s processor development flow. Before ISA-Formal could be deployed widely within ARM, we had make it work within the constraints of commercial processor development:

- (1) Processor development takes a long time (2 years or more) and it is important to be able to detect bugs at all stages of processor development. We have applied ISA-Formal all the way from incomplete designs that still contain bugs through to complete, heavily tested designs.

- (2) Verifying a processor takes longer than design: the long tail of processor development is developing new tests for the processor and fixing any bugs. It is important that useful results can be obtained even in the early stages of verification — before the complete test infrastructure has been developed. ISA-Formal is able to find bugs involving instructions for which we do not have a specification; all we need is a specification of any instruction whose result could be affected by the bug.

(3) Verification teams work in parallel with design teams so it is important that verification teams are able to continue searching for new bugs even when there are multiple outstanding bugs waiting to be fixed. Some bugs can take months to be fixed if they are not critical to immediate project milestones. ISA-Formal is able to work round known bugs in the processor.

(4) Any verification technique requires significant investment so reusability not only of the technique but also of the infrastructure is critical. We are able to reuse the tools across ARM v8-A/R (Application/Real-time) class and across v8-M (Microcontroller) class processors. The only part that needs to be customized for each processor is the Verilog abstraction function that extracts the effective architectural state from the micro-architectural state of a processor. This portability has been a great benefit while developing the technique because it allowed several processor teams to pool resources: one team worked on how to verify floating point instructions while another worked on branches and another worked on load-store instructions.

(5) Modern processor architectures and modern processors are large: the ARM v8-A ISA specification is over 2500 pages long, the v7-M ISA specification is over 600 pages long (almost half the length of the entire specification). It is important that verification techniques scale both in terms of human effort and computing resources. We have written a tool to automatically translate the source of the ARM Architecture Specifications to Verilog; and we split the verification task into thousands of small properties allowing effective use of large compute clusters.

We demonstrated these properties in three small-scale trials on different processors and have since refined and applied the technique on five further ARM processors: checking almost the complete instruction set architecture of these processors ranging from simple 3-stage microcontrollers up to sophisticated 64-bit out-of-order processors. ISA-Formal is now a key part of ARM’s formal verification strategy.

We characterise our approach as “end-to-end verification” because it focusses on directly verifying the path from instruction decode through to instruction retire against the architectural specification in contrast to hierarchical or block-level verification which focusses on verifying individual blocks against micro-architectural specifications and then verifying that the composition of those blocks meets the overall specification.

ISA-Formal is strongly based on techniques developed in the academic community; our contribution is a description of the techniques needed to make it scale and of the challenges and solutions in creating a portable approach which can be applied in a commercial setting to a wide range of processor micro-architectures.

The remainder of this paper is structured as follows: Sect. 2 discusses related work; Sect. 3 illustrates the basic idea, demonstrating how ISA-Formal can be applied manually, to a single instruction and discusses the kinds of bugs it was able to discover in real processors; Sect. 4 describes how we scaled this idea up to handle full ISA specifications; Sect. 5 describes adaptations to handle a variety of

different micro-architectural features; Sect. 6 reports on the results of applying this method to multiple processors; Sect. 7 concludes.

2 Related Work

Our work builds heavily on the pioneering work from the ‘90’s such as Burch-Dill’s automatic verification based on flushing refinements [5] and Srinivasan’s verification based on completion refinements [19]. These and many other works used different notions of correctness of which Aagard et al. [1, 2] give a useful taxonomy and establish conditions under which different notions of correctness are equivalent.

Our approach focusses on verifying RTL (Verilog) in contrast to work which verifies a high-level model of the microarchitecture design against a specification. For example, Lahiri et al. [14] verified the microarchitecture of the M*-core processor core (an early RISC-style architecture) and [13] verified the microarchitecture for an out-of-order processor through a series of successive refinements but neither verified against the RTL of an actual processor. In our experience, most errors are introduced while translating the microarchitecture design into RTL and during subsequent optimisation so verifying before RTL misses a lot of bugs. The challenge of verifying actual RTL is that it makes it hard to use abstraction techniques such as using uninterpreted functions because the actual RTL of an efficient processor tends not to have convenient blocks which match directly with parts of the original specification.

Many approaches to verifying pipeline control logic have used theorem proving techniques to tackle the difficult problems of handling pipeline forwarding and hazards in in-order processors [12, 21] and, later, for out-of-order processors [7–9, 16]. Theorem proving techniques are powerful and tend to suffer less machine-scaling issues than more automated techniques but their reliance on verification experts leads to severe human-scaling issues: it is hard to hire enough experts. We prefer to ride Moore’s law and use more CPU-intensive but more automatable approaches.

There has been considerable commercial interest recently in formal verification of floating point units such as Kaivola et al. [10], KiranKumar [11] and Slobodova et. al [18]. This is impressive and important work but essentially orthogonal to our own: while it tackles the scaling issues that occur when verifying commercial processors, it focusses on individual blocks processing a single instruction with relatively simple input-output signals while our approach focusses on the entire pipeline and especially the control logic to handle interactions between instructions. We describe how we deal with verification of pipelines containing floating point units in Sect. 5.1.

3 Illustration: Hand-Written Properties

The basic approach to verification that we use in ISA-Formal is based on the above prior work. We start with the processor in a simple, well-defined state

$uArch_0$ with no instructions in the pipeline. We then execute for a number of cycles where each cycle may issue an instruction. This serves to put the processor into a more complex state where hazards, forwarding, etc. can occur. And finally, we execute an instruction I_n and test whether the instruction executes correctly. This is done by applying an abstraction function abs which extracts the architectural state of the processor immediately before I_n executes and immediately after I_n executes. We do not flush the pipe before or after I_n .

$$\begin{array}{ccccccc}
 uArch_0 & \xrightarrow{I_1} & uArch_1 & \text{-----} & uArch_{n-1} & \xrightarrow{I_n} & uArch_n \\
 & & & & \downarrow_{abs} & & \downarrow_{abs} \\
 & & & & Arch_{n-1} & \xrightarrow{I_n} & Arch_n
 \end{array}$$

A key part of making this scalable is that, instead of allowing the formal verification tool to choose any instruction for I_n , we enumerate all the instruction classes supported by the architecture and perform a separate check for each instruction class. Proving these simpler results is helpful early in processor development by making it easy to focus on checking the currently implemented instructions. Later in development, the pattern of failing instructions is a useful guide in localizing the fault: if all branch instructions are failing, there is no need to worry about bugs in the ALU. And as the size of the verification task scales up, splitting the verification task into many small properties lets us make more effective use of our verification cluster which is optimized for running many independent processes across hundreds of machines.

To make this more concrete, consider the task of checking an addition instruction in the classic 5-stage pipeline illustrated in Fig. 1. This consists of 5 pipeline stages responsible for instruction fetch (IF), decode (ID), execute (EX), memory access (MEM) and writeback of results (WB). Values are read from the register file at the ID/EX boundary and results are written to the register file at the MEM/WB boundary. Forwarding paths (aka bypass logic) are used to reduce the number of stalls by allowing the result of one instruction to be used as an input to the ALU if required by the next instruction. Conventionally, most of the control signals from decode and those that control the pipeline and forwarding paths are not shown — although that is where many of the most difficult bugs lie. We use this simple microarchitecture to explain the technique, Sect. 5 discusses how we adapt the approach to handle more realistic microarchitectures including dual issue, out-of-order retire and register renaming.

Our first challenge is to implement the abstraction function abs which is responsible for converting the micro-architectural state of the processor into an architectural state. To verify an addition instruction, the function abs must extract the current values of the integer registers.

Many simple processors commit their results in order in a single pipeline stage. This means that, at the beginning of the cycle where the add instruction commits, the micro-architectural register file should contain the same values as the architectural register file *before* the add executes and, at the end of the cycle, the micro-architectural register file should contain the same values as the

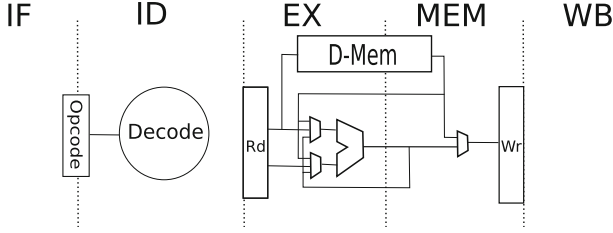


Fig. 1. A 5-stage processor pipeline, with forwarding paths, omitting I-Fetch

architectural register file *after* the add executes. We can therefore obtain the state *before* by reading the state at the end of the writeback stage and the state *after* by reading from the end of the Mem stage.

The other part of the input state of the processor that we require is the opcode of the current instruction. The opcode is normally discarded shortly after instruction decode and is not available at the point where an instruction commits. We therefore need to implement a “pipeline follower” which copies the opcode from one stage to the next and implements the same pipeline stall/flush logic as the datapath. This is similar to the introduction of “ghost state” in Lahiri et al. [13]. The followers and abstraction logic for the pre/post-states are illustrated in Fig. 2

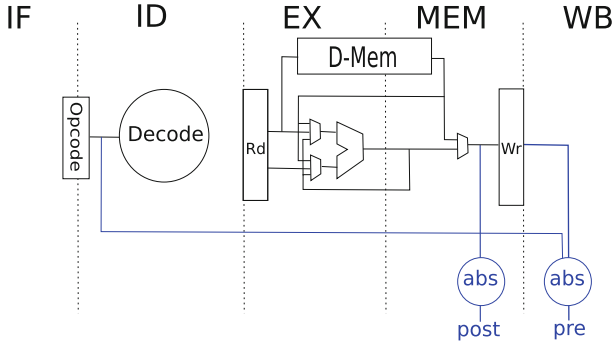


Fig. 2. A 5-stage processor pipeline with state abstraction and follower

Of course, modern ARM processors are considerably more challenging than a simple 5-stage pipeline: Sect. 5 describes the variations on the above approach required to apply ISA-Formal in practice.

Our second challenge is to create a specification of the addition instruction. For any individual instruction, the specification can often be written as a short piece of purely combinational logic. For example, ARM’s 16-bit encoding of the instruction “ADD Rd, Rn, Rm” has opcode $0b0001100 \mid Rm \ll 6 \mid Rn \ll 3 \mid Rd$ and adds the contents of registers Rn and Rm and writes the result to register Rd.

This can be implemented by the following System-Verilog.

```
assign ADD_retiring = (pre.opcode & 16'b1111_1110_0000_0000)
                    == 16'b0001_1000_0000_0000;
assign ADD_result   = pre.R[pre.opcode[8:6]]
                    + pre.R[pre.opcode[5:3]];
assign ADD_Rd       = pre.opcode[2:0];
```

To complete the example, we add assertions that the abstracted result matches the result of the specification when retiring an add instruction.

```
assert property (@(posedge clk) disable iff (~reset_n)
  ADD_retiring |-> (ADD_result == post.R[ADD_Rd]));
```

The above specification is remarkably simple so it is worth examining what kinds of defect this specification could catch.

Decode Errors. Most obviously, this specification would detect any error in instruction decoding. But many decode errors are also caught by other verification methods such as directed or random testing so, at first sight, this does not seem especially useful. However, the instruction decoder is responsible not just for determining how to execute the current instruction but also for setting signals that determine whether it is safe to apply optimizations involving later instructions. A property like the above found a decoder bug involving one such signal that determined whether two adjacent instructions could be fused into a single micro-op: the signal was being incorrectly set for one instruction. This defect had been missed despite extensive testing of the processor: there were tests to ensure that the optimization did happen but testing is ill-suited to checking that it never happens in any other circumstance.

Datapath Errors. An error in a datapath would be caught by this kind of check although, in practice, many errors of this kind are caught by other verification methods already in use.

Interactions between Instructions. Most usefully, and unlike methods based on Burch-Dill flushing, this specification will detect errors caused by interactions between instructions such as errors in the forwarding logic that can supply inputs to this instruction. The example given in the introduction of a sequence of five instructions which triggered an error in the forwarding control logic was detected by a hand-written property like the above. Bugs like this are significantly more important to catch because the forwarding paths vary from one processor to another, the control logic is difficult to get right and the errors are hard to catch by conventional tests.

We currently use bounded model checking which verifies that a sequence of n instructions does not go wrong but to show that any sequence does not go wrong, we would need to find invariants about the processor and use those to get unbounded proofs. Going further, in order to complete ISA verification, we would need to verify that instructions are not lost, duplicated or reordered (we have done this for some processors) and, to complete verification of the core, we would need to verify exception taking mechanisms, the instruction fetch unit and the memory management unit.

4 Generating Verification IP with Architecture Explorer

The main challenge in applying the above approach to a full processor is one of scaling. The ARM v8-M architecture has 384 instruction encodings and the instruction set part of the architecture specification is over 600 pages long [4]; and the ARM v8-A/R architecture has 1280 instruction encodings and is over 2500 pages long [3]. Some of the encodings explicitly disallow using certain registers as sources or destinations to the instructions, many of the instructions are conditional and there are a variety of other complications and corner cases. In addition, changes are regularly added to the architecture specification. All these reasons make the prospect of writing, testing and maintaining a Verilog specification like that shown above unattractive.

Over the last 5 years we have developed tools which transform ARM’s official Architecture Reference Manuals into executable specifications of the v8-A/R and v8-M architectures [17]. A key part of making this specification useful was to test it thoroughly before using the specification to verify anything else. In many ways, this is like Fox and Myreen’s testing of their ARM ISA specification [6] except that we were able to use ARM’s internal architecture conformance test suite (that is normally used to test processors) to test the specifications with billions of instructions that probe each instruction’s corner cases.

The core of this specification is ARM’s Architecture Specification Language (ASL) that grew out of the pseudocode used in earlier versions of the architecture reference manuals. At a high level, ASL is an indentation-sensitive, imperative, strongly typed, first-order language with type inference, exceptions, enumerations, arrays, records, and no pointers. All integers in ASL are unbounded and there is direct support for N-bit bitstrings and functions are allowed to be polymorphic in the width of a bitstring. For example, memory read returns a value of type `bits(8*size)` where `size` is constrained to be 1, 2, 4 or 8.

The task of scaling the ISA-Formal approach up to handle the full instruction sets with all their complexities is therefore one of translating the rich, expressive ASL language to combinational System-Verilog using the synthesizable subset of Verilog that is accepted by commercial Verilog model checkers. The challenge in doing this is that synthesizable Verilog is intended to describe hardware and imposes several limitations upon us; (1) Verilog integers are finite and the bitwidth is a part of the type; (2) Combinational Verilog is normally written in a declarative style with no assignments or control flow and few function calls; (3) Synthesizable Verilog does not support unbounded for-loops or while-loops; (4) Synthesizable Verilog does not support exceptions; (5) The width of bitstrings in Verilog must always be a manifest constant and there is no form of polymorphism over bitwidths of functions.

We were able to overcome the first four issues using relatively conventional compiler techniques. (1) We use a global flow-insensitive value range analysis to compute the required width of most integer variables and use a large, but safe bound for any integers with unknown range. (2) Verilog includes a rarely used procedural subset which most of the language can be translated into. (3) User-supplied bounds on loops can be used to unroll all loops. (4) A whole-program

transformation which adds additional flags and control flow to make exception and return-related control flow explicit.

The most challenging problem was dealing with bitstring polymorphism. Virtually all polymorphism was caused by instructions which could operate on data of different widths such as 8, 16, 32 or 64-bit load instructions. This observation enabled us to eliminate almost all polymorphism by automatically specializing such instruction encodings to create a separate instruction for each data width and then to use alternate passes of constant propagation and a “monomorphization” pass which identifies calls to polymorphic functions where the bitwidth is a manifest constant and replaces the call with a call to a monomorphic instance of the polymorphic function. The remaining polymorphism is handled by a set of ad-hoc transforms in the Verilog backend.

5 Applying ISA-Formal to CPUs

In practice, few processors are as simple as the 5-stage pipeline sketched in Fig. 1 and we have had to develop a number of techniques in writing abstraction functions to deal with complex functional units, out-of-order retire, dual issue pipelines, instruction fusion, and register renaming.

5.1 Complex Functional Units

For the most part, our end-to-end approach to verification works: commercial model checkers are able to handle the complexity of most components without assistance. However, for complex functional units such as floating point and the memory system we choose to use other more scalable verification techniques such as the end-to-end memory-system verification technique described by Stewart et al. [20]. This modular approach lets ISA-Formal verification focus on control logic and forwarding paths that controls, feeds and is fed by these complex units.

In order to make ISA-Formal modular, we partition the specification on function call boundaries into different parts “Instruction Set Architecture (ISA),” “Floating Point,” “Exception,” “Address Translation,” etc. and only generate Verilog for the “ISA” part. Any functions on the interfaces to other partitions are written by hand and many are just a few lines long: returning some component of the result of the pre-state or changing some component of the post-state.

On the interfaces, we adopt a variety of approaches to filling the resulting gaps in the generated Verilog using interface properties, subset behaviour checking and abstract functions. In general, these approaches will prevent us from detecting bugs in some parts of the processor using ISA-Formal. We tackle this by tracking which parts of the processor are not being checked by ISA-Formal and ensuring that an alternative verification technique is used on those parts.

Interface Properties. For some components such as the memory system, we were already creating interface specifications which were sufficiently strong that we could use the interface specification instead of the memory system. This only

required us to convert the architectural view of the memory system to the micro-architectural view by translating requests/responses between representations.

Subset Behaviour Checking. For components such as floating point units, a specification of the full behaviour would still be too complex to use in verification but is quite simple if we restrict ourselves to a subset of the full behaviour. For example, if we restrict the inputs to $\pm\{0, 1, \infty, S\text{-NaN}, Q\text{-NaN}\}$ then it is easy to create specifications of all the FP instructions for this subset and perform some verification. Obviously, this would not be sufficient to detect errors in the floating point unit itself, but this subset gives enough different values that errors in the control and forwarding logic can be detected.

We could use SystemVerilog assumptions to restrict inputs to the chosen set of inputs, but this would restrict all of the checks that ISA-Formal performs on instructions: whether the instruction sets condition flags, raises an exception, accesses memory, which registers are written, etc. Instead, we add an additional signal indicating whether the inputs are in the supported subset and use that signal only to restrict checks of the values written to floating point registers.

Abstract Functions. The final option is to use the processor as an oracle. That is, we add logic to track the inputs and outputs from some functional unit and then use the output value if the inputs of a function in the architectural specification match the actual inputs of a functional unit in the processor. Since we are choosing to trust the behaviour of that unit, this cannot detect errors in the unit but it can detect errors in the surrounding control and forwarding logic.

5.2 Out of Order Completion

In an in-order core, all instructions retire strictly in-order, but some slower instructions may complete out of order. Retiring a load (say) after the memory protection check but before the data returns from the memory system allows independent instructions to continue without waiting for the access to complete. Such optimizations are important to verify because they introduce difficult corner cases in the design such as ensuring that the result of the load is written back even if the processor takes an exception.

The difficulty in verifying out-of-order completing instructions is that it is hard to construct the post-state: by the time that the load instruction completes, some of the instructions issued *after* load will also have completed. This is further complicated because some load instructions may be split into multiple micro-ops which complete independently.

Our solution to this is to take a snapshot of the pre-state when the load instruction *retires*. As each micro-op for the instruction under test completes, the snapshot is updated with the change. Finally, when the last micro-op completes, the final post-state is available and the instruction can be checked against the architecture specification.

5.3 Dual Issue Pipelines

Dual issue pipelines decode and execute two consecutive instructions in parallel. To handle dual issue pipelines, we add a further abstraction function to extract the intermediate state between execution of the two instructions. Our initial approach to checking these was to create two copies of the combinational logic implementing the specification: one copy for each pipeline. This worked but consumed a lot of memory and would scale badly for 3 or more-issue processors so, instead, we use a single copy of the specification and insert multiplexors to select which pre/post state is used with the specification.

The most serious problem encountered occurs if the second instruction can suppress part of the behaviour of the first instruction. For example, if both instructions modify the carry flag, then the final value written will be the result of the second instruction. In this case, the carry flag value from the first instruction may not be available at the writeback stage and we need to identify the correct signal to use and add a pipeline follower to propagate the value down to the point of serialization. Any error in choice of signal is detected when that signal is used as part of the pre-state of the second instruction.

5.4 Instruction Fusion

A high-performance processor might wish to fuse commonly occurring pairs of consecutive instructions into a single instruction. For example Malik et al. [15] describes a processor that detects sequences of dependent ALU instructions such as

```
SUB R4, R1, R2    ; R4 := R1 - R2
ADD R4, R4, R3    ; R4 := R4 + R3
```

and fuses them into a single macro-operation that reads three inputs from the register file and performs two add/subtract operations.

Optimizations of this kind raise a potential problem in sequences where the results of the first instruction are overwritten by the second instruction because the processor may not calculate the post-state of the first instruction or the pre-state of the second instruction.

Our solution is to add additional verification logic to calculate the missing intermediate state. The correctness of this logic is verified when checking that all uses of the SUB instruction (i.e., the first instruction of the pair) is correct and that justifies use of the result when checking that the SUB/ADD fused pair (i.e., the first/second instruction pair) gives the correct overall result.

5.5 Register Renaming

Processors with out-of-order instruction issue differ significantly from processors with in-order issue because they speculatively execute instructions past branch instructions. To allow them to recover from mis-speculation, they use a register rename table that maps architectural registers such as “X0” to one of a

large pool of physical registers. As instructions are decoded, source registers are “renamed” using this table; free physical registers are allocated and the rename table is updated with mappings from destination register names to these physical registers. Instructions typically execute as soon as their input dependencies are satisfied but, to preserve the illusion that instructions execute in program order, a reorder buffer (ROB) only commits instructions in program order.

Despite the added complexity of speculative execution, register renaming and reorder buffers, it is actually simpler to apply ISA-Formal to out-of-order processors because they have a single clearly identified point of serialization implemented in the reorder buffer. In contrast, in-order processors have a variety of different mechanisms to support a limited degree of out-of-order execution such as varying pipeline length or supporting out-of-order completion of slow instructions and these different mechanisms are scattered across the processor.

5.6 Debugging Abstraction Functions

From the above, it should be apparent that creating the abstraction code remains a difficult task and involves a lot of work with the CPU designers to get right. While debugging these abstraction functions, we have found that it is useful to start by using hand-written properties like those described in Sect. 3 for instructions that touch the major parts of the processor. For example, a data-processing instruction, a load, a store, a floating point move, etc.

It is significantly easier to debug the abstraction function using hand-written specifications than using a mechanical translation from the specification. Once we have debugged the abstraction functions, we switch to using the machine-generated specifications exclusively, and rarely look at the generated code.

5.7 Handling Known Problems

One of the major difficulties we experienced before developing ISA-Formal was that formal verification tools would report variations on the same defect over and over again. This was a problem early in development when we might know that part of the processor was missing or incomplete; and it is a problem at any stage that once the bug report has been filed, the verification team wants to focus on finding other problems until the bug has been dealt with.

A critical technique for handling known problems is to maintain a list of assumptions corresponding to each individual bug or feature. As each bug is fixed, we remove the corresponding assumption and confirm that the bug has been fixed. Using assumptions is a simple technique but it greatly increases our ability to use formal verification to detect errors early in development and it very effectively decouples processor design from verification allowing the tasks to proceed in parallel.

6 Results

This section describes the results of applying ISA-Formal in three small-scale trials and five full-scale uses. These eight trials and uses cover the full lifetime of

ARM processor developments; they cover both application processor targetted at mobile phones, etc. and microcontrollers targetted at embedded uses; and they cover micro-architectures ranging from 3-stage, in-order pipelines through dual-issue, in-order pipelines to out-of-order pipelines.

6.1 ARM’s Development Phases

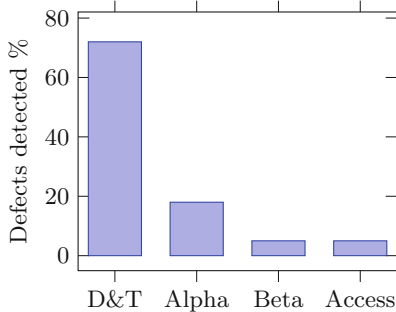
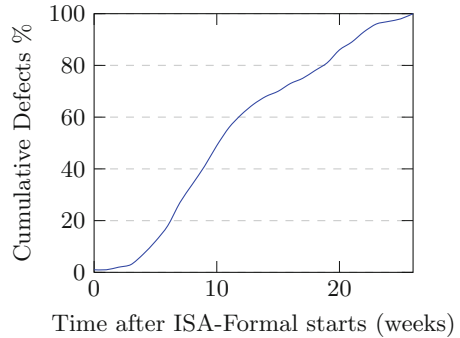
ARM’s development process involves four stages of roughly equal length: Develop and Test (D&T), Alpha, Beta and Access. The goal of each stage is to create a basic pipeline design in D&T; make it feature complete by the end of Alpha; improve power, performance and area through Beta; and to improve confidence in the design through the access period where the design is made available to the lead partners for that processor for evaluation and feedback. Testing steadily increases throughout this process and each stage applies roughly an order of magnitude more testing than the previous stage.

6.2 Small-Scale Trials

We carried out three small-scale trials on processors that were already in the access phase to demonstrate the ability of ISA-Formal to detect defects that were hard to detect by other means. These trials consisted of developing hand-written properties like those described in Sect. 3 and demonstrated the ability to detect defects that had been found by other means as well as new defects.

The defect described in the introduction is an example of a bug we detected during this trial process. The trigger sequence of the defect is conditional execution of instructions executing in two pipeline stages with a combination of taken and not-taken instructions. In a 2-pipeline design, the size of the smallest trigger sequence is 5 instructions: one to set up the condition, two (one per pipe) to generate values that might be forwarded, and two (one per pipe) to consume forwarded values. (There are several variations on that basic pattern.) Using traditional simulation-based verification, patterns like this would have to be tested on all combinations of instructions that have forwarding paths between them in that particular micro-architecture and each processor will have a different set of forwarding paths. There are many, many sequences of instructions like this to be tested so defects of this form are typically only found during soak-testing during the Access phase. Using ISA-Formal, we created hand-written properties for one or two instructions corresponding to each major unit in the datapath (the ALU, shifter, multiplier, etc.), we created abstraction functions for each of the two pipelines, and, since we left the opcode received from the fetch unit unconstrained, the commercial bounded-model-checker explored sequences of instructions up to some bound. We ran about a dozen properties through the model checker and after two minutes proof time detected the failing trigger sequence.

The same experience was repeated on all three processors: bugs were found with relatively little effort with the bulk of the work being done by junior engineers supervised by formal experts and with input from the microarchitects.

**Fig. 3.** Defect detection by phase**Fig. 4.** Defect detection by time

The consistent combination of low human effort and low machine effort was an important part of demonstrating that ISA-Formal could detect difficult defects that, at best, would have been caught only during the Access phase.

6.3 Production Usage

Based on the success of the small-scale trials, ARM decided to adopt ISA-Formal as part of the formal verification strategy on five processors that were in earlier stages in their development: three in D&T, one in Alpha and one in Access. This work used the tool described in Sect. 4 to generate Verilog for all instructions directly from ARM’s official Architecture Reference Manuals allowing engineers to focus on developing abstraction functions and testing the processor.

Defects have been found in all five processors with the distribution roughly in proportion to the effort invested in that processor. The small-scale trials had demonstrated that ISA-Formal can detect difficult to detect defects late in processor development; the production usage demonstrated that ISA-Formal is effective at detecting defects in earlier phases of development. Figures 3 and 4 show the distribution of confirmed, distinct defects detected using ISA-Formal by phase and by time. Figure 3 shows that ISA-Formal is capable of catching many defects early in development (overcoming the problem of being able to find many distinct defects in parallel with development) and that it is capable of finding defects late in development even after extensive testing by other methods. Figure 4 shows that ISA-Formal is able to start detecting defects in just a few weeks work and continues to find bugs as processors are developed.

We also found that ISA-Formal was able to detect issues affecting all areas of the instruction set: FP/SIMD, Memory, Branches, Integer, Exceptions and System instructions (e.g., memory fence instructions). Figure 5 shows the distribution of bugs found by ISA-Formal by the area of the processor affected (combining results for all processors). (The “Integer” category includes both integer datapath instructions and basic pipeline control issues — it is often hard to separate the two since integer instructions are so fundamental to a processor.)

FP/SIMD	25%
Memory	21%
Branch	21%
Integer	18%
Exception	8%
System	7%

Fig. 5. Defect detection by area

Processor	Lines of code
#1	2400
#2	2250
#3	4600
#4	1000
#5	2500

Fig. 6. Size of verification code

It is encouraging to note that the two largest sources of detected bugs were FP/SIMD instructions and memory instructions. As Sect. 5.1 explains, we do not test the FPU or the memory subsystem but, despite this, we are still able to test and find defects in the forwarding, pipeline control and register logic connected to these units.

The effort of creating, testing and debugging the machine-readable specification and a tool to translate it to Verilog is considerable but can be shared across multiple processors and can be used for other purposes within the company (e.g., documentation, testing of architecture extensions, etc.). The primary cost of implementing ISA-Formal on a new processor is the effort required to implement the pipeline follower and abstraction function on each processor. As a rough indication of the effort required, Fig. 6 shows the number of lines of code required for each (anonymized) processor. Most processors need around 2,500 lines of support code: a fairly modest cost. The outliers are processor #4 which has not yet added a follower for floating point registers and processor #3 which is a more complex processor than the other four.

Beyond the bug numbers, we found that applying ISA-Formal early in the development was capable of finding bugs that would not normally be caught until much later. For example, very early in development of an out-of-order processor, ISA-Formal found a bug that occurred when all the free registers in the physical register pool were in use. This was found before the processor could even execute load-store instructions so we would not normally be catching such bugs that early.

7 Conclusions

Two barriers to widespread industry adoption of formal verification techniques to check processors are scaling and return on investment issues. The end-to-end approach to verification that we adopt tackles both issues: it allows machine-generation of verification IP from the architecture specification, it allows engineers to detect bugs that affect actual instruction sequences very early in deployment, and it encourages creation of reusable tools, techniques and IP that can be used across an unusually wide range of micro-architectural styles.

This paper describes the steps needed to turn the basic idea into a scalable, reusable technique: automation, dealing with a range of different micro-architectural design techniques, and initial bringup issues. We have applied this

method to 8 different ARM processors spanning all stages of development up to release. In all processors, this has found bugs that would have been hard for conventional simulation-based methods to find and ISA-Formal is now a key part of ARM's formal verification strategy.

To the best of our knowledge, this is the most broadly applicable formal verification technique for verifying processor pipeline control in mainstream commercial use.

References

1. Aagaard, M.D., Cook, B., Day, N.A., Jones, R.B.: A framework for microprocessor correctness statements. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 433–448. Springer, Heidelberg (2001). <http://dl.acm.org/citation.cfm?id=646705.702043>
2. Aagaard, M.D., Jones, R.B., Melham, T.F., O'Leary, J.W., Seger, C.-J.H.: A methodology for large-scale hardware verification. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 300–319. Springer, Heidelberg (2000)
3. ARM Ltd: ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile). ARM Ltd (2013)
4. ARM Ltd: (In Preparation) ARM Architecture Reference Manual (ARMv8, for ARMv8-M architecture profile). ARM Ltd (2016)
5. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 68–80. Springer, Heidelberg (1994). <http://dl.acm.org/citation.cfm?id=647763.735662>
6. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the ARMv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-14052-5_18](https://doi.org/10.1007/978-3-642-14052-5_18)
7. Higgins, J.T., Aagaard, M.D.: Simplifying design and verification for structural hazards and datapaths in pipelined circuits. In: Ninth IEEE International Proceedings of the High-Level Design Validation and Test Workshop, HLDVT 2004, pp. 31–36 (2004). <http://dx.doi.org/10.1109/HLDVT.2004.1431229>
8. Hunt Jr., W.A., Sawada, J.: Verifying the FM9801 microarchitecture. IEEE Micro **19**(3), 47–55 (1999). doi:[10.1109/40.768503](https://doi.org/10.1109/40.768503)
9. Jhalal, R., McMillan, K.L.: Microarchitecture verification by compositional model checking. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 396. Springer, Heidelberg (2001). doi:[10.1007/3-540-44585-4_40](https://doi.org/10.1007/3-540-44585-4_40)
10. Kaivola, R., et al.: Replacing testing with formal verification in Intel[®] Core[™] i7 processor execution engine validation. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 414–429. Springer, Heidelberg (2009). http://dx.doi.org/10.1007/978-3-642-02658-4_32
11. KiranKumar, V., Gupta, A., Ghughal, R.: Symbolic trajectory evaluation: the primary validation vehicle for next generation Intel processor graphics FPU. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 149–156. IEEE (2012)
12. Kroening, D., Paul, W., Mueller, S.: Proving the correctness of pipelined micro-architectures. In: Waldschmidt, K., Grimm, C. (eds.) Proceedings of ITG/GI/GMM-Workshop “Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen”, pp. 89–98. VDE Verlag (2000)

13. Lahiri, S.K., Bryant, R.E.: Deductive verification of advanced out-of-order microprocessors. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 341–354. Springer, Heidelberg (2003). doi:[10.1007/978-3-540-45069-6_33](https://doi.org/10.1007/978-3-540-45069-6_33)
14. Lahiri, S.K., Pixley, C., Albin, K.: Experience with term level modeling and verification of the M*CORETM microprocessor core. In: Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop 2001, Monterey, California, USA, 7–9 November 2001, pp. 109–114 (2001). <http://dx.doi.org/10.1109/HLDVT.2001.972816>
15. Malik, N., Eickemeyer, R.J., Vassiliadis, S.: Interlock collapsing ALU for increased instruction-level parallelism. In: Proceedings of the 25th Annual International Symposium on Microarchitecture, pp. 149–157. MICRO 25, CA (1992). <http://dl.acm.org/citation.cfm?id=144953.145794>
16. McMillan, K.L.: Verification of an implementation of Tomasulo’s algorithm by compositional model checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 110–121. Springer, Heidelberg (1998). <http://dl.acm.org/citation.cfm?id=647767.733764>
17. Reid, A.: Creating trustworthy specifications of ARM v8-A and v8-M system level architecture. In: preparation (2016)
18. Slobodová, A., Davis, J., Swords, S., Hunt Jr., W.: A flexible formal verification framework for industrial scale validation. In: 2011 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 89–97. IEEE (2011)
19. Srinivasan, S.K.: Automatic refinement checking of pipelines with out-of-order execution. IEEE Trans. Comput. **59**(8), 1138–1144 (2010)
20. Stewart, D., Gilday, D., Nevill, D., Roberts, T.: Processor memory system verification using DOGR_{EL}: a language for specifying end-to-end properties. In: International Workshop on Design and Implementation of Formal Tools and Systems, DIFTS 2014 (2014)
21. Windley, P.J.: Formal modeling and verification of microprocessors. IEEE Trans. Comput. **44**(1), 54–72 (1995)

Chapter 4

Who guards the guards? Formal Validation of the ARM v8-M Architecture Specification (**Paper III**)

Alastair Reid. Who guards the guards? Formal validation of the ARM v8-M architecture specification. *In Proceedings of the ACM on Programming Languages, volume 1 of OOPSLA 2017, New York, NY, USA, October 2017.* ACM.

© 2017 Alastair David Reid.

doi: <https://dx.doi.org/10.1145/3133912>.

Who Guards the Guards? Formal Validation of the Arm v8-M Architecture Specification

ALASTAIR REID, Arm Ltd, United Kingdom

Software and hardware are increasingly being formally verified against specifications, but how can we verify the specifications themselves? This paper explores what it means to formally verify a specification. We solve three challenges: (1) How to create a secondary, higher-level specification that can be effectively reviewed by processor designers who are not experts in formal verification; (2) How to avoid common-mode failures between the specifications; and (3) How to automatically verify the two specifications against each other.

One of the most important specifications for software verification is the processor specification since it defines the behaviour of machine code and of hardware protection features used by operating systems. We demonstrate our approach on ARM's v8-M Processor Specification, which is intended to improve the security of Internet of Things devices. Thus, we focus on establishing the security guarantees the architecture is intended to provide. Despite the fact that the ARM v8-M specification had previously been extensively tested, we found twelve bugs (including two security bugs) that have all been fixed by ARM.

CCS Concepts: • **Computer systems organization** → *Architectures*; Reduced instruction set computing; • **Hardware** → *Theorem proving and SAT solving*; • **Software and its engineering** → *Consistency*; *Software verification*; *Formal software verification*;

Additional Key Words and Phrases: ISA, Specification, Formal Verification

ACM Reference Format:

Alastair Reid. 2017. Who Guards the Guards? Formal Validation of the Arm v8-M Architecture Specification. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 88 (October 2017), 24 pages. <https://doi.org/10.1145/3133912>

1 INTRODUCTION

The last decade has seen formal verification techniques scaling to the point where it is possible to formally verify realistic compilers [Leroy 2009], operating system kernels [Klein et al. 2009], hypervisors [Dam et al. 2013] and processors [Reid et al. 2016]. These efforts are impressive but we must beware that the correctness of their proofs ultimately rests on the correctness of the specifications they depend on. This is worrying because these specifications are, themselves, large and complex artifacts with all the risks of bugs that we expect in large, complex software. This risk is only likely to increase as more effective formal verification techniques and tools allow larger, more complex projects to be verified against larger, more complex specifications.

Bugs in specifications are not just a theoretical possibility. In previous work [Reid 2016], we reported that correcting errors in the ARM v8-A Architecture Reference Manual [ARM Ltd 2013] resulted in changes to 12% of the lines of code in ARM's processor specification. The CompCert compiler [Leroy 2009] required a bugfix *despite being formally verified* and the bug can be traced to the architecture specification not describing the full behaviour of an instruction [CompCert 2016]. In an empirical study of the correctness of three formally verified distributed systems [Fonseca

Author's address: Alastair Reid, Arm Research, Arm Ltd, 110 Fulbourn Road, Cambridge, CB1 9NJ, United Kingdom, alastair.reid@arm.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2017 Copyright held by the owner/author(s).

2475-1421/2017/10-ART88

<https://doi.org/10.1145/3133912>

et al. 2017], no protocol bugs were found in the verified systems but 16 bugs were found in the Trusted Computing Base (i.e., the unverified glue code, build system, and specifications used to build the code and in the proof) including two bugs in the specifications. If we are to trust the guarantees claimed for formally verified software, it is essential that we verify the large, complex specifications on which our formal correctness claims are founded.

Three common ways that bugs are found in specifications are by testing specifications against existing implementations [Flur et al. 2016; Fox and Myreen 2010; Goel et al. 2014]; by testing specifications using test suites used to test implementations [Reid 2016]; or as a side effect of attempting to formally verify an implementation against a specification [Reid et al. 2016]. Unfortunately, these approaches can still miss bugs either because test suites are incomplete or because of common mode failure (i.e., the specification and the implementation do the same wrong thing).

This situation is bad for programmers relying on specifications because, no matter how careful they are, they are reliant on the quality of the specification available to them. For example, Dunlap [Dunlap 2012] describes a bug in the Xen hypervisor that arose because of an inconsistency between the Intel and AMD specifications of the SYSRET instruction allowing a privilege escalation when run on Intel processors. The problem is that neither AMD’s specification of x86-64 nor Intel’s specification of x86-64 fully captures the range of implementations of the architecture.

The situation is also bad for architects extending specifications. Reliance on test suites or verification against implementations creates a “chicken and egg” problem because implementors and test writers do not want to work on unstable, incomplete specifications but architects want to test changes to the specification while they are still developing the changes.

Our solution to this problem is to write high-level properties about the specification and to formally verify that the specification satisfies those properties.

One of the most important specifications that formal verification of software depends on is the processor specification that defines the boundary between software and hardware and on which formal proofs about the entire software and hardware stack are founded. In this paper, we consider properties about ARM’s v8-M architecture specification [ARM Ltd 2016] that extends ARM’s microcontroller specification with additional security features that software can use to improve the trustworthiness of Internet of Things devices.

We focus on cross-cutting features of the architecture and specify properties of the architecture as a whole involving exceptions, privilege and security.

Cross-cutting features [Kiczales et al. 1997] are difficult for humans because they require understanding of interactions between many disparate parts of the architecture and so subtle errors can slip through the cracks. The flip side of this is that writing cross-cutting properties can also scale well: a single cross-cutting property can catch an error in many parts of the architecture. We do not attempt to state properties about more cleanly decomposable parts of the architecture such as whether an ADD instruction performs addition. We believe that these properties are adequately served by existing techniques and that they would not give the same degree of leverage as our cross-cutting properties.

We developed sets of properties by examination of natural language text in the ARM architecture reference manual, by examining recently discovered bugs in the specification, and by discussion with the architects of the specification.

The central design challenge we face is to create a set of properties that:

- express the major guarantees that programmers depend on;
- are concise so that architects can easily review and remember the entire set of properties;
- are stable so that architecture extensions don’t invalidate large numbers of rules;

- and that describe the architecture differently from the existing specification to reduce the risk of common-mode failure.

The ARM formal specification is split into many functions all rooted in a single top-level transition function that specifies any state change that the processor can make due to executing instructions, taking interrupts, etc. The classic compositional approach would be to tackle the problem hierarchically: stating and proving properties about the little functions at the bottom of the call tree of the specification, then using these properties as the basis for proofs about the functions in the next layer up the call tree and gradually working our way up the tree until all the properties of the specification have been stated and proved.

This conventional approach has a number of problems:

- *It is not how the architects view the architecture.* The architects describe the architecture at a high level in terms of the net effect of the architecture using statements like “if X happens then Y will happen” or “Y cannot happen unless X is enabled.” Walking through the call tree or the individual lines of code in the specification is a secondary activity during their internal discussions. This observation is reflected in the natural language part of ARM’s architecture specification and, we believe, reflects the way the architects think about the specification. Our experience is that, *when dealing with domain experts, there is significant benefit from formalizing their view of the domain instead of forcing them to use a different view.*
- *It does not aid understanding.* The problem we face in gaining confidence that the ARM specification is correct stems from all the fine detail in the existing specification: it is hard to see the forest for the trees. Adding properties to each function continues this problem: we better understand each tree but we still cannot see the forest.
- *It increases the maintenance burden.* Annotating every function with what would typically be multiple preconditions and postconditions would require significant specification and review effort. In addition, the internal structure of the specification is less stable than the boundary of the specification: functions are refactored, function arguments and results are added or removed, etc. The more that is written and proved about each individual function, the more there is to update as the specification evolves.

Accordingly, we adopt an “end to end” approach to writing properties: we only write properties that apply to the whole system. To make this practical, we extended conventional specification techniques based on predicates over the state with a novel kind of property inspired by coverage-based testing techniques.

Our verification is based on translating the specification plus the properties into verification conditions that an SMT solver can check. This allows the verification process to be entirely automated and requires no expert intervention.

The specification had previously been extensively tested [Reid 2016; Reid et al. 2016] but, despite this, we found a dozen bugs including two security bugs. Due to extensive test suites used in ARM processor development and to redundancy between the natural language part of the specification and the formal part of the specification, these bugs had not impacted processors but they are important to anyone verifying software or hardware against the formal part of the specification.

To our knowledge, no realistic architecture specification has been subjected to this degree of formal verification before.

The remainder of this paper is structured as follows: Section 2 describes the coverage properties we use to write end-to-end properties; Section 3 sketches the ARM microcontroller architecture, describes how ARM writes architecture specifications and provides an introduction to how we write end-to-end properties; Section 4 further illustrates our approach with examples; Section 5 describes the design and implementation of our system; Section 6 describes our experience of using

the system; Section 7 describes related work; Section 8 describes limitations and future work; and Section 9 concludes.

2 COVERAGE PROPERTIES

The classic approach to writing properties is to write invariant properties and function properties using predicates that refer only to the state of the system before and after the state transition function. Our decision to limit ourselves to writing end-to-end properties makes it very hard to capture key properties only in terms of states. For example, some properties will only apply if the system takes a certain kind of transition such as taking a reset or an exception but these are hard properties to observe from the state alone. In principle, we could reverse engineer the initial conditions under which these events could occur but then we would be reasoning about when we *think* certain transitions occur instead of what the specification actually says.

Our solution to this problem takes its inspiration from the approach used in coverage-based testing techniques that use measurement of the coverage of the system under test to determine whether the tests are hitting the relevant parts of the system. For example, many programmers have added an ad-hoc “debug printf” to a program to confirm that a test hits some line of code or some condition. More rigorous applications of this approach are built into hardware design languages such as System Verilog [IEEE 2013] that provides a rich set of functional coverage mechanisms for tracking how many tests hit each case or combination of cases. Inspired by these test-based mechanisms, we augment the traditional Hoare-style properties about states with the ability to observe execution paths. The property $CALLED(f)$ is satisfied for any execution of the function under test that calls the function f . For example, to state that an exception causes register $R[0]$ to be set to zero, one could write the following (where `ExceptionEntry` is the name of the function that is called when an exception is taken).

$$CALLED(ExceptionEntry) \Rightarrow R[0] = 0$$

In some cases, finer-grained observation is important and we specify an additional predicate P that tests the values of the parameters when a function is called. For example, the function `ExceptionEntry` has a boolean parameter `isSecure` that specifies whether a secure or non-secure exception should be taken so the property

$$CALLED(ExceptionEntry \text{ when } isSecure) \Rightarrow R[0] = 0$$

weakens the statement to say that $R[0]$ is set to zero for *secure* exceptions.

Similarly, it is useful to write properties about function return and the values returned. We write $RETURNED(f \text{ when } P)$ to say that a function f returned successfully with values that satisfy the predicate P . (ARM’s specification language includes exceptions so it is possible for a function to be called but not to return.)

We feel that observing execution paths in this way is a satisfactory compromise on our commitment to writing end-to-end properties: we mostly focus on the overall properties of the architecture but we allow references to some of the inner structure of the specification when required. In practice, we find that only a small fraction of the functions need to be observed in this way.

3 FORMALIZING ARM SPECIFICATIONS

The Internet of Things (IoT) adds network access to microcontroller-based systems: a class of devices that are small, cheap and energy efficient but not previously required to be secure. To meet this challenge, ARM created the “M-class” of processor that retains the positive characteristics but adds extensive security features. This does not eliminate the IoT security challenge but it gives software developers a sound foundation to build on.

The challenge in designing security features is that security is asymmetric: the designer has to get everything right but the attacker only has to find a single weakness to gain access. This makes the design and programming of Internet of Things devices appealing targets of formal verification research.

This section gives a brief introduction to the essential aspects of ARM's microcontroller specification referred to in this paper, describes how ARM writes architecture specifications and provides an introduction to their formalization.

3.1 The ARM v8-M Security, Privilege and Exception Model

ARM's v8-M architecture specification applies to ARM's 32-bit microcontrollers such as the recently announced Cortex-M23 and Cortex-M33 processors that are designed for embedded, low-cost devices at different performance points with a focus on security. Our focus in this paper is on writing and proving properties about specifications but, in order to explain the examples, it is necessary to describe a few key architectural concepts.

- An *exception* can be triggered by memory protection faults, security faults, interrupts, etc. If the cause of the exception was a fault then an appropriate field of the *Fault Status Register* is set to 1. Each exception has a priority and the processor selects the highest priority exception to work on. On taking an exception, the processor automatically saves the current context (user registers) onto the current stack and reads the address of the exception handler from an exception vector table in memory.
- A *derived exception* occurs if the act of taking an exception triggers a further exception. Two particular ways that derived exceptions occur are if saving the current context to the stack triggers a fault such as a stack overflow or if reading the exception vector triggers a fault. A cascade of up to two derived exceptions can result from an initial exception.
- *Lockup* occurs if a derived exception has lower priority than the original exception since there is then no way to report the derived exception. When in lockup, the processor sets the program counter to a distinctive lockup address and stops executing instructions.
- A processor can be in *Privileged* or *Unprivileged* mode. *Privilege* corresponds to the traditional protection mechanisms used by operating systems: only privileged execution mode is allowed to access system registers (including the memory protection registers).
- Orthogonally, a processor can be in *Secure* or *NonSecure* mode. The two modes share user registers but the stack pointer and many of the system control and status registers are *banked*: there are two copies and which copy is accessed depends on the current mode. *Security* goes beyond traditional processor-based protection and enforces access checks in peripherals and memory devices so that when a DMA controller or processor is executing in non-secure mode they cannot access secure peripherals or memory containing secrets such as crypto keys.
- An external debugger may request that the processor *halt* and can then examine and modify the processor and memory state. When the processor is halted, it is said to be in *Debug State*. It is possible to disable debugging of the processor when it is in Secure mode.

Inclusion of all these features makes the architecture more complex than a classic RISC architecture. There are multiple motivations for these features ranging from optimisations that can be performed in stacking/unstacking registers that make interrupt response faster and more deterministic; enabling interrupt handlers to be written in plain C code; and adding security features. It also means that some of the corner cases arising from the interaction of features only have to be handled correctly once by the hardware designers instead of having to be handled in many different software stacks. However, the combination of four different privilege/security modes, priority,

derived exceptions, debug, lockup and security adds considerable complexity to the architecture that makes testing and formal verification of the architecture specification desirable.

3.2 ARM's Specification Language

ARM's architecture specifications consist of two parts: a detailed, executable formal specification and a natural language part.

The formal part of ARM's specifications is written in ARM's Architecture Specification Language (ASL) that grew out of the pseudocode used in earlier versions of architecture reference manuals. At a high level, ASL is an indentation-sensitive, imperative, strongly-typed, first-order language with type inference, exceptions,¹ enumerations, arrays, records, and no pointers. All integers in ASL are unbounded and there is direct support for N-bit bitvectors and functions are allowed to be polymorphic in the width of a bitvector. For example, memory read returns a value of type `bits(8*size)` where `size` is constrained to be 1, 2, 4 or 8.

To make this more concrete, here is a small example of an ASL function that is called when a processor exception is triggered. The function pushes the current state onto the stack and, if this does not trigger a memory access fault, it calls the function `ExceptionTaken` that adjusts registers, swaps stacks, reads the address of the exception handler from memory and branches to it.

```
ExclInfo ExceptionEntry(integer exceptionType, boolean toSecure, boolean commitState)
    // PushStack() can abandon memory accesses if a fault occurs during the stacking sequence.
    exc = PushStack(commitState);
    if exc.fault == NoFault then
        exc = ExceptionTaken(exceptionType, FALSE, toSecure, FALSE);
    return exc;
```

The ARM v8-M formal specification is over 15,000 lines of code consisting of over 300 instructions and over 250 functions. This makes it one of the largest formal specifications we are aware of. The most important functions in the specification are: (1) The function that defines the initial state of the system. This function is called `TakeColdReset` and it specifies how the processor performs a “cold” reset (i.e., when first powered up). (2) The transition function. This function is called `TopLevel` and it specifies all types of transition that the specification can make: instruction fetch, instruction execute, entering and returning from processor exceptions, warm reset, entering/leaving Debug State, etc.

3.3 Rule Based Specification

ARM's architecture reference manuals also contain natural language statements about the architecture. Starting with the v8-M Architecture Reference Manual [ARM Ltd 2016], these are structured into a number of labelled “rules.” Labels begins with the letter “R” for normative statements and with the letter “I” for informative statements and are followed by four randomly chosen letters.

We found that many of these rules simply repeated information found in the formal specification in much the same structure as the formal specification. These were not very useful for our purposes because they were somewhat low level and, worse, they were prone to common-mode failure wrt the formal specification. However, a small number of the rules stated high level properties about the architecture. For example, the following rule describe properties of how a processor can exit the Lockup state.

¹The presence of “exceptions” in both the processor architecture and in ASL can lead to confusion as to which kind of exception we are referring to. In the remainder of this paper, we use “processor exception” and “ASL exception” to avoid confusion.

R_{JRJC}

Exit from lockup is by any of the following:

- *A Cold reset.*
- *A Warm reset.*
- *Entry to Debug state.*
- *Preemption by a higher priority processor exception.*

We interpret this to mean that the *only* way to exit from a lockup state is if one of the four listed conditions occurs. By examining the ASL specification, we found tests that could be used to formalize this statement:

- The variable `LockedUp` indicates whether the processor is in lockup.
- A cold reset is specified by `TakeColdReset` and a warm reset is specified by the function `TakeReset`;
- The variable `Halted` indicates whether the processor is in Debug state.
- Taking a processor exception is initiated by the function `ExceptionEntry`.

Based on this, one could choose to formalize the original statement in the following Hoare-triple

```
{ Invariants ∧ LockedUp }
TopLevel();
{ ¬ LockedUp ⇒ CALLED(TakeColdReset) ∨ CALLED(TakeReset) ∨ (¬Halted' ∧ Halted) ∨ CALLED(ExceptionEntry) }
```

where `Invariants` is the conjunction of all the invariants for the system and `Halted'` represents the value of `Halted` before the function is called. (This omits the requirement that preemption must be by a higher priority exception. This is a general requirement on all processor exceptions and we chose to specify it in a separate property.)

Even for this simple rule, we found this notation to be quite unwieldy so we introduced some syntactic sugar to let us write properties in a more structured way.

- Each property is labelled for ease of reference.
- Instead of using the v' convention for accessing the old value of a variable v , we provide an operator $PAST(e)$ that refers to the value of an expression e before the function under test was called.
- Following the example of System Verilog Assertions [IEEE 2013], we define syntactic sugar for some common uses of the $PAST$ operator.

$$STABLE(e) \triangleq PAST(e) = e$$

$$CHANGED(e) \triangleq PAST(e) \neq e$$

$$ROSE(e) \triangleq PAST(e) < e$$

$$FELL(e) \triangleq PAST(e) > e$$

By abuse, $ROSE$ and $FELL$ can also be applied to boolean expressions.

- We separate the assumptions from the consequences of those assumptions to improve readability.
- We omit the name of the function under test because we wish to test the same invariants on both the reset function and the transition function and because there is only one transition function.

In our notation the above property is written as follows.²

²To avoid distraction, we have simplified the ASL language slightly in this paper: omitting explicit type conversions and using mathematical symbols such as $=$, \neq and \wedge where the concrete syntax uses conventional programming notation such as `==`, `!=` and `&&`.

```

property R_JRC
  assume FELL(LockedUp);
  CALLED(TakeColdReset)  $\vee$  CALLED(TakeReset)  $\vee$  ROSE(Halted)  $\vee$  CALLED(ExceptionEntry);

```

We feel that this is a reasonably close match to the structure of the original natural language statement.

Invariants: Invariants are properties that should initially be valid and then their validity should be preserved by any action the processor takes. An example invariant is that a processor can be in Lockup or it can be Halted but it cannot be both. In our notation, this property is written like this.

```

invariant dbg_lockup_mutex
   $\neg(\text{Halted} \wedge \text{LockedUp})$ ;

```

Unpredictable Behaviour: ARM’s specifications are deliberately incomplete and do not specify what a processor should do in all circumstances. ARM labels these gaps in the specification as *UNPREDICTABLE* and the processor is free to do anything that can be achieved at the current or a lower level of privilege using instructions that are not *UNPREDICTABLE* and that does not halt or hang the processor or parts of the system.

Most unpredictable behaviour in the ARM specification is associated with attempting to do something that is nonsensical and that can be easily avoided by the programmer. In ASL, unpredictable behaviour is marked by the statement *UNPREDICTABLE*;. To let us distinguish executions that do not execute this statement, we add a new property *Predictable* that is true for executions that do not execute *UNPREDICTABLE*;

Implicit assumptions: All of the properties that we wish to prove about the transition function only hold under the restrictions that the initial state satisfies the invariant, and the execution is *Predictable*. These restrictions could be added to each individual property by adding the following assumptions:

```

assume PAST(Invariants);
assume Predictable;

```

where *Invariants* represents the conjunction of all the invariant properties. Such assumptions would be the same for all properties and would only serve to add noise to our properties so, instead, we choose to leave these restrictions implicit and add them in our proof tool (see Section 5.3).

4 EXAMPLES

This section illustrates the use of the notation introduced in the previous section to write further properties about the architecture and it will look at the challenges in formalizing rules found in the natural language part of the specification.

4.1 The Exception Entry Bug

One of our motivating examples in this work was trying to detect a bug that had recently been found in the v8-M specification and to prove that any bugfix does, indeed, fix the bug.

In order to write a property that would detect what the specification did wrong we asked the v8-M architects how they could tell that the bug had occurred (but not to describe the bug itself). They told us that the bug involved what state is saved on taking a processor exception. From testing, they knew that the state was usually saved correctly but, under some circumstances, the specification was not saving information about which stack the interrupted context was saved on.

The current stack selection is recorded in the field `SPSEL` of the register `CONTROL`. On entry to a processor exception handler, the current stack selection that was active before the exception is recorded in bit two of the register `LR`. Using our notation, we formalized the property like this.

```
property exn_entry_spsel
  assume CALLED(ExceptionEntry);
  assume ¬CALLED(TakeReset);
  assume ¬CALLED(ExceptionReturn);
  PAST(CONTROL.SPSEL) = LR<2>;
```

Having specified the required property, we used our tool to attempt to rediscover the bug. Our tool generated a counterexample that the architecture development team confirmed as a possible symptom of the bug they had previously found. In particular, the bug occurred if the attempt to save the state of the processor on the original stack failed and the processor exception could not be escalated to an appropriate handler (e.g., because the processor was already in the highest priority exception handler). In this case, the processor enters the `Lockup` state and stops execution but even in this desperate circumstance, it is required that the originating mode and security level are saved correctly in `LR` to enable the problem to be diagnosed through the debugger.

After confirming that the properties could detect the original bug, we applied a bugfix proposed by the architecture team and repeated the check. To our relief, all of the processor exception entry properties were found to hold: our first formal verification that a bugfix actually fixed a specification bug.

4.2 Property Groups

Properties often share a number of assumptions and triggering conditions so we find it useful to group multiple properties together to allow them to share common antecedents.

For example, when a processor exception is taken, the processor doesn't just save the current stack selection, it also saves the current security state, the exception mode, whether the floating point state is "dirty", etc. We provide some syntactic sugar for writing sets of related properties sharing a common set of antecedents. The first sub-property in the following is equivalent to the `exn_entry_spsel` property above.

```
rule exn_entry
  assume CALLED(ExceptionEntry);
  assume ¬CALLED(TakeReset);
  assume ¬CALLED(ExceptionReturn);

  property spsel: PAST(CONTROL.SPSEL) = LR<2>;
  property secure: FELL(IsSecure()) ⇔ (LR<0> = '1');
  property mode: PAST(CurrentMode() = PEMode_Handler) ⇔ LR<3> = '0';
  property ftype: PAST(CONTROL.FPCA) = NOT LR<4>;
```

4.3 Entry to Lockup

One of the more challenging rules to formalize was the following rule that describes entry to `Lockup`.

R_{VGNW}

Entry to lockup from a processor exception causes:

- *Any Fault Status Registers associated with the exception to be updated.*
- *No update to the exception state, pending or active.*
- *The PC to be set to 0xEFFFFFFE.*
- *EPSR.IT to be become UNKNOWN.*

In addition, HFSR.FORCED is not set to 1.

Each bullet in this rule required careful interpretation/debugging in order to understand it.

- Other rules detail which fields of the Fault Status Registers should be set but one consequence of the first bullet is that at least one bit in Fault Status Register CFSR should be set after entry to lockup (in configurations that provide the CFSR register).
- The second bullet turned out to be false: pending and active processor exception state should be updated to reflect the attempted exception entry. The statement had been true in the previous version of the architecture but had not been updated. We filed a bug against the documentation.
- The third bullet suggested that we check that PC = 0xEFFFFFFE but this property failed with counterexamples where PC was equal to 0xF0000002. On investigation, we found that the rule was implicitly referring to the “debug view” of the program counter that, for historical reasons, reads as four less than the “program view” that is accessed as PC. We filed a clarification request against the documentation.
- The fourth bullet is untestable because setting a register to *UNKNOWN* is allowed to choose any value — including the current value of the register. We are currently unable to formalize this statement.
- The final sentence seemed to allow multiple interpretations including HFSR.FORCED must become 0 or may become 0 or must not be changed. After consulting the architects, we learned that it meant that HFSR.FORCED is not modified. We filed a clarification request against the documentation.

Of course, the task of determining which interpretation to use is not quite as direct as suggested above and in practice, we followed a more experimental methodology. We would typically formalize several different interpretations of each clause of a rule; we test which interpretations hold for the specification; and we consult the architects to confirm that the winning interpretation is the intended interpretation. This led to the following set of properties

```

rule lockup_entry
  assume ROSE(LockedUp);
  assume ¬CALLED(TakeReset);

  property R_VGNWa: HaveMainExt() ⇒ CFSR ≠ 0;
  property R_VGNWc: _RName[RNamesPC] = 0xEFFFFFFE;
  property R_VGNWe: STABLE(HFSR.FORCED);

```

4.4 Exit from Lockup

The example rule in Section 3 described when a processor could exit the *Lockup* state. The following rule describes one part of what a processor should do when that happens.

R_{SPPN}

On an exit from lockup by entry to Debug state, or by preemption by a higher priority processor exception, the return address is 0xEFFFFFFE.

We initially formalized the debug part of this rule with the following property.

```
property R_SPPNa
  assume FELL(LockedUp);
  assume ROSE(Halted);
  LR = 0xEFFFFFFE;
```

Our tool reported that this property did not hold and, on investigation, we realized that we had misinterpreted the phrase “return address.” When an ARM processor is executing instructions, the return address is normally held in register LR but when an ARM processor is in Debug state, the return address is held in the program counter and when an exception is taken, the return address is held on the stack. The amended formalization read as follows for the debug case

```
property R_SPPN
  assume FELL(LockedUp);
  assume ROSE(Halted);
  _R[RNamesPC] = 0xEFFFFFFE;
```

We filed a clarification request against the documentation and recommended splitting these two cases.

4.5 Lockup Invariants

Lockup occurs when a fault occurs and it is not possible to report the fault because the appropriate fault handler is lower priority than the current execution priority. A consequence of this is that, under normal circumstances, Lockup can only occur in the highest priority processor exception handlers: Non-maskable Interrupt NMI and HardFault. We formalized this as follows using the Interrupt Program Status Register IPSR to read the current processor exception handler and adding the additional assumption that execution priority had not been boosted using the FAULTMASK register.

```
invariant lockup_IPSR
  assume LockedUp;
  assume FAULTMASK.FM = 0;
  IPSR ∈ {NMI, HardFault};
```

A further rule about Lockup states

R_{MBTM}

When the PE is in lockup:

- *DHCSR.S_LOCKUP reads as 1.*
- *The PC reads as 0xEFFFFFFE. This is an execute never (XN) address.*
- *The PE stops fetching and executing instructions.*
- *If the implementation provides an external LOCKUP signal, LOCKUP is asserted HIGH.*

We formalized this as follows.

```

rule R_MBTM
  assume LockedUp;
  invariant a DHCSR.S_LOCKUP = 1;
  invariant b PC == 0xEFFFFFFE;
  property c
    assume PAST(LockedUp);
     $\neg$  CALLED(FetchInstr)  $\wedge$   $\neg$  CALLED(DecodeExecute);

```

Attempting to prove these properties found that the DHCSR register was only partially implemented in the specification and that PC referred to the debug view of the program counter and we filed bugs against the specification and the documentation.

4.6 Preemption by Processor Exceptions

As a final example, an important property of the processor exception mechanism is that execution can only be preempted by higher priority exceptions. Since higher priority is represented by smaller numbers, this says that if a processor exception is successfully taken then the priority value must be lower than it was before the transition. In formalizing and proving this statement, we found a counterexample: the priority need not increase if the program triggers a derived exception while attempting to perform a processor exception return (e.g., because of a memory fault while popping the exception frame off the stack). Our amended statement is as follows.

```

property priority_increase
  assume CALLED(ExceptionEntry);
  assume !CALLED(ExceptionReturn);
  ExecutionPriority() < PAST(ExecutionPriority());

```

Interestingly, the complementary property does not always hold: exception return does not always lead to a decrease in priority (that is, an increase in priority number) because an exception handler can dynamically change the priority of an interrupt before returning.

4.7 Summary

This section described several properties we created by talking to the architects or by translating natural language “rules” to our property notation. The process of formalizing and of attempting to prove the properties found several bugs in both the formal part of the specification and in the natural language part.

The bugs we found in the formal specification typically involved corner cases that trigger cascades of derived processor exceptions, exceptions triggered when returning from an exception, exceptions triggered because the vector table is in an unreadable part of the memory space, etc. These bugs tend to creep into a specification because humans find it hard to think about all of the corner cases and because it is natural to focus on your current task when extending the architecture and to forget about all of the cross-cutting issues.

It is not surprising to find ambiguous, misleading and erroneous statements in natural language specification — even one as heavily reviewed as the ARM specification. It took a process of experimentation to find the correct interpretation of some statements although, a bit like a good crossword puzzle, our final solution was obvious once we knew what it was. Our property language and checker allows us to perform those experiments and to confirm that those results are consistent with the formal specification; and the act of formalizing the statements helps us formulate clearer, more accurate natural language statements.

$\langle \text{definition} \rangle ::=$	$\text{'rule' } \langle \text{ident} \rangle$	$\langle \text{expr} \rangle ::=$	$\text{'Past' '(' } \langle \text{expr} \rangle \text{')'}$
	$\{ \text{'var' } \langle \text{ident} \rangle \text{' : ' } \langle \text{type} \rangle \text{' ; ' } \}$		$ \text{'Rose' '(' } \langle \text{expr} \rangle \text{')'}$
	$\{ \text{'assume' } \langle \text{prop} \rangle \text{' ; ' } \}$		$ \text{'Fell' '(' } \langle \text{expr} \rangle \text{')'}$
	$\{ (\text{'property' } \text{'invariant'}) \langle \text{ident} \rangle \langle \text{expr} \rangle \text{' ; ' } \}$		$ \text{'Stable' '(' } \langle \text{expr} \rangle \text{')'}$
	$\}$		$ \text{'Changed' '(' } \langle \text{expr} \rangle \text{')'}$
	$ (\text{'property' } \text{'invariant'}) \langle \text{ident} \rangle$		$ \text{'Called' '(' } \langle \text{ident} \rangle [\text{'when' } \langle \text{expr} \rangle] \text{')'}$
	$\{ \text{'var' } \langle \text{ident} \rangle \text{' : ' } \langle \text{type} \rangle \text{' ; ' } \}$		$ \text{'Returned' '(' } \langle \text{ident} \rangle [\text{'when' } \langle \text{expr} \rangle] \text{')'}$
	$\{ \text{'assumes' } \langle \text{expr} \rangle \text{' ; ' } \}$		$ \text{'PREDICTABLE'}$
	$\langle \text{expr} \rangle \text{' ; ' }$		$ \text{'Invariants'}$

Fig. 1. Property Syntax Extensions

5 DESIGN AND IMPLEMENTATION

This section describes the semantics and implementation of the property notation described in earlier sections.

5.1 Property Language

Our notation for specifying invariants and properties extends the ASL specification language with the ability to refer to the values of expressions before execution of the code under test; to test whether an execution performs an action such as calling a function; and to name properties for ease of reference. It also adds some syntactic sugar for defining groups of larger properties. The grammar for these extensions is shown in Figure 1 which defines additional productions for the $\langle \text{definition} \rangle$ and $\langle \text{expr} \rangle$ non-terminals.

Our property language blends two different notions: conditions involving the *state* of the processor before and after a processor transition; and conditions involving the execution path taken while executing the state transition function. Defining the semantics of this combination requires two steps:

- We extend the stateful semantics of the ASL language with generation of a trace during execution. The details of this extension are unsurprising and results in a trace of function call and return events. Function call events are represented by $C\langle f, \bar{a} \rangle$ consisting of the name f of the function and a binding \bar{a} of the function's formal parameters to the values of each actual parameter of the call. Function return events are represented by $R\langle f, \bar{a}, \bar{r} \rangle$ consisting of the name f of the function and bindings \bar{a} and \bar{r} of the function's formal parameters and results to the names of each function argument and return variable in the function.
- We define the semantics of the *CALLED* and *RETURNED* operators in terms of this trace. For any terminating execution producing a trace T , the $\text{CALLED}(f \text{ when } P)$ operator is satisfied if T contains an element $C\langle f, \bar{a} \rangle$ such that $\llbracket P \rrbracket_{\bar{a}}$ is satisfied where $\llbracket _ \rrbracket_{\rho}$ is the semantics of evaluating an expression wrt a binding ρ . Similarly, the $\text{RETURNED}(f \text{ when } P)$ operator is satisfied if T contains an element $R\langle f, \bar{a}, \bar{r} \rangle$ such that $\llbracket P \rrbracket_{\bar{a} \cup \bar{r}}$ is satisfied.

5.2 Implementation

A key requirement for practical deployment is that all proofs should be performed automatically without needing to train the authors of the architecture in the use of an interactive proof assistant. Our implementation therefore is based on translating the architecture specification and the properties to be checked into verification conditions suitable for SMT solvers. This translation consists of three major steps: converting property specifications to ASL; a number of “lowering passes” that convert complex language features into simpler language features; and converting the simplified ASL specification to a verification condition expressed in the SMT-Lib language [Barrett et al. 2016].

5.2.1 Converting Properties to ASL. Properties are written using an extension of ASL so we convert each extension into the original ASL language. This is done by introducing “ghost variables” to collect information needed by the properties and adding code to initialize, update and test these variables that should execute before, during and after the function under test. We introduce two new functions *SMTPre* and *SMTPost* to hold the statements that execute before and after execution of the function under test.

- The *PAST*(*e*) operator is implemented by introducing a fresh global variable *v* and adding an assignment $v = e$; to the *SMTPre* function. Occurrences of *PAST*(*e*) are replaced by *v*.
- The *CALLED*(*f when P*) and *RETURNED*(*f when P*) operators are implemented by introducing a fresh global boolean variable *v* initialized to *FALSE* and instrumenting each function with an assignment $v = v \vee P$;. The assignment is placed at the start of the function for *CALLED* and before each *Return* statement for *RETURNED*. Occurrences of the operator are replaced by *v*.
- Invariant properties are evaluated before and after the function under test by creating two fresh global variables *pre* and *post* adding assignments $pre = P$; and $post = P$; to *SMTPre* and to *SMTPost*, respectively. Our proof frontend uses the conjunction of all the pre-variables and (separately) all the post-variables when proving that properties hold.
- All function properties are evaluated after the function under test by creating a fresh global variable *v* and adding an assignment $v = P$; to *SMTPost*. Our proof frontend replaces the property name with *v*.

With this conversion, testing whether a property holds for some function *f* consists of checking whether the corresponding global variable is *TRUE* after executing the sequence *SMTPre*(); *f*(); *SMTPost*();.

5.2.2 Simplifying ASL. The challenge in translating the rich, expressive ASL language to an SMT problem is that SMT-Lib [Barrett et al. 2016] is a pure expression language and lacks polymorphic types, dependent types, function calls, control flow, assignments, exceptions and structured data types.

Before starting translation, we apply a number of “lowering passes” that convert complex language features into simpler language features. The primary transformations performed in these passes are

- Eliminating dependent types and polymorphism by specializing all instructions and creating monomorphic instances of all polymorphic functions. For example, the memory load instruction can perform an 8, 16, 32 or 64-bit memory access based on a 2-bit size field of the instruction encoding. This results in many intermediate variables and function arguments whose width is dependent on the value of the size field. The specialization pass creates 4 separate instances of the instruction each of which accesses a single data width.
- Unrolling all loops. In our application, we were fortunate that it was always possible and often trivial to find an appropriate loop bound. There was one use of recursion but the architects were easily persuaded that rewriting it would make it easier to understand. Had this not been the case, we would have resorted to bounded unrolling and bounded recursion depths as is common practice elsewhere [Clarke et al. 2004].
- Eliminating unstructured control flow using a simplified form of if-conversion [Allen et al. 1983]. ASL does not have *goto* but it provides functions that return in the middle of a function and provides exception throwing that can exit in the middle of a function. This is converted to structured control flow by introducing an additional control variable into each function. This variable is initially true but it is set to false in the event of function return or an ASL exception and the variable is used as a guard to disable actions of statements if the variable is false.

- Global context-insensitive, flow-insensitive, structure-insensitive constant propagation and dead code elimination to exploit the large number of constants introduced by the previous passes. The choice of global/local and sensitive/insensitive propagation is based on our understanding of the structure of the specifications we wish to reason about.

These preprocessing steps reduce the ASL specification to a simple monomorphic, imperative language with functions, structured control flow and no loops or recursion.

5.2.3 Converting ASL to Verification Conditions. The remainder of the transformation is performed by symbolically executing the specification using SMT expressions as symbolic values and with each step of the evaluation extending a graph of SMT expressions representing the data/control flow of the program. When control-flow splits, the control expression is remembered, both control paths are executed using separate copies of the current execution environment, and when control-flow joins, the two execution environments are merged by introducing if-then-else nodes to select values from one path or the other. Function calls are handled in the usual way for an interpreter: a fresh environment is created containing the values of the function arguments and the function body is evaluated in that environment. Uninitialized variables and *UNKNOWN* expressions are handled by introducing oracles (that is, fresh variables that are unconstrained).

Unfortunately, this conventional translation resulted in excessively large SMT problems and we were unable to generate SMT problems for even the smallest architecture configuration.

To overcome this, we implemented four important optimisations:

- When merging environments, we omit the if-then-else node if neither environment has changed the value of a variable.
- We perform “hash-consing” to avoid creating nodes that are identical to a previously constructed node. This increases the effectiveness of the first optimization in the case that both branches set a variable to the same value.
- When evaluating an if-statement, if the control expression is definitely true or definitely false then we avoid exploring the dead branch. This is a significant optimization.
- We perform a limited amount of constant folding to catch constant propagation opportunities that were missed during preprocessing. Our primary goal in doing this is to evaluate boolean conditions to make the third optimization more effective.

After implementing these optimisations, the generated SMT expression was still large: approximately 30,000 terms for *TakeColdReset* and between 360,000 terms and 860,000 terms for *TopLevel* depending on the architecture configuration tested. We found that the Z3 SMT solver [de Moura and Bjørner 2008] was able to handle problems of this size but we found that even proving very shallow properties took 30-60 minutes: this put the feasibility of tackling interesting properties in question. To resolve this, we consulted one of the Z3 developers [Wintersteiger 2017] who suggested that we further simplify the SMT expression by avoiding use of high-level constructs such as enumerated types and arrays whenever possible. Replacing enumerated types with small bitvectors was an easy change but to avoid arrays we had to construct expressions that closely resemble the way that register files are typically implemented in hardware using address decoder trees to write array elements and using trees of multiplexors to read array elements. These additional optimisations reduced the need to switch between different theories when solving problems and resulted in a performance improvement of approximately 5x. Solution times with the above optimisations are detailed in Section 6.3.

5.3 Proof Frontend

The final part of our implementation is a proof frontend that uses the Z3 solver to prove that invariant properties and function properties hold.

For each invariant I , we check two properties (expressed here as Hoare triples)

$$\{\} \text{ TAKECOLDRESET() } \{I\}$$

$$\{\text{INVARIANTS}\} \text{ TOPLEVEL() } \{\text{PREDICTABLE} \Rightarrow I\}$$

For each function property P , we check the property

$$\{\text{INVARIANTS}\} \text{ TOPLEVEL() } \{\text{PREDICTABLE} \Rightarrow P\}$$

For each assertion or bounds check P , we check the properties

$$\{\} \text{ TAKECOLDRESET() } \{P\}$$

$$\{\text{INVARIANTS}\} \text{ TOPLEVEL() } \{P\}$$

5.4 Debugging Properties

Given the large size of the state space, we found it hard to debug failing properties just by examining the initial and final states. To help us understand counterexamples, we added the ability to emit code that would set the processor registers to the final state.

A minor challenge in doing this is that the generated SMT problem loses several type distinctions that were present in the original ASL: we solved this by emitting a file containing the ASL-level type of every SMT variable that our proof tool could use to generate type-correct ASL code. This was used with an interpreter for ASL that provides useful debugging features such as displaying the call tree of an execution, displaying register reads/writes, an interactive mode, etc. Using the interpreter to animate counterexamples proved to be essential for understanding bugs in the specification and when testing speculative properties and invariants. It was also useful while developing the transformation from ASL to SMT for identifying differences between the transformation and the interpreter that indicated bugs in the transformation.

A more significant challenge is that ASL allows underspecification (i.e., the specification does not completely constrain the behaviour in some circumstances). Our ASL interpreter handles this by choosing just one possible behaviour whenever the specification provides a choice. In contrast, the SMT solver explores all possible behaviours and may find a counterexample that is allowed by the specification but that is not the behaviour chosen by the ASL interpreter. When the underspecification affects the control path in the specification we can see significant divergence between the interpreter and the SMT solver. This has prevented us from debugging some of the failing properties found by our tool (Section 6) and is the subject of future work.

6 EXPERIENCE

We subjectively feel that our properties closely reflect the rules we formalized and, hence, the way that architects view the architecture. More objectively, we evaluate the effectiveness of our approach based on the ability of our properties to find bugs, and the efficiency of proof.

6.1 Formalizing Natural Language Specifications

Our original intention in this project was to focus on verifying properties that would be useful to programmers or that the architects identified as having been hard to get right (e.g., based on bugfixes to the specification). When we realized that some of the natural language rules in ARM's existing architecture specification could also be formalized using our tools, we shifted our focus to formalizing those rules and added more structure to our notation to better match the style of those rules.

We are also working with the team responsible for creating and maintaining the natural language part of ARM's architecture documents about two improvements to the rule style. The first

improvement is adoption of a more structured approach where rules are categorized according to the type of constraint expressed and each type is then written in a consistent way. For example, responses to exceptional events might be written in a sentence structure like this:

$R_{\langle \text{label} \rangle}$
<i>IF $\langle \text{optional preconditions} \rangle$ $\langle \text{trigger} \rangle$, THEN the $\langle \text{system name} \rangle$ shall $\langle \text{system response} \rangle$</i>

(This approach is inspired by the “EARS” requirements specification style [Mavin et al. 2009] used by Rolls Royce PLC to write the requirements of their avionics engine control systems.) The second improvement is adoption of a standardized terminology to describe the triggers, actions and responses. For example, there should be only one way to describe a signal becoming ‘1’, only one way to describe a signal changing from ‘0’ to ‘1’ and only one way to describe a signal remaining unchanged. These changes to improve consistency are motivated by a desire for clarity and ease of understanding and is especially valuable for customers who are not native English speakers.

The more formal notation described in this paper is also a structured way of capturing rules: the notation for properties and the assumptions they rely on provides a high level structure while the ASL notation coupled with temporal operators such as *ROSE* and *STABLE* provides a standard way to describe signal values and their changes.

We are starting to look at extending the formal notation with structures and operators directly corresponding to the sentence structures and terminology used in the natural language rules. Our hope is that we can narrow the gap between the two notations so that our formal properties are “eyeball close” to the corresponding informal rule: that is, identically structured and using corresponding terminology/notation so that humans can easily see that they have the same meaning. We don’t expect this to be possible for all rules but the experience reported in Section 4 suggests that it should be possible.

An obvious further step would be to write rules in a style that can always be directly translated to formal properties or, conversely, to write properties that can be automatically converted from our formal notation to English sentences [Burke and Johannisson 2005, for example]. Our current feeling is that this would be a step too far: it is possible and desirable to narrow the gap between natural language and formal notation but there is a tension between the best way to express rules so that humans from different technical backgrounds can understand them and the best way to express properties to enable machine proofs. This tension is especially strong when the specification has to deal with new concepts for which we do not have a good mathematical theory and may not yet know how to formalize or prove a property. For example,

- We cannot currently formalize statements about *UNKNOWN* (see Section 4.3).
- The best way to formalize memory concurrency semantics is still an active area of research with no clear agreement between an operational approach (e.g., [Flur et al. 2016]) and an axiomatic approach (e.g., [Alglave et al. 2014]).
- It is not clear how to formalize statements about security properties of the architecture.

In such cases, we must start with a natural language specification, then formalize rules as techniques and understanding develop and only then hope to find a way of structuring both the rules and the properties to be “eyeball close.”

6.2 Bugs Found

We checked the properties on two configurations of the v8-M architecture: one with security extensions enabled and one with security extensions disabled. The configuration with security extensions had previously been heavily tested [Reid 2016] but the configuration without security

extensions was relatively untested. In addition, debug features had only recently been implemented and only partially tested.

We checked all properties on both configurations and, in the process, we found twelve bugs in the formal part of the specification and 9 issues in the natural language part including:

- *Trivial Bugs*: Array bounds failures, a guarding test that was placed after the action it was meant to guard instead of before the action, and an uninitialized variable.
- *Unimplemented/untested functionality*: Some parts of the debug specification were ignoring a “debug disable” control signal;
- *System register problems*: We found several bugs in the machine-readable description of the system registers. Some status fields in system registers gave incorrect information about the state of the processor while others should have masked the status information based on a control field but did not.
- *Ambiguity of Natural Language*: Some bugs were not in the ASL part of the specification but in the natural language part that specifies the rules.
- *Imprecision of Natural Language*: Section 4 contains several examples where the specification referred to the PC but meant the “debug view of the program counter.” These lead to considerable confusion and attempts at bugfixes failed until we understood the subtle distinction between the two.
- *Processor exception entry*: The example discussed in Section 4.1 was already known to the architects but we were able to detect it using very high level properties without knowledge of the details of the bug.
- *Mixed logic polarity*: The security parts of the specification use boolean variables where TRUE indicates that something *is* secure and they also use variables where TRUE indicates that something *is not* secure. That is, it uses both positive logic and negative logic. We found a bug where a variable of one polarity was passed to a function that expected a variable of the opposite polarity.
- *Secure accesses from NonSecure processor*: The most serious of the bugs we found was in the configuration with security extensions disabled. In this configuration, the processor should behave as though it was in the NonSecure state: all accesses should be non-secure. Our tool found a case where the processor was treating accesses as secure.

The most difficult and tedious part of this process was in creating invariant properties. Many of the invariant properties were added in response to puzzling counterexamples involving processor states that seemed to be nonsensical. After some time staring at these examples, we would convince ourselves that these nonsense states were unreachable and we would add and prove another invariant.

6.3 Proof Time

We wrote a total of 59 function properties and invariants. In addition the specification already contained assertions and we added additional array bounds checks during SMT generation. We test each of the invariants on both the initial function `TakeColdReset` and on the transition function `TopLevel` and we test the function properties on `TopLevel` as detailed in Section 5.3. We applied our tool to two architecture configurations: “NS” with security extensions disabled and “S” with security extensions enabled. Our SMT generator omits checks for assertions and bounds checks that are provably satisfied at generation time so the number of assertions and of bounds checks varies slightly between configurations.

For this experiment, we ran all properties on an Intel(R) Xeon X5670 at 2.93GHz equipped with 48GB memory. We attempted to prove 315 verification conditions. Each proof attempt was run

Table 1. Number of properties of different classes for the TakeColdReset and TopLevel functions and number of proofs that pass, fail or timeout in 30000s.

	TakeColdReset			TopLevel			
	Asserts	Bounds	Invariant	Asserts	Bounds	Invariant	Properties
Configuration = NS							
Total	21	2	38	36	2	38	23
Passed	21	2	38	36	2	36	21
Failed							1
Timeout						2	1
Configuration = S							
Total	18	3	38	32	3	38	23
Passed	18	3	38	29	3	33	19
Failed							2
Timeout				3		5	2

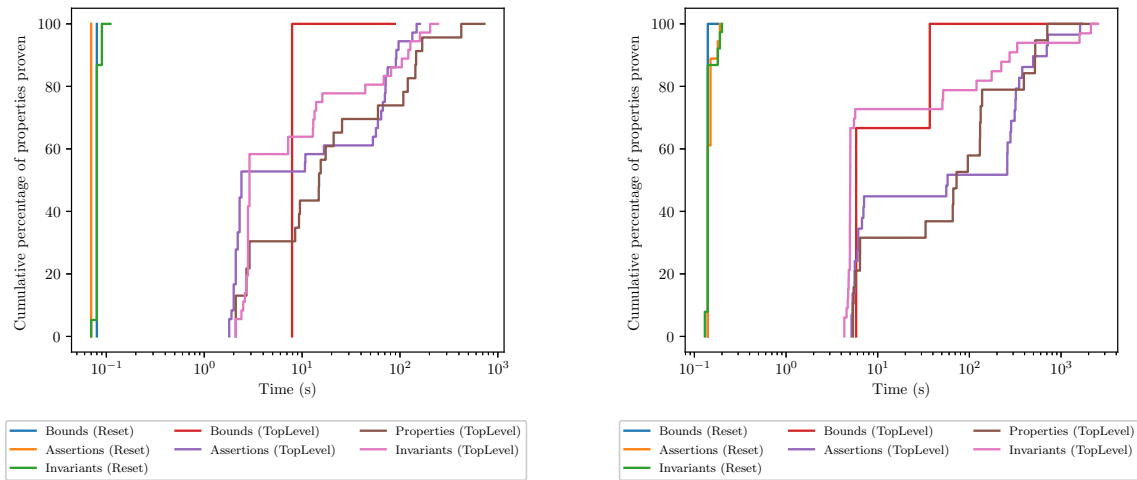
with a timeout of one day and we were able to prove 299 of them within the timeout. The results are summarized in Table 1. The 3 failing properties are still being diagnosed but are probably due to missing invariants. The presence of 7 timeouts on the invariants means that our proofs are not yet sound but this has not prevented us from using the tool for finding bugs. The presence of 6 timeouts on assertions and properties means that some of those properties could yet fail if given enough time to run the checks. Though less worrying, we hope to reduce the size of the SMT problem we generate and that that will allow these proofs to terminate one way or the other in an acceptable time.

To help understand the timeouts in Table 1, Figures 2a and 2b summarize what fraction of properties can be proved in a given time interval for the “NS” and “S” configuration, respectively. As one might expect, proofs about the reset function are fairly trivial and take just a fraction of a second while the amount of choice present in the transition function makes proofs about TopLevel take longer. The graphs show that the properties for the “NS” configuration are typically proved 3-10x times faster than the properties for the “S” configuration but even for the “S” configuration most proofs are generated within 1000 seconds. The total time taken for all passing proofs is under 5 hours and using a 1000 second timeout would result in the tests that fail or timeout taking another 4 hours. In practice, the proof effort should parallelize nicely so the total elapsed time is primarily bounded by the number of properties that fail or timeout.

6.4 Notation

As Section 5.2.2 shows the ASL language used in the main specification is a little awkward for the work described in this paper: it would be easier to translate ASL to an SMT problem if all loops had explicit bounds, if ASL did not support exceptions and did not allow return from the middle of a function, if ASL did not provide unbounded integers, etc. On the other hand, simplifying ASL in such ways would make the language less readable, less robust or require a more subtle semantics (whose finer details might be lost on some readers).

ASL is a compromise specification language intended to be useful to multiple communities inside and outside ARM: OS engineers, compiler engineers, hardware engineers, hardware verification engineers, authors of tests, JIT writers, creators of simulators, documentation teams and formal verifiers of software. Enabling new user groups and applications increases the utility of the specification, detects previously undiscovered bugs in the specification and, through successful use, increases our confidence in the specification. The cost of these benefits is that each individual use



(a) Time to prove properties about configuration NS

(b) Time to prove properties about configuration S

Fig. 2. Cumulative time to prove properties

is more difficult than if the specification and tools were optimized for that individual purpose. For example, ARM's hardware engineers would generally find an unoptimized processor implementation written in Verilog to be easier to understand and easier to verify against than the ASL specification. This would lead to fragmentation of the specification since a specification written in Verilog would only be useful to hardware engineers and other groups would use one or more separate and incompatible specifications.

7 RELATED WORK

There are two areas of closely related work: formal specification of processors and formal validation of requirement specifications.

This paper is concerned with the general problem of trusting large specifications but its particular focus is on specifications of processors. Most recent papers on creating processor specifications describe how they tested their specification. The most extensively tested processor specifications are the executable ARM specifications described in our previous work [Reid 2016; Reid et al. 2016] and the executable x86-64 specifications created by Goel et al. [Goel et al. 2014]. Both have been verified using substantial programs: Reid uses ARM's architecture conformance testsuite while Goel runs real programs including (amusingly) a SAT solver. ARM has publicly released their v8-A processor specification in machine readable form. Reid et al. also formally verify ARM processor pipelines against the instruction set part of the specification: this increases confidence in the instruction set part of the specification but it says little about the system architecture part of the specification that is our primary concern in this paper.

Other notable processor specifications are the Fox/Myreen ARM v7-A ISA specification in HOL [Fox and Myreen 2010] and Flur et al.'s ISA and concurrency specification in SAIL [Flur et al. 2016] both of which were tested against actual processors using random and directed tests (8400 tests in Flur et al., 281,307 tests in Fox/Myreen). The other major ARM ISA specification that we are aware of is embedded in the CompCert compiler and is used in the proof that the compiler faithfully translates the input C program to ARM assembly code. This specification is limited to a subset of the user-mode ARMv6 specification and there is no published statement of how it was validated.

It is clear that testing of processor specifications is becoming standard practice but all of the above work consists of testing or formally verifying the specification against implementations of that specification. They are therefore vulnerable to the problems we identified in the introduction: (1) Testing of the specification cannot begin until the first implementation or a test suite is produced; and (2) Testing against an implementation of the specification is vulnerable to common mode failure. Our approach avoids these pitfalls by relying on formalization of high level properties that the specification is intended to meet and it avoids the well known limitations of testing by using formal verification techniques.

We believe that our *CALLED* operator is a novel feature in formal specifications but it can be seen as an adaptation of the coverage measurement features found in System Verilog [IEEE 2013] that allow verification engineers to annotate Verilog programs with specific coverage goals. Verilog simulation tools generate reports of how many times each coverage goal was hit allowing verification engineers to confirm that their test harness is exercising the desired behaviour. Alternatively, in software testing, it is common to add debug printf statements to a program to confirm that a certain path is being tested.

Another important part of processor architecture is the specification of the memory concurrency semantics [Algave et al. 2014; Flur et al. 2016; Sarkar et al. 2011]. These specifications are tested extensively against commercial processors. More recently, the MemAlloy tool has been created for automatically comparing memory consistency models [Wickerson et al. 2017]. Although different in almost every detail, we see this as solving a similar problem: understanding if a specification is correct without the need to wait until an implementation is available.

Our work can also be seen as a variant on formal validation of requirements specifications³. The Alloy language and analyzer [Jackson 2002] is closest to the system described here. Alloy is a simplified and improved descendant of the Z notation that allows definition of a model consisting of some state and operations on those states and one can verify expected properties using a SAT solver. In some ways, Alloy is considerably more general and sophisticated than the system described here: it provides a simple, mathematically clean language for specification. In other ways the ASL language is more powerful because it provides specialized concepts for the task of defining processor semantics such as bitvectors and dependent types, concepts like instructions and bitfields of registers, etc. and it is imperative: these features allow the creation of detailed specifications of large, complex architectures and proofs about specifications with very large state spaces.

The Formal Tropos language [Fuxman et al. 2001] is specifically designed to allow the form of loose specification that characterizes the early stages of requirements engineering: it focusses on entities and the relationships between them and allows the addition both of hard goals specified using first order linear temporal logic and soft goals that might be subjective (e.g., a company may have a goal of attracting new customers). The language provides a number of high level abstractions of events such as notions of object creation, fulfillment of a goal, etc. that could be expressed in temporal logic but whose inclusion improved readability; specifications can be animated to check understanding; and model checking can be used to formally verify that properties of the specification are true or can be satisfied. Support for temporal logic is the most obvious difference from our system but we are not sure that model checkers would be able to cope with the large state spaces of our specification because, even with explicit invariants, some of the properties we wish to check are barely provable by an SMT solver. However, the rich set of abstractions for

³Strictly speaking, ARM says that the natural language rules and the formal ASL specification in ARM's specifications have equal weight: they are both part of the specification and both must be satisfied by an implementation. However, in this work we have treated the natural language rules as a loose specification of the properties that the more precise ASL specification is required to satisfy — much as a requirements specification can be seen as a loose specification for more refined specifications developed as design and implementation proceeds.

describing events appears useful and we are considering whether they can be adopted without requiring the use of model-checking.

More broadly, it is common practice when creating specifications in a theorem prover to prove that a new definition satisfies sets of properties such as commutativity, associativity, etc. [Pierce et al. 2016]. Like our properties about specifications, such properties may not completely characterize the functions being developed but they give increased confidence that the functions are correct and they are often useful when using those functions. The difference is that our specifications are somewhat larger and that our properties make use of the ability of our property language to restrict the set of execution paths taken by the function being checked.

8 LIMITATIONS AND FUTURE WORK

This is part of a long program of creating a complete and precise specification of the ARM architecture. The most significant limitation is that it has not yet been integrated with parallel work on concurrent memory semantics [Alglave et al. 2014; Flur et al. 2016; Sarkar et al. 2011]. This limitation shows up most clearly in situations where `TopLevel` performs multiple memory accesses in a single transition (either because of executing ARM’s “load/store multiple” instructions or because of pushing/popping context on/off the stack during exceptions). Our reasoning treats the entire execution/exception as a single atomic transition while an external observer would see multiple independent memory accesses.

We see this work as a step towards creating a set of properties that can be used to verify low-level system code such as interrupt handlers, memory protection, etc. We hope that this could be used to plug the gaps in formal proofs of software such as the seL4 OS kernel [Klein et al. 2009, Section 4.4] that rely on manual inspection and thorough testing of a few pieces of low-level code instead of providing a formal proof.

The current performance of our tool is adequate for daily or weekly checking of the specification but it is currently too slow to use as a check on every commit to the specification repository. We plan to implement a variation on DAG inlining [Lal and Qadeer 2015] to improve scalability.

We are considering how we could formalize the statement in Section 4.3 that says “EPSR.IT to be become UNKNOWN.” This property cannot be checked in our current implementation because it is a 2-safety property: detecting a violation would require comparing the result of traces from two program traces [Clarkson and Schneider 2010].

9 CONCLUSION

Formal verification of programs is becoming more and more practical but, if the verification is to be meaningful, it must be based on correct architecture specifications for the hardware that the programs run on. That is, the specifications are a critical part of the Trusted Computing Base. Unfortunately, the size and complexity of architecture specifications is such that it seems inevitable that specifications will contain bugs and our previous work confirms this supposition [Reid 2016].

While it is common to debug specifications by *testing* the specification, this paper proposes a different approach: we define a set of formal properties that should hold for the specification and we *formally verify* that the architecture specification satisfies these properties. We think of the relationship between the properties and the specification as being like the relationship between a nation’s constitution and a nation’s laws: the constitution is concise enough that everyone can read them while the laws are too large for effective review; the constitution can be used to test whether existing or proposed laws are compatible with high level goals; and the constitution is stable and changes very, very slowly.

We have extended ARM’s Architecture Specification Language with a property language that is able to concisely express many of the properties currently written in natural language. Our

extension's power comes from a novel coverage operator that lets us express cross-cutting, end-to-end properties. We are able to check that these properties hold by converting both the specification and the properties to verification conditions that can be checked in a push-button manner using an SMT solver. We have used this system to check ARM's v8-M specification. Despite the fact that the ARM v8-M specification had previously been extensively tested and reviewed, we found twelve bugs in it, that have all been fixed by ARM.

To our knowledge, no realistic architecture specification has been subjected to this degree of formal verification before.

ACKNOWLEDGMENTS

We wish to thank Christoph Wintersteiger for his suggestions for using Z3 more effectively. We are grateful to John Regehr, Peter Sewell and Nathan Chong and to the anonymous referees for their comments and suggestions about the content and presentation of this paper.

REFERENCES

- Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. DOI : <http://dx.doi.org/10.1145/2627752>
- J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '83)*. ACM, New York, NY, USA, 177–189. DOI : <http://dx.doi.org/10.1145/567067.567085>
- ARM Ltd. 2013. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile) (DDI0487)*. ARM Ltd. <https://developer.arm.com/docs/ddi0487/a/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile>
- ARM Ltd. 2016. *ARMv8-M Architecture Reference Manual (DDI0553)*. ARM Ltd. <https://developer.arm.com/docs/ddi0553/latest/armv8-m-architecture-reference-manual>
- Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org. (2016).
- D. A. Burke and K. Johannisson. 2005. Translating Formal Software Specifications to Natural Language / A Grammar-Based Approach. *Logical Aspects of Computational Linguistics (LACL 2005)* 3402 (2005), 51–66. DOI : http://dx.doi.org/10.1007/11422532_4
- Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. *A Tool for Checking ANSI-C Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176. DOI : http://dx.doi.org/10.1007/978-3-540-24730-2_15
- Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (Sept. 2010), 1157–1210. DOI : <http://dx.doi.org/10.3233/JCS-2009-0393>
- CompCert. 2016. Release Notes for CompCert 2.7 (Bugfixes). (29 June 2016). <http://compcert.inria.fr/release/Changelog>
- Mads Dam, Roberto Guanciale, and Hamed Nemati. 2013. Machine Code Verification of a Tiny ARM Hypervisor. In *Proc. Workshop on Trustworthy Embedded Devices (TrustED '13)*. ACM, 3–12. DOI : <http://dx.doi.org/10.1145/2517300.2517302>
- Leonardo de Moura and Nikolaj Bjørner. 2008. *Z3: An Efficient SMT Solver*. Springer Berlin Heidelberg, 337–340. DOI : http://dx.doi.org/10.1007/978-3-540-78800-3_24
- George Dunlap. 2012. The Intel SYSRET Privilege Escalation (Xen Project Blog). (2012). <https://blog.xenproject.org/2012/06/13/the-intel-sysret-privilege-escalation/>
- Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proc. Principles of Programming Languages, POPL 2016*. ACM, 608–621. DOI : <http://dx.doi.org/10.1145/2837614.2837615>
- Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 328–343. DOI : <http://dx.doi.org/10.1145/3064176.3064183>
- Anthony C. J. Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Proc. Interactive Theorem Proving ITP 2010 (LNCS)*, Vol. 6172. Springer, 243–258. DOI : http://dx.doi.org/10.1007/978-3-642-14052-5_18
- Ariel Fuxman, Marco Pistore, John Mylopoulos, and Paolo Traverso. 2001. Model checking early requirements specifications in Tropos. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. IEEE, IEEE, 174–181. DOI : <http://dx.doi.org/10.1109/ISRE.2001.948557>
- Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. 2014. Simulation and formal verification of x86 machine-code programs that make system calls. In *Formal Methods in Computer-Aided Design, FMCAD*. 91–98. DOI :

88:24

Alastair Reid

- <http://dx.doi.org/10.1109/FMCAD.2014.6987600>
- IEEE. 2013. IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language. *IEEE Std. 1800-2012* (2013). DOI : <http://dx.doi.org/10.1109/IEEESTD.2013.6469140>
- Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 2 (April 2002), 256–290. DOI : <http://dx.doi.org/10.1145/505145.505149>
- Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer, Berlin, Heidelberg, 220–242. DOI : <http://dx.doi.org/10.1007/BFb0053381>
- Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. DOI : <http://dx.doi.org/10.1145/1629575.1629596>
- Akash Lal and Shaz Qadeer. 2015. DAG Inlining: A Decision Procedure for Reachability-modulo-theories in Hierarchical Programs. In *Programming Language Design and Implementation (PLDI)*, Vol. 50. ACM, New York, NY, USA, 280–290. DOI : <http://dx.doi.org/10.1145/2813885.2737987>
- Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. DOI : <http://dx.doi.org/10.1145/1538788.1538814>
- Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. 2009. Easy Approach to Requirements Syntax (EARS). In *17th IEEE International Requirements Engineering Conference (RE'09)*. IEEE. DOI : <http://dx.doi.org/10.1109/RE.2009.9>
- Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. 2016. *Software Foundations*. Electronic textbook. <http://www.cis.upenn.edu/~bcpierce/sf> Version 4.0.
- Alastair Reid. 2016. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In *Proceedings of Formal Methods in Computer-Aided Design, (FMCAD 2016)*, Mountain View, CA, USA. 161–168. DOI : <http://dx.doi.org/10.1109/FMCAD.2016.7886675>
- Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of ARM Processors with ISA-Formal, In *Proceedings of the 2016 International Conference on Computer Aided Verification (CAV'16)*, S. Chaudhuri and A. Farzan (Eds.). *CAV 2016, Part II, Lecture Notes in Computer Science* 9780 (July 2016), 42–58. DOI : http://dx.doi.org/10.1007/978-3-319-41540-6_3
- Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 175–186. DOI : <http://dx.doi.org/10.1145/1993498.1993520>
- John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 190–204. DOI : <http://dx.doi.org/10.1145/3009837.3009838>
- Christoph M. Wintersteiger. 2017. Private communication. (2017).

Part II

Creating high performance hardware-software interfaces

One paper and one patent are presented in this section.

[Paper IV](#) “SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip” [102] describes the design and implementation of a set of extensions to the C programming language that direct the mapping of that program to different parts of a heterogeneous multiprocessor system.

[Patent I](#) “Reducing inter-task latency in a multiprocessor system” [103] describes an extension of the technique described in [Paper IV](#) that is able to exploit simple task triggering hardware to reduce the latency between tasks running on the different processors within the system.

Chapter 5

SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip (**Paper IV**)

Alastair D. Reid, Krisztián Flautner, Edmund Grimley-Evans, and Yuan Lin. SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip. In Erik R. Altman, editor, *Proceedings of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2008, Atlanta, GA, USA, October 19–24, 2008*, pages 95–104. ACM, October 2008.

© 2008 Association for Computing Machinery, Inc. Reprinted by permission. License number: 4296051467934.

doi: <https://dx.doi.org/10.1145/1450095.1450112>.

SoC-C: Efficient Programming Abstractions for Heterogeneous Multicore Systems on Chip

Alastair D. Reid
Krisztian Flautner
Edmund Grimley-Evans
ARM Ltd

Yuan Lin
University of Michigan

ABSTRACT

The architectures of system-on-chip (SoC) platforms found in high-end consumer devices are getting more and more complex as designers strive to deliver increasingly compute-intensive applications on near-constant energy budgets. Workloads running on these platforms require the exploitation of heterogeneous parallelism and increasingly irregular memory hierarchies. The conventional approach to programming such hardware is very low-level but this yields software which is intimately and inseparably tied to the details of the platform it was originally designed for, limiting the software's portability, and, ultimately, the architectural choices available to designers of future platform generations. The key insight of this paper is that many of the problems experienced in mapping applications onto SoC platforms come not from deciding how to map a program onto the hardware but from the need to restructure the program and the number of interdependencies introduced in the process of implementing those decisions. We tackle this complexity with a set of language extensions which allows the programmer to introduce pipeline parallelism into sequential programs, manage distributed memories, and express the desired mapping of tasks to resources. The compiler takes care of the complex, error-prone details required to implement that mapping. We demonstrate the effectiveness of SoC-C and its compiler with a "software defined radio" example (the PHY layer of a Digital Video Broadcast receiver) achieving a 3.4x speedup on 4 cores.

Categories and Subject Descriptors: D.3.3 [Software]: Programming Languages

General Terms: Languages

1. INTRODUCTION

In the next five years the peak available bandwidth to mobile phones is expected to increase from less than 5 Mbps today to 100 Mbps in 2012. The signal-processing throughput to implement these protocols is expected to increase to beyond 25 giga-operations per second. Commodity cameras on phones already support 10M pixel resolution which further drives the need for high-speed multimedia image processing, high-definition video coding and 3D graphics. To maintain the same form-factor, this massive performance must be achieved without increasing battery size which limits the power consumption to around 1 Watt.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.

Copyright 2008 ACM 978-1-60558-469-0/08/10 ...\$5.00.

Modern DSP designs are starting to achieve the required energy efficiency. For example, ARM's prototype data processing engine can sustain over 10 GMAC/s at less than 300mW in 65nm technology. *The main problem is not creating energy-efficient hardware but creating efficient, maintainable programs to run on them.* In order to save energy and, to some extent, silicon area, high performance embedded systems eschew features that characterize today's high-end multiprocessor systems: Homogeneous processors are replaced by a heterogeneous mix of specialized processors tuned to particular parts of the expected workload; General-purpose processors programmed in C, C++, etc. are supplemented by special-purpose accelerator engines which may be fixed-function, configurable or programmable using a C subset; Shared memory is replaced by multiple private memories to decrease latency and energy and increase bandwidth; and Hardware cache coherency is omitted to save area and power consumed by cache coherence protocols. Omitting these features from high performance embedded systems requires programmers to adopt a very low-level, error-prone programming style that limits portability and maintainability. *The key insight of this paper is that these problems come not from deciding how to map the application onto the hardware but from the restructuring of the code and the number of interdependencies introduced in the process of implementing those decisions.* Rather than abandon features because of their hardware cost, SoC-C moves their implementation into the language so that the programmer can reason about and optimize the mapping at a high level while the compiler takes care of the complex, error-prone details required to implement that mapping.

SoC-C is a set of language extensions that enables programmers to express efficient system-on-chip programs that exploit the parallelism available in the platform, provides programmers with control over how the many different processing elements in the platforms are used, and requires little or no restructuring when the application is subsequently ported within a family of platform architectures.

This paper makes the following contributions: We describe channel-based decoupling: a novel spin on existing ways to automatically introduce pipeline parallelism that allows programmers to tradeoff determinism for scheduling freedom and is capable of handling the complex control flow that real applications require. We propose a novel way of expressing the data copying that must happen in a distributed memory system. Our annotations express the programmer's intent allowing the compiler to detect missing or incorrect copy operations. We describe an inference mechanism that

```

// Data placement
declaration ::= type variable @ { memory1, ... memoryn } ;
expression ::= variable @ memory
statement ::= SYNC(variable[,memory[,memory]]
                ) @ processor ;

// Code placement
expression ::= identifier( expression, ... expression
                ) @ processor

// Fork-join parallelism
statement ::= parallel_sections {
    section { compound-statement } ;
    ...
    section { compound-statement } ;
}

// Pipeline parallelism
statement ::= pipeline { compound-statement }
statement ::= FIFO ( variable ) ;

```

Figure 1: SoC-C syntax extensions.

significantly reduces the amount of annotation required to map an application onto a hardware platform. We identify the critical optimizations required to support the high level programming model. With these optimizations, SoC-C can achieve accelerator utilization levels of 94% and a speedup of 3.4x on a platform with 4 accelerators on a real workload.

The paper is structured as follows. Section 2 describes a set of obvious minimal extensions to C to support heterogeneous, distributed parallel systems and introduces an example to illustrate why these extensions are *necessary* but *insufficient* for programming complex SoCs. Thus motivated, Sections 3–6 make a series of improvements showing how each extension improves the running example and we evaluate the expressiveness of the extensions in Section 7. Sections 8 and 9 discuss optimizations and performance. Section 10 discusses related work and Section 11 concludes.

This paper does not address how the best application mapping can be generated automatically using program analysis, profiling, iterative compilation, etc. for two reasons. The first is that the mechanism used to choose a mapping is largely orthogonal to the mechanism used to act on those decisions. The second is that there is no single obvious property to optimize for in embedded systems. Depending on the system one may want to optimize for some combination of battery life, low-latency user experience, meeting real-time deadlines, reducing number of retransmits, code size, etc.

2. A MINIMAL EXTENSION TO C

This Section considers minimal extensions to C to support heterogeneous multiprocessor systems with distributed memory and shows that whilst these or similar extensions are *necessary* (and form the basis of SoC-C), they are not *sufficient* for creating high performance, maintainable programs. This sets the stage for later sections which describe further extensions and optimizations to tackle these problems.

The extensions considered in this Section are those required to introduce parallelism, control sharing of resources and variables, communicate between threads, map data onto memories and map code onto processors/accelerators. Our descriptions of the extensions are brief because they are based on extensions found in other languages such as OpenMP (which inspired our notation), Concurrent Pascal, etc. Figure 1 summarizes all the extensions discussed in this paper.

Parallel sections introduce fork-join parallelism where a single master thread forks multiple child tasks (which may also fork child tasks) and waits for all children to complete.

```

complex_t samples[2048];
bool      bits[3024];
int8_t    bytes[378];
int      timing_correction = 0;
while (1) {
    ADC_get(&adc,&samples,2048);
    AdjustTiming(timing_correction,samples);
    FFT(samples);
    timing_correction += FindTimeOffset(samples);
    Demodulate(bits,samples);
    ErrorCorrect(bytes,bits);
}

```

Figure 2: A simplified OFDM radio receiver.

The statement

```

parallel_sections{
    section{ statement1 }
    section{ statement2 }
}

```

executes `statement1` and `statement2` in parallel and completes when both statements complete. Parallel sections can be implemented by forking one thread per section and then waiting for all threads to complete. Since this is the basic mechanism for expressing all parallelism, it is the programmer’s responsibility to avoid race conditions, deadlock, etc.

Channels synchronize/communicate between threads. FIFO channels provide two operations: “`fifo_put`” atomically transfers data into the channel and “`fifo_get`” operations atomically transfers data out (blocking if the channel is full/empty). This atomic-transfer semantics ensures that each thread has exclusive access to the data.

Data placement annotations map variables to memories. A variable declaration of the form

```
type V @ M ;
```

instructs the SoC-C compiler and linker to place the variable ‘V’ in memory ‘M’.

Code placement annotations perform RPCs. A function call of the form

```
function(expr1, ... exprm) @ P
```

is compiled into a synchronous remote procedure call: the function is invoked on processing element ‘P’. Unlike most RPC implementations, the call-frame (i.e., which function to call and any scalar and pointer arguments) is copied to the processing element but bulk data structures are not copied. This reflects our design goal of giving the programmer control over data copying to let them tune memory use and the impact on timing.

To illustrate these minimal extensions, consider mapping the sequential program in Figure 2 onto the architecture shown in Figure 3. This program displays two different types of data dependency which must be handled when parallelizing the program. There is forward dataflow within a loop iteration carrying complex samples from the ADC through timing correction, an FFT, demodulation and error correction. There is also feedback loop from one iteration to the next which continuously monitors changes in the timing offset between the transmitter and the receiver (caused by slight differences in clock rates, Doppler effects, etc.) which is used to control timing correction in future iterations. For simplicity, this example deals with fine timing correction (errors less than half the sample rate which are dealt with by applying a rotation to the complex samples) but ignores coarse timing correction (which would adjust the ADC in-

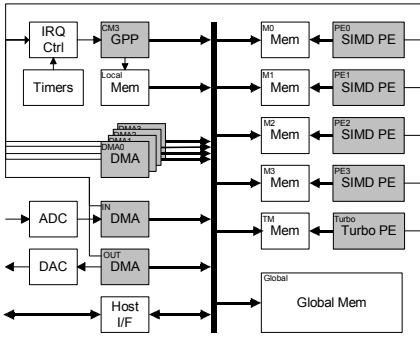


Figure 3: The architecture of a communication-processing subsystem. This system consists of one RISC processor and five processing elements (highlighted in gray), five private memories and one shared memory.

teraction) and channel equalization (which would correct for some frequencies being received more strongly than others).

In SoCs with heterogeneous processors, the principal form of parallelism used is pipeline parallelism: each engine is dedicated to performing a set of tasks and engines communicate with each other via FIFO channels. Figure 4 shows how the program can be rewritten to express pipeline parallelism using the minimal subset of SoC-C described in this Section. As can be seen, the parallel version of the program is significantly longer than the sequential version and has several more variables (both buffers and FIFOs). Looking more closely, we identify the following problems:

FIFO channels create excessive synchronization In the sequential version of the program, a feedback loop carries the timing correction back for use in the next iteration. To achieve exactly the same semantics, the parallel program would need to use a FIFO channel but this would have forced the first three sections to run sequentially because section 1 could not start the next iteration until section 3 had sent the new timing correction — a problem known as *loss of decoupling* [1]. To avoid this, the programmers used their knowledge of radio systems to confirm that timing corrections change slowly and so it would be acceptable to use a slightly older timing correction if that leads to greater parallelism. Since FIFO channels create too much synchronization, they chose the other inter-thread communication method of a shared variable accessed in a critical section. Section 3 addresses this by supporting user defined channels and zero-copy optimization.

Structure of the program is significantly changed. While the sequential program was a single, short loop, the parallel program contains four loops, the code is dominated by communication and parallelism constructs and it takes more effort to determine the flow of data through the program.

Equally seriously, this major restructuring was performed to suit one particular architecture and achieve a reasonable load balance given the current speed of each function. If the architecture were to change or a function were optimized, the program would have to be restructured accordingly — a significant and error-prone undertaking. Section 4 addresses this using decoupling to automatically introduce pipeline parallelism under programmer control.

Fragmentation of variables. Each individual variable in the sequential program has been fragmented into many sep-

```
int timing_correction = 0;
parallel_sections{
  section{
    complex_t samples1[2048] @ {M0};
    int t1;
    while (1) {
      ADC_get(&adc,&samples1,2048);
      critical(offset){
        t1 = timing_correction;
      }
      AdjustTiming(t1,samples1) @ P0;
      fifo_put(&f1,samples1);
    }
  }
  section{
    complex_t samples2[2048] @ {M0};
    complex_t samples3[2048] @ {M1};
    while (1) {
      fifo_get(&f1,samples2);
      memcpy(samples3,samples2,sizeof(samples2)) @ DMA;
      FFT(samples3) @ P1;
      fifo_put(&f2,samples3);
    }
  }
  section{
    complex_t samples4[2048] @ {M1};
    complex_t samples5[2048] @ {M2};
    bool bits1[3024] @ {M2};
    int t2 = 0;
    while (1) {
      fifo_get(&f2,samples4);
      memcpy(samples5,samples4,sizeof(samples4)) @ DMA;
      t2 += FindTimeOffset(samples5)@P2;
      critical(offset){
        timing_correction2 = t2;
      }
      Demodulate(bits1,samples5) @ P2;
      fifo_put(&f3,bits1);
    }
  }
  section{
    bool bits2[3024] @ {M2};
    bool bits3[3024] @ {M3};
    int8_t bytes[378] @ {M3};
    while (1) {
      fifo_get(&f3,bits2);
      memcpy(bits3,bits2,sizeof(bits2)) @ DMA;
      ErrorCorrect(bytes,bits3) @ Viterbi;
    }
  }
}
```

Figure 4: A parallel version of the program in Figure 2.

arate variables in the parallel program. This fragmentation comes from two distinct sources: replicating variables that are communicated between threads and replicating variables between memory spaces. Whilst the replication is necessary, the burden on the programmer is significant: they may use the wrong version of a variable, they may fail to copy from one version of a variable to the other, or they may perform the copy but in the wrong direction. Sections 4 and 5 separately address the two sources of fragmentation.

Performance issues In addition to the impact on the structure of the application, the choice of synchronous RPCs, threads and FIFOs to express parallelism is convenient but runs the risk of a high overhead from copying data and swapping thread contexts. Section 8 shows that existing optimizations can be used to allow the use of high-level constructs without excessive overhead.

3. USER DEFINED CHANNELS

In Section 2, we observed that FIFO channels introduced too much synchronization and therefore used shared vari-

```

typedef struct { lock_t lock; int data; } atomic_int_t;

void atomic_int_init(atomic_int_t *a, int x)
{ lock_init(&a->lock); a->data = x; }

void atomic_int_put(atomic_int_t *a, int x)
__attribute__(( PUT(a, x) IN(x) ))
{ lock(&a->lock); a->data = x; unlock(&a->lock); }

void atomic_int_get(atomic_int_t *a, int *x)
__attribute__(( GET(a, x) OUT(x) ))
{ lock(&a->lock); *x = a->data; unlock(&a->lock); }

```

Figure 5: Implementation of atomic channels showing their dataflow annotations.

ables instead to achieve parallelism. The problem with using shared variables to communicate between threads is that it is harder to determine the direction of dataflow through shared variables, which makes it harder for programmers to understand and makes dataflow analysis less precise. To address this issue, SoC-C allows programmers to define new channel types to express directional dataflow.

SoC-C provides the usual array of thread primitives (locks, condition variables, etc.) to allow programmers to create their own channel operations. More importantly, SoC-C provides annotations to allow the programmer to specify that a function performs directional communication. Figure 5 contains a simple example: an “atomic channel” which allows one thread to pass data to another thread atomically. The most important aspect of this example is the annotations. The `PUT(a, x)` attribute specifies that the function argument ‘a’ is a channel used to communicate between threads and that the function argument ‘x’ is the data transferred into the channel. The `PUT` and `GET` attributes provide important information for the decoupling transformation described in Section 4 and the zero-copy transformation described in Section 8.1. `IN` and `OUT` attributes indicate dataflow through arguments in the usual way.

The `PUT` and `GET` attributes were originally added to SoC-C to let us quickly prototype new types of channel without the usual effort of having to add new intrinsic functions to tables in the compiler. We have since realized that most inter-thread communication and communication between threads and stream-oriented devices like Analog-to-Digital Convertors (ADCs) is directional and can be modelled as channels using our annotations. In addition to atomic channels, some examples of channels we use are:

Channel interfaces to ADCs and DACs High rate ADCs usually write data continuously into a circular buffer in memory. In addition to the channel and buffer arguments, it takes a size argument indicating how many samples are required.

```
void ADC_get(adc_t *adc, buffer_t *buf, unsigned sz);
```

Although it interacts with hardware, this function has the same semantics as any other “get” function: if the data is not yet available, the thread blocks until the data is available.

Timed channels provide time-indexed access to data. FIFO channels and atomic channels are at opposite ends of the spectrum on how puts and gets are matched: a FIFO channel matches each get with a unique put; while an atomic channel matches gets with the most recent put. Timed channels provide an intermediate semantic: data is timestamped and a get is matched with the put closest to the requested time.

```

void ts_put(tschan_t *c, int timestamp, void* v);
void ts_get(tschan_t *c, int timestamp, void* v);

```

The `ts_get` operation returns the entry with the closest timestamp to the one specified. All `ts_put` operations must use strictly increasing times and all `ts_get` operations must use strictly increasing times. This restriction allows entries to be discarded when they can no longer be accessed. Timed channels allow for more parallelism between threads since, after the first `ts_put` is performed, `ts_get` operations never block because there is always an entry with a closest timestamp. The cost of this increased performance is less precise synchronization between threads than with FIFO channels: applications that use timed channels are unlikely to give deterministic results.

4. DECOUPLING

In Section 2, we observed that manually introducing pipeline parallelism requires a significant restructuring of the program. There are many papers on avoiding this cost by automating the transformation: Smith [10] applies the technique manually to Cray assembly code which they referred to as “decoupling”; and Palacharla and Smith [9] use program slicing to automatically decouple a program into two threads communicating via FIFO channels: one containing load-store operations, the other containing all other operations. Subhlok et al. [11] have proposed syntax extensions for marking the start and end of pipeline stages within a loop body. These tools allow the programmer to identify what code should be in each section and then the compiler inserts FIFO channel operations as required.

SoC-C’s approach is similar in that it requires the programmer to indicate the boundaries between threads. It differs in that the programmer indicates the boundaries by inserting the communication between sections and the compiler determines which code must be in each section — the exact opposite of previous work. In practice, the difference in annotations is usually small: we insert similar annotations at similar parts of the program. We believe our emphasis on communication brings an important benefit: *the programmer is able to select an appropriate channel type in order to reduce synchronization between sections*. These decisions necessarily involve the programmer because using any channel other than a FIFO channel can change the meaning of the program. A secondary benefit is that our channel-based decoupling transformation can be applied to code containing complex control flow and is not restricted to being applied to loops — constraints applied by most prior work.

Figure 6 shows the program in Figure 4 rewritten to use our pipeline construct. There are three major differences. (1) It is possible to write the program as a single loop because decoupling can automatically split it into four parallel copies of the loop. (2) It is not necessary to introduce as many intermediate variables (`samples2`, `samples4`, `bits2`) because our transformation performs range-splitting to split any local variable with disjoint live ranges into multiple variables. (3) It is necessary to use an atomic channel to express the direction of dataflow for the `a_timing` variable.

SoC-C uses the syntax

```

pipeline{
  compound_statement
}

```

to indicate that the body of the compound statement is to be transformed into an equivalent set of parallel sections which

```

atomic_int_t a_timing;
atomic_int_init(&a_timing,0);
pipeline {
    complex_t samples1[2048] @ {M0};
    complex_t samples3[2048] @ {M1};
    complex_t samples5[2048] @ {M2};
    bool      bits1[3024]   @ {M2};
    bool      bits3[3024]   @ {M3};
    int8_t    bytes[378]    @ {M3};
    int       t1;
    int       t2 = 0;
    while (1) {
        ADC_get(&adc,&samples1,2048);
        atomic_int_get(&a_timing,&t1);
        AdjustTiming(t1,samples1) @ P0;
        fifo_put(&f1,samples1);
        fifo_get(&f1,samples1);
        memcpy(samples3,samples1,sizeof(samples1)) @ DMA;
        FFT(samples3) @ P1;
        fifo_put(&f2,samples3);
        fifo_get(&f2,samples3);
        memcpy(samples5,samples3,sizeof(samples3)) @ DMA;
        t2 += FindTimeOffset(samples5)@P2;
        atomic_int_put(&a_timing,t2);
        Demodulate(bits1,samples5) @ P2;
        fifo_put(&f3,bits1);
        fifo_get(&f3,bits1);
        memcpy(bits3,bits1,sizeof(bits1)) @ DMA;
        ErrorCorrect(bytes,bits3) @ Viterbi;
    }
}

```

Figure 6: A version of the program in Figure 2 written using the pipeline construct.

communicate (only) using the channel operations already present in the program. Since communication lies at the boundaries between threads, the compiler uses a dataflow analysis which “colors in” the code that lies between the boundaries. This analysis identifies the set of operations that are on the “producer” side of a channel and the set of operations on the “consumer” side of a channel. Repeating this for all channels and considering shared variables, the compiler partitions the operations into sets of operations which must be in the same thread. The body of the `pipeline` construct is then transformed into parallel sections replicating control flow and variables as required.

The decoupling algorithm must make two essential decisions: “What variables and operations to replicate?” and “What operations to place in the same thread?”

The task of decoupling is to split the region of code into as many threads as possible, without introducing timing-dependent behaviour, using channels to communicate between threads. Comparing Figure 4 with Figure 6 we observe that the generated threads do not strictly partition the statements in the original code: some variables and operations (principally those used for control) have been *privatized* (i.e., replicated in multiple threads) while others remain *shared*. The choice of what to privatize is an essential part of the transformation: if too much code or data is privatized, the transformed program can run more slowly and use more memory than the original program. While these decisions could be controlled using annotations on every variable and statement, SoC-C applies some simple default rules that give the programmer control without requiring excessive annotation. By default, scalar variables and variables declared inside the `pipeline` annotation may be privatized. Operations other than function calls may be privatized unless they have side-effects or modify a non-duplicable variable.

The other essential decision that the transformation must

make is “What operations must be in the same thread as each other?”. To avoid introducing timing-dependent behaviour, the compiler applies the following three rules:

1. To preserve data and control dependencies, any dependent operations must be in the same thread as each other unless the dependency is from a ‘put’ operations to a ‘get’ operation on the same channel. This special treatment of dependencies on channel operations has the effect of cutting edges in the dataflow graph.
2. To avoid introducing race conditions, any operations which write to a shareable, non-channel variable must be in the same thread as all operations which read from or write to that variable. Channels are excluded because all channel operations are required to atomically modify the channel.
3. To avoid introducing unwanted non-determinism, all puts to a given channel must be in one thread and all gets from a given channel must be in one thread.

Our implementation of decoupling finds the unique, maximal solution in four stages: live range splitting of privatizable variables, dependency analysis, merging, and thread production. To illustrate our method, we consider the following simple example.

```

1 pipeline{
2   for(int i=0; i<100; ++i) {
3     int x = foo();
4     if (i % 2 != 0) {
5       fifo_put(&f,x);
6       fifo_get(&f,&x);
7       bar(x);
8     }
9   }
10 }

```

The *dependency analysis stage* forms a large number of candidate threads by computing a backward data and control slice [16] from each unprivatized operation ignoring data dependencies on channel operations but including all other operations in the slice. That is, we repeatedly apply rules (1–3) to form candidate threads. In our running example, there are four candidates: one each for `foo()`, `fifo_put(&f,x)`, `fifo_get(&f,&x)` and `bar(x)`.

For example, the candidate for `foo()` is:

```

2   for(int i=0; i<100; ++i) {
3     int x = foo();
9   }

```

the candidate for `fifo_put(&f,x)` is:

```

2   for(int i=0; i<100; ++i) {
3     int x = foo();
4     if (i % 2 != 0) {
5       fifo_put(&f,x);
8     }
9   }

```

and the candidate for `bar(x)` is:

```

2   for(int i=0; i<100; ++i) {
3     int x;
4     if (i % 2 != 0) {
6       fifo_get(&f,&x);
7       bar(x);
8     }
9   }

```

The *merging stage* combines candidate threads by merging threads that contain the same un-privatizable operation or variable. For example, the candidate for `foo()` is merged

with the candidate for the operation `fifo_put(&f,x)` because they both contain the operation `foo()`. This results in the candidate thread:

```

2  for(int i=0; i<100; ++i) {
3      int x = foo();
4      if (i % 2 != 0) {
5          fifo_put(&f,x);
6      }
7  }
8  }
9  }
```

This is identical to the candidate for `fifo_put(&f,x)` because the candidate already contained the `x=foo()` operation. Similarly, the result of merging the candidate for `bar(x)` with the candidate for the operation `fifo_get(&f,&x)` is identical to the candidate for `bar(x)`.

The *thread production* stage converts candidate threads to threads by privatizing variables and combining the result using parallel sections.

Syntactic sugar We have found that it is common for pairs of put and get operations to be adjacent. In recognition of this, we added a small piece of syntactic sugar: “`FIFO(x);`”. This is equivalent to a put followed by a get on variable `x` and that also declares and initializes a fifo channel. This syntax is illustrated in Figure 7.

5. COMPILER-SUPPORTED COHERENCY

In Section 2, we saw that distributed memory leads to variable fragmentation: if functions running on different processors access the same variable, we must create a version of the variable for each memory region. This is tedious and error prone because it is hard to understand the original design intent. To address this problem, we extend the data placement notation to allow the programmer to express the fact that the additional variables are just versions of the same variable. This preserves the structure of the original design and allows the compiler to detect errors using a single compile-time coherence protocol.

We allow the programmer to assign a variable to multiple memory regions. For example, the declaration

```
bool bits[2048] @ {M2,M3};
```

introduces two copies of the variable `bits`: one in memory M2 (written `bits@M2`) and one in memory M3 (written `bits@M3`).

Semantically, the different versions of a variable behave like copies in a coherent cache: an assignment to one version of `bits` (logically) invalidates the contents of the other version. An invalid version of a variable can be made valid by synchronizing it with a valid version of the same variable. The statement

```
SYNC(bits,M3,M2) @ DMA;
```

makes `bits@M2` valid if `bits@M2` was already valid and is an error if `bits@M0` was invalid. A `SYNC` statement is compiled into a copy operation which, in this example, is to be performed on engine DMA. Figure 7 illustrates how using variable coherency annotations simplifies the task of keeping track of all the different variables in Figure 6.

Adding support for multiple coherent versions of a variable required the following compiler changes. Various trivial changes to support the new syntax; to transform uses of variables to use the appropriate version; and to transform `SYNC` constructs into `memcpy` operations. Checking for coherence errors is performed using a flow-sensitive, context-insensitive, field-insensitive forward dataflow analysis:

```

atomic_int_t a_timing;
atomic_int_init(&a_timing,0);
pipeline {
    complex_t samples[2048] @ {M0,M1,M2};
    bool      bits[3024]   @ {M2,M3};
    int8_t    bytes[378]   @ {M3};
    int       t1;
    int       t2 = 0;
    while (1) {
        ADC_get(&adc,&samples@M0,2048);
        atomic_int_get(&a_timing,&t1);
        AdjustTiming(t1,samples@M0) @ P0;
        FIFO(samples@M0);
        SYNC(samples,M1,M0) @ DMA;
        FFT(samples@M1) @ P1;
        FIFO(samples@M1);
        SYNC(samples,M2,M1) @ DMA;
        t2 += FindTimeOffset(samples@M2)@P2;
        atomic_int_put(&a_timing,&t2);
        Demodulate(bits@M2,samples@M2) @ P2;
        FIFO(bits@M2);
        SYNC(bits,M3,M2) @ DMA;
        ErrorCorrect(bytes@M3,bits@M3) @ Viterbi;
    }
}
```

Figure 7: The effect of rewriting Figure 6 using variable coherency annotations. Changes are highlighted in bold.

1. Each version of each variable can be valid or invalid;
2. A kill makes all versions of a variable invalid;
3. A def to a version of a variable makes that version valid and all others invalid;
4. A `SYNC` statement copies validity from one version of a variable to another;
5. A use checks that the version used is valid; and
6. When flow merges, a version is valid only if it is valid in all incoming edges.

To illustrate the kind of errors these checks detect, suppose the programmer had accidentally reversed the first two arguments in the first call to `memcpy` in Figure 6. Since the programmer’s intent is not clear, it would be hard for a compiler to detect this error. In contrast, reversing the M0 and M1 arguments in the first `SYNC` statement in Figure 7 is detected as a coherence error by our compiler: the `FIFO` on the previous line defines `samples@M0` which invalidates `samples@M1` but the `SYNC` reads from `samples@M1`.

This coherency mechanism meets our primary goal of supporting safe, statically checked use of distributed memory between processors but within a single thread. Inter-thread coherency checking would require dynamic checking of the ownership of a variable and synchronization — which we think is better expressed using channels. A consequence of not supporting inter-thread coherence is that we perform coherence checking and transformation before decoupling to eliminate coherence annotations before creating new threads.

6. PLACEMENT INFERENCE

Supporting multiple coherent versions of a variable helps communicate the intent of the programmer and, hence, detect errors but it requires that every single use of a variable is annotated. To reduce this annotation burden, we replace *coherence checking* with *placement inference* which exploits the observation that the annotations contain a high degree of redundancy:


```

atomic_int_t a_timing;
atomic_int_init(&a_timing,0);
pipeline {
  complex_t samples[2048];
  bool      bits[3024];
  int8_t    bytes[378];
  int       t1;
  int       t2 = 0;
  while (1) {
    ADC_get(&adc,&samples,2048);
    atomic_int_get(&a_timing,&t1);
    AdjustTiming(t1,samples) @ P0;
    FIFO(samples);
    SYNC(samples) @ DMA;
    FFT(samples) @ P1;
    FIFO(samples);
    SYNC(samples) @ DMA;
    t2 += FindTimeOffset(samples) @ P2;
    atomic_int_put(&a_timing,t2);
    Demodulate(bits,samples) @ P2;
    FIFO(bits);
    SYNC(bits) @ DMA;
    ErrorCorrect(bytes,bits) @ Viterbi;
  }
}

```

Figure 8: An OFDM radio receiver mapped onto a complex architecture using the full set of SoC-C annotations.

- If P can only access one memory M, and the program contains an RPC “foo(x)@P,” then variable x must be placed in memory M and that x must have a version in memory M.
- If there is only one valid version x@M of a variable at the site of a SYNC(x) statement, then the only legal source of the SYNC is x@M.
- If x@M is the only version of a variable whose use is reachable from a SYNC(x) statement, then the only sensible target of the SYNC is x@M.

These three observations follow a common pattern: if there is only one valid choice, assume that choice is true. Our coherence inference algorithm is similar to flow-sensitive type inference: it uses annotations and the memory topology to add constraints to the system (e.g., an RPC ‘f(x)@P’ provides the constraint that ‘x’ must be in a memory accessible by ‘P’ while a reference to ‘x@M0’ provides the constraint that ‘x@M0’ is valid. Having gathered all the constraints, we use forward-chaining inference to add additional constraints. When no more constraints can be inferred, we choose an open question and test all possible answers to see if they break the constraints. If precisely one possible solution does not break the constraints, then we assume that it is the correct solution and repeat the inference process. This is repeated until no more open questions can be resolved in this way. This process results in a unique solution if there is one because it makes an assumption only if it can prove that all other alternatives are invalid.

In practice we find that SoCs which have multiple memory regions also have sufficiently restricted memory topologies that the compiler can infer most annotations. For example, Figure 8 shows the effect of applying our annotations to the code in Figure 7: all data placement annotations can be inferred in this example.

7. EVALUATING SOC-C ANNOTATIONS

Having completed our presentation of SoC-C, we consider how effective the annotations are at expressing SoC programs. Comparing Figure 8 with the sequential code, we

see that to map and parallelize we added: 8 code placement annotations; 0 data placement annotations; 3 SYNC statements; 3 FIFO statements to pass data between threads; and 2 put/get operations on atomic operations.

While annotations and additional statements have been inserted, the structure of the code is unchanged; to port the parallelized code to a new platform, the worst case is that one would delete all the annotations and start again.

Most importantly, we claim that there is little redundancy in the code: most of the changes are independent of the other changes. This suggests that SoC-C allows programmers to express design decisions rather than focussing on the mechanics of making the program correct and consistent.

8. KEY OPTIMIZATIONS

In Section 2 we identified two potential performance issues in our choice of abstraction: the cost of copying buffers into and out of channels; and the cost of using synchronous RPC and threads. This Section describes the (previously known) optimizations we apply to make the cost of these abstractions acceptable.

8.1 Optimizing channels

The semantics of channels is that **put** operations transfer data into the channel and **get** operations transfer data out. This simple semantics ensures that each thread has exclusive access to the data but a literal implementation would require a lot of unnecessary data copying. Network stacks, filesystems and embedded systems often provide a “zero copy” interface which pass pointers instead of copying data. For example the Task Transaction Interface [14, section 4.1.5] splits “put” operations into two operations. “**acquireRoom**” allocates the next empty buffer in a channel; the producer should then write data into the buffer and call “**releaseData**” to make the data available to the consumer. Similarly, “get” operations are split into “**acquireData**” and “**releaseRoom**”.

Supporting this style of channel interface required three changes. First, for all channels which hold large buffers, we rewrote the channel implementation to support a zero-copy interface. For example, the zero copy operations corresponding to **fifo_put** are:

```

void fifo_acquireRoom(fifo_t *f, void* *room);
void fifo_releaseData(fifo_t *f, void* data);

```

Secondly, we added attributes to the “put” and “get” functions identifying that these functions could be zero-copy optimized and naming the two associated functions. The augmented set of attributes on the **fifo_put** function is:

```

void fifo_put(fifo_t *fifo, void *data)
__attribute__(( PUT(fifo, data) IN(data)
                ZEROCOPY(fifo1_acquireRoom,
                        fifo1_releaseData) ));

```

Finally, we modified the compiler to analyze the live range of buffers passed to functions with ZEROCOPY attributes and insert calls to the two functions at the starts of the live range and at the ends of the live range.

The optimization cannot be used if the live range does not end at a **put** (or start with a **get**), for example, if a variable is put into multiple channels or is used after the **put** operation. For bulk data types, the cost of not optimizing away the copy may be significant so, when zero-copying cannot be used, our compiler reports a warning prompting the programmer to change their code.

```

section{
  complex_t *p_samples2;
  complex_t *p_samples3;
  while (1) {
    fifo_acquireData(&f1,&p_samples2);
    fifo_acquireRoom(&f2,&p_samples3);
    memcpy(p_samples3,p_samples2,...) @ DMA;
    fifo_releaseRoom(&f1,p_samples2);
    FFT(p_samples3) @ P1;
    fifo_releaseData(&f2,p_samples3);
  }
}

```

Figure 9: The effect of zero-copy optimization.

Figure 9 shows the effect of applying zero-copy optimization to the second code section in Figure 4. Using this transformation typically reduces the channel operations to just a small amount of book-keeping. For example, the `put` operation on an atomic channel is split into an operation to wait for the buffer to become available followed later by an operation to release the buffer to other users of the channel. These operations are exactly the same as the lock/unlock operations on a mutex that would normally have been used: our optimizations result in the same low-level, efficient implementation that embedded programmers normally use without the semantic complexity of using shared variables directly.

8.2 Optimizing thread implementation

SoC-C provides synchronous RPCs and uses threads to express sequencing of operations and parallelism. In embedded systems, it is more usual to provide asynchronous (aka non-blocking) RPCs and use an event-driven programming style to express sequencing of operations and parallelism. SoC-C’s choice is simpler to use but a conventional thread implementation would incur a large space overhead to store thread contexts and a large time overhead performing context switches when an engine sends an interrupt to signal that it has completed a function call.

To achieve the simplicity of threads with the efficiency of event-driven programming, we use an old trick: we transform threads into event-driven programs [6]. Our compiler transforms each thread into a state machine where states represent points where the program may block on an event and edges are labelled with event handlers which execute code from the thread and update the current state. For example, Figure 10 shows the state machine corresponding to the code in Figure 9.

A typical execution sequence is as follows. The processor spends most of its time in a low-power state waiting for an interrupt with all threads either blocked on a condition variable waiting for a processing element to complete execution or blocked on a channel. On receiving an interrupt signalling completion of a task, the processor invokes an interrupt handler which acknowledges the interrupt and invokes an event handler for the thread currently waiting for that task to complete. This handler typically starts a new task on a processing element and blocks waiting for the task to complete or waiting for the processing element to become available. If the event handler released a lock or put data into a channel before it blocked, the handler may trigger other event handlers: the completion of a single processing element can cause a cascade of event handlers as results propagate to other threads and buffers are freed. When all event handlers in this cascade have executed, all threads are

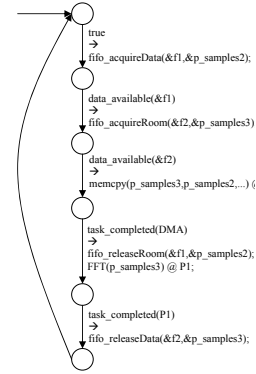


Figure 10: The state machine for Figure 9.

once more blocked on a condition variable or a channel and the control processor returns from the interrupt handler and reenters a low-power state.

8.3 Dataflow Analysis and Phase Ordering

Our compiler relies on the ability to perform a sufficiently accurate dataflow analysis. Since we wish to keep the programmer “in the loop”, we limited ourselves to a simple dataflow analysis that was easy to understand. Accordingly, our analysis is flow sensitive, field-insensitive, context-insensitive. Our pointer analysis is just sufficient to analyze programs that use pointers to pass arguments by reference; programmers are encouraged to create abstract data types to hide any other use.

The dataflow analysis used for decoupling and other transformations requires programmer annotations on function arguments. We rely on programmer annotation to determine whether a pointer argument is an ‘in’ argument (indicated by C’s `const` qualifier), an ‘out’ argument (indicated by an attribute) or an ‘in-out’ argument (the default). Similarly, if a function accesses a global variable, the function prototype must be annotated to indicate whether it is an ‘in’, ‘out’ or ‘in-out’ variable. If a function modifies just one field in a struct or array, the argument is recorded as an ‘in-out’ argument indicating that the function does not “kill” the entire argument. In practice, we find that most of this information is already documented informally or can be obtained as a side-effect of function compilation.

Our compiler performs the transformations described in this paper in the following order: dataflow analysis, placement inference/checking, splitting variables with multiple placements into separate variables, zero-copy optimization, decoupling and transforming threads into state machines. Performing dataflow analysis early is important because it allows us to analyze the code before additional pointers are introduced and to give accurate programmer feedback. Coherency checking is performed before decoupling because coherency checking can only be applied within a thread. Zero-copy optimization can be performed either before or after decoupling; to allow it to be performed before decoupling, the ‘`releaseData`’ and ‘`acquireData`’ operations need to be annotated with ‘`PUT`’ and ‘`GET`’ attributes.

Our SoC-C compiler is written as a source to source compiler implemented using Necula et al.’s wonderful CIL [7] C processing framework, 5800 lines of O’Caml code and around 5000 lines of runtime support code including device drivers.

activity	cycles
enter irq handler	10
clearing interrupts	20
start data engine	39
lock overhead	34-38
FIFO transfer overhead	54-55

Figure 11: Performance of SoC-C Implementation

9. PERFORMANCE EVALUATION

This Section evaluates the performance of SoC-C using two criteria: we establish the efficiency of our implementation using microbenchmarks; and we measure how performance of a high performance “software defined radio” application scales with the number of processors.

All measurements were based on a multiprocessor system being developed by ARM Ltd. to implement the physical layer processing of 3.9G mobile phones. This platform centres around a configurable number of moderate-frequency, highly parallel C-programmable data processing engines implemented using ARM’s OptimoDE design technology. These processors exploit both data-parallelism using a very wide SIMD (512-bit) datapath and exploit instruction-level parallelism using VLIW instruction decoding. These OptimoDE engines have a 512-bit data bus to memory and are supported by a DMA engine capable of 512-bit wide transfers. We evaluated using a platform configured to use between 1 and 4 of these data engines as shown in Figure 3. SoC-C code runs on a Cortex-M3 RISC processor with a 32-bit tightly coupled memory. The primary task of the RISC processor is to control data engines, DMA, etc. and to interact with processors executing the higher layers of the network protocol stack. All measurements were made using cycle-accurate models of the data engines, DMA engine, RISC processor and memory system and cycle-approximate models of the peripherals.

9.1 Performance of the runtime system

One of the most important metrics is how long a data engine spends idle between tasks. The control processor must perform the following steps: 1) Complete the current instruction and enter the interrupt handler; 2) Acknowledge the interrupt to the device; 3) Execute the appropriate event handler including constructing a call frame and starting the data engine. Using the simulator we monitored the start/stop signals from data engines, the interrupt signals and the program counter on the control processor and obtained very precise, repeatable measurements to be made (Figure 11). The total time that a data engine is idle between tasks is 69 cycles.

In practice, it is usually necessary to use locks to prevent two threads from using the same engine at once. Locking increases the idle time by 50% to 103–107 cycles. When two threads communicate via a FIFO queue, the time between the completion of a task on one thread and the start of a task in the other thread is 157–162 cycles.

In comparison, our experience is that commercial RTOSs require more than 300 cycles to enter an interrupt handler and trigger a thread context switch. The extra 150–200 cycles may appear negligible until one considers that in that time, our SIMD data engine could have performed another 4500–6000 fixed point multiply operations.

cores	ideal time	actual time	utilization	speedup
1	29286	31101	94%	1.00
2	15013	16865	89%	1.84
4	7876	9077	87%	3.43

Figure 12: Scaling of DVB application.

9.2 Scalability

This Section evaluates how well performance scales as the number of processors is varied using the inner receiver of a Digital Video Broadcast (DVB) physical layer as a benchmark. This has a similar structure and dataflow to our running example but, in addition, it performs: coarse-timing correction to maintain synchronization over long time periods, demultiplexing of data, control bits and pilot channels; channel equalization to correct for fading of individual frequency channels; de-interleaving of the data to reduce sensitivity to bursts of noise. Odd and even symbols require slightly different processing requiring the compiler to decouple code containing if-statements and the two paths have slightly different execution times. Our receiver consists of around 9000 lines of C code split into 17 DSP functions which execute on the SIMD data engines. The total number of cycles of the functions and three DMA transfers is 29286 cycles of which 740 cycles are DMA transfer. Task granularity varies considerably: there are 2 tasks of almost 7000 cycles, 3 tasks of more than 3000 cycles, 1 task of 1000 cycles and the remainder are 500 cycles or less.

We used SoC-C to combine these functions into a single-threaded application and created two pipelined versions of the program for platforms with two and with four SIMD data engines by inserting FIFOs and atomic channels and changing the placement annotations.

We measured the maximum sustainable rate at which a stream of 2K point DVB symbols can be processed measured in cycles per symbol and calculated the best possible time for a system with one DMA engine and N cores given our code placement decisions and function runtimes and ignoring data dependencies which would prevent perfect parallelization. Ignoring data dependencies makes this number a little conservative (too low). We calculated utilization as the ratio between the ideal rate and the actual rate and calculated speedup as the ratio of actual rate against the actual rate of the 1-core variant. The results are summarized in Figure 12. The results for a single core demonstrate the effectiveness of our implementation strategy: the overhead of using SoC-C is just 1800 cycles (6%) which matches our expectation from the microbenchmarks. On two cores, the application speeds up by a factor of 1.84 compared with the single core version. We were unable to achieve perfect speedup because the coarse granularity of tasks made it impossible to perfectly balance the load. On four cores, the application speeds up by a factor of 3.43 compared with the single core version despite coarse task granularity.

10. DISCUSSION AND RELATED WORK

SoC-C’s major influences are stream programming languages such as StreamIt [5] which emphasize pipeline parallelism and have a clear separation of the communication language from the kernel language. We maintain the separation of communication/control layer (SoC-C) from computation (code called by RPCs) but we chose a sequential communication language instead of a dataflow language because we found

it hard to express global control (i.e., conditionals that span multiple pipeline stages) over pipeline stages that execute asynchronously with respect to each other. Using decoupling to introduce parallelism, gives the ease of expression of global control that imperative languages provide combined with the pipeline parallelism that stream languages provide.

Decoupling has been applied many times; we cite a representative sample. Smith [10] applies the technique manually to Cray assembly code to separate load-store operations from other operations to program Access-Execute processors; [9] automated this transformation; [4, 8, 2, 3] decouple programs automatically based on load-balancing heuristics; [11, 13] rely on programmer annotations to mark the beginning and end of pipeline stages. All these papers rely on partitioning of the operations into pipeline stages and then inserting FIFO channels. Our channel-based decoupling algorithm does the opposite: it relies on the programmer inserting channels and partitions the operations accordingly. The difference is small but significant: making the channels first-class concepts, instead of mere implementation details, lets the programmer use different channel types to explicitly relax synchronization between pipeline stages to avoid loss of decoupling. We are not aware of any work that uses non-FIFO channels when automatically decoupling a sequential program though StreamIt's "teleport messaging" [12] provides a related feature for dataflow languages.

Although SoC-C borrows syntax from OpenMP, the two languages target very different systems and parallelism patterns: OpenMP targets SMP systems and supports data parallelism using annotations on for-loops; SoC-C targets AMP systems and, hence, supports pipeline parallelism.

EXOCHI [15] also tackles the problem of programming heterogeneous multicore systems but is complementary since they focus on coping with multiple instruction sets/toolchains, providing shared virtual memory and dynamically allocating tasks to accelerators whereas we focus on distributed memory, static allocation of tasks and decoupling.

There has been a large body of work on software distributed shared memory and on reducing cache-coherency traffic between threads using compiler techniques. SoC-C's approach is to express inter-thread communication (which requires dynamic checks) using channels and restrict coherence checking for intra-thread, inter-processor communication (which our compiler checks statically).

SoC-C handles data copying differently from many systems: RPCs normally copy bulk data structures; FIFO channels normally copy data both on a put and a get; private memories are often used to store local copies of variables whose master copy is in shared memory. Instead, SoC-C gives explicit control over data copying and SoC-C provides support to make this less burdensome and error-prone.

11. CONCLUSIONS

Mapping an application onto low-power, high-performance SoCs is a challenging problem due to the architectural complexity needed to achieve high energy efficiency. A common approach to the problem of complex hardware is to use software libraries to hide the complexity from the user. To achieve significantly higher energy efficiency we take a different approach: SoC-C provides the programmer with explicit control over how an application is mapped onto an architecture without requiring significant manual restructuring. Any language requires careful implementation and choice of

optimizations to minimize overhead: our compiler is able to speedup a coarse-grained, real-world application by a factor of 3.4 on a four-core platform achieving utilization of 87%.

12. REFERENCES

- [1] P. L. Bird, A. Rawsthorne, and N. P. Topham. The effectiveness of decoupling. In *International Conference on Supercomputing*, pages 47–56, 1993.
- [2] M. Bridges et al. Revisiting the sequential programming model for multi-core. In *MICRO 2007: Proc. of Symposium on Microarchitecture*, 2007.
- [3] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *PLDI 2005*, pages 237–248, 2005.
- [4] W. Du, R. Ferreira, and G. Agrawal. Compiler support for exploiting coarse-grained pipelined parallelism. In *Proc. of Conf. on High Performance Networking and Computing (SC2003)*, page 8, 2003.
- [5] M. I. Gordon et al. A stream compiler for communication-exposed architectures. In *Proc. Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.
- [6] H. Lauer and R. Needham. On the duality of operating system structures. In *Proc. Symposium on Operating Systems*, 1978.
- [7] G. C. Necula et al. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proc. Int. Conf. on Compiler Construction*, pages 213–228. Springer-Verlag, 2002.
- [8] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO '05: Proc. Int. Symposium on Microarchitecture*, Nov 2005.
- [9] S. Palacharla and J. E. Smith. Decoupling integer execution in superscalar processors. In *MICRO 28: Proc. of International Symposium on Microarchitecture*, pages 285–290, 1995.
- [10] J. E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, 1984.
- [11] J. Subhlok et al. Exploiting task and data parallelism on a multicomputer. In *Proc. of Symp. on Principles and Practice of Parallel Programming*, 1993.
- [12] W. Thies et al. Teleport messaging for distributed stream programs. In *PPoPP '05: Proc. of Symposium on Principles and Practice of Parallel Programming*, pages 224–235. ACM Press, 2005.
- [13] W. Thies et al. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO 2007*, 2007.
- [14] P. van der Wolf et al. Design and programming of embedded multiprocessors: An interface-centric approach. In *CODES+ISSS'04: Hardware/Software Codesign and System Synthesis*, 2004.
- [15] P. H. Wang et al. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proc. PLDI*, 2007.
- [16] M. Weiser. Program slicing. In *ICSE '81: Proc. of International Conference on Software Engineering*, pages 439–449, 1981.

Chapter 6

Reducing inter-task latency in a multiprocessor system (**Patent I**)

Alastair David Reid. Reducing inter-task latency in a multiprocessor system, January 22 2013.
US Patent 8,359,588.

(12) **United States Patent**
Reid

(10) **Patent No.:** **US 8,359,588 B2**
(45) **Date of Patent:** **Jan. 22, 2013**

(54) **REDUCING INTER-TASK LATENCY IN A
MULTIPROCESSOR SYSTEM**

2008/0114937 A1 5/2008 Reid et al.
2008/0215768 A1 9/2008 Reid et al.
2010/0153934 A1* 6/2010 Lachner 717/146

OTHER PUBLICATIONS

(75) Inventor: **Alastair David Reid**, Cambridgeshire
(GB)

Title: Reducing Memory ordering overheads in software transactional memory, author: Spear, M. F, dated: Mar. 22, 2009 source: IEEE.*

(73) Assignee: **Arm Limited**, Cambridge (GB)

Title: Reducing memory latency using a samll software driven array cache, author: Chi-Hung Chi, dated: Jan. 3, 1995, source: IEEE.*
Eide et al., "Flick: A Flexible, Optimizing IDL Compiler", *ACM SIGPLAN '97 Conference*, Jun. 15-18, 1997, pp. 1-13.

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 662 days.

Bershad et al., (Presentation) "Lightweight RPC", 17 pages, Dated: 1990.

(21) Appl. No.: **12/591,661**

Don Sandler, (Presentation) "Optimizing RPC", *Comp 520*, Sep. 9, 2004, 35 pages.

(22) Filed: **Nov. 25, 2009**

Allen et al., "Conversion of Control Dependence to Data Dependence", *Dept. of Mathematical Sciences*, Rice University, 1983, pp. 177-189.

(65) **Prior Publication Data**

US 2011/0125986 A1 May 26, 2011

* cited by examiner

(51) **Int. Cl.**
G06F 9/45 (2006.01)

Primary Examiner — Chameli Das

(52) **U.S. Cl.** **717/159**; 717/151; 717/157; 717/149;
717/133; 719/330

(74) *Attorney, Agent, or Firm* — Nixon & Vanderhye P.C.

(58) **Field of Classification Search** None
See application file for complete search history.

(57) **ABSTRACT**

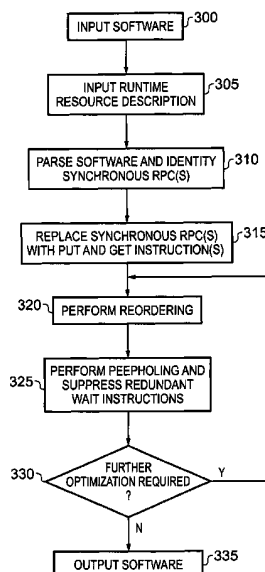
A method of reducing inter-task latency for software comprising a sequence of instructions including a synchronous remote procedure call to be executed on a multiprocessor system comprising a calling processor and at least one remote engine. The method comprises the steps of: inputting the software; inputting a runtime resource description describing a runtime environment of the multiprocessor system; identifying the synchronous remote procedure call in the sequence of instructions; replacing the synchronous remote procedure call in the sequence of instructions with an initiation instruction and a wait instruction to generate a substitute sequence of instructions; reordering the substitute sequence of instructions with reference to the runtime resource description and the dependencies to generate a reordered sequence of instructions; and outputting the reordered sequence of instructions.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,404,523 A * 4/1995 DellaFera et al. 718/101
5,822,563 A * 10/1998 Sitbon et al. 719/330
5,860,010 A * 1/1999 Attal 717/137
6,088,511 A * 7/2000 Hardwick 717/149
6,106,575 A * 8/2000 Hardwick 717/119
6,438,551 B1 * 8/2002 Holmskar 708/105
6,826,763 B1 * 11/2004 Wang et al. 719/330
7,624,398 B2 * 11/2009 Wang et al. 719/315
2002/0078255 A1 * 6/2002 Narayan 709/316
2005/0273772 A1 * 12/2005 Matsakis et al. 717/136

37 Claims, 11 Drawing Sheets



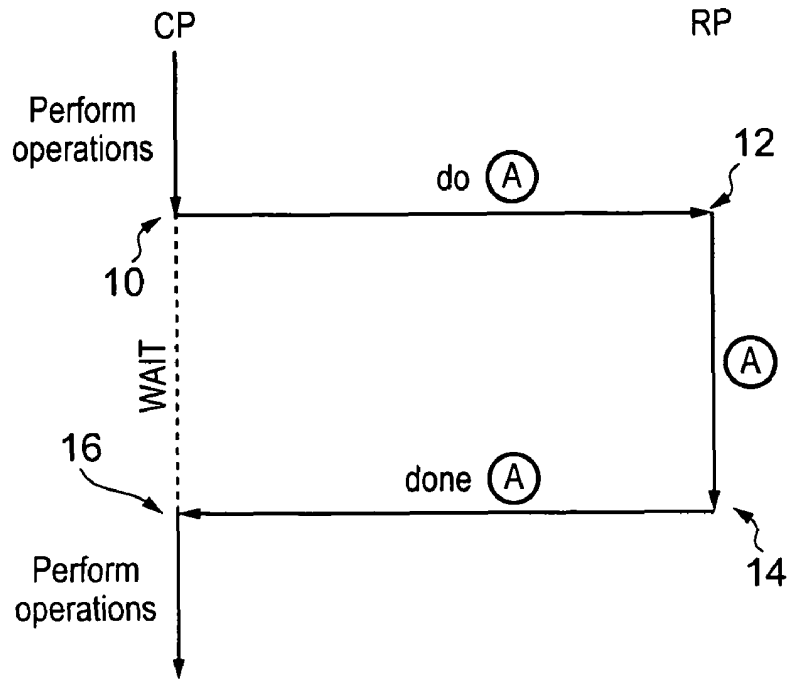


FIG. 1A (PRIOR ART)

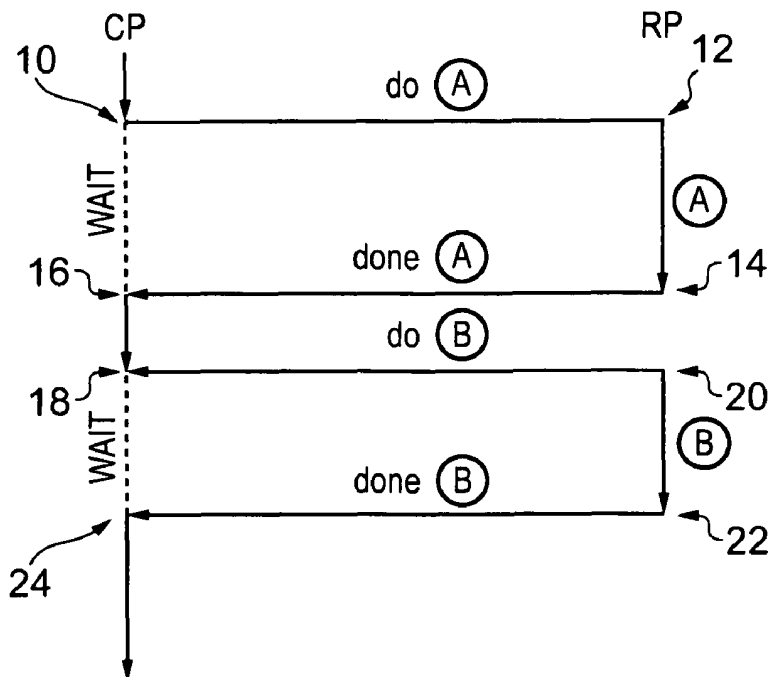


FIG. 1B (PRIOR ART)

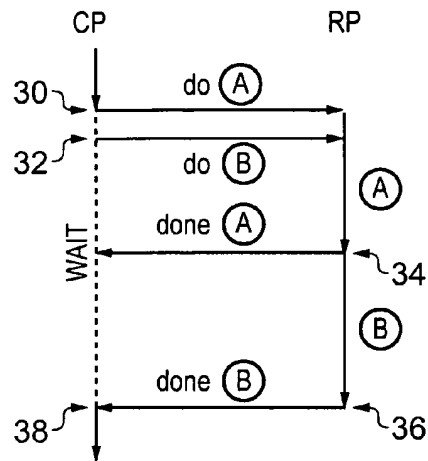


FIG. 2A (PRIOR ART)

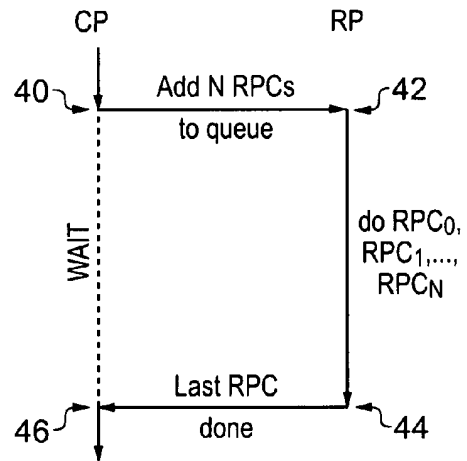


FIG. 2B (PRIOR ART)

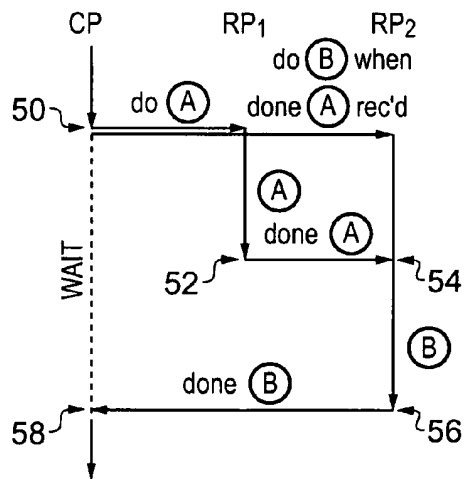


FIG. 2C (PRIOR ART)

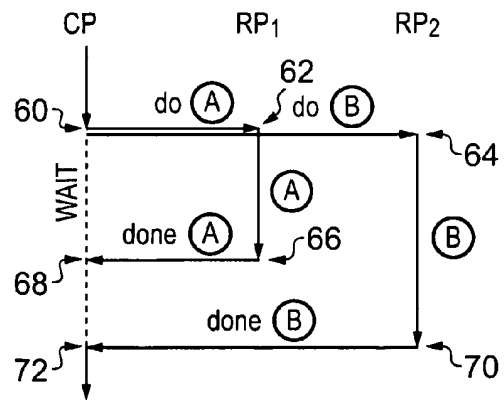


FIG. 2D (PRIOR ART)

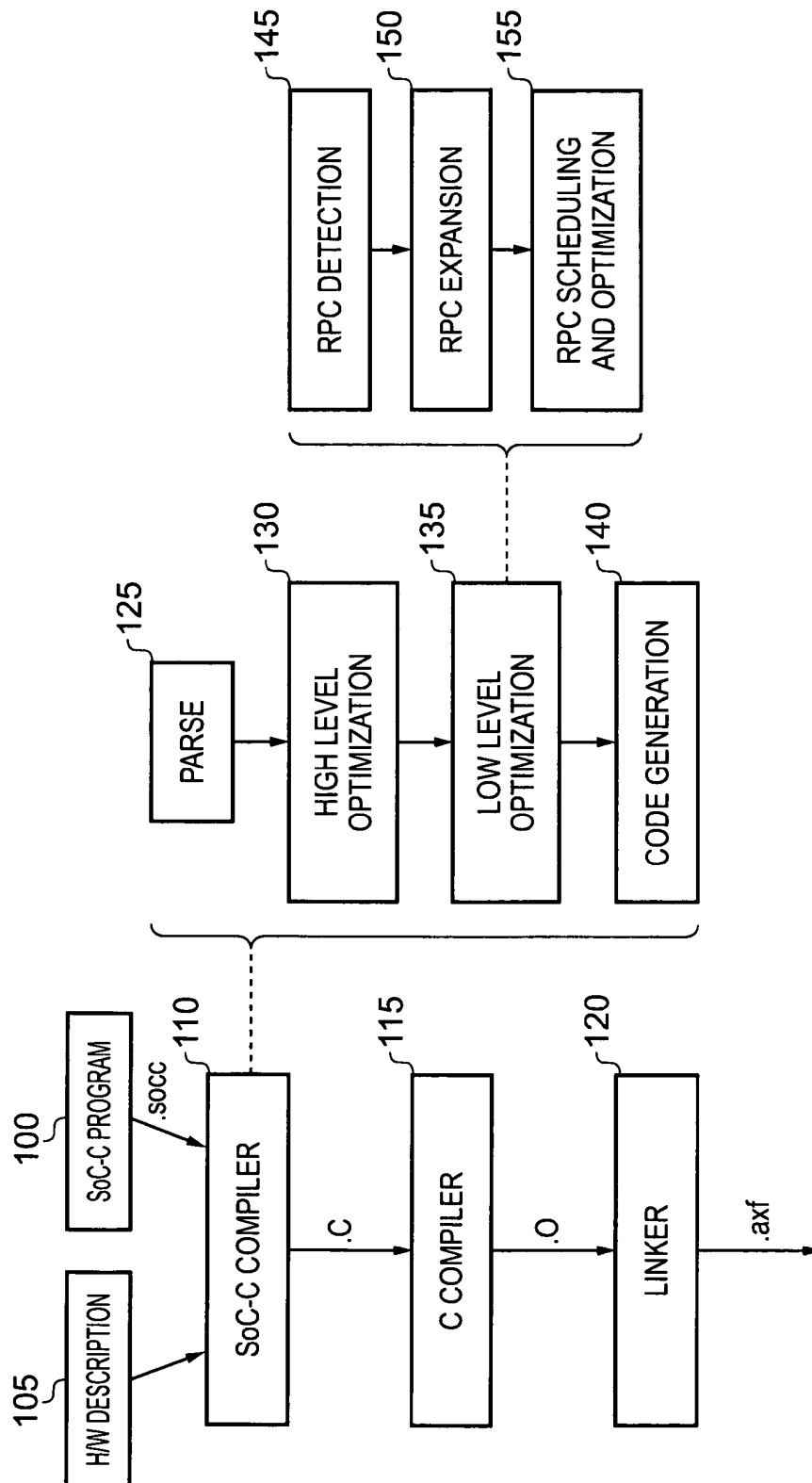


FIG. 3

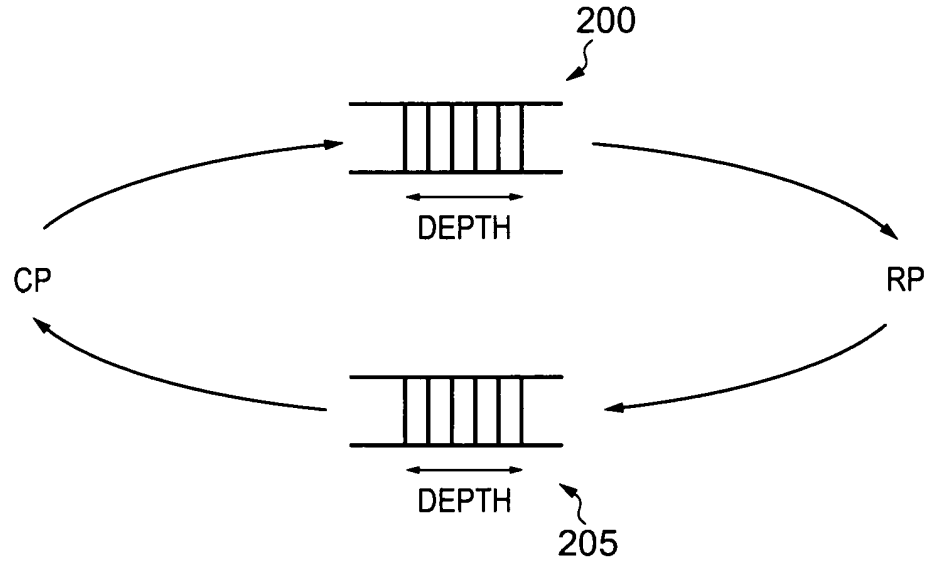


FIG. 4A

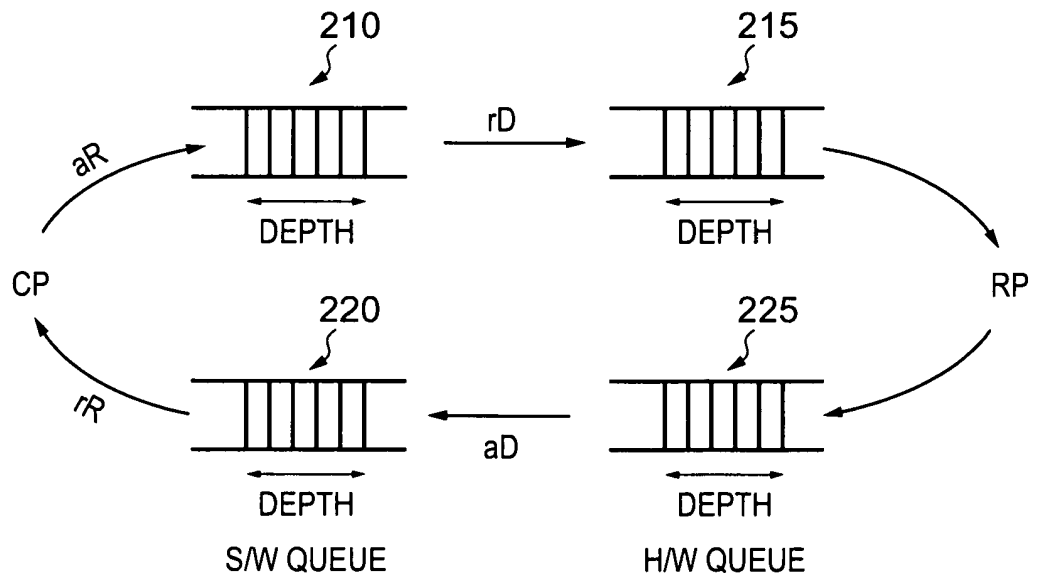


FIG. 4B

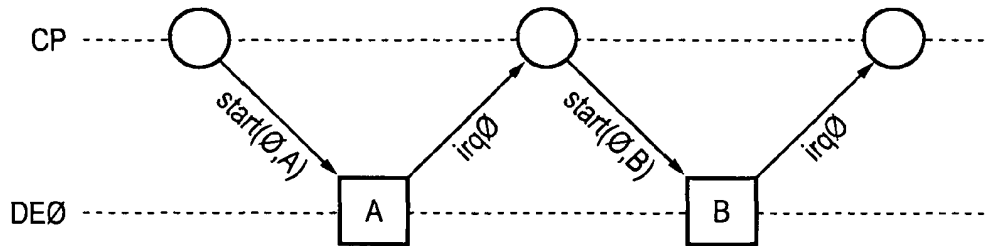


FIG. 5A

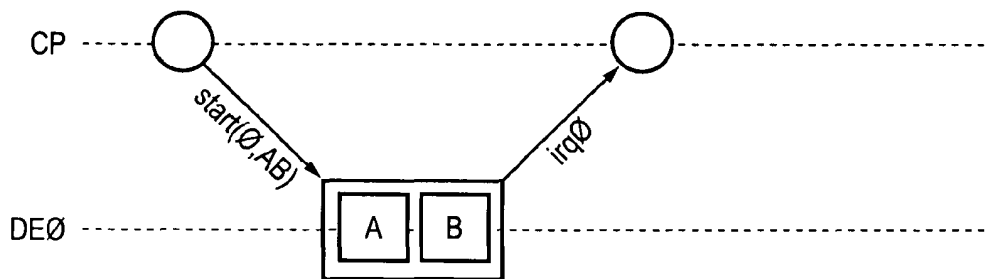


FIG. 5B

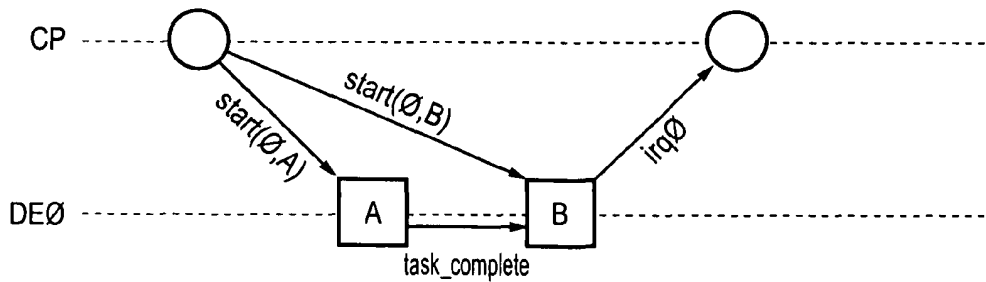


FIG. 5C

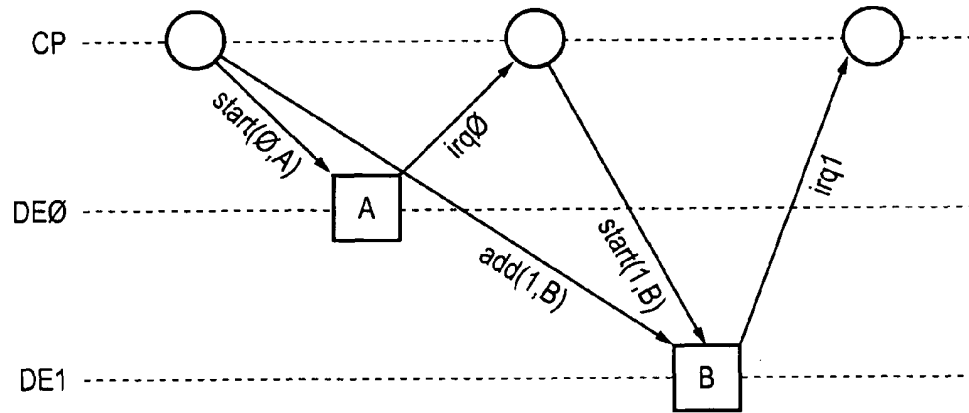


FIG. 6A

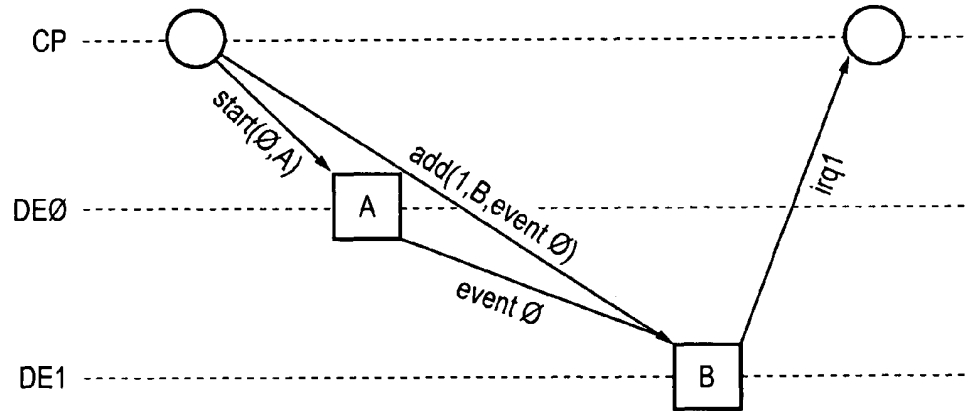


FIG. 6B

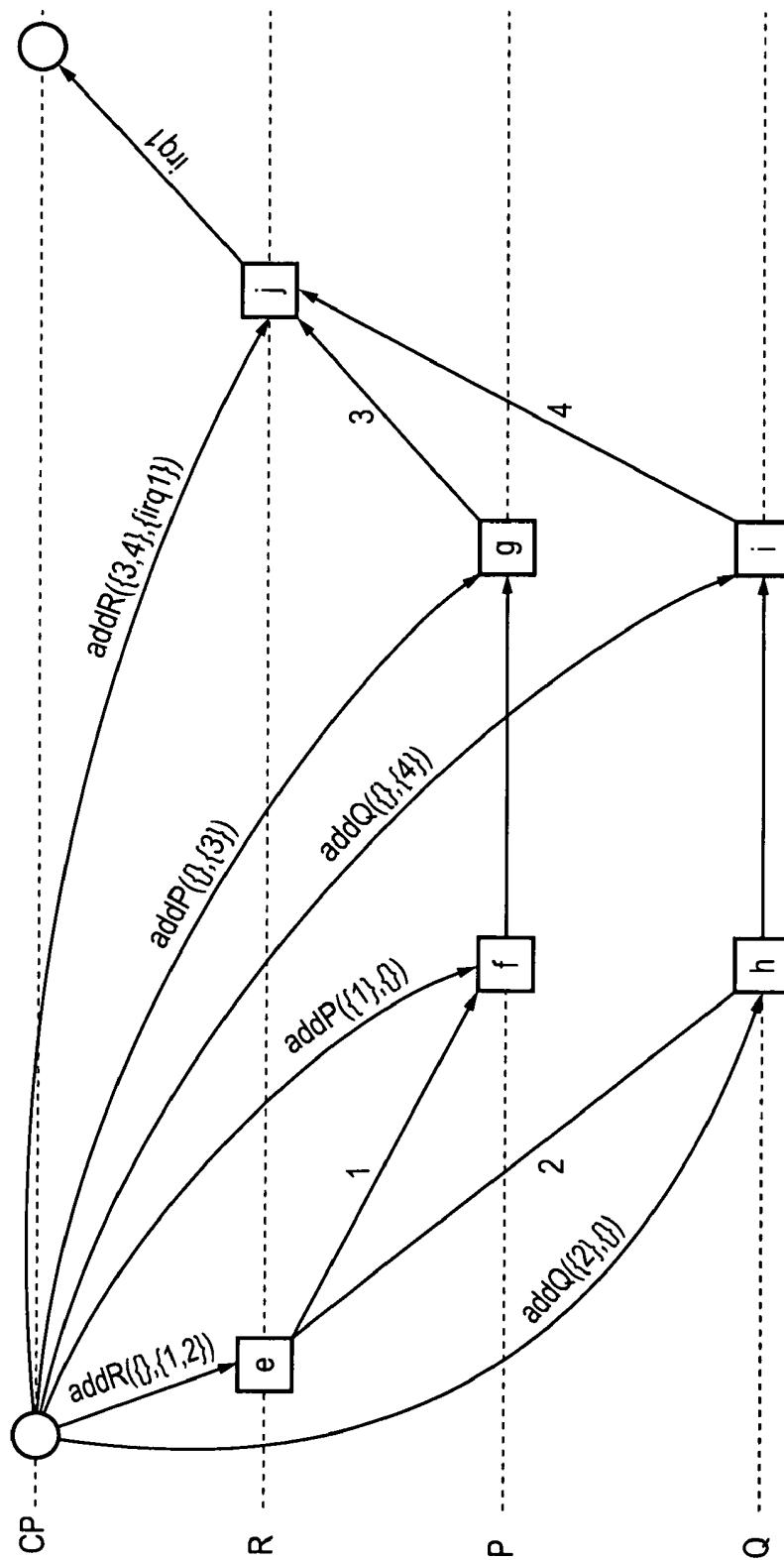


FIG. 7

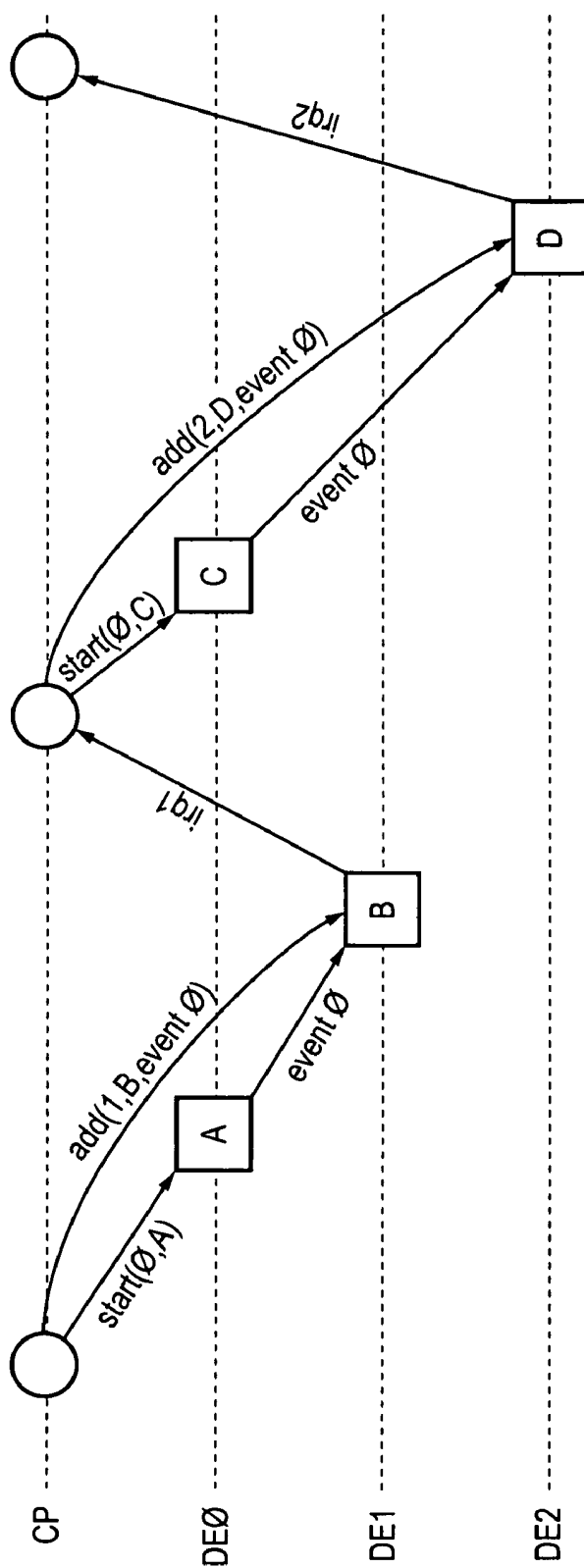


FIG. 8

U.S. Patent

Jan. 22, 2013

Sheet 9 of 11

US 8,359,588 B2

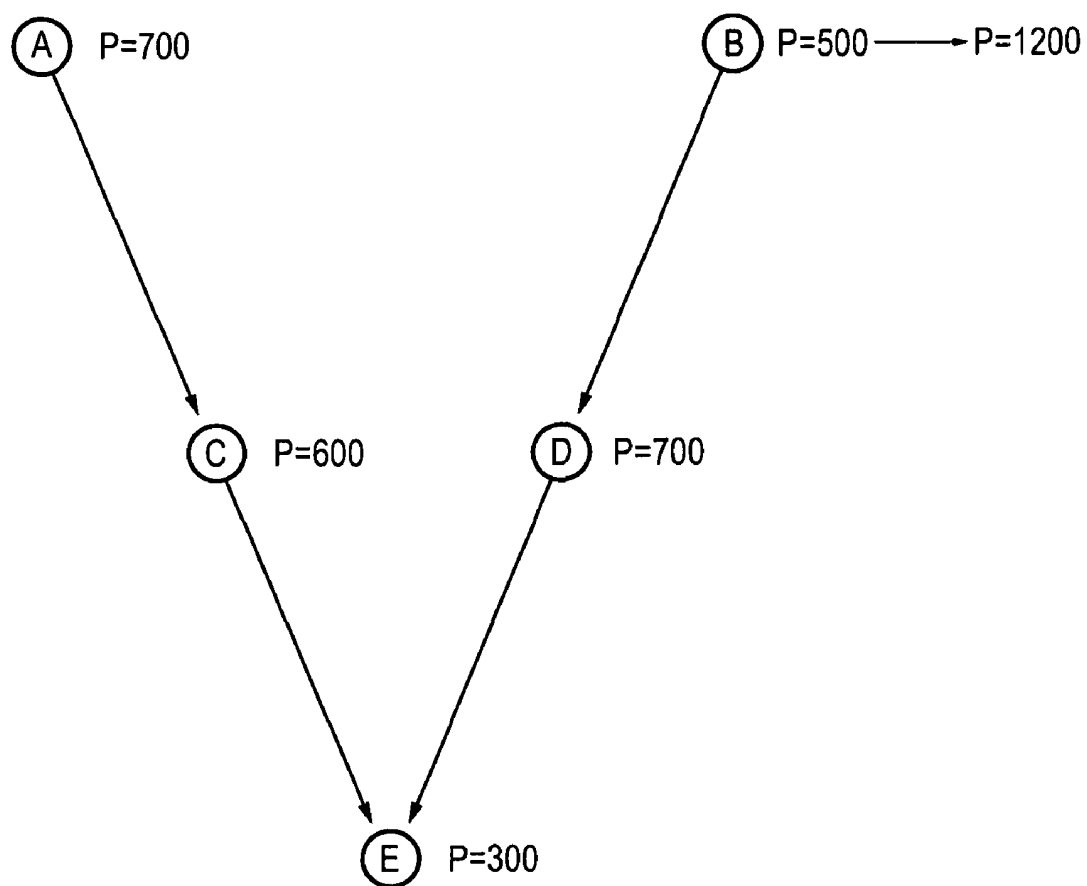


FIG. 9

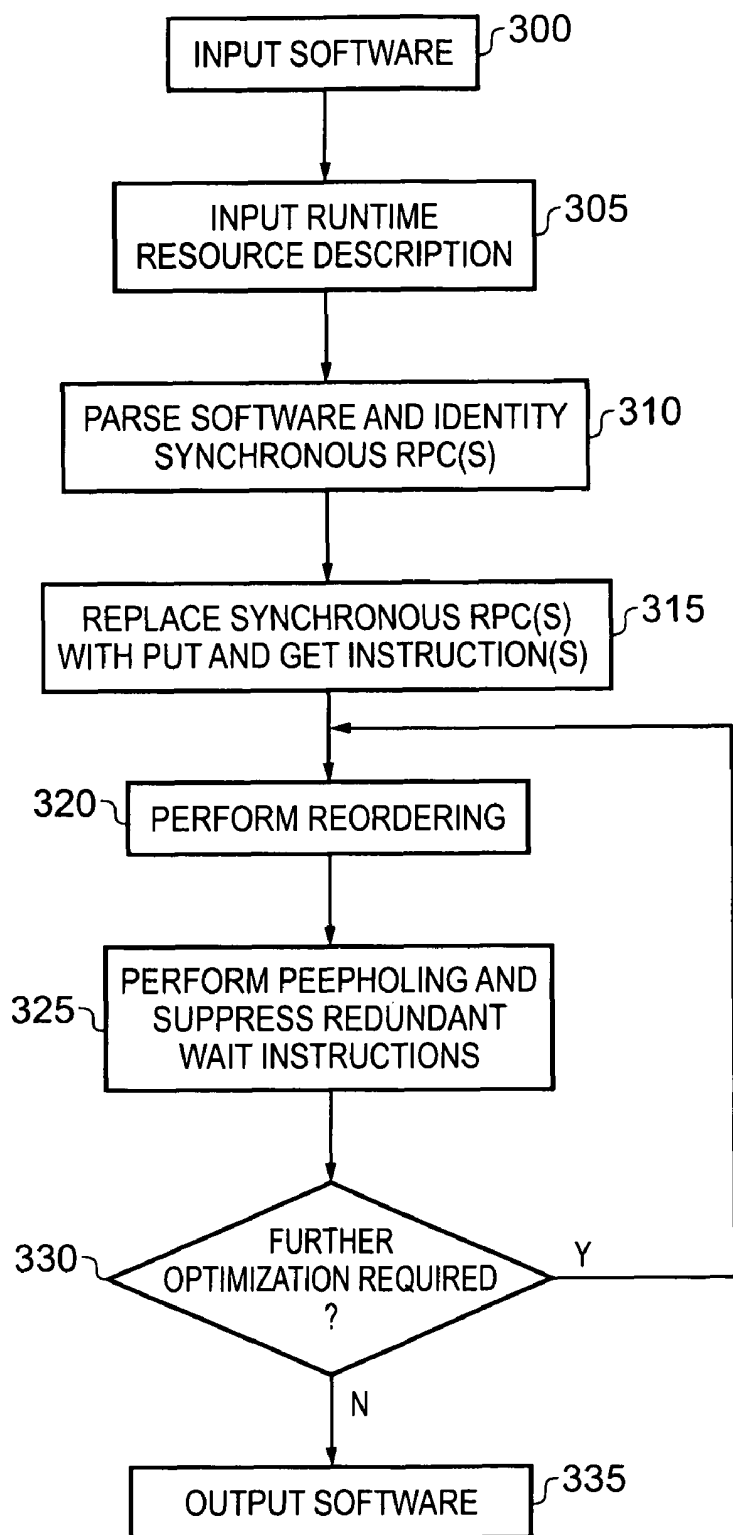


FIG. 10

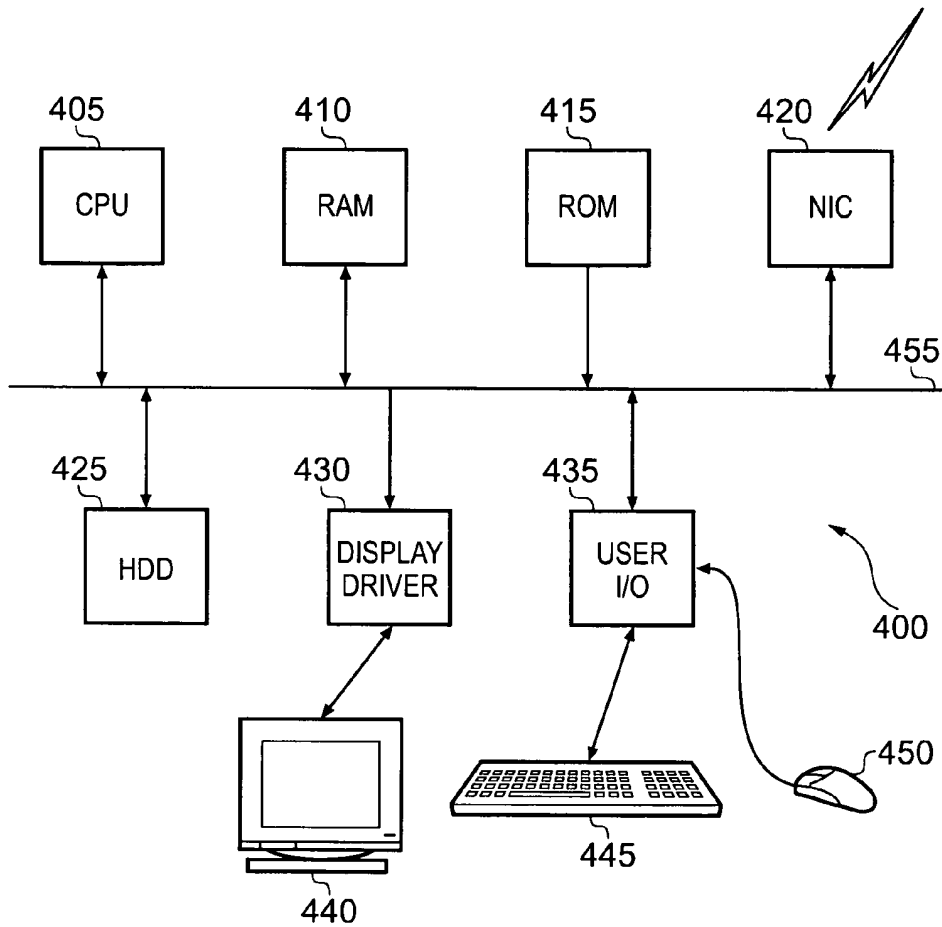


FIG. 11

US 8,359,588 B2

1

REDUCING INTER-TASK LATENCY IN A MULTIPROCESSOR SYSTEM

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to reducing inter-task latency in a multi-processor system. More particularly, this invention relates to reducing inter-task latency in a multiprocessor system on which software is executed which includes at least one synchronous remote procedure call.

2. Description of the Prior Art

Remote procedures calls (RPCs) are a known technique for programming multiprocessor systems. An RPC typically allows a program executing on one processor to cause a task to be executed by another processor in the multiprocessor system. In U.S. patent application Ser. No. 11/976,315 the concept of using RPCs to cause the execution of tasks on accelerators (such as DMA engines, data engines etc.) in the multiprocessor system is discussed.

RPCs may be categorised as either synchronous RPCs or asynchronous RPCs. From a programming point of view a synchronous RPC is the simpler of the two and operates much like a function call, except that the function is performed remotely on another processor or engine as is illustrated in FIG. 1A. In this example, the control processor (CP) is performing a sequence of operations in accordance with the instructions it is executing and at point 10 begins execution of an instruction to execute a synchronous RPC, which comprises triggering a remote processor (RP) to perform function A. CP sets up the required inputs to function A, for example by placing them in a memory space that RP can access, and then triggers RP to begin execution of function A by sending an appropriate signal (illustrated here as "do A"). Hence at 12, RP begins execution of function A. Whilst RP is executing function A CP waits until it receives a completion signal from RP. RP completes function A at 14 and sends the completion signal ("done A") to CP. At 16, CP wakes up in response to the completion signal from RP, reads the outputs of the function A (which RP has placed in a memory space accessible to CP) and continues execution of its sequence of instructions.

The feature of a synchronous RPC which makes it "synchronous" is the fact that the control processor waits for the remote processor before continuing execution (i.e. CP waits between 10 and 16 in FIG. 1). This synchronisation between the controller and remote processors makes synchronous RPCs easier to program, because there is no parallelism between the two processors. However, this ease of programming also results in some inter-task latency between tasks carried out in such a multiprocessor system.

For example, as illustrated in FIG. 1B, if CP executes a first synchronous RPC to cause RP to execute function A, and once it receives the signal "done A" from RP it executes a second synchronous RPC to cause RP to execute function B, there will be a delay between 14 (when RP sends signal "done A" to CP), 16 (when CP receives this signal), 18 (when CP executes the second synchronous RPC and sends signal "do B" to RP), and 20 (when RP begins execution of function B). Finally, once RP completes function B at 22, signals CP with "done B" and CP has received this signal at 24, CP can recommence execution of its instruction sequence.

The inter-task latency associated with synchronous RPCs can be reduced by exploiting parallelism between CP and RP, as illustrated in FIGS. 2A, 2B, 2C and 2D, and instead making use of asynchronous RPCs. In particular, if RP supports a task queue which allows multiple RPCs to be queued ready for

2

execution, this allows CP to place multiple RPCs requests into the task queue which can have various benefits as explained in the following.

FIG. 2A illustrates a situation in which CP requires two functions A and B to be executed by RP (as in FIG. 1B). RP in this example has a task queue which can accept a pending task to be executed whilst a current task is already executing. Hence, having initiated function A at 30, at 32 CP is able to signal RP to execute function B, this pending task being placed in RP's task queue, such that at 34 (when function A completes) RP can both signal CP that function A has completed ("done A") and immediately initiate execution of function B. When function B completes at 36, RP signals CP that B has completed ("done B") and on receipt of this signal at 38 CP continues execution on a sequence of instructions. Hence it can be seen that the use of asynchronous RPCs can substantially reduce the inter-task latency when a sequence of RPCs are executed on a single remote processor.

If CP wishes to perform a sequence of N synchronous RPCs, it must wait for each task to complete and therefore the RP must signal N times and CP must wait N times. However, as illustrated in FIG. 2B, when using asynchronous RPCs with a task queue with a capacity of N (i.e. the task queue is capable of holding N RPCs requests) CP need only wait for the last task to complete. As illustrated in FIG. 2B, at 40 CP signals to RP that N RPCs should be added to its task queue and at 42 RP begins execution of this series of RPCs. On completion of RPC_N at 44, RP signals that the last RPC is complete to CP, which at 46 continues execution of its sequence of instructions. The cost associating with signalling and waiting can be significant, so using asynchronous RPCs can reduce the task invocation overhead by up to N times.

The multiprocessor system can consist of more than one remote processor, as illustrated in FIG. 2C. In the situation where CP wishes to perform an RPC on RP_1 and then to execute another RPC on RP_2 this can be done (using synchronous or asynchronous RPCs) by causing RP_1 to signal CP when the first RPC completes, and CP then initiating the second RPC on RP_2 . However, if the two remote processors are able to signal each other, then asynchronous RPCs allow CP to initiate the first RPC on RP_1 , further indicating to RP_1 that it should signal RP_2 when it has completed its task. At the same time CP can queue up the second RPC on RP_2 , indicating that it should not start execution until the signal from RP_1 arrives. This way of executing is only possible if asynchronous RPCs are used and if a suitable signalling mechanism exists, as illustrated in FIG. 2C. At 50, CP signals RP_1 ("do A") and further signals RP_2 to begin function B when RP_2 receives "done A" from RP_1 (i.e. CP sends "do B when done A received"). At 52, when RP_1 completes A, it signals this fact to RP_2 ("done A") and at 54 RP_2 begins execution of function B. At 56, when RP_2 has completed function B, it signals this fact to CP ("done B") and at 58 CP continues execution of its sequence of instructions. Hence it can be seen that the use of asynchronous RPCs allows inter-task latency to be reduced, even if the sequence of RPCs is spread across multiple processors.

If CP wishes to perform two synchronous RPCs on different remote processors in parallel with each other, the only mechanism available is to execute two parallel threads, each of which performs a synchronous RPC. However, using asynchronous RPCs CP can (in one thread) start two RPCs and wait for both to complete. As illustrated in FIG. 2D, at 60 CP signals RP_1 to execute function A ("do A") and signals RP_2 to execute function B ("do B"). Hence at 62 RP_1 begins execution of function A and at 64 RP_2 begins execution of function B. At 66 RP_1 completes execution of function A and signals

US 8,359,588 B2

3

this fact ("done A") to CP, which receives this message at 68 but continues to wait for the completion of function B. RP₂ completes execution of function B at 70, and signals this fact ("done B") to CP, which at 72 continues execution of its sequence of instructions. Hence it can be seen that asynchron-

ous RPCs allow parallelism between processors. Nevertheless, in practice asynchronous RPCs can be difficult for the programmer to use. Various asynchronous RPC libraries are known, but they all suffer from the problem of being hard to program. Some common errors include: suppressing the signalling of task completion, but still waiting for the task to complete; not suppressing the signalling of task completion and not waiting for the task to complete; writing too many RPC requests into a task queue of finite capacity; introducing a deadlock condition where the next RPC request on each of two different remote processors cannot start until it receives a signal indicating that the other RPC request has completed; and introducing race conditions where the behaviour of the program depends on the relative speeds of tasks running on different processors.

The IBM RPC library allows sequences of RPCs to be sent as one group. This reduces the inter-task latency associated with signalling and waiting identified above, but it cannot reduce inter-task latency when the RPCs execute on multiple processors. Furthermore it does not assist the programmer in avoiding the problems described above, such as the introduction of race conditions or omitting waits.

As such the programmer is typically faced with a choice between the simplicity and reliability of programming using synchronous RPCs and the performance benefits of using asynchronous RPCs.

Some discussions of the use of RPCs in the prior art can be found in the following: "Optimizing RPC", Sandler D., COMP 520, Sep. 9, 2004; "Lightweight RPC", Bershad B., Anderson T., Lazowska E., Levy H., 1990; and "Flick: A flexible, optimizing IDL compiler", Eide E., Frei K., Ford B., Lepreau J. and Lindstrom G., *ACM SIGPLAN '97*, pages 44-56, Las Vegas, Nev., June 1997.

U.S. patent application Ser. Nos. 11/976,314 and 11/976,315 discuss the programming of multiprocessor systems. Some background information on the analysis of dependencies in the compilation of program code for such systems can be found in Chapters 9.0 to 9.2 of "Advanced Compiler Design and Implementation", S. Muchnick, Morgan Kaufmann, 1997 and in "Conversion of control dependence to data dependence", Allen J., Kennedy K., Porterfield C. and Warren J., 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (Austin, Tex., Jan. 24-26, 1983), ACM, New York, N.Y., 177-189.

It would be desirable to provide an improved technique for programming multiprocessor systems, which combined the simplicity of programming with synchronous RPCs and the performance benefits of using asynchronous RPCs.

SUMMARY OF THE INVENTION

Viewed from a first aspect, the present invention provides a method of reducing inter-task latency for software comprising a sequence of instructions including a synchronous remote procedure call to be executed on a multiprocessor system, said multiprocessor system comprising a calling processor and at least one remote engine, the method comprising the steps of: inputting said software; inputting a runtime resource description, said runtime resource description describing a runtime environment of said multiprocessor system; identifying said synchronous remote procedure call in said sequence of instructions; replacing said synchronous

4

remote procedure call in said sequence of instructions with an initiation instruction and a wait instruction to generate a substitute sequence of instructions; identifying dependencies between instructions in said substitute sequence of instructions; reordering said substitute sequence of instructions with reference to said runtime resource description and said dependencies to generate a reordered sequence of instructions; and outputting said reordered sequence of instructions.

According to the techniques of the present invention software may be written comprising a sequence of instructions which includes a synchronous remote procedure call, that software intended to be compiled for execution on a multiprocessor system, and yet the reduced inter-task latency associated with the use of an asynchronous remote procedure call may be achieved. The multiprocessor system comprises a calling processor (also known as a control processor) from which the synchronous remote procedure call is to be called and at least one remote engine to execute the procedure thus remotely called. Note that the programmer is able to write program code which includes the easier-to-program synchronous remote procedure call.

According to the techniques of the present invention, a synchronous remote procedure call is identified in the sequence of instructions and replaced with an initiating instruction and a wait instruction to generate a substitute sequence of instructions. This pair of instructions corresponds in function to the synchronous remote procedure call, the initiating instruction causing the required task (the remote procedure) to be signalled to the remote engine on which it will execute, and the wait instruction causing the calling processor to wait until completion of that remote procedure is signalled. The method then further comprises identifying dependencies in the substitute sequence of instructions and reordering the substitute sequence of instructions with reference to a runtime resource description, which describes a runtime environment of the multiprocessor system, and with reference to the identified dependencies. This reordering of the sequence of instructions is performed to reduce inter-task latency when the software is executed on the multiprocessor system. The reordered sequence of instruction is then outputted ready to be compiled for execution on the multiprocessor system.

The inventor of the present invention realised that inter-task latency when executing software on a multiprocessor system could be reduced by taking advantage of asynchronous remote procedure call techniques, whilst still allowing the programmer the simplicity and transparency of writing software with a synchronous remote procedure call. Expanding the sequence of instructions into a substitute sequence of instructions in which the synchronous remote procedure call has been split up into an initiation instruction and a wait instruction allows a greater degree of freedom in the subsequent reordering of instructions, and hence inter-task latency can be more effectively reduced. The advantages of the present invention may be realised even when there is only one synchronous remote procedure call in the software, for example, when the software also comprises a FIFO instruction following a synchronous remote procedure call. The expansion of the synchronous remote procedure call into an equivalent initiation instruction and wait instruction allows a reordering in which the FIFO instruction comes between the initiation instruction and the wait instruction. This reordering is beneficial if the FIFO instruction is usually able to complete before the wait instruction completes, because it allows the FIFO instruction to execute in parallel to the remote procedure call, thus reducing inter-task latency.

US 8,359,588 B2

5

In other embodiments said sequence of instructions includes a plurality of synchronous remote procedure calls, said identifying said remote procedure call step comprises identifying said plurality of synchronous remote procedure calls, and said replacing step comprises replacing each instruction of at least a subset of said plurality of synchronous remote procedure calls with a corresponding initiation instruction and wait instruction. To take a simple example, the software may include two synchronous remote procedures calls, and by replacing each synchronous remote procedure call with an equivalent initiation instruction and wait instruction, a reordering of the instruction sequence becomes possible which allows inter-task latency to be reduced, for example by scheduling the first and second initiation instructions to be sequential, followed by the wait instructions in sequence.

In some embodiments, the method further comprises a step of suppressing at least one wait instruction determined to be redundant following said reordering. In the above mentioned example, where the software includes at least two synchronous remote procedure calls and the initiation and wait instructions have been reordered such that the pair of wait instructions follows the pair of initiation instructions, the first of the wait instructions may be determined to be redundant. For example each wait instruction may be configured to comprise an argument indicating whether a "complete" signal is required when the remote procedure call completes. When two synchronous remote procedure calls are arranged in sequence, according to the techniques of the present invention, the first initiation instruction (corresponding to the first synchronous remote procedure call) may be chosen such that it is indicated that no "complete" signal is required when that remote procedure call has completed. Hence, the wait instruction corresponding to the first remote procedure call is no longer required, can be determined to be redundant and may be suppressed. This suppression could comprise removing that wait instruction from the sequence of instructions or could simply comprise marking it such that it will not be executed.

In some embodiments, the method further comprises a step of identifying at least two wait instructions, and reordering said at least two wait instructions to be adjacent to each other in said substitute sequence. In the situation where there are at least two synchronous remote procedure calls in the sequence of instructions, and hence after the replacing step there are then at least two wait instructions in the substitute sequence, subsequent handling of those wait instructions is simplified by reordering of the sequence of instructions such that the at least two wait instructions are adjacent to each other in the substitute sequence. This handling could take a number of forms, but according to one embodiment said reordering said at least two wait instructions is followed by a peepholing step in which adjacent instructions are examined. Peepholing represents an optimisation step which is advantageously straightforward to implement, comprising the comparison of adjacent pairs of instructions in the sequence of instructions.

It will be recognised that the multiprocessor system could be configured in a variety of ways, but in some embodiments the multiprocessor system comprises at least two remote engines and the method further comprises introducing signalling between said at least two remote engines. As discussed in the introduction, when a multiprocessor system comprises at least two remote engines, one source of inter-task latency may be the delay introduced by a first remote engine signalling to the calling processor that the first remote procedure call has completed, the calling processor handling this signal and instructing the second remote engine to begin its own remote

6

procedure call. By introducing signalling between the at least two remote engines, it can be provided that completion of the first remote procedure call on the first remote engine may be directly signalled to the second remote engine, in order to initiate processing of the second remote procedure call. The inter-task latency is thus reduced, by avoiding the "round trip" back to the calling processor.

The signalling that may be introduced between at least two remote engines may take a variety of forms. In one embodiment, the signalling comprises task triggering signals. Thus, a first remote engine may send a signal to second remote engine to trigger a task on that second remote engine. In one embodiment, the signalling further comprises data provision. This data may result from a recently completed task on a first remote engine, and/or may comprise data required by a second remote engine for the processing task it has been given. In one embodiment, the signalling comprises an idle status notification, this idle status notification providing the remote engine receiving this signal with the information that the remote engine which sent the signal is currently idle. This idle status may result from the completion of a task, and in one embodiment the signalling comprises a task completion signal. Such a task completion signal may take a number of forms, but in one embodiment the task completion signal is specified by the calling processor. The ability for the calling processor to specify the task completion signal may be useful in a number of ways, for example it may be necessary to ensure the correct sequence of operations on the remote engines to distinguish between different situations in which one remote engine may signal to another. By specifying the task completion signal, the calling processor can ensure that the remote engine receiving that task completion signal can distinguish between different tasks or different time points at which a given task is completed. In a related manner, in one embodiment at least one remote engine is configured by said calling processor to begin a predetermined task on receipt of said task completion signal. In this way, the calling processor may also control which task a remote engine will be begin when it receives a particular task completion signal from another remote engine. In other embodiments, the task completion signal indicates completion of a predetermined number of tasks. For example a remote engine may be required to perform a number of tasks, and only on completion of all of those tasks to signal this fact to another remote engine. In embodiments of the present invention, the runtime resource description comprises a description of available signals between said at least two remote engines. This then facilitates the optimisation steps that may be performed to make use of inter-engine signalling.

When certain sequences of instructions are arranged for execution on a multiprocessor system, it may be the case that reordering that sequence of instructions may result in task interdependencies which could cause data hazards when at least a degree of parallelism is introduced. Accordingly, in one embodiment the reordering step further comprises identifying task interdependencies which could cause at least one data hazard when said software is executed on said multiprocessor system, and performing said reordering to avoid said at least one data hazard. Hence, the opportunity for data hazards to occur may be avoided, if task interdependencies are recognised and respected, when reordering instructions in the reducing inter task latency step.

In one embodiment, said at least one remote engine comprises a pending task queue configured to hold at least one indication of a pending task for subsequent execution by that remote engine and said initiation instruction is configured to place an indication of a pending task in said pending task

US 8,359,588 B2

7

queue. The provision of the pending task queue for a remote engine enables that remote engine to accept more than one task to perform, tasks still to be performed being queued in the pending task queue. The provision of a pending task queue for a remote engine provides a significant degree of flexibility for the multi-processor system. In particular, the ability to have the next task which a remote engine must perform already queued up whilst a current task is being performed means that delays between the execution of one task and next may be kept to a minimum, since such a remote engine does not need to signal completion of a first task to the calling processor and wait for allocation of a second task before it can begin processing that second task. It will be recognised that the pending task queue can take a variety of forms, but in one embodiment the pending task queue is a FIFO queue. In some embodiments, the at least one remote engine comprises a completed task queue configured to hold at least one indication of a task completed by that remote engine. According to this arrangement, when the remote engine completes a particular task it adds an indication of that completed task into its completed task queue. The ability to queue up indications of completed tasks in this manner means that the consequences of completing that task (e.g. signalling this completion to the calling processor, passing resulting data to another remote engine, etc) need not be actioned immediately and this adds a further degree of flexibility and configurability to the multi-processor system. It will be recognised that this completed task queue can take a number of forms, but in one embodiment it is a FIFO queue.

According to one embodiment the runtime resource description comprises a depth of said pending task queue, and said reordering is performed such that when said software is executed on said multiprocessor system a number of pending tasks queued in said pending task queue will not exceed said depth. Incorporating the depth of the pending task queue into the runtime resource description enables the optimisations performed, in particular the reordering step, to be performed taking this step into account such that the number of pending tasks that will be added to this pending task queue when the software is executed on the multiprocessor system will not exceed the depth of the pending task queue.

According to one embodiment the instructions each have an associated priority, and said reordering comprises positioning instructions having a higher priority before instructions having lower priorities to an extent allowed by inter-instruction dependencies. Amongst the many instructions to be executed on a multiprocessor system, it may be the case that certain instructions are more important than others, in the sense that they should be executed with minimal delay, whilst other instructions may be less time-critical. The importance of instructions can be parameterised by a priority associated with each instruction and in this embodiment when reordering instructions, instructions having a higher priority are positioned as early in the sequence of instructions as is possible, whilst still respecting inter-instruction dependencies. The priorities associated with the instructions may be fixed, but according to one embodiment the method further comprises a priority adjustment step, when at least one priority is adjusted. Incorporating the ability to adjust the priorities of instructions enables a further degree of flexibility, in which the priority of a given instruction may be adjusted, if it is established that this adjustment would improve the inter-task latency when the software is executed on the multiprocessor system, without altering the semantic meaning of the software. This priority adjustment may be performed for a number of reasons, but according to one embodiment the at least one priority is associated with a lower priority instruction

8

upon which a higher priority instruction depends, and said priority adjustment step comprises raising said at least one priority associated with said lower priority instruction. It has been recognised that higher priority instructions may be held up by the fact that they depend on the outcome of lower priority instructions, and according to this embodiment such lower priority instructions are identified and may have their associated priority raised, in order to promote the execution of the higher priority instruction dependent thereon.

When at least one remote engine in the multiprocessor system comprises a pending task queue, according to one embodiment the initiation instruction is a pending task queue entry acquisition instruction, and said replacing step further comprises introducing a pending task queue data release instruction. According to this embodiment, the initiation instruction is further split into a pending task queue entry acquisition instruction and a pending task queue data release instruction. Sub-dividing the initiation instruction in this manner allows a further degree of flexibility in the scheduling of instructions for execution and hence allows the inter-task latency to be further reduced. In the situation where the at least one remote engine in the multiprocessor system comprises a completed task queue, according to one embodiment the wait instruction is a completed task queue data acquisition instruction, and said replacing step further comprises introducing a completed task queue entry release instruction. Hence the wait instruction is further sub-divided into a task queue data acquisition instruction and a completed task queue entry release instruction. Sub-dividing the wait instruction in this manner brings a further degree of flexibility to the reordering and scheduling of instructions for execution on the multiprocessor and thus may allow inter-task latency to be further reduced.

It may be the case that the at least one remote engine comprises only a single pending task queue, but in some embodiments the at least one remote engine comprises sequential pending task queues. Similarly it may be the case that the at least one remote engine comprises only a single completed task queue, but in embodiments the at least one remote engine comprises sequential completed task queues. These sequential task queues bring a further degree of flexibility to the configuration of the system, allowing the remote engine to more efficiently handle pending and completed tasks. For example, if a remote engine has an associated pending task queue, but a further task queue is interposed between the calling processor and the remote engine, then the calling processor will still be able to release a task to be queued for the remote engine, by adding it to that further task queue, even if the task queue of the remote engine is currently full to capacity.

The runtime resource description with reference to which the reordering of the sequence of instructions is performed may take a variety of forms. In one embodiment the runtime resource description comprises a description of hardware available in the multiprocessor system, but in another embodiment the runtime resource description comprises a software controlled view of said hardware available in said multiprocessor system. Hence, the runtime resource description may be a direct representation of the hardware available in the multi processor system, or may be a partial view thereof under the control of software. For example it may be the case that particular hardware components, although physically existing in the system, may be hidden under software control. As another example, the hardware may only provide a two-entry queue, but software may implement a four-entry queue.

In one embodiment, the method further comprises reducing a number of interrupts that will be received by said calling

processor when said software is executed on said multiprocessor system. As discussed above, remote procedure calls may have inherent delays due to the requirement to signal completion of the remote procedure call and waiting for that signal to arrive. In low level remote procedure call mechanism implementations, each signal may generate an interrupt, and the number of interrupts received by the calling processor may be a significant burden. Accordingly, reducing the number of interrupts that will be received by the caller processor can alleviate this burden and allow the calling processor to perform its operations more efficiently.

It may be the case that at least one synchronous remote procedure call is arranged only to be executed if a particular runtime condition is true. Accordingly, in one embodiment said identifying said at least one synchronous remote procedure call in said sequence of instructions further comprises determining if said at least one synchronous remote procedure call is only executed when a runtime condition is true; and said replacing each said at least one synchronous remote procedure call in said sequence of instructions with an initiation instruction further comprises introducing said runtime condition as an argument of said initiation instruction. Transforming the initiation instruction in this manner enables such synchronous remote procedure calls to be handled by the techniques of the present invention, despite their dependency on a given runtime condition.

It will be recognised that the multiprocessor system could be configured in a variety of ways. In one embodiment the at least one remote engine comprises a processor, whilst in other embodiments the at least one remote engine comprises a hardware engine. Various permutations are of course possible, for example multiple individual processors, a mixture of processors and hardware engines, dedicated hardware engines such as DMAs or analogue-to-digital converters (ADCs) and so on.

According to one embodiment, wherein said sequence of instructions includes a plurality of synchronous remote procedure calls and said identifying said remote procedure call step comprises identifying said plurality of synchronous remote procedure calls, said method further comprising a merging step after said identifying said remote procedure call step in which: at least two sequential remote procedure calls to single remote engine are merged into one remote procedure call to said single remote engine. Hence, when at least two remote procedure calls are identified, if these are arranged to be executed sequentially on a single remote processor, they can be replaced by one remote procedure call to that remote processor, in which the two or more tasks are merged into a unit which will be handled as a single remote procedure call, which will cause those two or more tasks to be sequentially executed on that remote processor.

Viewed from a second aspect, the present invention provides a computer-readable medium storing a program which when executed on a computer causes the computer to carry out a method of reducing inter-task latency for software comprising a sequence of instructions including a synchronous remote procedure call to be executed on a multiprocessor system, said multiprocessor system comprising a calling processor and at least one remote engine, the method comprising the steps of: inputting said software; inputting a runtime resource description, said runtime resource description describing a runtime environment of said multiprocessor system; identifying said synchronous remote procedure call in said sequence of instructions; replacing said synchronous remote procedure call in said sequence of instructions with an initiation instruction and a wait instruction to generate a substitute sequence of instructions; identifying dependencies

between instructions in said substitute sequence of instructions; reordering said substitute sequence of instructions with reference to said runtime resource description and said dependencies to generate a reordered sequence of instructions; and outputting said reordered sequence of instructions.

Viewed from a third aspect, the present invention provides a data processing apparatus for transforming reducing inter-task latency for software comprising a sequence of instructions including a synchronous remote procedure call to be executed on a multiprocessor system, said multiprocessor system comprising a calling processor and at least one remote engine, the apparatus comprising: a software input configured to input said software; a runtime resource description input configured to input a runtime resource description, said runtime resource description describing a runtime environment of said multiprocessor system; an identification unit configured to identify said synchronous remote procedure call in said sequence of instructions; a replacement unit configured to replace said synchronous remote procedure call in said sequence of instructions with an initiation instruction and a wait instruction to generate a substitute sequence of instructions; an identification unit configured to identify dependencies between instructions in said substitute sequence of instructions; a reordering unit configured to reorder said substitute sequence of instructions with reference to said runtime resource description and said dependencies to generate a reordered sequence of instructions; and an output configured to output said reordered sequence of instructions.

Viewed from a fourth aspect, the present invention provides a data processing apparatus for reducing inter-task latency for software comprising a sequence of instructions including a synchronous remote procedure call to be executed on a multiprocessor system, said multiprocessor system comprising a calling processor and at least one remote engine, the apparatus comprising: software input means for inputting said software; runtime resource description input means for inputting a runtime resource description, said runtime resource description describing a runtime environment of said multiprocessor system; identifying means for identifying said synchronous remote procedure call in said sequence of instructions; replacing means for replacing said synchronous remote procedure call in said sequence of instructions with an initiation instruction and a wait instruction; identification means for identifying dependencies between instructions in said substitute sequence of instructions; reordering means for reordering said substitute sequence of instructions with reference to said runtime resource description and said dependencies to generate a reordered sequence of instructions; and output means for outputting said reordered sequence of instructions.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be described further, by way of example only, with reference to embodiments thereof as illustrated in the accompanying drawings, in which:

FIGS. 1A and 1B schematically illustrates the use of synchronous remote procedure calls according to the prior art;

FIGS. 2A, 2B, 2C and 2D schematically illustrate the use of asynchronous remote procedure calls according to the prior art;

FIG. 3 schematically illustrates the compilation and linking of a source SoC-C program for execution on a multiprocessor system according to one embodiment;

FIG. 4A schematically illustrates a calling processor and remote engine communicating via pending and completed task queues;

US 8,359,588 B2

11

FIG. 4B schematically illustrates a calling processing and remote engine communicating via sequential pending task queues and sequential completed task queues;

FIGS. 5A, 5B and 5C schematically illustrate different communication patterns between a calling processor and a remote engine executing two tasks;

FIGS. 6A and 6B schematically illustrate different communication patterns between a calling processor and two remote engines each having a task to execute;

FIG. 7 schematically illustrates communication in a multiprocessor system;

FIG. 8 schematically illustrates communication in a multiprocessor system;

FIG. 9 schematically illustrates five interdependent instructions and their relative priorities;

FIG. 10 schematically illustrates a series of steps taken in one embodiment; and

FIG. 11 schematically illustrates a general purpose computer on which some embodiments may be implemented.

DESCRIPTION OF EMBODIMENTS

FIG. 3 schematically illustrates the process by which a source SoC-C (System-on-Chip-C) program is compiled and linked for execution on a multiprocessor system. The SoC-C program 100 provides one input (in .socc format) to SoC-C compiler 110, whilst the other input is provided by the hardware description 105, which provides the compiler with information about the runtime configuration of the multiprocessor system on which the SoC-C program will ultimately be executed. This hardware description may be a true description of the hardware composing the system, or may be a software controlled view of the configuration of the system. This software controlled view can, for example, hide certain components of the system if they should not be used, or can alternatively present the compiler with information about components which, whilst appearing to be hardware components of the system for the purposes of compilation and execution, are in fact instantiated in software. The SoC-C compiler 110 transforms the SoC-C program 100 into a format (in this example .c format) suitable to input into C compiler 115. C compiler 115 compiles this transformed source code into an object file (in .o format) which is passed to linker 120. Linker 120 takes this object file and (with reference to one or more libraries as appropriate) generates an executable program (in this case in .axf format), which may be executed on the multiprocessor system.

The operation of the SoC-C compiler 110 may be conceptually broken down into the illustrated steps of parsing 125, high level optimisation 130, low level optimisation 135 and code generation 140. As illustrated in FIG. 3, the parse step 125 represents the initial step at which the input 100 and 105 are analysed to determine their structure with respect to predetermined rules of the languages in which they are written. Thus analysed, at step 130 various high level optimisations of the program may be carried out, for example the transformation of a sequential program into a program with multiple parallel program sections containing RPC calls which communicate via FIFO queues (as is described in U.S. patent application Ser. No. 11/976,315) or the transformation of a program for execution on a multiprocessor system where each processor may have a private memory (as is described in U.S. patent application Ser. No. 11/976,314). Thereafter at step 135 the SoC-C compiler 110 carries out various low level optimisations (described in more detail below) and finally at step 140 the revised program code is generated (in .c format) for passing to C compiler 115.

12

The low level optimisations at step 135 may, according to one embodiment, comprise the identification of remote procedure calls (RPCs) in the SoC-C program 100 which have been written by the programmer as synchronous RPCs and transforming them into asynchronous RPCs in a manner which will improve inter-task latency. Conceptually, the low level optimisation 135 may be broken down into stages. Firstly, at step 145 the synchronous RPCs are detected. Then at step 150 these synchronous RPCs are expanded (described in more detail below). Finally, at step 155 scheduling and optimisation of the RPCs is carried out (also described in more detail below).

Before the process of RPC detection, expansion and scheduling/optimisation is described in more detail, reference is first made to FIGS. 4A and 4B which schematically illustrate two configurations of task queues between a calling processor CP and a remote processor RP. CP and RP may also be referred to as a control processor and a remote engine, respectively, depending on their relative configurations. For example both CP and RP may be largely equivalent processors, or RP may be a hardware engine configured to perform certain tasks on behalf of CP. FIG. 4A schematically illustrates a configuration having a single pending task queue 200 and a single completed task queue 205. In the configuration illustrated in FIG. 4A task queues 200 and 205 may be considered to be associated with the remote processor RP, i.e. calling processor CP places tasks for RP to execute into RP's pending task queue 200 and RP places indications of its completed tasks into its completed task queue 205. However, the configuration may also be considered to be more symmetrical, for example where CP and RP are equivalent processing units in the multiprocessor system. Note that both task queues 200 and 205 have an illustrated depth, which indicates the maximum number of tasks (or indications of tasks) which can be placed into each task queue. In some embodiments the optimisation steps carried out, for example the reordering of instructions, is performed with reference to the available task queue depths, such that the number of tasks that will end up being queued in a given task queue will not exceed the available depth of that task queue.

FIG. 4B schematically illustrates a configuration of sequential task queues. Task queue 200 in FIG. 4A has been replaced by task queues 210 and 215, whilst task queue 205 in FIG. 4A has been replaced by task queues 220 and 225. Note in particular that in the example configuration illustrated in FIG. 4B the task queues 210 and 220 are embodied in software, whilst (in this embodiment) the task queues 215 and 225 are embodied in hardware. Various permutations of hardware-only, software-only or a mix of hardware-software are possible in other embodiments are possible. The task queues of FIG. 4B can in principle be considered to be shared between CP and RP, or associated with only one of the two. For example, in the embodiment illustrated in FIG. 4B, the software queues 210 and 220 are associated with CP, whilst the hardware queues 215 and 225 are associated with RP. In other words, CP instantiates task queues 210 and 220 in software running on CP, whilst task queues 215 and 225 form part of the hardware of RP. According to this arrangement, the process of CP passing a task to RP for execution is formed of two stages. This means that a task descriptor (to be placed in a task queue) can be constructed some time before the task should actually be initiated, and the task is only initiated some time later.

Firstly, CP constructs a task in software pending task queue 210 ("RPC_acquireRoom aR"). This setting up of the task may take a relatively long time (e.g. 100 cycles). At this stage the task is not ready to be run, for example it may be depen-

dent on the completion of another task that is currently running on another processor. Some time later it becomes safe (from a data hazard point of view) to run the task (for example, the other task on which it depends has completed). Then, at this point the (indication of) the task is transferred from software pending task queue **210** to hardware pending task queue **215** ("RPC_releaseData rD"), indicating to RP that the task is ready to be run. Similarly, the process by which RP signals a completed task to CP may be broken down into two stages. Firstly the (indication of) the completed task in hardware completed task queue **225** is passed to software completed task queue **220** ("RPC_acquireData aD"). Secondly, the (indication of) the completed task is transferred from the software completed task queue **220** to CP ("RPC_releaseRoom rR").

A further advantage of this illustrated example of FIG. 4B is that the four queues can be efficiently implemented as a single circular buffer with four pointers into the buffer indicating the boundary between the four queues. In such an arrangement, transferring a task from one queue to another can be performed by merely incrementing a single pointer which might take 10 cycles.

Hence, splitting the pending task queue **200** in FIG. 4A into two separate queues **210** and **215** in FIG. 4B can be seen as another mechanism for reducing inter-task latency. With a single pending task queue (such as **200** in FIG. 4A), the task descriptor cannot be constructed until the task is ready to run. With two separate queues (such as **210** and **215** in FIG. 4B), it is possible to construct a task descriptor in advance and only transfer it into the pending queue **215** when any tasks it depends on have complete. This takes the **100** cycles taken to construct the task out of the critical path.

The process of RPC detection **145**, RPC expansion **150** and RPC scheduling/optimization **155** schematically illustrated in FIG. 3 is now described in more detail.

An RPC can be split into 2 or more phases (the number depending on how detailed a level of scheduling optimization one wishes to be able to perform). The present discussion starts by considering an RPC split into two phases: initiating an RPC and waiting for the RPC to complete. Consider the following a sequence of code:

```
RPC(P,f);
RPC(P,g);
```

which indicates remote invocation of two functions "f" and "g" in order, both executing on a processor "P". Each RPC can be split into two phases: "RPC_put" (initiation) which puts an RPC request into P's task queue and "RPC_get" (waiting) which waits for a response from "P". Rewriting the example code sequence using these operations gives:

```
RPC_put(P,f);
RPC_get(P,f);
RPC_put(P,g);
RPC_get(P,g);
```

Importantly it should be recognized that the following dependencies then exist between the four operations (where A→B should be read as "B depends on A"):

```
RPC_put(P,f); → RPC_get(P,f);
RPC_put(P,g); → RPC_get(P,g);
RPC_put(P,f); → RPC_put(P,g);
```

Since there is no dependency between the 2nd and 3rd operations, it is legal (i.e. it does not change the semantic meaning of the sequence of instructions) to reorder the operations. The field of instruction scheduling has developed many algorithms for optimizing sequences of dependent operations that typically work as follows: identify dependencies between operations; prioritize operations; generate a new sequence of operations by reordering operations such that high priority operations occur before lower-priority operations on which they do not depend. If RPC_put operations are given higher priority than RPC_get operations, such instruction scheduling yields the following sequence:

```
RPC_put(P,f);
RPC_put(P,g);
RPC_get(P,f);
RPC_get(P,g);
```

This reordering of the operations required to carry out the invocation of RPCs to perform functions f and g on processor P reduces the inter-task latency between functions f and g, in other words bringing the benefits of asynchronous RPC calls that were identified above.

When reordering operations, it is important to ensure that the data dependencies of the split RPCs reflect the data dependencies of the original RPCs. In one embodiment, this is handled as follows.

Suppose that RPC(P,f) reads a global variable 'r', writes a global variable 'w' and modifies a global variable 'm'. After splitting, both the RPC_put(P,f) and RPC_get(P,f) operations should be modelled as reading 'r' and modifying 'm', but they differ in their treatment of 'w'. RPC_put(P,f) is modelled as writing to 'w' while 'RPC_get(P,f)' is modelled as modifying 'w'. Treating RPC_get(P,f) as though it modifies 'w' ensures that 'w' is live throughout the period from when RPC_put(P,f) is initiated until when the RPC_get(P,f) completes.

It is also common for RPC operations to operate on data which is passed by reference. For example, we can write RPC(f,&x) to pass some variable 'x' to the RPC function 'f' (using the standard C notation of '&x' for variable 'x' being passed by reference). When optimizing program code involving RPCs, it is important to know whether an RPC function reads, writes or modifies any data buffers passed by reference. This might be accomplished by analyzing the implementation of the RPC function or it might be indicated by a programmer annotation.

When an RPC function RPC(P,f,&x) is split, it is important that both the RPC_put and RPC_get parts of the operation record that they read, write or modify 'x'. This is done by specifying '&x' in both the RPC_put and RPC_get calls. That is to say, the code:

```
RPC(P,f,&x);
```

US 8,359,588 B2

15

is transformed into two calls:

```
RPC_put(P,f,&x);
RPC_get(P,f,&x);
```

If the original RPC function reads or modifies *x*, then the `RPC_put` and `RPC_get` operations are treated as though they read or modify ‘*x*’ respectively. On the other hand, if the original RPC function writes to ‘*x*’, then the `RPC_put` is treated as though it writes ‘*x*’ and the `RPC_get` operation is treated as though it modifies ‘*x*’.

With these changes, it is possible then to track the dependencies in the same way that they are normally tracked in compilers (see, for example, Chapters 9.0-9.2 of “Advanced Compiler Design and Implementation”, by S. Muchnick, Morgan Kaufmann, 1997). The reduction in inter-task latency resulting from replacing synchronous RPC calls, and subsequently reordering the substituted instructions, may even be realised when a particular sequence of code only comprises a single synchronous RPC. Consider the following code sequence:

```
RPC(P,f,&x);
fifo_acquireRoom(&q,&py);
```

which indicates remote invocation of function “*P*”, executing on a processor “*P*” and the acquisition of an entry in a FIFO data queue ‘*q*’ by placing a pointer to the entry in variable ‘*py*’ (both ‘*q*’ and ‘*py*’ being passed by reference) (e.g. see the step aR acquiring an entry in task queue 210 in FIG. 4B). Performing the above-described expansion of the RPC call into its initiation and waiting phases yields:

```
RPC_put(P,f,&x);
RPC_get(P,f,&x);
fifo_acquireRoom(&q,&py);
```

Then performing the same reordering as described above, scheduling instructions according to priority whilst respecting the dependencies between the instructions, gives:

```
RPC_put(P,f,&x);
fifo_acquireRoom(&q,&py);
RPC_get(P,f,&x);
```

This reordering swaps two blocking operations (`RPC_get` and `fifo_acquireRoom`). This particular reordering is beneficial if the `fifo_acquireRoom` operation is usually able to complete before `RPC_get` completes, because it allows the `fifo_acquireRoom` call to execute in parallel to the RPC.

As noted above, synchronous RPC invocation has an overhead due to signalling completion of the RPC and waiting for that signal to arrive. Some asynchronous RPC mechanisms provide ways to suppress waiting for an RPC to complete. This is especially beneficial in low level RPC mechanisms where each signal generates an interrupt.

One way to do this is for the `RPC_put` operation to take an additional argument indicating whether a signal is desired. Whether the `RPC_put` operation supports signal suppression is indicated by the runtime resource description. When per-

16

forming signal suppression, it is necessary for the RPC splitting step to keep track of corresponding operations, i.e. `RPC_put` and `RPC_get` operations that result from the splitting of a single original RPC operation. For example, an additional argument with values “`NO_SIGNAL`” or “`WITH_SIGNAL`” could be used. Returning to the example sequence of code discussed above, comprising `RPC(P,f)` and `RPC(P,g)`, and introducing the argument “`WITH_SIGNAL`” to each `RPC_put` instruction, results in the following (reordered) code sequence:

```
RPC_put(P,WITH_SIGNAL,f);
RPC_put(P,WITH_SIGNAL,g);
RPC_get(P,f);
RPC_get(P,g);
```

Subsequently, a peephole optimization procedure may be carried out, which identifies sequences of adjacent “`RPC_get`” operations. Whenever such a sequence is found, all `RPC_get` operations except the last can be removed, and the `RPC_put` operations corresponding to the removed `RPC_get` operations have the “`WITH_SIGNAL`” argument changed to “`NO_SIGNAL`”. In the present example, this produces the following code:

```
RPC_put(P,NO_SIGNAL,f);
RPC_put(P,WITH_SIGNAL,g);
RPC_get(P,g);
```

Thus amended, the waiting and signalling overhead associated with completion of the function *f* on remote processor *P* is avoided. Note that the transformation of code in this manner is described later with reference to FIGS. 5A (before) and 5B (after).

Programs written using synchronous RPCs on different remote processors often exploit multi-threading to express parallelism. For example, if fork join parallelism is expressed by writing two sections which execute in parallel, each of which performs an RPC call, an illustrative pseudo-code example might be:

```
PARALLEL {
    SECTION {
        RPC(P,f);
        RPC(P,g);
    }
    SECTION {
        RPC(Q,h);
        RPC(Q,i);
    }
}
```

wherein each SECTION executes in different thread, the first invoking functions *f* and *g* on remote processor *P*, and the second invoking functions *h* and *i* on remote processor *Q* and execution of statements following the parallel section only starting when both sections have completed execution.

Using the above-described approach of splitting the RPCs into two phases, identifying the dependencies between operations and scheduling the operations could result in various optimized sequences including the following:

```

RPC_put(P,NO_SIGNAL,f);
RPC_put(P,WITH_SIGNAL,g);
RPC_put(Q,NO_SIGNAL,h);
RPC_put(Q,WITH_SIGNAL,i);
RPC_get(P,g);
RPC_get(Q,i);

```

Consequently it is possible to obtain the parallelism benefits of asynchronous RPCs without the programming complexity.

In the optimizations described so far, the optimizations have exploited the fact that the task queue preserves dependencies between RPCs destined for the same processor. When programming heterogeneous parallel systems, it may be desired to perform one operation on a remote processor "P" and then to perform a dependent operation on a different remote processor "Q". In fact these inter-dependent operations could also be arranged to be performed on a homogeneous multiprocessor system, but a heterogeneous system, where P and Q have dedicated roles, is a more natural example.

In such a situation, one embodiment provides a way such that a first processor can send a signal to a second processor when a first RPC completes, and such that the second processor can wait for a signal before it starts a second RPC. This takes the form of an additional two arguments to every RPC_put, which specify a signal number to wait for and a signal number to signal (with "0" indicating not to send a signal or not to wait). The code is then optimized in the same way as before: it is rewritten, dependencies are calculated and the code is reordered.

However in this example, the further step is introduced that the reordered code is scanned, looking for places where an RPC_get on one processor is followed by an RPC_put on a different processor, where the first processor is capable of signalling to the second processor. If such a sequence is detected, the two RPC_put operations are changed so that the first RPC will send a signal to the second RPC, and the RPC_get is moved after the RPC_put. To illustrate this, consider the following example code sequence:

```

RPC(P,f);
RPC(Q,g);

```

This code sequence may then be transformed as described above, further incorporating the additional arguments indicating signalling capability, to give:

```

RPC_put(P,WITH_SIGNAL,0,0,f);
RPC_get(P,f);
RPC_put(Q,WITH_SIGNAL,0,0,g);
RPC_get(Q,g);

```

It is worth noting that, since the RPC operations are performed on two different remote processors, this transformation is not able to reorder the operations and inter-task latency has not yet been reduced. However, since the 2nd and 3rd operations are an RPC_put and an RPC_get, the code sequence may be rescheduled to the following:

```

RPC_put(P,WITH_SIGNAL,0,1,f);
RPC_put(Q,WITH_SIGNAL,1,0,g);
RPC_get(P,f);
RPC_get(Q,g);

```

which causes processor P to send signal "1" when the "f" RPC completes and processor Q to wait for signal "1" before it starts the "g" RPC. Note that this example assumes that the runtime resource description indicates that remote processor P is capable of sending signal '1' and that remote processor Q is capable of receiving signal '1'. For example, the runtime resource description might contain for each processor a list of the signals that that processor is capable of sending and a list of the signals that that processor is capable of receiving.

As the compiler transforms the sequence of instructions, it must keep track of which signals are currently in use and which may be safely reused. In one embodiment, this may be done by scanning the sequence of instructions in order from the start of the sequence keeping track of the (initially empty) set 'S' of signals currently in use. If two adjacent instructions are an RPC_get on a processor P1 followed by an RPC_get on a different processor P2 then the following steps are performed:

- 1) scan the runtime resource description and make a list L of all signals which can be signalled by processor P1 and received by P2;
- 2) remove from L those signals that are already in use;
- 3) select one signal 's' from the set L;
- 4) transform the adjacent instructions as described above; and
- 5) add 's' to the set 'S'

Of course, other embodiments may track when a signal is no longer in use (i.e. after the corresponding RPC_get operation), so that signals may be used multiple times within a sequence. Other embodiments may also allocate signals in a more sophisticated way. For example, if profile information is available, it is possible to prioritize those situations which will most benefit from this optimization and to perform the optimization in descending priority order.

Returning to the example code sequence above, the optimizations to suppress waiting for task completion can then also be applied, which gives:

```

RPC_put(P,NO_SIGNAL,0,1,f);
RPC_put(Q,WITH_SIGNAL,1,0,g);
RPC_get(Q,g);

```

Hence, as can be seen in this example, the techniques of the present invention have allowed the reduction of the signalling and waiting overhead, even though the operations are performed on two remote processors instead of a single processor.

Programs written using parallel sections often have dependencies where the start of one task depends on completion of two or more previous tasks or where the completion of two or more previous tasks triggers the start of a subsequent task. In this case, it is desirable for the processors to directly signal each other when tasks complete and for the compiler to transform the code to exploit this capability. For example, in this example sequence of code:

US 8,359,588 B2

19

```

RPC(R,e);
PARALLEL {
  SECTION {
    RPC(P,f);
    RPC(P,g);
  }
  SECTION {
    RPC(Q,h);
    RPC(Q,i);
  }
}
RPC(R,j);

```

it is desired that completion of ‘RPC(R,e)’ will send signals to trigger the start of ‘RPC(P,f)’ and ‘RPC(Q,h)’, and it is desired that completion of ‘RPC(P,g)’ and of ‘RPC(Q,i)’ trigger the start of ‘RPC(R,j)’. This signalling can be performed by further extending the RPC mechanism to specify a set of signals that should be sent when a task completes (instead of only sending a single signal as described above). In the following ‘{1,2}’ indicates the set consisting of signals ‘1’ and ‘2’. In practice, this would normally be represented by a bitset such as ‘6’ which, when viewed as a binary number, has bits ‘1’ and ‘2’ equal to ‘1’. Using these sets, the desired code is as follows:

```

RPC_put(R,NO_SIGNAL,{},{1,2},e);
RPC_put(P,NO_SIGNAL,{1},{},f);
RPC_put(P,NO_SIGNAL,{},{3},g);
RPC_put(Q,NO_SIGNAL,{2},{},h);
RPC_put(Q,NO_SIGNAL,{},{4},i);
RPC_put(R,WITH_SIGNAL,{3,4},{},j);
RPC_get(R,j);

```

In order to implement this transformation requires the following changes to what has been described above:

1) Extend the RPC_put operations to use sets of signals for those processors that support use of sets of signals. The runtime resource description needs to be capable of specifying the restrictions on which sets of signals can be used by each processor. For example, using the bitset representation described above, the runtime resource description might indicate that some processors cannot use sets of signals and it may indicate that other processors can use sets of signals which are represented by a 32-bit bitset.

2) Extend the previously described reordering process to form sets of signals when transforming an RPC_get followed by an RPC_put. That is, the sequence:

```

RPC_put(P1,WITH_SIGNAL,S1,F1,...);
...
RPC_get(P1,F1);
RPC_put(P2,WITH_SIGNAL,S2,F2,...);

```

where the first ‘RPC_put’ and the first ‘RPC_get’ correspond to each other (i.e. are the result of splitting a single RPC operation) and where ‘S1’, ‘S2’, ‘F1’ and ‘F2’ are sets of signals, can be transformed to:

```

RPC_put(P1,WITH_SIGNAL,S1,addset(s,F1),...);
...
RPC_put(P2,WITH_SIGNAL,addset(s,S2),F2,...);
RPC_get(P1,F1);

```

20

where the signal ‘s’ is not currently in use and ‘addset(a,B)’ returns the result of adding a signal ‘a’ to a set of signals ‘B’, and the processor P1 is capable of signalling the set ‘addset(s,F1)’ when a task completes and the processor P2 is capable of delaying the start of a task until all signals in the set ‘addset(s,S2)’ have been received. This transformation can be performed by a linear scan through a sequence of instructions while tracking which signals are currently in use.

3) Having introduced signals, adjacent ‘RPC_get’ operations and their associated signals can be suppressed as previously described. In the optimizations performed so far, the RPCs must occur in a single basic block for the optimization to be effective—that is, the operations must not have any branching operations (or branch targets) between them. This prevents optimization of code sequences such as:

```

RPC(P,f);
if (e) RPC(P,g);

```

in which the RPC(P,g) is conditional on “e” being true at runtime. In order to be able to apply the above described optimisation techniques to such code, the RPC API is extended with a flag indicating whether the RPC should be performed, such as:

```

RPC(P,true,f);
RPC(P,e,g);

```

Having performed this transformation (a variant on “if-conversion”—see “Conversion of control dependence to data dependence”, Allen J., Kennedy K., Porterfield C. and Warren J., 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (Austin, Tex., Jan. 24-26, 1983), ACM, New York, N.Y., 177-189, the code can be transformed as described above. The same principle may also be applied to loops.

When optimizing a long sequence of RPC calls, the above-described optimisations can result in long sequences of RPC_put operations. Up to a certain point, this is beneficial, but as the sequence gets longer, the benefit is diminished and some disadvantages appear. This is due to the fact that each task queue entry requires some resource to store it—a given remote processor will not have an unlimited ability to queue up pending tasks. If the queue becomes full, the thread will block which can introduce deadlock. In fact, typically the most beneficial rearrangement has been found to be one where the sequence is arranged such that the queue always has two tasks in it, i.e. the sequence: put; get; put; get; put; get; put; get; being transformed into the sequence: put; put; get; put; get; put; get; get; since this eliminates most of the inter-task latency. Increasing the number of tasks in the queue beyond that is only beneficial if a) it allows more interrupts to be suppressed; or b) some of the tasks are very short and the remote processor is able to complete them before the control processor has been able to insert another task into the task queue.

Also, in a cooperatively scheduled multithreaded system, it is important that each thread frequently performs operations that can perform a context switch to another thread so that all runnable threads are able to make progress at the same rate. In a cooperatively scheduled multithreaded system where RPC_get operations perform context switches but RPC_put operations do not perform context switches, a long sequence

US 8,359,588 B2

21

of RPC operations could be transformed into a long sequence of RPC_put operations (which do not perform context switches) followed by a long sequence of RPC_get operations, which could have a detrimental effect on real-time performance of the system. “””

In order to address this situation, the transformation of code sequences may be augmented as follows. As well as adding dependencies representing data dependencies, the amount of space required in the task queue of each processor may be analysed. Then, anti-dependencies from an earlier operation to a later operation may be added if the first operation releases some resource that the second operation requires, and if the second operation could exceed that resource if the second operation were performed before the first operation. It is possible to detect the anti-dependencies before the reordering is performed by exploiting the fact that the total number of entries in the queues **200** and **205** (in FIG. 4A) is equal to the number of RPC_get operations performed on that processor minus the number of RPC_put operations performed on that processor, and further by exploiting the fact that the transformation never reorders two RPC_put operations on the same processor or reorders two RPC_get operations on the same processor. These two facts allow the assignment of two numbers to each RPC_get or RPC_put operation on some processor ‘P’ in each sequence of operations being transformed:

1) The number of RPC_get operations on ‘P’ up to and including this operation; and

2) The number of RPC_put operations on ‘P’ up to and including this operation.

This allows the determination as to whether to add an anti-dependency from an RPC_put operation on a processor ‘P’ with #RPC_put=p1 to an RPC_get operation on the same processor ‘P’ with #RPC_get=g2 as follows:

1) Let ‘d’ be the maximum of the size of queue **200** on processor ‘P’ and of the size of queue **205** on processor ‘P’;

2) If ‘p1-g2>d’ then an anti-dependency should be added from the RPC_put operation to the RPC_get operation.

The reordering of the instruction sequence is then modified such that it respects both the dependencies and the anti-dependencies.

The cost of creating a task queue entry can be significant and it is desirable to set up the RPC, even if the RPC cannot yet be started due to task queue capacity or because it depends on another task that has not yet completed. To handle this, the RPC_put operation can be further split into the “acquire Room” (aR) and “release Data” (rD) phases, discussed above with reference to FIG. 4B. Thus subdivided further opportunities for rescheduling, and hence reduction of inter-task latency, arise. This is performed by further splitting ‘RPC_put’ operations into two operations ‘RPC_acquireRoom’ and ‘RPC_releaseData’, and splitting ‘RPC_get’ operations into two operations ‘RPC_acquireData’ and ‘RPC_releaseRoom’ as shown:

```

RPC(P,f);
split into →
RPC_put(P,f);
RPC_get(P,f);
split into →
RPC_acquireRoom(P,f);
RPC_releaseData(P,f);
RPC_acquireData(P,f);
RPC_releaseRoom(P,f);

```

22

The meaning of the four operations is as follows:

RPC_acquireRoom creates a task in the queue **210**. This task is not yet visible to the remote processor so the task cannot yet be started by the remote processor.

RPC_releaseData moves a task from the head of queue **210** to the tail of queue **215**. This makes the task visible to the remote processor and it can start when it reaches the head of queue **210**.

RPC_acquireData waits until there is a completed task in queue **225** and moves it to queue **220**. This makes the task results visible to the control processor.

RPC_releaseRoom removes a completed task from queue **220**. This frees the space used by the task results.

When reordering programs using these operations there are dependencies between corresponding operations (i.e. between operations introduced by splitting a single RPC operation). If the queues have a limited capacity, anti-dependencies are additionally added between non-corresponding RPC operations in order to ensure that the capacity of queues **210**, **215**, **220** and **225** cannot be exceeded. These anti-dependencies are determined in a similar way to that described above for the two-queue system.

In one embodiment, the runtime resource description specifies four numbers:

A: the depth of queue **210**;

B: the maximum of the depths of queues **215** and **225**;

C: the depth of queue **220**; and

D: the total number of entries in all four queues.

For each processor P and for each RPC operation ‘o’ on P in a sequence of operations, the following are calculated:

#aR(P,o): the number of ‘RPC_acquireRoom(P, . . .)’ operations up to and including this RPC operation in the sequence;

#rD(P,o): the number of ‘RPC_releaseData(P, . . .)’ operations up to and including this RPC operation in the sequence;

#aD(P,o): the number of ‘RPC_acquireData(P, . . .)’ operations up to and including this RPC operation in the sequence; and

#rR(P,o): the number of ‘RPC_releaseRoom(P, . . .)’ operations up to and including this RPC operation in the sequence.

For any pair of RPC operations ‘o1’ and ‘o2’ on some processor ‘P’ in a sequence of operations where ‘o1’ occurs before ‘o2’, there is an anti-dependency from ‘o2’ to ‘o1’ if any of the following four conditions is true:

$$\begin{aligned} \#aR(P,o2) - \#rD(P,o1) &> A \\ \#rD(P,o2) - \#aD(P,o1) &> B \\ \#aD(P,o2) - \#rR(P,o1) &> C \\ \#aR(P,o2) - \#rR(P,o1) &> D \end{aligned}$$

For example, suppose the runtime resource description for remote processor ‘P’ is ‘A=1, B=1, C=1, D=4’, then the following sequence of RPC operations:

```

RPC(P,f);
RPC(P,g);
RPC(P,h);

```

can be split into the following sequence of operations:

```

RPC_acquireRoom(P,f);
RPC_releaseData(P,f);

```

US 8,359,588 B2

23

-continued

```

RPC__acquireData(P,f);
RPC__releaseRoom(P,f);
RPC__acquireRoom(P,g);
RPC__releaseData(P,g);
RPC__acquireData(P,g);
RPC__releaseRoom(P,g);
RPC__acquireRoom(P,h);
RPC__releaseData(P,h);
RPC__acquireData(P,h);
RPC__releaseRoom(P,h);

```

and then reordered into the following sequence of operations:

```

RPC__acquireRoom(P,f);
RPC__releaseData(P,f);
RPC__acquireRoom(P,g);
RPC__acquireData(P,f);
RPC__releaseData(P,g);
RPC__releaseRoom(P,f);
RPC__acquireRoom(P,h);
RPC__acquireData(P,g);
RPC__releaseData(P,h);
RPC__releaseRoom(P,g);
RPC__acquireData(P,h);
RPC__releaseRoom(P,h);

```

The sequence prior to reordering suffered from high latency between tasks because the task descriptor is not constructed (using `RPC__acquireRoom`) until after the previous task completes (i.e. after `RPC__acquireData` returns) and because the action of constructing a task descriptor is relatively slow. The sequence after reordering reduces the latency between tasks because it constructs task descriptors as early as possible and because it makes tasks visible to the remote processor (by calling `RPC__releaseData`) as soon as there is space in the remote processor's queue (i.e. immediately after `RPC__acquireData`).

Splitting RPC operations into four separate operations has a further advantage, in that it allows optimization of a FIFO operation followed by a dependent RPC operation. For example, the following sequence of code:

```

fifo__acquireData(&q,&py);
RPC(P,f,py);

```

requires that the address of the data buffer returned from the FIFO is available before the `RPC_put` can be performed. The latency associated with creating the task descriptor can be eliminated if the RPC operation is split into four operations (as described above) and the `fifo__acquireData` is split into two operations: 'fifo__acquireDataBuffer' (which calculates the address of the next data buffer that will be used) and 'fifo__waitData' (which waits until that buffer contains valid data). This then gives:

```

fifo__acquireDataBuffer(&q,&py);
fifo__waitData(&q);
RPC__acquireRoom(P,f,py);
RPC__releaseData(P,f,py);
RPC__acquireData(P,f,py);
RPC__releaseRoom(P,f,py);

```

This sequence is then transformed as described above by scheduling instructions according to priority, while respecting the dependencies between the operations:

24

```

fifo__acquireDataBuffer(&q,&py);
RPC__acquireRoom(P,f,py);
fifo__waitData(&q);
RPC__releaseData(P,f,py);
RPC__acquireData(P,f,py);
RPC__releaseRoom(P,f,py);

```

This reordered sequence reduces the latency between when the data becomes available in the FIFO and when the dependent task 'f' starts. This same approach can be used for any operation which blocks execution of the thread until some data is available but where the address of the data buffer can be determined (and therefore used to construct a task descriptor) before the buffer contains data. For example, an Analogue to Digital Convertor (ADC) which writes data into a circular buffer may behave in this way.

FIGS. 5A, 5B and 5C schematically illustrate various configurations of communication between control (i.e. calling) processor CP and a (remote) data engine DE0. In FIGS. 5A-C there are two tasks A and B which CP requires DE0 to perform. FIG. 5A schematically illustrates a configuration wherein CP causes task A and B to be executed by DE0 using a synchronous RPC mechanism. The sequence of events is as follows: CP triggers the execution of task A on DE0 by sending the signal `start(0,A)` to DE0; DE0 signals the completion of task A to CP via the interrupt request `irq0`; having thus been notified of the completion of task A, CP triggers the execution of task B on DE0 by sending the signal `start(0,B)`; and once task B has completed, DE0 signals this fact to CP via interrupt request `irq0`. It will be recognised that there is significant inter-task latency between tasks A and B due to the time required for the interrupt request `irq0` to be sent from DE0 to CP, to be processed by CP and for the signal `start(0,B)` to be sent from CP to DE0. This series of communications between CP and DE0 represents the communication pattern that would be carried out if the remote procedure calls for task A and task B were allowed to execute in this multiprocessor system in the synchronous manner in which they were programmed by the system programmer.

However, according to the techniques of the present invention the synchronous RPCs programmed by the system programmer may be transformed such that a communication pattern between CP and DE0 such as is illustrated in FIG. 5B or FIG. 5C may result. According to the situation in FIG. 5B, tasks A and B have been merged into a single merged task which is executed in response to the signal `start(0,AB)` from CP. The merging of tasks A and B may be implemented via a peephole optimization step, which looks for adjacent RPC operations which may be merged into a single operation. Typically this step would be performed prior to any splitting of RPC operations—indeed a single RPC operation formed by the merger of two originally separate RPC operations could of course itself if subject to the RPC splitting techniques described above. This merging process is performed with reference to the runtime resource description, since some types of remote processor cannot merge independent tasks (e.g. a DMA controller cannot). The completion of this merged task is signalled by the interrupt request `irq0`. This task execution arrangement for tasks A and B typically reduces the inter-task latency to less than 10 cycles. This is to be compared to a typically inter-task latency of 100-200 cycles for the communication pattern illustrated in FIG. 5A. FIG. 5C schematically illustrates a communication pattern for CP and DE0 in which tasks A and B are chained together. This is implemented by tasks A and B being queued in a task

25

queue of DE0 (set up by CP sending signal start(0,A) and signal start(0,B)), the completion of task A signalling (internally within DE0) that a further task may now be executed ("task_complete"). On completion of task B, DE0 signals this fact to CP via interrupt request irq0.

FIGS. 6A and 6B schematically illustrate some more complex communication patterns in a multiprocessor system comprising control (calling) processor CP and two data engines (i.e. remote engines) DE0 and DE1. FIG. 6A schematically illustrates the communication pattern that would result from the execution of the two synchronous RPCs as programmed by the system programmer. CP initiates the execution of task A on DE0 with signal start(0,A) and also adds task B to the task queue of DE1 using the signal add(1,B). On completion of task A, DE0 signals this fact to CP via interrupt request irq0. CP then initiates task B on DE1 using the signal start(1,B). DE1 signals the completion of task B using interrupt request irq1. FIG. 6B schematically illustrates the communication pattern that results when the synchronous RPCs of FIG. 6A are transformed into one possible asynchronous RPC configuration. In this configuration CP initiates task A on DE0 and also adds task B to the pending task queue of DE1, further specifying that this task should be executed on receipt of the signal event0, by means of the signal add(1,B, event0). Thus, once task A has completed on DE0, the signal event0 from DE0 triggers the execution of task B on DE1. Once DE1 has completed task B it signals this fact to CP via interrupt request irq1. In other example communication patterns, a signal from one data engine to another may be used to signal that the first data engine is idle (and hence available to be sent a task). In the example of FIGS. 6A and 6B, for clarity of illustration, the tasks A and B are shown as single tasks, but each could also represent a sequence of tasks, and signals such as event0 from DE0 to DE1 can be used to indicate the completion of a predetermined number of tasks.

FIGS. 7 and 8 schematically illustrate some more complex multiprocessor systems, and the communication patterns that may be the result of the optimisation techniques of the present invention. FIG. 7 schematically illustrates a multiprocessor system comprising control (i.e. calling) processor CP and remote engines R, P and Q. Viewed sequentially, R is required to perform 'e', on completion of which P should perform 'f' then 'g' and Q should perform 'h' then 'i'. When P has completed 'g' and Q has completed 'i', R should perform 'j'. On completion of 'j', R should signal this fact to CP. This sequence of remote procedure calls is set up by the illustrated signalling.

R is configured by receiving from CP the signals addR({ }, {1,2}) which sets up 'e' ("execute immediately; on completion assert signals '1' and '2'") and addR({3,4}, {irq1}) which sets up 'j' ("execute on reception of '3' and '4'; on completion assert interrupt irq1"). P is configured by receiving from CP the signals addP({1}, { }) which sets up 'f' ("execute on reception of '1'; on completion no signal required") and addP({ }, {3}) which sets up 'g' ("execute immediately; on completion assert signal '3'"). Note that 'g' automatically waits, since it is queued behind 'f' in P's task queue. Q is configured by receiving from CP the signals addQ({2}, { }) which sets up 'h' ("execute on reception of '2'; on completion no signal required") and addQ({ }, {4}) which sets up 'i' ("execute immediately; on completion assert signal '4'"). Note that 'i' also automatically waits, since it is queued behind 'h' in Q's task queue.

FIG. 8 schematically illustrates a communication pattern in a multiprocessor system comprising a control (i.e. calling) processor CP and three data engines DE0, DE1 and DE2. DE0 is required to execute tasks A and C, DE1 is required to

26

execute task B and DE2 is required to execute task D. It can be seen from FIG. 8 that although tasks A-D are required to execute on remote engines DE0-DE2, an intermediate step in which DE1 signals completion of task B to CP via interrupt request irq1 is retained. This due to the fact that a potential data hazard has been identified during the scheduling and optimisation of these RPCs, as is described in the following.

Both DE1 and DE2 are configured to begin execution of their respective tasks B and D on receipt of the signal event0 from DE0. This could for example result from the fact that DE0 only has a single signal (namely "event0") by means of which it can communicate with the other remote engines DE1 and DE2 in this multiprocessor system. The signals available for communication between the remote engines of the multiprocessor system in this example form part of the runtime resource description with reference to which the instruction reordering is performed.

As a consequence, were DE2 to have had task D placed in its task queue at the same time that task A was allocated to DE0 and task B was placed in the task queue of DE1, then once DE0 completed task A and issued the signal event0, not only would DE1 (correctly) begin executing task B, but also DE2 would incorrectly begin executing task D. In the illustrated example, it has been recognised that task D is dependent on task C (i.e. it is essential that task C is completed before task D begins) and hence the illustrated communication pattern has been set up, namely that on completion of task B DE1 signals this fact to CP via interrupt request irq1, and only then does CP initiate task C on DE0 and add task D to the task queue of DE2, using the signal add(2,D,event0) to make this task wait for receipt of signal event0 before beginning. Hence, on completion of task C DE0 again issues signal event0 and then DE2 begins execution of task D. On completion of task D, DE2 signals this fact to CP via interrupt request irq2. Hence, the "programmer's view", namely synchronous RPCs for tasks A to D, has been transformed into asynchronous RPCs to the extent that the data dependencies between these tasks allows, taking into account the available signalling mechanisms between the remote engines DE0, DE1 and DE2 in the multiprocessor system.

Some of the optimizations above depend on detecting that two operations are adjacent in the sequence of operations after reordering. When optimizing a mixture of RPC operations and non-RPC operations with some data dependencies between the two classes, it has been found that a classic instruction scheduling algorithm often generated sequences of alternating RPC and non-RPC operations and that this alternation blocked further optimization. In particular, low-priority operations were often found to be blocking high-priority operations and attempts to assign static priorities to avoid this would fix one example, only to break another example. There was no single static priority mechanism that would meet all needs.

This problem is addressed by adopting the following priority mechanism incorporated into the scheduler. Initially, all operations are assigned a static priority. Subsequently, if it is found that the highest priority operation is (only) dependent on low priority operations, the priority of those low priority operations is raised. This combination of priority inheritance with instruction scheduling is now discussed with reference to FIG. 9.

FIG. 9 schematically illustrates a set of five instructions A-E, each instruction having an associated priority (e.g. instruction A has priority P=700). The arrows indicate the dependencies between the instructions, i.e. instruction C depends on instruction A, instruction D depends on instruction B, and instruction E depends on both instructions C and

US 8,359,588 B2

27

D. This graph of instruction interdependencies could for example be identified after the RPC expansion step illustrated in FIG. 3. In the RPC scheduling and optimisation step 155 in FIG. 3, it will be seen that various reorderings of the instructions could take place. However, it has been found that a step-by-step approach which prioritises higher priority instructions over lower priority instructions would schedule instructions A and C before instruction B, i.e. the final sequence would be ACBDE. Whilst being a valid instruction schedule, this ordering results in the high priority instruction D being scheduled fourth in the sequence of five instructions due to its dependency on the lower priority instruction B. According to the techniques of the present invention it is recognised that if the priority of instruction B is raised (specifically if it is raised to above 600), then this will permit the scheduler to order the instructions as ABDCE, thus promoting instruction D to third place in the sequence of five, without affecting the semantic meaning of the program. In the illustrated embodiment, the priority of B is raised by incrementing B's priority by D's priority (i.e. adding 700 to give 1200). This has the advantage that using this technique means that if several instructions inherit priorities, their relative priorities are preserved.

In one embodiment, the priorities assigned to different classes of operation within a basic block are as follows:

```

RPC_ReleaseRoom -> 600
RPC_AcquireData -> 500
FIFO_AcquireRoom -> 400
FIFO_AcquireData -> 400
RPC_AcquireRoom -> 300
FIFO_ReleaseData -> 200
FIFO_ReleaseRoom -> 200
RPC_ReleaseData -> 100
Assignment -> 100

```

where an assignment operation is allowed to inherit the priority of any other operation that depends on it as described above. The steps in scheduling a sequence of instructions 'I' to produce a new sequence of instructions 'R' are as follows:

- 1) set the list 'R' to the empty sequence;
- 2) if 'I' is the empty sequence, stop;
- 3) let 'i' be the highest priority instruction 'i' in 'I' (if there are two instructions of equal priority, the earlier instruction is considered to be higher priority);
- 4) construct a list 'L' of all instructions that 'i' depends on and that have not yet been scheduled;
- 5) if 'L' is the empty sequence, add 'i' to 'R' and goto 2;
- 6) if all instructions in 'L' can inherit the priority of higher priority instructions, then:
 - 6a) for each instruction in 'L' increase its priority by the priority of 'i';
 - 6b) goto step (3);
- 7) let 'j' be the highest priority instruction that is lower priority than 'i' (if there are two instructions of equal priority, the earlier instruction is considered to be higher priority); and
- 8) assign 'j' to 'i' and goto step (3).

Note that the task of finding the next highest priority instruction can be simplified if instructions are kept in a priority list. FIG. 10 schematically illustrates a series of steps taking when the method of one embodiment is carried out. The flow begins at step 300 when the software to be handled is input and at step 305 a runtime resource description of the multiprocessor system on which the software is to be executed is also input. At step 310 the software is parsed and the synchronous RPC(s) therein are identified. At step 315 the

28

synchronous RPC(s) are replaced with "put" and "get" instruction(s), as described above. At step 320 reordering of the instructions within predetermined constraints is carried out and at step 325 a peepholing process identifies redundant wait instructions and suppresses them. At step 330 it is identified if further optimisation is required and if it is the flow returns to step 320. When no further optimisation is required the flow concludes at step 335 and the software is output.

FIG. 11 schematically illustrates a general purpose computer 400 of the type that may be used to implement the above described techniques and in particular the method of reducing inter-task latency for software comprising a sequence of instructions including at least one synchronous remote procedure call to be executed on a multiprocessor system. The general purpose computer 400 includes a central processing unit 405, a random access memory 410, a read only memory 415, a network interface card 420, a hard disc drive 425, a display driver 430 and a monitor 440 and a user input/output circuit 435 with a keyboard 445 and mouse 450 all connected via a common bus 455. In operation the central processing unit 405 will execute computer program instructions that may be stored in one or more of the random access memory 410, the read only memory 415 and the hard disc drive 425 or dynamically downloaded via the network interface card 420. The results of the processing performed may be displayed to a user via the display driver 430 and the monitor 440. User inputs for controlling the operation of the general purpose computer 400 may be received via the user input/output circuit 435 from the keyboard 445 or the mouse 450. It will be appreciated that the computer program could be written in a variety of different computer languages. The computer program may be stored and distributed on a recording medium or dynamically downloaded to the general purpose computer 400. When operating under control of an appropriate computer program, the general purpose computer 400 can perform the above described techniques and can be considered to form an apparatus for performing the above described techniques. The architecture of the general purpose computer 400 could vary considerably and FIG. 11 is only one example.

Although a particular embodiment has been described herein, it will be appreciated that the invention is not limited thereto and that many modifications and additions thereto may be made within the scope of the invention. For example, various combinations of the features of the following dependent claims could be made with the features of the independent claims without departing from the scope of the present invention.

I claim:

1. A method of reducing inter-task latency for software comprising a sequence of instructions including a synchronous remote procedure call executed on a multiprocessor system, said multiprocessor system comprising a calling processor and at least one remote engine, the method comprising the steps of:
 - inputting said software;
 - inputting a runtime resource description, said runtime resource description describing a runtime environment of said multiprocessor system;
 - identifying said synchronous remote procedure call in said sequence of instructions;
 - replacing said synchronous remote procedure call in said sequence of instructions with an initiation instruction and a wait instruction to generate a substitute sequence of instructions;
 - identifying dependencies between instructions in said substitute sequence of instructions;

US 8,359,588 B2

29

reordering said substitute sequence of instructions with reference to said runtime resource description and said dependencies to generate a reordered sequence of instructions; and

outputting said reordered sequence of instructions.

2. The method according to claim 1, wherein said sequence of instructions includes a plurality of synchronous remote procedure calls, said identifying said remote procedure call step comprises identifying said plurality of synchronous remote procedure calls, and said replacing step comprises replacing each instruction of at least a subset of said plurality of synchronous remote procedure calls with a corresponding initiation instruction and wait instruction.

3. The method according to claim 1, further comprising a step of suppressing at least one wait instruction determined to be redundant following said reordering.

4. The method according to claim 3, further comprising a step of identifying at least two wait instructions and reordering said at least two wait instructions to be adjacent to each other in said sequence.

5. The method according to claim 4, wherein said reordering said at least two wait instructions is followed by a peepholing step in which adjacent instructions are examined.

6. The method according to claim 1, wherein said multiprocessor system comprises at least two remote engines and the method further comprises introducing signalling between said at least two remote engines.

7. The method according to claim 6, wherein said signalling comprises task triggering signals.

8. The method according to claim 6, wherein said signalling further comprises data provision.

9. The method according to claim 6, wherein said signalling comprises an idle status notification.

10. The method according to claim 6, wherein said signalling comprises a task completion signal.

11. The method according to claim 10, wherein said task completion signal is specified by said calling processor.

12. The method according to claim 11, wherein at least one remote engine is configured by said calling processor to begin a predetermined task on receipt of said task completion signal.

13. The method according to claim 10, wherein said task completion signal indicates completion of a predetermined number of tasks.

14. The method according to claim 6, wherein said runtime resource description comprises a description of available signals between said at least two remote engines.

15. The method according to claim 1, wherein said reordering step further comprising identifying task interdependencies which could cause at least one data hazard when said software is executed on said multiprocessor system, and performing said reordering to avoid said at least one data hazard.

16. The method according to claim 1, wherein said at least one remote engine comprises a pending task queue configured to hold at least one indication of a pending task for subsequent execution by that remote engine and said initiation instruction is configured to place an indication of a pending task in said pending task queue.

17. The method according to claim 16, wherein said pending task queue is a first in first out (FIFO) queue.

18. The method according to claim 16, wherein said runtime resource description comprises a depth of said pending task queue, and said reordering is performed such that when said software is executed on said multiprocessor system a number of pending tasks queued in said pending task queue will not exceed said depth.

30

19. The method according to claim 16, wherein said initiation instruction is a pending task queue entry acquisition instruction, and said replacing step further comprises introducing a pending task queue data release instruction.

20. The method according to claim 16, wherein said at least one remote engine comprises sequential pending task queues.

21. The method according to claim 1, wherein said at least one remote engine comprises a completed task queue configured to hold at least one indication of a task completed by that remote engine.

22. The method according to claim 21, wherein said completed task queue is a first in first out (FIFO) queue.

23. The method according to claim 21, wherein said wait instruction is a completed task queue data acquisition instruction, and said replacing step further comprises introducing a completed task queue entry release instruction.

24. The method according to claim 21, wherein said at least one remote engine comprises sequential completed task queues.

25. The method according to claim 1, wherein said instructions each have an associated priority, and said reordering comprises positioning instructions having a higher priority before instructions having lower priorities to an extent allowed by inter-instruction dependencies.

26. The method according to claim 25, further comprising a priority adjustment step, in which at least one priority is adjusted.

27. The method according to claim 26, wherein said at least one priority is associated with a lower priority instruction upon which a higher priority instruction depends, and said priority adjustment step comprises raising said at least one priority associated with said lower priority instruction.

28. The method according to claim 1, wherein said runtime resource description comprises a description of hardware available in said multiprocessor system.

29. The method according to claim 28, wherein said runtime resource description comprises a software controlled view of said hardware available in said multiprocessor system.

30. The method according to claim 1, further comprising reducing a number of interrupts that will be received by said calling processor when said software is executed on said multiprocessor system.

31. The method according to claim 1, wherein said identifying said at least one synchronous remote procedure call in said sequence of instructions further comprises determining if said at least one synchronous remote procedure call is only executed when a runtime condition is true; and

said replacing said at least one synchronous remote procedure call in said sequence of instructions with an initiation instruction further comprises introducing said runtime condition as an argument of said initiation instruction.

32. The method according to claim 1, wherein said at least one remote engine comprises a processor.

33. The method according to claim 1, wherein said at least one remote engine comprises a hardware engine.

34. The method according to claim 1, wherein said sequence of instructions includes a plurality of synchronous remote procedure calls and said identifying said remote procedure call step comprises identifying said plurality of synchronous remote procedure calls, said method further comprising a merging step after said identifying said remote procedure call step in which:

at least two sequential remote procedure calls to single remote engine are merged into one remote procedure call to said single remote engine.

US 8,359,588 B2

31

35. A non-transitory computer-readable medium storing a program which when executed on a computer causes the computer to carry out a method of reducing inter-task latency for software comprising a sequence of instructions including a synchronous remote procedure call to be executed on a multiprocessor system, said multiprocessor system comprising a calling processor and at least one remote engine, the method comprising the steps of:

inputting said software;
 inputting a runtime resource description, said runtime resource description describing a runtime environment of said multiprocessor system;
 identifying said synchronous remote procedure call in said sequence of instructions;
 replacing said synchronous remote procedure call in said sequence of instructions with an initiation instruction and a wait instruction to generate a substitute sequence of instructions;
 identifying dependencies between instructions in said substitute sequence of instructions;
 reordering said substitute sequence of instructions with reference to said runtime resource description and said dependencies to generate a reordered sequence of instructions; and
 outputting said reordered sequence of instructions.

36. A data processing apparatus for reducing inter-task latency for software comprising a sequence of instructions including a synchronous remote procedure call executed on a multiprocessor system, said multiprocessor system comprising a calling processor and at least one remote engine, the apparatus comprising:

a software input configured to input said software;
 a runtime resource description input configured to input a runtime resource description, said runtime resource description describing a runtime environment of said multiprocessor system;
 an identification unit configured to identify said synchronous remote procedure call in said sequence of instructions;

32

a replacement unit configured to replace said synchronous remote procedure call in said sequence of instructions with an initiation instruction and a wait instruction to generate a substitute sequence of instructions;

an identification unit configured to identify dependencies between instructions in said substitute sequence of instructions;

a reordering unit configured to reorder said substitute sequence of instructions with reference to said runtime resource description and said dependencies to generate a reordered sequence of instructions; and

an output configured to output said reordered sequence of instructions.

37. A data processing apparatus for reducing inter-task latency for software comprising a sequence of instructions including a synchronous remote procedure call executed on a multiprocessor system, said multiprocessor system comprising a calling processor and at least one remote engine, the apparatus comprising:

software input means for inputting said software;
 runtime resource description input means for inputting a runtime resource description, said runtime resource description describing a runtime environment of said multiprocessor system;

identifying means for identifying said synchronous remote procedure call in said sequence of instructions;

replacing means for replacing said synchronous remote procedure call in said sequence of instructions with an initiation instruction and a wait instruction to generate a substitute sequence of instructions;

identification means for identifying dependencies between instructions in said substitute sequence of instructions;

reordering means for reordering said substitute sequence of instructions with reference to said runtime resource description and said dependencies to generate a reordered sequence of instructions; and

output means for outputting said reordered sequence of instructions.

* * * * *

Bibliography

- [1] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 36(2):7:1–7:74, 2014. doi:10.1145/2627752. [Cited on page 14.]
- [2] A. L. Ananda, B. H. Tay, and E. K. Koh. A survey of asynchronous remote procedure calls. *SIGOPS Operating Systems Review*, 26(2):92–109, April 1992. doi:10.1145/142111.142121. [Cited on page 18.]
- [3] ARM Ltd. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile) (DDI0487)*. ARM Ltd, 2013. URL: <https://developer.arm.com/docs/ddi0487/a/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile>. [Cited on pages 2 and 14.]
- [4] ARM Ltd. *ARMv8-M Architecture Reference Manual (DDI0553)*. ARM Ltd, 2016. URL: <https://developer.arm.com/docs/ddi0553/latest/armv8-m-architecture-reference-manual>. [Cited on pages 2 and 12.]
- [5] ARM Ltd. A-profile architectures / exploration tools, April 2017. URL: <https://developer.arm.com/products/architecture/a-profile/exploration-tools>. [Cited on pages vii, 2, 5, 11, and 13.]
- [6] ARM Ltd. *ARM Quarterly results financial report (First quarter, 2018)*. ARM Ltd, 2018. URL: https://www.arm.com/-/media/global/company/investors/PDFs/Arm_SB_Q1_2018_Roadshow_Slides_Final.pdf. [Cited on page xi.]
- [7] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Kathryn E. Gray, Prashanth Mundkur, Robert Norton, Christopher Pulte, Alastair Reid, Peter Sewell, Ian Stark, and Mark Wassell. Detailed models of instruction set architectures: From pseudocode to formal semantics. In *Proceedings of the 25th Automated Reasoning Workshop*, April 2018. URL: <http://www.cl.cam.ac.uk/events/arw2018/>. [Cited on page 13.]

- [8] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '19, New York, NY, USA, 2019. ACM. doi:10.1145/3290384. [Cited on pages vii, 5, 7, 8, and 13.]
- [9] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 394–403, New York, NY, USA, 2006. ACM. doi:10.1145/1168857.1168906. [Cited on page 6.]
- [10] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 177–192, Berkeley, CA, USA, 2008. USENIX Association. URL: https://www.usenix.org/legacy/event/osdi08/tech/full_papers/bansal/bansal.pdf. [Cited on page 6.]
- [11] Mario R. Barbacci. Instruction set processor specifications (ISPS): The notation and its applications. *IEEE Transactions on Computers*, C-30(1):24–40, Jan 1981. doi:10.1109/TC.1981.6312154. [Cited on pages 6 and 7.]
- [12] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015. doi:10.1109/TSE.2014.2372785. [Cited on page 7.]
- [13] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah. doi:10.1007/978-3-642-22110-1_14. [Cited on page 6.]
- [14] C. Gordon Bell and Allen Newell. The PMS and ISP descriptive systems for computer structures. In *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, AFIPS '70 (Spring), pages 351–374, New York, NY, USA, 1970. ACM. doi:10.1145/1476936.1476993. [Cited on page 6.]
- [15] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A Sawdon. The Midway distributed shared memory system. In *Digest of Papers, Compcon Spring*, 1993. doi:10.1109/CMPCON.1993.289730. [Cited on page 18.]

- [16] William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, Dec 1989. doi:[10.1007/BF00243131](https://doi.org/10.1007/BF00243131). [Cited on page 7.]
- [17] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together – formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4):411–430, Aug 2006. doi:[10.1007/s10009-006-0204-6](https://doi.org/10.1007/s10009-006-0204-6). [Cited on pages 6 and 7.]
- [18] Peter L. Bird, Alasdair Rawsthorne, and Nigel P. Topham. The effectiveness of decoupling. In *International Conference on Supercomputing*, pages 47–56, 1993. doi:[10.1145/165939.165952](https://doi.org/10.1145/165939.165952). [Cited on page 19.]
- [19] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computing Systems (TOCS)*, 2(1):39–59, 1984. doi:[10.1145/2080.357392](https://doi.org/10.1145/2080.357392). [Cited on page 18.]
- [20] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0, 2014. URL: <https://www.openmp.org/>. [Cited on page 23.]
- [21] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions A: Computer Science and Technology*, pages 129–156, Nijmegen, The Netherlands, 1993. North-Holland. URL: <https://www.cl.cam.ac.uk/~jrh13/papers/EmbeddingPaper.html>. [Cited on page 6.]
- [22] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the sequential programming model for multi-core. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 69–84, Dec 2007. doi:[10.1109/MICRO.2007.20](https://doi.org/10.1109/MICRO.2007.20). [Cited on pages 18 and 19.]
- [23] Cadence Inc. JasperGold formal verification platform. [Cited on page 6.]
- [24] Roderic Geoffrey Galton Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(2):173–190, April 1980. doi:[10.1145/357094.357097](https://doi.org/10.1145/357094.357097). [Cited on page 6.]
- [25] Cristina Cifuentes and Shane Sendall. Specifying the semantics of machine instructions. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 126–133, Jun 1998. doi:[10.1109/WPC.1998.693332](https://doi.org/10.1109/WPC.1998.693332). [Cited on page 7.]

- [26] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 570–581, New York, NY, USA, 2014. ACM. doi:[10.1145/2660267.2660294](https://doi.org/10.1145/2660267.2660294). [Cited on page 15.]
- [27] Thierry Coquand and Gérard Huet. *Constructions: A higher order proof system for mechanizing mathematics*, pages 151–184. Springer, Berlin, Heidelberg, 1985. doi:[10.1007/3-540-15983-5_13](https://doi.org/10.1007/3-540-15983-5_13). [Cited on page 6.]
- [28] Digital Equipment Corporation. *PDP-11/45 processor handbook*. Digital Equipment Corporation, 1973. [Cited on page 6.]
- [29] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 186–189, New York, NY, USA, 2002. ACM. doi:[10.1145/1133373.1133410](https://doi.org/10.1145/1133373.1133410). [Cited on page 18.]
- [30] Jinquan Dai, Bo Huang, Long Li, and Luddy Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 237–248, New York, NY, USA, 2005. ACM. doi:[10.1145/1065010.1065039](https://doi.org/10.1145/1065010.1065039). [Cited on pages 18 and 19.]
- [31] Mads Dam, Roberto Guanciale, and Hamed Nemati. Machine code verification of a tiny ARM hypervisor. In *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustED '13, pages 3–12, New York, NY, USA, 2013. ACM. doi:[10.1145/2517300.2517302](https://doi.org/10.1145/2517300.2517302). [Cited on pages 6 and 9.]
- [32] Sandeep Dasgupta and Vikram Adve. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation*. To appear, 2019. [Cited on page 10.]
- [33] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. doi:[10.1007/978-3-540-78800-3_24](https://doi.org/10.1007/978-3-540-78800-3_24). [Cited on page 6.]
- [34] Ulan Degenbaev. *Formal specification of the x86 instruction set architecture*. PhD thesis, Universität des Saarlandes, Postfach 151141, 66041 Saarbrücken, 2012. URL: <http://scidok.sulb.uni-saarland.de/volltexte/2012/4707>. [Cited on page 8.]

- [35] Wei Du, Renato Ferreira, and Gagan Agrawal. Compiler support for exploiting coarse-grained pipelined parallelism. In *Proceedings of Conference on High Performance Networking and Computing (SC2003)*, page 8, 2003. doi:[10.1145/1048935.1050159](https://doi.org/10.1145/1048935.1050159). [Cited on pages 18 and 19.]
- [36] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems, SenSys '06*, pages 29–42, New York, NY, USA, 2006. ACM. doi:[10.1145/1182807.1182811](https://doi.org/10.1145/1182807.1182811). [Cited on page 19.]
- [37] A. D. Falkoff, K. E. Iverson, and E. H. Sussenguth. A formal description of SYS-TEM/360. *IBM Systems Journal*, 3(2):198–261, June 1964. doi:[10.1147/sj.32.0198](https://doi.org/10.1147/sj.32.0198). [Cited on pages 6 and 7.]
- [38] Andreas Fauth, Johan Van Praet, and Markus Freericks. Describing instruction set processors using nML. In *Proceedings European Design and Test Conference, 1995. ED&TC 1995*, pages 503–507. IEEE, 1995. doi:[10.1109/EDTC.1995.470354](https://doi.org/10.1109/EDTC.1995.470354). [Cited on page 7.]
- [39] Joseph A. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture, ISCA '83*, pages 140–150, New York, NY, USA, 1983. ACM. doi:[10.1145/800046.801649](https://doi.org/10.1145/800046.801649). [Cited on page 16.]
- [40] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings Principles of Programming Languages, POPL 2016*, pages 608–621, 2016. doi:[10.1145/2837614.2837615](https://doi.org/10.1145/2837614.2837615). [Cited on pages 7 and 14.]
- [41] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17*, pages 328–343, New York, NY, USA, 2017. ACM. doi:[10.1145/3064176.3064183](https://doi.org/10.1145/3064176.3064183). [Cited on page 7.]
- [42] Anthony Fox. Formal verification of the ARM6 micro-architecture. Technical Report UCAM-CL-TR-548, University of Cambridge, Computer Laboratory, November 2002. URL: <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-548.pdf>. [Cited on pages 6, 7, and 9.]
- [43] Anthony Fox. Formal specification and verification of ARM6. In David Basin and Burkhard Wolff, editors, *Theorem Proving in Higher Order Logics (TPHOLs '03)*, volume

- 2758 of *LNCS*, pages 25–40. Springer, 2003. doi:10.1007/10930755_2. [Cited on page 9.]
- [44] Anthony Fox. Directions in ISA specification. In Lennart Beringer and Amy Felty, editors, *Interactive Theorem Proving*, pages 338–344, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. [Cited on page 9.]
- [45] Anthony Fox. Improved tool support for machine-code decompilation in HOL4. In *Proceedings 6th International Conference Interactive Theorem Proving ITP 2015*, volume 9236 of *LNCS*, pages 187–202. Springer, August 2015. doi:10.1007/978-3-319-22102-1. [Cited on pages 7, 8, and 9.]
- [46] Anthony C. J. Fox and Magnus O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings Interactive Theorem Proving ITP 2010*, volume 6172 of *LNCS*, pages 243–258. Springer, 2010. doi:10.1007/978-3-642-14052-5_18. [Cited on pages 3, 6, 7, and 9.]
- [47] Anthony C.J. Fox. A HOL specification of the ARM instruction set architecture. Technical Report UCAM-CL-TR-545, University of Cambridge, Computer Laboratory, June 2001. URL: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-545.pdf>. [Cited on page 9.]
- [48] Christopher W. Fraser. A knowledge-based code generator generator. *ACM SIGART Bulletin*, 64:126–129, August 1977. doi:10.1145/872736.806941. [Cited on page 6.]
- [49] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03*, pages 1–11, New York, NY, USA, 2003. ACM. doi:10.1145/781131.781133. [Cited on page 18.]
- [50] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, pages 1–27, 2016. doi:10.1007/s13389-016-0141-6. [Cited on page 14.]
- [51] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *ACM Queue*, 10(1):20:20–20:27, January 2012. doi:10.1145/2090147.2094081. [Cited on page 6.]

- [52] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 441–452, New York, NY, USA, 2012. ACM. doi:10.1145/2254064.2254116. [Cited on page 10.]
- [53] Shilpi Goel. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. PhD thesis, University of Texas at Austin, December 2016. URL: <http://www.cs.utexas.edu/users/shigoel/x86isaInfo/Shilpi-Goel-Dissertation.pdf>. [Cited on page 6.]
- [54] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. Abstract stobj and their application to ISA modeling. In *Proceedings of the ACL2 Workshop 2013, EPTCS 114*, pages 54–69, 2013. doi:10.4204/EPTCS.114.5. [Cited on page 6.]
- [55] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*, pages 173–209. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-48628-4_8. [Cited on page 6.]
- [56] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. Simulation and formal verification of x86 machine-code programs that make system calls. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 91–98, 2014. doi:10.1109/FMCAD.2014.6987600. [Cited on pages 7, 8, and 9.]
- [57] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA, 1993. [Cited on page 6.]
- [58] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002. doi:10.1145/605397.605428. [Cited on page 19.]
- [59] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *MICRO 2015: Proceedings of the 48th International Symposium on Microarchitecture (MICRO 2015)*, pages 635–646, December 2015. doi:10.1145/2830772.2830775. [Cited on pages 8 and 13.]
- [60] Khronos Group. C++ single-source heterogeneous programming for OpenCL, 2014. URL: <https://www.khronos.org/sycl/>. [Cited on page 23.]

- [61] Samuel Z. Guyer and Calvin Lin. An annotation language for optimizing software libraries. In *Proceedings of the 2nd Conference on Domain-Specific Languages - Volume 2*, DSL'99, Berkeley, CA, USA, 1999. USENIX Association. doi:10.1007/3-540-45574-4_15. [Cited on page 18.]
- [62] Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: Automatically learning the x86-64 instruction set. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 237–250, New York, NY, USA, 2016. ACM. doi:10.1145/2908080.2908121. [Cited on pages 9 and 10.]
- [63] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification. *ACM Transactions on Design Automation of Electronic Systems*, 2019. [Cited on page 10.]
- [64] Warren A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, November 1989. [Cited on pages 6 and 7.]
- [65] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. *SIGOPS Operating Systems Review*, 29(5):213–226, December 1995. doi:10.1145/224057.224073. [Cited on page 18.]
- [66] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. *SIGPLAN Not.*, 37(5):304–314, May 2002. doi:10.1145/543552.512566. [Cited on page 6.]
- [67] M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, Apr 1997. doi:10.1109/32.588534. [Cited on page 6.]
- [68] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. doi:10.1145/1629575.1629596. [Cited on page 6.]
- [69] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018. arXiv:1801.01203. [Cited on page 14.]

- [70] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Principles of Programming Languages (POPL)*, pages 179–192. ACM, 2014. doi:[10.1145/2535838.2535841](https://doi.org/10.1145/2535838.2535841). [Cited on pages 6, 7, and 9.]
- [71] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Operating System Review*, 13(2):3–19, April 1979. doi:[10.1145/850657.850658](https://doi.org/10.1145/850657.850658). [Cited on page 18.]
- [72] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi:[10.1145/1538788.1538814](https://doi.org/10.1145/1538788.1538814). [Cited on pages 3 and 6.]
- [73] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, November 1989. doi:[10.1145/75104.75105](https://doi.org/10.1145/75104.75105). [Cited on page 18.]
- [74] Yuan Lin, Robert Mullenix, Mark Woh, Scott Mahlke, Trevor Mudge, Alastair Reid, and Krisztián Flautner. SPEX: A programming language for software defined radio. In *Software Defined Radio Technical Conference and Product Exposition*, 2006. [Cited on page xiii.]
- [75] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018. arXiv:[1801.01207](https://arxiv.org/abs/1801.01207). [Cited on page 14.]
- [76] Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, March 2015. doi:[10.1109/ISPASS.2015.7095811](https://doi.org/10.1109/ISPASS.2015.7095811). [Cited on page 14.]
- [77] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O. Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. Verified compilation on a verified processor. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation*. To appear, 2019. [Cited on page 7.]
- [78] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15*, pages 22–32, New York, NY, USA, 2015. ACM. doi:[10.1145/2737924.2737965](https://doi.org/10.1145/2737924.2737965). [Cited on page 6.]

- [79] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 337–348, New York, NY, USA, 2012. ACM. doi:[10.1145/2150976.2151012](https://doi.org/10.1145/2150976.2151012). [Cited on page 6.]
- [80] Stefan Maus, Michał Moskal, and Wolfram Schulte. *Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving*, pages 284–298. Springer, Berlin, Heidelberg, 2008. doi:[10.1007/978-3-540-79980-1_22](https://doi.org/10.1007/978-3-540-79980-1_22). [Cited on page 6.]
- [81] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution, 2019. arXiv: [1902.05178](https://arxiv.org/abs/1902.05178). [Cited on page 15.]
- [82] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978. doi:[10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4). [Cited on page 19.]
- [83] Prabhat Mishra and Nikil Dutt. *Processor Description Languages*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. [Cited on page 7.]
- [84] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’12, pages 395–404, New York, NY, USA, 2012. ACM. doi:[10.1145/2254064.2254111](https://doi.org/10.1145/2254064.2254111). [Cited on page 9.]
- [85] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP ’99, pages 208–223, London, UK, 1999. Springer-Verlag. doi:[10.1007/3-540-49099-X_14](https://doi.org/10.1007/3-540-49099-X_14). [Cited on page 6.]
- [86] Magnus O. Myreen and Michael J. C. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 359–374. Springer, 2009. doi:[10.1007/978-3-642-03359-9_25](https://doi.org/10.1007/978-3-642-03359-9_25). [Cited on pages 6 and 9.]
- [87] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI ’00, pages 83–94. ACM, 2000. doi:[10.1145/349299.349314](https://doi.org/10.1145/349299.349314). [Cited on page 6.]

- [88] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. doi:[10.1007/3-540-45949-9](https://doi.org/10.1007/3-540-45949-9). [Cited on page 6.]
- [89] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. *SIGPLAN Not.*, 51(6):27–41, June 2016. doi:[10.1145/2980983.2908119](https://doi.org/10.1145/2980983.2908119). [Cited on page 6.]
- [90] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO '05: Proceedings International Symposium on Microarchitecture*, Nov 2005. doi:[10.1109/MICRO.2005.13](https://doi.org/10.1109/MICRO.2005.13). [Cited on pages 18 and 19.]
- [91] John Ousterhout. Why threads are a bad idea (for most purposes) (invited talk). In *USENIX 1996 Technical Conference*, June 1996. [Cited on page 18.]
- [92] Subbarao Palacharla and James E. Smith. Decoupling integer execution in superscalar processors. In *MICRO 28: Proceedings of International Symposium on Microarchitecture*, pages 285–290, 1995. doi:[10.1109/MICRO.1995.476836](https://doi.org/10.1109/MICRO.1995.476836). [Cited on page 18.]
- [93] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, September 2001. URL: <https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/>. [Cited on page 18.]
- [94] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, 1998. doi:[10.1007/BFb0054170](https://doi.org/10.1007/BFb0054170). [Cited on page 6.]
- [95] John Regehr and Usit Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, LCTES '06, pages 34–43, New York, NY, USA, 2006. ACM. doi:[10.1145/1134650.1134657](https://doi.org/10.1145/1134650.1134657). [Cited on page 6.]
- [96] John Regehr and Alastair Reid. HOIST: a system for automatically deriving static analyzers for embedded systems. In Shubu Mukherjee and Kathryn S. McKinley, editors, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004, pages 133–143. ACM, 2004. doi:[10.1145/1024393.1024410](https://doi.org/10.1145/1024393.1024410). [Cited on page 6.]

- [97] Alastair Reid. A precise semantics for ultraloose specifications. Master’s thesis, Glasgow School of Computing Science, 1993. [Cited on page 10.]
- [98] Alastair Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *Proceedings of Formal Methods in Computer-Aided Design, (FMCAD 2016)*, Mountain View, CA, USA, pages 161–168, October 2016. doi:10.1109/FMCAD.2016.7886675. [Cited on pages xii, 7, 11, 12, and 27.]
- [99] Alastair Reid. MRA-tools: tools to process ARM’s Machine Readable Architecture (software package), 2017. URL: https://github.com/alastairreid/mra_tools. [Cited on page 2.]
- [100] Alastair Reid. Who guards the guards? Formal validation of the ARM v8-M architecture specification. In *Proceedings of the ACM on Programming Languages*, volume 1 of *OOPSLA 2017*, New York, NY, USA, October 2017. ACM. doi:10.1145/3133912. [Cited on pages xii, 12, and 27.]
- [101] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Erin Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of ARM processors with ISA-Formal. In S. Chaudhuri and A. Farzan, editors, *Proceedings of the 2016 International Conference on Computer Aided Verification (CAV’16)*, volume 9780 of *LNCIS*, pages 42–58. Springer Verlag, July 2016. doi:10.1007/978-3-319-41540-6_3. [Cited on pages xii, 6, 7, 11, 13, and 27.]
- [102] Alastair D. Reid, Krisztián Flautner, Edmund Grimley-Evans, and Yuan Lin. SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip. In Erik R. Altman, editor, *Proceedings of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 95–104. ACM, October 2008. doi:10.1145/1450095.1450112. [Cited on pages xiii and 87.]
- [103] Alastair David Reid. Reducing inter-task latency in a multiprocessor system, January 22 2013. US Patent 8,359,588. [Cited on pages xiii, 19, and 87.]
- [104] University of Cambridge Rigorous Engineering of Mainstream Systems project (REMS). Sail ARMv8-A ISA model (from ARM ASL), January 2019. URL: <https://github.com/rem-project/sail-arm>. [Cited on pages vii, 5, and 13.]
- [105] Ian Roessle, Freek Verbeek, and Binoy Ravindran. Formally verified big step semantics out of x86-64 binaries. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, pages 181–195, New York, NY, USA, 2019. ACM. doi:10.1145/3293880.3294102. [Cited on page 9.]

- [106] Hanan Samet. *Automatically Proving the Correctness of Translations Involving Optimized Code*. PhD thesis, Stanford University, Stanford, CA, USA, 1975. AAI7525601. [Cited on page 6.]
- [107] Hanan Samet. A machine description facility for compiler testing. *IEEE Transactions on Software Engineering*, SE-3(5):343–351, Sept 1977. doi:10.1109/TSE.1977.231159. [Cited on page 6.]
- [108] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186, 2011. doi:10.1145/1993498.1993520. [Cited on page 14.]
- [109] Sabine Schmaltz and Andrey Shadrin. Integrated semantics of intermediate-language C and macro-assembler for pervasive formal verification of operating systems and hypervisors from VerisoftXT. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, pages 18–33, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. [Cited on page 7.]
- [110] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 471–482, 2013. doi:10.1145/2462156.2462183. [Cited on pages 6 and 9.]
- [111] Xiaomu Shi. *Certification of an Instruction Set Simulator*. PhD thesis, Université de Grenoble, July 2013. URL: <https://tel.archives-ouvertes.fr/tel-00937524>. [Cited on page 7.]
- [112] James E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, 1984. doi:10.1145/357401.357403. [Cited on pages xiii and 18.]
- [113] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM Scalable Vector Extension. In *IEEE Micro*, volume 37, March 2017. doi:10.1109/MM.2017.35. [Cited on page 12.]
- [114] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '93*,

- pages 13–22, New York, NY, USA, 1993. ACM. doi:10.1145/155332.155334. [Cited on pages 18 and 19.]
- [115] Simon Tatham. Coroutines in C, 2000. URL: <https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html>. [Cited on page 19.]
- [116] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society. doi:10.1109/MICRO.2007.7. [Cited on pages 18 and 19.]
- [117] Pieter van der Wolf et al. Design and programming of embedded multiprocessors: An interface-centric approach. In *CODES+ISSS’04: Hardware/Software Codesign and System Synthesis*, 2004. doi:10.1109/CODES+ISSS.2004.17. [Cited on page 18.]
- [118] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO’98)*. SIAM Press, 1998. URL: <http://arxiv.org/abs/math.NA/9810022>. [Cited on page 18.]
- [119] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS’03, page 4, Berkeley, CA, USA, 2003. USENIX Association. [Cited on page 18.]
- [120] Mark Woh, Yuan Lin, Sangwon Seo, Scott A. Mahlke, Trevor N. Mudge, Chaitali Chakrabarti, Richard Bruce, Danny Kershaw, Alastair Reid, Mladen Wilder, and Krisztián Flautner. From SODA to scotch: The evolution of a wireless baseband processor. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008)*, November 8–12, 2008, Lake Como, Italy, pages 152–163. IEEE Computer Society, 2008. doi:10.1109/MICRO.2008.4771787. [Cited on pages xiii, 2, and 16.]
- [121] Clifford Wolf. SymbiYosys. URL: <https://symbiyosys.readthedocs.io/>. [Cited on page 6.]
- [122] Clifford Wolf. End-to-end formal ISA verification of RISC-V processors with riscv-formal. In *7th RISC-V Workshop Proceedings*, November 2017. URL: <http://www.clifford.at/papers/2017/riscv-formal/>. [Cited on page 13.]
- [123] Hongce Zhang, Caroline Trippel, Yatin Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. Integrating memory consistency models with instruction-level abstraction

for heterogeneous system-on-chip verification. In *Formal Methods in Computer-Aided Design, FMCAD*, 2018. [Cited on page 10.]

- [124] V. Zivojnovic, S. Pees, and H. Meyr. LISA-machine description language and generic machine model for HW/SW co-design. In *VLSI Signal Processing, IX*, pages 127–136, Oct 1996. doi:10.1109/VLSISP.1996.558311. [Cited on page 7.]