THE UNIVERSITY *of* EDINBURGH

Speeding up Dynamic Compilation

Concurrent and Parallel Dynamic Compilation

Igor Böhm

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2013

Abstract

The main challenge faced by a dynamic compilation system is to detect and translate frequently executed program regions into highly efficient native code as fast as possible. To efficiently reduce dynamic compilation latency, a dynamic compilation system must improve its workload throughput, i.e. compile more application hotspots per time. As time for dynamic compilation adds to the overall execution time, the dynamic compiler is often decoupled and operates in a separate thread independent from the main execution loop to reduce the overhead of dynamic compilation.

This thesis proposes innovative techniques aimed at effectively speeding up dynamic compilation. The first contribution is a generalised region recording scheme optimised for program representations that require dynamic code discovery (e.g. binary program representations). The second contribution reduces dynamic compilation cost by incrementally compiling several hot regions in a concurrent and parallel task farm. Altogether the combination of generalised light-weight code discovery, large translation units, dynamic work scheduling, and concurrent and parallel dynamic compilation ensures timely and efficient processing of compilation workloads. Compared to state-of-the-art dynamic compilation approaches, speedups of up to 2.08 are demonstrated for industry standard benchmarks such as BioPerf, Spec Cpu 2006, and Eembc.

Next, innovative applications of the proposed dynamic compilation scheme to speed up architectural and micro-architectural performance modelling are demonstrated. The main contribution in this context is to exploit runtime information to dynamically generate optimised code that accurately models architectural and micro-architectural components. Consequently, compilation units are larger and more complex resulting in increased compilation latencies. Large and complex compilation units present an ideal use case for our concurrent and parallel dynamic compilation infrastructure. We demonstrate that our novel micro-architectural performance modelling is faster than state-of-the-art Fpga-based simulation, whilst providing the same level of accuracy.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- **I.Böhm**, T.E. von Koch, S.C.Kyle, B.Franke, and N.Topham, "Generalized Just-in-Time Trace Compilation using a Parallel Task Farm in a Dynamic Binary Translator," in *Proceedings of the Conference on Programming Language Design and Implementation* (PLDI'11), 2011.

- S.Kyle, **I.Böhm**, B.Franke, H.Leather, and N.Topham, "Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-Core Processors using Region-based Just-in-Time Dynamic Binary Translation," in *Proceedings of the International Conference on Languages, Compilers, Tools and Theory for Embedded Systems* (LCTES'12), 2012.

- O.Almer, **I.Böhm**, T.E. von Koch, B.Franke, S.Kyle, V.Seeker, C. Thompson, and N.Topham, "Scalable Multi-Core Simulation using Parallel Dynamic Binary Translation," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* (SAMOS'11), 2011.

- **I.Böhm**, B.Franke, and N.Topham, "Cycle-Accurate Performance Modelling in an Ultra-fast Just-in-Time Dynamic Binary Translation Instruction Set Simulator," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* (SAMOS'10), 2010.

- **I.Böhm**, B.Franke, and N.Topham, "Cycle-Accurate Performance Modelling in an Ultra-Fast Just-In-Time Dynamic Binary Translation Instruction Set Simulator," in *Transactions on High-Performance Embedded Architectures and Compilers* (HiPEAC'11), vol. 5, no. 4, 2011.

(*Igor Böhm*)

Table of Contents

Table of Contents . . . . . **i**

1 Introduction **1**
1.1 Problem - Dynamic Compilation Latency 2
1.1.1 Code Discovery 2
1.1.2 Dynamic Compilation 3
1.1.3 Compilation Latency 3
1.2 Solution - Speeding up Dynamic Compilation 4
1.3 Contributions 6
1.4 Thesis Structure 7
1.5 Summary 8

2 Dynamic Compilation Background **9**
2.1 Introduction 9
2.2 Code Discovery 11
2.2.1 Static Code Discovery 11
2.2.2 Dynamic Code Discovery 11
2.2.3 Hybrid Code Discovery 12
2.3 Dynamic Compilation 12
2.3.1 Deciding When to Compile 14
2.3.2 Handling Dynamic Compilation Workload 14
2.3.3 Scheduling Dynamic Compilation Workload 16
2.4 Summary 17

3 Related Work **19**
3.1 Dynamic Binary Translation and Optimisation 19
3.2 Trace- and Region-based Dynamic Compilation 21
3.3 Parallel and Concurrent Dynamic Compilation 22
3.4 Architectural Performance Modelling 24
3.5 Micro-architectural Performance Modelling 25
3.6 Summary 27

4 Concurrent and Parallel Dynamic Compilation 29
4.1 Motivating Example 30
4.2 Trace Based Dynamic Region Detection 32
4.2.1 Tracing Regions 33
4.2.2 Partitioning Regions 34
4.3 Concurrent and Parallel Dynamic Compilation 36
4.3.1 Dynamic Compilation Flow 36
4.3.2 Interpretive - Native Execution Mode Transition 39
4.3.3 Scheduling Compilation Workload 40
4.3.4 Adaptive Hotspot Threshold Selection 41
4.4 Self-Referencing and Self-Modifying Code 42
4.5 Scaling for Multi-threaded Execution Environments 44
4.5.1 Code Discovery in Multi-threaded Applications 45
4.5.2 Motivating Example 46
4.5.3 Thread Agnostic Code Generation 47
4.5.4 Inter-thread Region Sharing 49
4.5.5 Region Translation Caching 50
4.6 Evaluation and Analysis of Results 52
4.6.1 Benchmarks and Experimental Setup 52
4.6.2 Speedup 54
4.6.3 Scalability 57
4.6.4 Throughput 59
4.6.5 Region Sharing 60
4.7 Summary 63

5 Architectural and Microarchitectural Modelling 65
5.1 Motivating Example 68
5.2 Dynamic Compilation of Microarchitectural Components . . . 68
5.2.1 Microarchitecture 70
5.2.2 Pipeline Model 71
5.3 Instruction Operand Dependencies and Side Effects 74
5.3.1 Control Flow and Branch Prediction 74
5.3.2 Memory Model 75
5.4 Evaluation and Analysis of Results 76
5.4.1 Benchmarks and Experimental Setup 76
5.4.2 Speedup 77
5.5 Summary 81

6 Conclusions **83**
6.1 Contributions 83
6.1.1 Concurrent and Parallel Dynamic Compilation 84
6.1.2 Architectural and Microarchitectural Modelling 84
6.2 Critical Analysis 85
6.2.1 Dynamic Compilation Methodology 85
6.2.2 Microarchitectural Modelling 86
6.3 Future Work 87

List of Figures **89**

List of Tables **91**

List of Listings **93**

Bibliography **95**

—For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers.

Gene Amdahl - 1967

1 Introduction

This chapter gives a general introduction to the goals and main contributions of, and challenges addressed by this thesis.

Programming language implementations using portable program representations [26, 60, 67, 82], simulation systems (e.g instruction set simulators [14, 15]), dynamic binary translators [3, 16], and dynamic optimisation systems [9, 22] all rely on implementations that defer machine specific code generation and optimisation until runtime [5] to enable satisfactory execution speeds. Translation that occurs after a program begins execution is referred to as dynamic compilation or Just-In-Time (JIT) compilation [7]. Its purpose is to improve the time and space efficiency of programs. The technology is so effective that it has become a part of our everyday lives, embedded in web browsers (e.g. JavaScript engines), virtual machines (e.g. Oracle HotSpot/JRockit JVM, Common Language Runtime), mobile device operating systems (e.g. Google Android), and many more.

Dynamic compilation occurring at runtime inevitably incurs an overhead and thus contributes to the total execution time of a program. There is a trade-off between the time spent for dynamic compilation and total execution time. If, on the one hand, lots of effort is spent on aggressive dynamic compilation to ensure generation of highly efficient native code, too much compilation time will be contributed to the total execution time of a program. If, on the other hand, too little time is spent on optimising code for execution during dynamic compilation, the runtime performance of the target program is likely to be suboptimal. This thesis presents a dynamic code discovery and

compilation strategy that is aimed at reducing dynamic compilation latency by adaptively prioritising the hottest most recently executed program regions to be compiled in parallel, concurrently with the execution of the target program.

1.1 Problem - Dynamic Compilation Latency

The purpose of a dynamic compilation system is to achieve the lowest possible total execution time of a target program. The main challenge is to quickly discover executable code and translate frequently executed code as fast as possible into highly efficient native code. In this Section we first outline the problem of code discovery followed by strategies for dynamic translation of executable code. Finally, we conclude by recognising that dynamic compilation latency indeed is a problem and a metric we should try to optimise. Reducing dynamic compilation latency enables a reduction in total execution time (i.e. performance improvement) and improves response time for interactive applications.

1.1.1 Code Discovery

In general there are two classes of program representations. The first class includes program representations where instructions can be separated from data (e.g. Java byte-code, Common Intermediate Language, JavaScript). Such representations are amenable to static translation ahead of execution. The second class includes binary program representations in the form of hardware machine instructions where "the identification of executable code, i.e. the separation of instructions from data [...] is equivalent to the Halting Problem and is therefore unsolvable in general" [55].

Consequently it is harder to *detect* program regions for dynamic compilation for the latter class of program representations as this cannot be done statically. Instead, executable code discovery is performed dynamically, translating sections of code incrementally as the program reaches them [93]. In research literature dynamic discovery of instruction sequences is commonly referred to as *tracing* [9] and the process of dynamically compiling traces as *trace compilation*. The dynamic code discovery strategy presented in this thesis falls into this category, but can equally well be applied to program representations that allow static discovery of executable code.

The data-structure used for recording traces must be carefully chosen as it is manipulated on the critical path of program execution. It must be compact in terms of size, enable fast trace recording and analysis, and permit efficient native code generation. Various linear and tree-like data-structures have been proposed and are used in state-of-the-art systems [9, 40]. More recent implementations transform linear trace data-structures into graph-based structures [11, 49, 50] to work around the problem of excessive code duplication that occurs due the fact that a single basic block can be recorded as a part of multiple traces.

1.1.2 Dynamic Compilation

Once executable code has been discovered it must be translated into efficient native code as fast as possible. Deciding when and how to dynamically compile a region of code is the next challenge. State-of-the-art dynamic compilation systems [9, 26, 60, 67, 82] are threshold based. They consider a program region for compilation or re-compilation when a user defined execution threshold is reached. Choosing a low execution threshold results in the selection of many program regions for compilation and a high workload for the dynamic compilation subsystem. While a high execution threshold causes less pressure on the dynamic compiler as only very frequently executed regions of code are considered for compilation, more time is spent executing code in a slow interpreter or unoptimised machine code.

Application runtime parallelism is another factor affecting the workload of a dynamic compiler. A multi-threaded task parallel application executing different code in each thread puts much more pressure on a dynamic compilation system than a data-parallel or sequential application. Finally, a dynamic compiler must decide how to compile a region of code by committing to a set of optimisations it applies. Again, there is a trade-off between the quality of generated machine code and time spent on optimisation.

1.1.3 Compilation Latency

Faster availability of optimised native code minimises the time spent in an interpreter or unoptimised machine code, thereby improving application performance. As dynamic compilation adds to the overall execution time, it is often decoupled and operates concurrently in a separate thread independent from the main execution loop [47,63,64] (see ① in Figure 1.1). This approach

improves application responsiveness by reducing pause times due to dynamic compilation. It does not, however, reduce dynamic compilation latency.

Dynamic compilation workload has a significant impact on dynamic compilation latency. As new program regions are discovered and dispatched for dynamic compilation, the dynamic compiler may still be busy compiling previous code regions. To avoid waiting until the dynamic compiler has finished its current work, a common solution is to put pending translations into a queue [67] for later processing. While this solution avoids waiting, it does not solve the dynamic compilation latency problem. An overloaded dynamic compilation system will have many pending program regions in its queue as the dynamic compilation thread cannot keep up with the workload, causing a long delay between code region dispatch and native code availability.

Especially in the context of high-speed architectural and micro-architectural processor simulation, dynamic compilation latency is a significant problem. To enable effective exploration of the architectural and micro-architectural processor design space, customised code simulating architectural and micro-architectural components (e.g. processor pipeline, caches, closely coupled memories, memory management unit, brach prediction unit) is dynamically compiled. Consequently compilation units are larger and more complex resulting in high compilation latencies.

1.2 Solution - Speeding up Dynamic Compilation

In this Section the high-level idea for speeding up dynamic compilation is outlined followed by a number of key research innovations aimed at improving dynamic code discovery and reduction of dynamic compilation latency in state-of-the-art virtual machines and execution environments.

To effectively reduce dynamic compilation latency, a dynamic compilation system must improve its workload throughput, i.e. compile more application hotspots per unit of time. This problem is related to the workload static compilers face when compiling large scale applications. To improve compilation throughput, build systems or static compiler drivers identify independent translation units and exploit separate compilation, a feature supported by most programming languages, to compile independent translation units in parallel [111]. Given unlimited hardware parallelism, the speedup achievable by a parallel compiler is limited by the time it takes to compile the largest

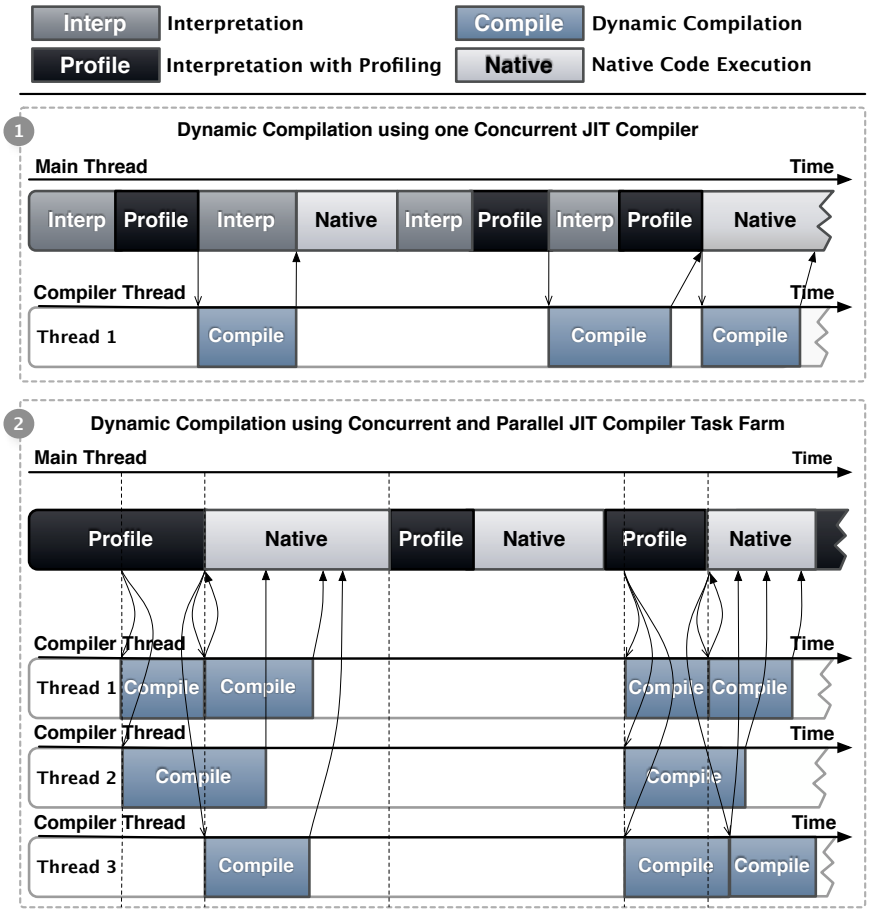

Figure 1.1: Motivating Example: ① Dynamic compilation using one concurrent JIT compiler decoupled from main execution thread. ② Demonstrates concurrent and parallel dynamic compilation task farm to effectively reduce dynamic compilation latency and enable earlier transition to native code execution.

translation unit, plus a sequential fraction that consists of discovering independent translation units and linking the final executable. Successfully exploiting concurrent and parallel compilation in the context of a dynamic compiler using dynamic code discovery is the key research contribution of this thesis.

First, a novel way to dynamically discover, i.e. trace, executable code incrementally for dynamic compilation is presented. This can be applied to program representations where static code discovery is not possible. It records dynamic control flow and is light-weight, enabling tracing right from the first instruction. This avoids a period of interpretation until either specific

structures (e.g. loop headers) are encountered or execution thresholds are exceeded. Consequently more opportunities for dynamic compilation can be discovered earlier.

Being able to discover more independent code regions for dynamic compilation, the idea of decoupled dynamic compilation is taken a step further. A scalable, truly parallel dynamic compilation scheme based on the parallel task farm design pattern [104] is proposed. It is designed to execute concurrently with the target program and effectively reduces dynamic compilation latency by exploiting the concept of parallel compilation. Figure 1.1 illustrates that improved code discovery and task parallel dynamic compilation ② improves compilation throughput and reduces the time until native code execution can start in comparison to current approaches ①.

Finally, it is paramount that the proposed techniques scale and adapt automatically to changing workloads, and efficiently exploit parallelism and concurrency available on contemporary multi-core architectures. Given resource constraints (e.g. number of dynamic compilation threads) execution frequency thresholds adapt automatically to achieve a good balance between dynamic compilation throughput and execution speed. Given several compilation units awaiting dynamic compilation an efficient dynamic work scheduling scheme ensures that the most important compilation units are priositised for compilation.

1.3 Contributions

This thesis presents new techniques and approaches for dynamic code discovery and dynamic compilation exploiting parallel design patterns to improve dynamic compilation throughput, better response times, and utilise multi-core architectures. Among the main contributions of this thesis are:

1. The introduction of a novel interval based and light-weight dynamic code discovery approach. Dynamic application control flow is recorded and analysed for incremental dynamic compilation, considering all frequently executed paths (i.e. hot regions) and not just traces restricted to regular loops,

2. the introduction of an innovative parallel task farming strategy for truly concurrent dynamic compilation of hot regions,

3. the development of dynamic work scheduling and adaptive hotspot threshold selection schemes that give priority to the most recent, and frequently executed regions in order to reduce time spent in interpreted or unoptimised execution mode, and

4. an extensive evaluation within a dynamic binary translator targeting the ARCompact ISA, and

5. the application of the proposed dynamic compilation scheme to architectural and micro-architectural simulation demonstrating how runtime information can be exploited to generate optimised code accurately modelling architectural and micro-architectural components at high speeds.

To demonstrate the effectiveness of the proposed techniques three industry standard benchmark suites, EEMBC, BIOPERF and SPEC CPU2006, as well as a variety of custom benchmarks based on real world applications such as operating system simulation, audio and video decoding have been used. Across short- and long-running benchmarks the proposed scheme is robust and never results in a slowdown. In fact, using four processors total execution time can be reduced by on average 11.5% over state-of-the-art concurrent dynamic compilation.

1.4 Thesis Structure

This thesis is organised as follows.

Chapter 2 introduces different dynamic compilation techniques and approaches used throughout this thesis.

Chapter 3 presents the related work. Prior work on dynamic binary translation, trace compilation and optimisation, as well as parallel and concurrent dynamic compilation is discussed. Then work related to architectural and micro-architectural performance modelling is reviewed, and in particular, approaches aimed at speeding up execution and simulation performance.

Chapter 4 investigates our concurrent and parallel dynamic compilation approach in the context of a dynamic binary translator using region based dynamic code discovery. Adaptive heuristics for selecting and scheduling units of compilation that adjust automatically to dynamic compilation workloads and the underlying hardware architecture are presented. The proposed

dynamic compilation scheme allows a significant reduction of compilation latency and a rigorous evaluation and analysis demonstrates its robustness, efficiency and good performance. This chapter is based on the work published in [3, 16, 66].

Chapter 5 applies dynamic compilation in the context of architectural and micro-architectural simulation. This exploration shows how runtime information can be exploited to generate optimised code modelling micro-architectural components. Again, the presence and complexity of additional dynamically generated code has a negative impact on compilation latency providing the perfect application scenario for the proposed dynamic compilation scheme. This is the first application of concurrent and parallel dynamic compilation to micro-architectural performance modelling resulting in significant speedups over state-of-the-art hardware and software solutions. This chapter is based on the work published in [14, 15].

Chapter 6 finally concludes this thesis by summarising the contributions, providing a critical analysis of this work and discussing future work.

1.5 Summary

This chapter has introduced this thesis, outlining the key problems and challenges. It advocates the use of a scalable, region-based, concurrent and parallel dynamic compilation scheme based on the parallel task farm design pattern to speedup dynamic compilation and exploit contemporary multi-core architectures. The key contributions of this work have been listed and an outline of the thesis described. The next chapter provides a short introduction to the dynamic compilation background used throughout this thesis.

2

Dynamic Compilation Background

This chapter presents a short overview of dynamic compilation techniques. First, a classification of dynamic compilation approaches is given. Then, the structure of a typical dynamic compiler is outlined, establishing common terminology. Finally, after demonstrating approaches for deciding when to compile program regions, the focus is directed to handling and scheduling of dynamic compilation workloads.

2.1 Introduction

Approaches to dynamic compilation differ in several aspects, including the degree of transparency, the extent and scope of dynamic compilation, and the encoding of the program representation. On the highest level, dynamic compilation systems can be divided into transparent and nontransparent systems. In a transparent system the executable program representation is not specially prepared for dynamic compilation or optimisation, and may execute with or without a dynamic compilation stage. In contrast, nontransparent approaches to dynamic compilation rely on staged runtime specialisation techniques and try to prepare for dynamic compilation as much as possible at static compilation time. Figure 2.1 shows a classification of various transparent and non-transparent dynamic compilation approaches as presented in [1].

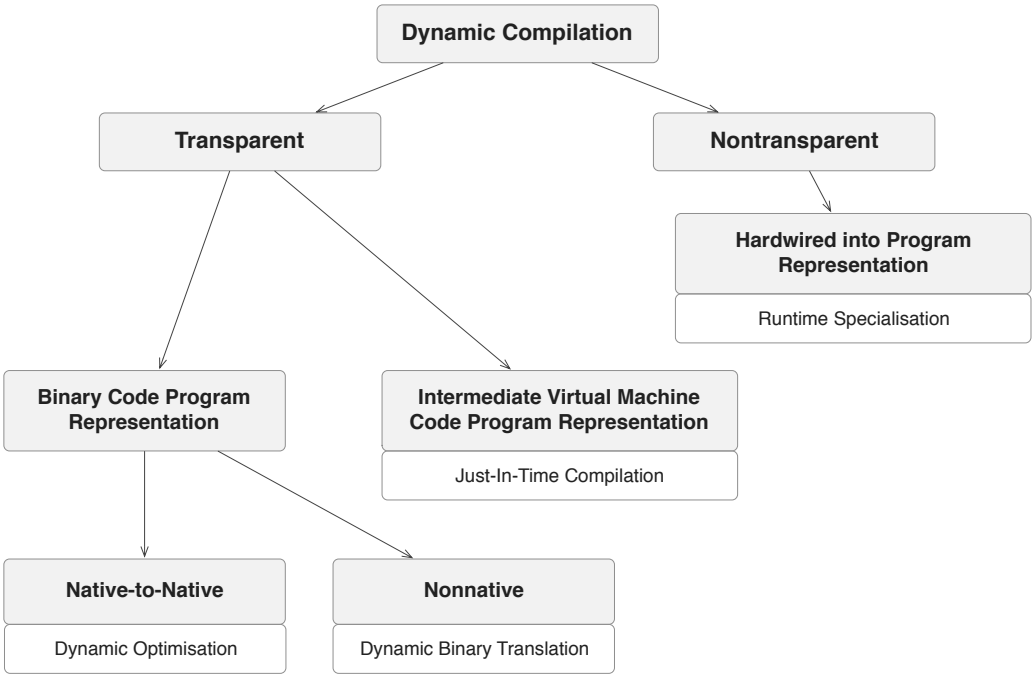

Figure 2.1: Classification of Dynamic Compilation approaches [1].

Transparent dynamic compilation systems can be divided into systems that operate on binary code and systems that operate on intermediate platform-independent program representations. A dynamic compiler for binary code program representations starts out with a loaded fully executable binary. In one scenario the dynamic compiler acts as a dynamic optimiser performing native-to-native optimising transformations based on runtime information such as runtime control and data flow values [9]. In the other scenario the loaded input binary is in nonnative format and dynamic compilation is used to retarget the code to a different host architecture. This process is referred to as dynamic binary translation and may also include optimisations [35,91,102].

Just-In-Time (Jit) compilers [26, 60, 67, 82] present a different class of transparent dynamic compilers. Their input is code in an intermediate, platform-independent representation that targets a virtual machine. The Jit compiler serves as an enhancement to the virtual machine improving target application runtime performance. This is done by compiling the intermediate input program to native code at runtime instead of executing it in an interpreter or emulator. Typically, semantic information is attached to the code, such as symbol tables or constant pools, which facilitates the compilation.

In the nontransparent dynamic compilation approach the dynamic compilation stage is integrated explicitly, i.e. hardwired, into the target program by earlier static compilation stages. The static compiler works together with the dynamic compiler by delaying certain parts of compilation until runtime to improve performance. The dynamic compilation agent compiled into the executable fills and links in prepared code templates for delayed compilation regions. Several techniques have been developed to perform runtime specialisation of a program in this manner [33, 46, 70, 85].

Runtime specialisation techniques are tightly integrated with the static compiler, whereas transparent dynamic compilation techniques are generally independent of the static compiler. Transparent dynamic compilation, however, still benefits from static information passed down by a static compiler such as a symbol table or other annotations [62]. While nontransparent dynamic compilation is mentioned in this Chapter for the sake of completeness, the work in this thesis focuses on transparent dynamic compilation.

2.2 Code Discovery

Identifying executable code is a precursor of dynamic compilation. The program representation determines whether executable code can be discovered statically or must be discovered dynamically as the program executes.

2.2.1 Static Code Discovery

For program representations where instructions can be statically separated from data, such as Java byte-code, Common Intermediate Language, C++, and JavaScript, executable code can be discovered statically. This means that a program can be compiled in its entirety before execution using only static program information. In the context of dynamic compilation this is known as ahead-of-time compilation [67].

2.2.2 Dynamic Code Discovery

For binary program representations in the form of hardware machine instructions "the identification of executable code [...] is equivalent to the Halting Problem and is therefore unsolvable in general" [55].

Consider the following example: A sequence of instructions to be translated contains an indirect jump instruction. The target of the jump instruction is held in a register that is assigned at runtime. In general it is impossible to determine the register contents statically. In addition, there is no guarantee that the locations immediately following the jump instruction contain valid instructions, as data can be interspersed with code in binary program representations.

Conventional binary program representations contain variable-length instructions, register indirect jumps, pads to align instructions, and data interspersed with instructions. For this reason executable code discovery is performed dynamically, discovering executable code incrementally as the program reaches them [93].

2.2.3 Hybrid Code Discovery

Even when static code discovery and therefore ahead-of-time compilation is possible, state-of-the-art execution environments [45, 60, 67, 82] combine static program knowledge with dynamically discovered program knowledge to enable optimisations that can not be inferred statically. For example, a dynamic compiler for an object oriented language may decide to de-virtualise a polymorphic call site as an optimisation, if at runtime it only observes monomorphic calls. If such an assumption is invalidated in the future, e.g. by loading new classes, the dynamically compiled code must be deoptimised and invalidated. Deoptimisation is the process of interrupting program execution while in optimised code and resuming it in an interpreter or unoptimised version. Dynamic deoptimisation was pioneered in Self [54] to allow for source-level debugging of globally optimised code. We use this technique to handle self-modifying dynamically compiled binary code (see Section 4.4).

2.3 Dynamic Compilation

Dynamic compilation or just-in-time (Jit) compilation is a means to improve runtime efficiency of programs. To the end user a dynamic compiler should behave like a black box taking a computer program representation as its input, and producing a dynamically translated and optimised version at runtime. This chapter deals with the structure of a dynamic compiler, its internal components, and their interaction. Figure 2.2 depicts a typical infrastructure found in most modern dynamic compilation systems:

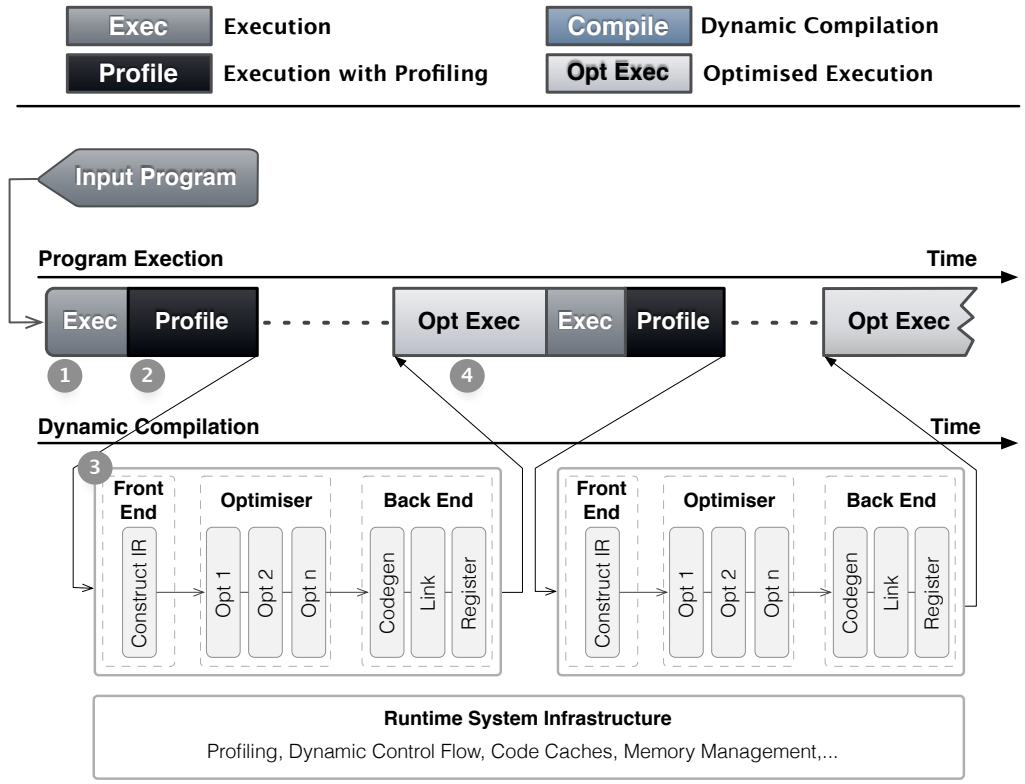

Figure 2.2: Structure of a typical Dynamic Compilation System.

- Initially the input program is executed or interpreted ① until a frequently executed program region is detected by exceeding an execution threshold (e.g. method execution count).
- Subsequently execution continues with profiling ② enabled. During profiling runtime information such as dynamic control flow and runtime type information is recorded.
- The recorded profiling data gathered from the currently running workload is used to drive the dynamic compiler ③ and focus optimisations on frequently executed parts of the program.
- After a frequently executed program region has been dynamically compiled and optimised it is linked and registered and program execution continues by running the optimised version ④.

The "Infrastructure" box at the bottom of Figure 2.2 highlights the importance of choosing efficient data structures and algorithms since those

choices greatly impact the performance, resource usage, and complexity of a dynamic compilation system.

2.3.1 Deciding When to Compile

Dynamic compilation schemes are based on compilation units reaching threshold frequencies of execution to determine when to trigger compilation. There exist two approaches to determine when threshold frequencies are exceeded, namely sampling-based and instrumentation-based profiling [95].

On the one hand a sampling-based profiling [107] approach keeps track of program regions where the application spends most of its time by periodically sampling program threads, thereby identifying which code regions are currently executed. At each sampling period a frequency counter associated with each region is incremented. On the other hand instrumentation based profiling instruments the target program by dynamically generating code for collecting specified counters from target code regions. After a program region counter update has reached its threshold, dynamic compilation kicks in and typically generates an optimised version without instrumentation code to minimise the performance impact.

In Section 4.2 of this thesis we propose a novel scheme to determine when execution frequencies are exceeded. It is a hybrid between sampling- and instrumentation-based profiling schemes. Initially execution starts with instrumentation based profiling enabled during emulation until a trace interval expires. The end of a trace interval constitutes a sampling point where all code regions profiled during that interval are analysed, and regions exceeding a certain execution threshold are dispatched for dynamic compilation. Code regions that have already been dynamically compiled or dispatched for dynamic compilation are not profiled any longer.

2.3.2 Handling Dynamic Compilation Workload

Given a set of selected compilation units for dynamic compilation, there exist several strategies aimed at handling dynamic compilation workload. Some state-of-the-art approaches [9, 12, 45, 77, 106] are blocking and wait for dynamic compilation of a frequently executed program region to finish before continuing execution (① in Figure 2.3) in the optimised version.

Another possible approach to dynamic compilation [39] is to split dynamic

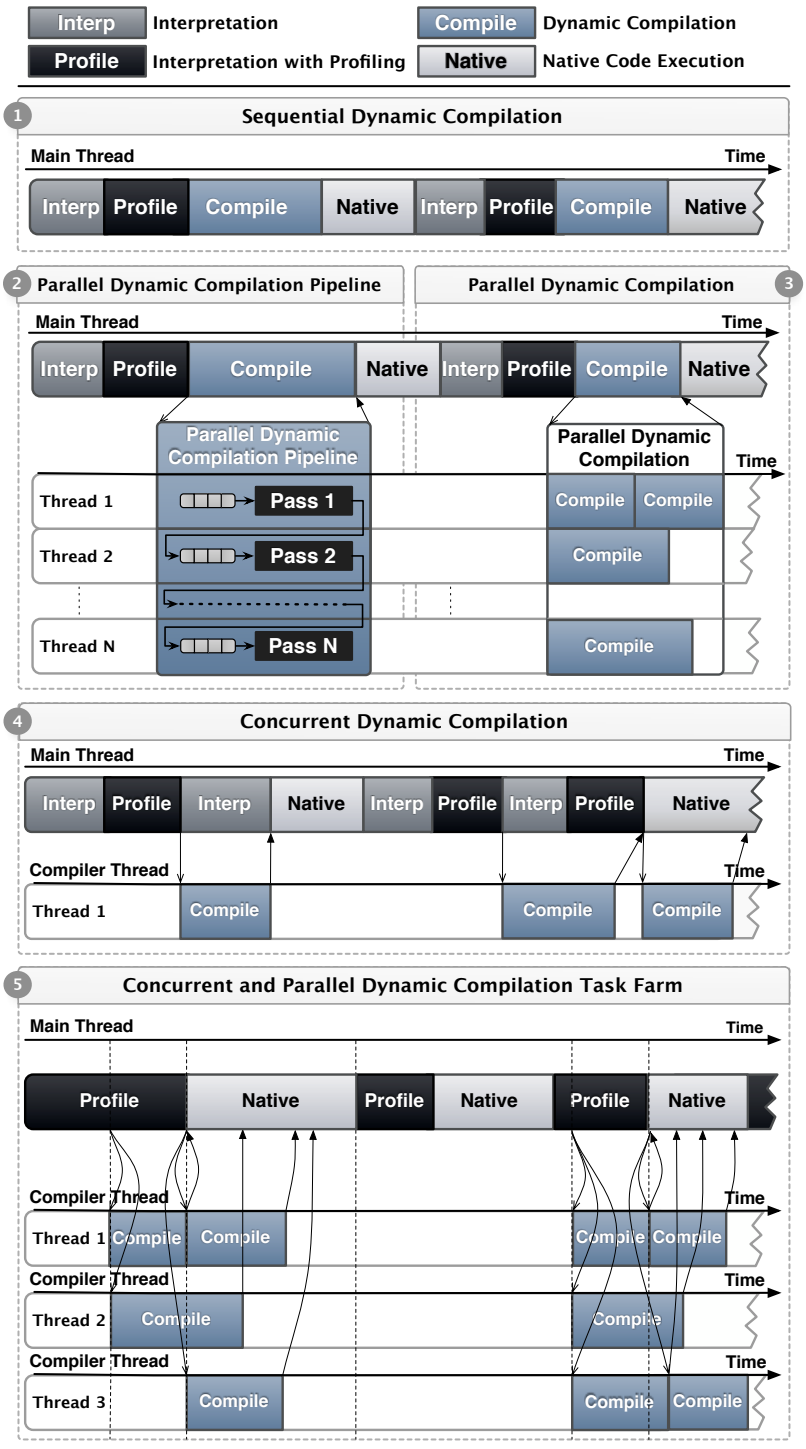

Figure 2.3: Dynamic compilation strategies.

compilation into multiple, parallel pipeline stages operating at instruction granularity (② in Figure 2.3). While the compiler pipeline can – in principle – be parallelised, synchronisation of up to 19 pipeline stages for every compiled instruction adds significantly to the overall compilation time while execution of the target program does not make any progress. This results in consistent slowdowns over all benchmarks in [39]. The approach in ③ is implemented by Oracle's JRockit Java VM [67]. It keeps synchronous dynamic compilation and optimisation requests in a queue. The queue is consumed by one or more dynamic compilation threads, depending on system configuration.

The HotSpot Jvm [82] and [47] are examples of systems that implement a concurrent dynamic compilation strategy by running the dynamic compiler in *one* separate helper thread whilst the master thread continues to interpret code (④ in Figure 2.3).

We build on this approach and extend it to run *several* dynamic compilers in a parallel task farm whilst target program execution continues (see Section 4.3). We do this to further hide dynamic compilation cost and switch to native execution of hot program regions even sooner and for longer (⑤ in Figure 2.3) than current state-of-the-art approaches.

The objective of our concurrent and dynamic compilation strategy ⑤ outlined in Figure 2.3 is translating frequently executed program regions into native code faster when compared to e.g. ①②③④ [39, 40, 47, 67, 77, 82, 106]. Figure 2.3 also illustrates that concurrent task parallel dynamic compilation ⑤ reduces the time until native code execution starts in comparison to current approaches ①②③④.

2.3.3 Scheduling Dynamic Compilation Workload

In any kind of dynamic compilation environment it is paramount to translate hot code regions as fast as possible into highly efficient native code. A good strategy for scheduling dynamic compilation tasks is needed when one or more code regions are pending dynamic compilation.

For sequential dynamic compilation systems (see ① in Figure 2.3) such as the V8 JavaScript engine [45], frequently executed program regions are not compiled immediately when execution thresholds are exceeded. Instead, they are marked for dynamic compilation at the function level, thereby delaying dynamic compilation or optimisation until the marked function is entered again. An advantage of using function entries as safe points for dynamic compilation and optimisation is that it simplifies the design of the runtime

system. A disadvantage of this design is that compilation is unnecessarily delayed for functions with hot loops.

Oracle's JRockit Java VM [67] (see ③ in Figure 2.3) keeps synchronous dynamic code generation requests in a code generation queue, and dynamic optimisation requests in an optimisation queue. The optimisation queue runs at a lower priority than the code generation queue as its work is not strictly necessary for code execution, but just for code performance. "Also, an optimisation request usually takes orders of magnitude longer than a standard code generation request to execute, trading compile time for efficient code" [67]. Approaches based on one concurrent dynamic compilation thread (see ④ in Figure 2.3) can schedule dynamic compilation workload by adapting dynamic compiler thread priorities [64].

In Section 4.3.4 we present a novel approach to adaptively handle dynamic compilation workloads by prioritising the most recently and frequently executed program regions first. The key idea behind our scheduling scheme is to consider temperature and recency of regions in order to prioritise hot regions that are being executed right now and to dynamically compile and optimise them first.

2.4 Summary

This chapter presented a short overview of dynamic compilation techniques. After providing a classification of dynamic compilation approaches the structure of a typical dynamic compiler was outlined to establish common terminology. Finally, dynamic compilation strategies for handling and scheduling dynamic compilation workloads have been presented together with the key contributions of this thesis. The next Chapter discusses related work.

3 Related Work

This chapter presents prior research relevant to this thesis. First, state of the art applications of dynamic compilation in the context of dynamic binary translation and optimisation are presented. Then, trace- and region-based dynamic compilation in the context of virtual execution environments and language runtimes are outlined. Finally, related work in the field of architectural and micro-architectural performance modelling exploiting dynamic compilation to achieve execution speedup is presented.

3.1 Dynamic Binary Translation and Optimisation

Dynamic binary translation (DBT) techniques are used to overcome the lack of flexibility and performance inherent in statically-compiled interpreters, emulators, and simulators. Among the main uses of DBT are cross-platform virtualisation for the migration of legacy applications to different hardware platforms. Prominent examples are DEC FX!32 [27], HP ARIES [115], QEMU [12], and solutions based on Transitive's QuickTransit technology such as Apple Rosetta and IBM POWERVM LX86 [94].

Dec Fx!32 [27] combines emulation and dynamic binary translation to provide both fast and transparent execution of x86 Cisc binaries on the Alpha Risc processor. The first time an x86 application is run, the entire application is emulated. Whilst transparently running the application, the emulator generates an execution profile that describes the application's execution history. The profile shows which parts of the application were heavily used and which parts are less important or rarely used. Later, after the application exits, the profile data directs the background optimiser to generate native Alpha code as replacement for all the frequently executed procedures. When the application is executed again, the native Alpha code is used and the application executes much faster.

Dynamo [9] is a dynamic optimisation system, implemented entirely in software. Its operation is transparent: no preparatory compiler phase or programmer assistance is required, and even legacy native binaries can be dynamically optimised by Dynamo. It focuses its efforts on optimisation opportunities that tend to manifest only at runtime. Our approach to concurrent and parallel dynamic compilation is also fully transparent. However, contrary to Dynamo, dynamic compilation runs concurrently with interpretation and is not on the critical execution path. Additionally, we enable parallel dynamic compilation by extracting independent compilation units encoding dynamic control flow during code discovery, further reducing dynamic compilation latency.

Qemu [12] is a fast emulator using an original dynamic translator. Each target instruction is divided into a simple sequence of micro-operations, the set of micro-operations having been pre-compiled offline into an object file. During emulation the code generator accesses the object file and concatenates micro-operations to form a host function that emulates the target instructions within a block. Compared to Qemu our dynamic compilation infrastructure is concurrent and parallel (i.e. non-blocking) and uses regions of multiple basic blocks instead of single basic blocks as compilation units. Regions expose more optimisation opportunities to the dynamic compiler by exploiting runtime control flow information (see Section 4.3.1).

Other current and emerging uses of Dbt include, but are not limited to, generation of cycle-accurate architecture simulators [14, 37], dynamic instrumentation [51], program analysis, cache modelling, and workload characterisation [72], software security [113], and transparent software support for heterogeneous embedded platforms [32].

3.2 Trace- and Region-based Dynamic Compilation

Tracing is a well established technique for dynamic profile guided optimisation of native binaries. Bala et al. [9] introduced tracing as a method for runtime optimisation of native program binaries in their Dynamo system. They use backward branch targets as candidates for the start of a trace, but do not attempt to capture traces of loops. Zaleski et al. [114] use Dynamo-like tracing in order to achieve inlining, indirect jump elimination, and other optimisations for Java. Their primary goal was to build an interpreter that could be extended to a tracing VM.

Whaley [108] uses partial method compilation to reduce the granularity of compilation to the sub-method level. His system uses profile information to detect never or rarely executed parts of a method and to ignore these during compilation. If such a part is reached at a later stage, execution continues in the interpreter. Compilation still starts at the beginning of a method. Similarly, Suganuma et al. [96] propose region-based compilation to overcome the limitations of method-based compilation. They use heuristics and profiles to identify and eliminate rarely executed sections of code, but rely on expensive runtime code instrumentation for trace identification. Our dynamic code discovery and incremental region construction scheme also avoids dispatching never or rarely executed regions of code using very low profiling overhead (see Section 4.2.1). In contrast to Whaley [108] and Suganuma et al. [96] we hide dynamic compilation latency by partitioning regions (see Section 4.2.2) in a way that enables effective parallel dynamic compilation.

Gal et al. [40, 41] propose an approach to building dynamic compilers in which no control flow graph (Cfg) is ever constructed, and no source code level compilation units such as methods are used. Instead, they use runtime profiling to detect frequently executed cyclic code paths in the program. The compiler then generates code for dynamically recorded code traces along these paths. It assembles these traces dynamically into a trace tree, a tree-like data-structure, that covers frequently executed (and thus compilation worthy) code paths through hot code regions. Trace trees can suffer from the problem of code explosion when many control-flow paths are present in a loop, causing them to grow to very large sizes due to excessive tail duplication as outlined in [11]. Compared to Gal et al. [40, 41] our system constructs a dynamic control flow graph during code discovery and profiling resulting in a compact program representation.

To solve the problem of excessive tail duplication Bebenita et al. [11] propose trace-regions as a data-structure for tracing in their implementation of Hotpath, a trace-based Java JIT compiler in the Maxine VM. Trace-regions are an extension to trace trees as they can include join nodes instead of using tail duplication. Locations where trace recording can be enabled, so called anchors, are determined statically during byte-code verification, and trace regions are restricted to method boundaries. In contrast, our approach does not rely on statically determined anchors for tracing and trace regions are not confined to method boundaries.

3.3 Parallel and Concurrent Dynamic Compilation

Dynamic compilation has a long history dating back to the 1960s [7]. The possibility of reducing the overhead of dynamic compilation by decoupling the JIT compiler from the main execution loop and executing it in a separate thread has been suggested by several researchers, e.g. [47, 63, 103].

Krintz et al. [63] pushed JIT compilation into the background while interpreted execution continued. Only single compiler and execution threads were employed and hot method detection was performed by offline profiling. More aggressive background compilation is exploited in [103] to choose the best way to compile a program on an embedded device as its battery level changes. Again, only single execution and compiler threads are used. Kulkarni et al. [64] dynamically increase the priority of its compilation thread to increase compiler throughput. Their technique is useful when the number of application threads is greater than the number of physical cores.

Some approaches [25, 39] attempted to exploit pipeline parallelism in the JIT compiler. However, pipelining of the JIT compiler has significant drawbacks. First, compiler stages are typically not well balanced and the overall throughput is limited by the slowest pipeline stage – this is often the front-end or IR generation stage. Second, unlike method based compilers, trace-based JIT compilers operate on relatively small translation units in order to reduce the compilation overhead to a bare minimum [40]. Small translation units and long compilation pipelines, however, increase the relative synchronisation costs between pipeline stages and, again, limit the achievable compiler throughput. Third, compilation pipelines are static and do not scale with the available task parallelism in inherently independent translation units.

Work by [87] pioneered the concept of parallel and concurrent JIT compilation workers to speed up dynamic binary translation, but suffers from a number of flaws. First, rather than taking a region- or trace-based compilation approach entire pages are translated – this is unnecessarily wasteful in a time-critical JIT compilation environment. Second, there are no provisions for a dynamic work scheduling scheme that prioritises compilation of hot code pages – this may defer compilation of critical code and lower overall efficiency. Third, JIT compilers reside in separate processes on remote machines – this significantly increases the communication overhead and limits its scalability. This last point is critical, as results shown in [87] are based solely on CPU time of the main simulation process rather than the more relevant wall clock time that includes CPU time, I/O time and communication channel delay.

Bruening et al. [20, 21] explore dynamic compilation in multi-threaded execution environments. In their work, the dynamic compilation of traces is performed on the execution thread, rather than asynchronously in the background as in our case, resulting in progress on that thread stalling during the compilation. Their compilation units are dynamically recorded linear traces, rather than regions. In multi-threaded execution environments Bruening et al. [20,21] consider the benefits of sharing linear traces in the context of multi-thread application. We build on this work and extend the concept of sharing linear trace compilation units to sharing region based compilation units to reduce dynamic compilation workload. Additionally, we propose novel and truly scalable dynamic code discovery approach for multi-threaded execution environments (see Section 4.5).

Ha et al. [47] use a tracing JavaScript interpreter together with a single background compiler thread to dynamically compile frequently executed traces. The transitions from interpreted execution to native execution are managed without locks by attaching a "Compiled State Variable" (CSV) to each trace anchor. We also build on this particular use of a state variable indicating the translation state of dynamic compilation units allowing for lock-free synchronisation between dynamic compilation threads and the execution loop.

Wimmer et al. [109] note that tracing enables simple phase change detection by comparing the ratio of side exits taken to the time spent in the trace itself. Traces can be discarded and recompiled when a phase change is detected. Their work uses a global trace cache and permits only one background compiler thread. Inoue et al. [57] implement a trace-based compiler retrofitted from a method-based compiler in the mixed-mode IBM J9/TR

JVM. Only one background compilation thread is used for JIT trace compilation.

Oracle's JROCKIT Java VM [67] is a commercial virtual machine without an interpreter and relies on total JIT compilation. All Java methods are compiled to native code immediately when they are first encountered. The advantage is that no interpreter has to be implemented, but the disadvantage is that compile time becomes an important factor in the total runtime. Execution cannot continue until the compilation of the currently executed method has finished, causing potentially large pause times when a large method is executed for the first time. For multi-threaded applications this problem is only exacerbated. For this reason JROCKIT uses a work queue based appraoch to keep track of dynamic compilation workloads, and enables the usage of several parallel dynamic compilation workers in an attempt to reduce excessive dynamic compilation latency for multi-threaded applications. The use of parallel dynamic compilation in JROCKIT, however, is only beneficial for multi-threaded applications. It does not resolve the problem that dynamic compilation is blocking, i.e. execution can not continue while the currently executed method is in translation. In contrast, our dynamic compilation system is non-blocking (i.e. concurrent) and capable of successfully exploiting parallel dynamic compilation even for single threaded applications (see Chapter 4).

3.4 Architectural Performance Modelling

Instruction set simulators (ISS) provide a platform on which experimental instruction set architectures can be tested, and new compilers and applications may be developed and verified. They help to reduce the overall development time for new microprocessors by allowing concurrent engineering during the design phase. This is especially important for embedded system-on-chip (SOC) designs, where processors may be extended to support specific applications. However, increasing size and complexity of embedded applications challenge current ISS technology and there is an increasing need for fast ISS technology to keep up with performance demands of real world applications.

To speed up instruction set simulation Mills et al. [76] employ in-line macro expansion in a statically-compiled simulator and demonstrate their system to run up to three times faster than an interpretive simulator. Target code is statically translated to host machine code which is then executed directly within a switch statement. The disadvantage of this approach however

is that applications containing self-modifying code (e.g. operating system, dynamic compilers) can not be simulated.

The Mimic simulator [74] simulates Ibm System/370 instructions on the Ibm Rt Pc and translates groups of target basic blocks into host instructions. Shade [31] and Embra [68] use Dbt with translation caching techniques in order to increase simulation speeds. The Ultra-fast Instruction Set Simulator [116] improves the performance of statically-compiled simulation by using low-level binary translation techniques to take full advantage of the host architecture.

The SimIcs full system simulator by Magnusson et al. [73] translates target machine-code instructions into an intermediate format before interpretation. They have developed a specification language to encode various aspects of the target instruction-set architecture, from which a simulation kernel based on threaded-interpretation is generated automatically. The Instruction Set Compiled Simulation (Ic-Cs) simulator [88] was designed to be a high performance and flexible functional simulator. To achieve this the time-consuming instruction decode process is performed during the compile stage, whilst interpretation is enabled at simulation time. Just-In-Time Cache Compiled Simulation (Jit-Ccs) [19] executes and caches pre-compiled instruction-operation functions for each function fetched.

More recent approaches using dynamic binary translation in instruction set simulation are presented in [18, 59, 87, 101]. Apart from different target platforms these approaches differ in the granularity of translation units (basic blocks *vs.* pages or Cfg regions) and their Jit code generation target language (Ansi-C *vs.* Llvm Ir).

The commercial xIss simulator [99] employs Dbt technology and targets the same ARCompact™ Isa that has been used in this paper. It achieves simulation speeds of 200+ Mips. In contrast, our dynamic binary translator operates at 500+ Mips in functional simulation mode.

3.5 Micro-architectural Performance Modelling

Dynamic binary translation combines interpretive and compiled simulation techniques in order to maintain high speed, observability and flexibility. Common to all approaches in the previous Section is that they implement

functional or instruction accurate instruction set simulation. They do not provide a detailed micro-architectural performance model as achieving accurate state, and even more so micro-architectural observability, remains in tension with high speed simulation.

A dynamic binary translation approach to architectural simulation has been introduced in [24]. The PowerPC Isa is dynamically mapped onto Pisa in order to take advantage of the underlying SimpleScalar [23] timing model. While this approach enables hardware design space exploration it does not provide a faithful performance model for any actual PowerPC implementation. Relevant to our work is an architectural and microarchitectural performance modeling approach based on a variant of the functional model/timing model partitioning implemented in systems such as Ramp [105] and Fast [28, 29]. This kind of microarchitectural performance modeling uses Fpga's to model microarchitectural components and thereby provide a timing model, while functional simulation is done using an Iss. The functional model sequentially executes the program, generating a functional path instruction trace, and pipes that stream to the Fpga based timing model. While this approach is accurate and faithfully models the simulated target microarchitecture, it is not a software only simulation strategy and relies on Fpga based simulation.

Most relevant to our work is the performance estimation approach in the HySim hybrid simulation environment [42–44, 61]. HySim merges native host execution with detailed Iss. For this, an application is partitioned and operation cost annotations are introduced to a low-level intermediate representation (Ir). HySim "imitates" the operation of an optimising compiler and applies generic code transformations that are expected to be applied in the actual compiler targeting the simulation platform. Furthermore, calls to stub functions are inserted that handle access to data managed within the Iss where the cache model is located. No executable for the target platform is ever generated and, hence, the simulated code is only an approximation of what the actual target compiler would generate. No detailed pipeline model is maintained. Hence, cost annotations do not reflect actual instruction latencies and dependencies between instructions, but assume fixed average instruction latencies. Even for relatively simple, non-superscalar processors this assumption does not hold. HySim has been evaluated against an Iss that does not implement a detailed pipeline model. Hence, accuracy figures reported in e.g. [43] only refer to how close performance estimates come to those obtained by this Iss, but it is unclear if these figures accurately reflect the actual target platform. A similar hybrid approach targeting software energy estimation has been proposed earlier in [78, 79].

Statistical performance estimation methodologies such as SimPoint and Smarts have been proposed in [48, 112]. The approaches are potentially very fast, but require preprocessing (SimPoint) of an application and do not accurately model the micro-architecture (Smarts, SimPoint). Unlike our accurate pipeline modelling this introduces a statistical error that cannot be entirely avoided.

Machine learning based performance models have been proposed in [10,17, 81] and, more recently, more mature approaches have been presented in [38, 86]. After initial training these performance estimation methodologies can achieve very high simulation rates that are only limited by the speed of faster, functional simulators. Similar to Smarts and SimPoint, however, these approaches suffer from inherent statistical errors and the reliable detection of statistical outliers is still an unsolved problem.

3.6 Summary

This chapter has presented prior work related to dynamic binary translation, concurrent and parallel dynamic compilation, and architectural and micro-architectural performance modelling. The work in the field of concurrent and parallel dynamic compilation has mostly been limited to program representations that allow static code discovery. Approaches targeted at execution environments requiring dynamic code discovery use the dynamic compiler on the critical execution path, effectively stoping emulation until the translation of the current application hotspot is finished. In contrast, the work presented in Chapter 4 of this thesis introduces an innovative parallel task farming strategy for truly concurrent dynamic compilation. It is designed to execute concurrently with the target program and effectively reduces dynamic compilation latency by exploiting the concept of parallel compilation. It has been used for execution environments that rely on dynamic code discovery, but can equally well be applied to execution environments and program representations that allow static code discovery.

Prior work on architectural performance modelling uses dynamic binary translation to speed up simulation but does not use a concurrent and parallel dynamic compiler to speed up the translation process. Exploiting dynamic compilation in the context of micro-architectural performance modelling has not been previously explored and provides an ideal application scenario for the dynamic compilation infrastructure presented in this thesis. In Chapter 5 we demonstrate how to use our concurrent and parallel dynamic compiler for

architectural and micro-architectural performance modelling, to achieve simulation speeds that surpass hardware based (FPGA) simulation approaches.

4

Concurrent and Parallel Dynamic Compilation

This chapter starts by demonstrating the workload a concurrent dynamic compiler has to handle in the context of a dynamic binary translator. This motivates the idea of concurrent and parallel dynamic compilation. Then, the selection of program regions for dynamic compilation is outlined. Next, a system architecture for concurrent and parallel dynamic compilation is proposed followed by a description of adaptive heuristics for selecting and scheduling units of compilation. Finally, a rigorous evaluation and analysis of the proposed scheme demonstrates its robustness, efficiency, and good performance.

Efficient dynamic binary translation (DBT) heavily relies on dynamic compilation for the translation of target machine instructions to host machine instructions. Although dynamically compiled code generally runs much faster than interpreted code, dynamic compilation incurs an additional overhead. For this reason, only the most frequently executed code regions are translated to native code whereas less frequently executed code is still interpreted. Using a single-threaded execution model (see ① in Figure 2.3), the interpreter pauses until the dynamic compiler has translated its assigned compilation unit and the generated native code is executed directly. However, it has been noted earlier [7, 47, 63, 84, 100] that program execution does not need to be paused to permit compilation, as a dynamic compiler can

operate in a separate thread while the program executes concurrently (see ④ in Figure 2.3). This decoupled or concurrent execution of the dynamic compiler increases complexity of the DBT, but is very effective in hiding the compilation latency – especially if the dynamic compiler can run on a separate processor.

The main contribution of this thesis is to demonstrate how to effectively reduce dynamic compilation overhead and speedup execution by performing concurrent and parallel dynamic compilation, exploiting the broad proliferation of multi-core processors. The key idea is to detect independent, large translation units in execution traces and to farm out work to multiple, parallel dynamic compilation workers. To ensure that the latest and most frequently executed code regions are compiled first, we apply a priority queue based dynamic work scheduling strategy where the most recent, hottest traces are given highest priority.

This novel, concurrent and parallel dynamic compilation methodology is integrated into the LLVM-based state-of-the-art ARCSIM [3, 14–16, 59, 101] DBT implementing the ARCompact ISA and its performance is evaluated using three benchmark suites: EEMBC [36], BIOPERF [8] and SPEC CPU2006 [52]. The concurrent and parallel dynamic compilation approach yields an average speedup of 1.17 across all 61 benchmarks – and up to 2.08 for individual benchmarks – over decoupled dynamic compilation using only a single compilation worker thread on a standard quad-core Intel Xeon host platform. At the same time the proposed scheme is robust and never results in a slowdown even for very short- and long-running applications.

4.1 Motivating Example

Consider the full-system simulation of a Linux OS configured and built for the ARC 700 processor family (RISC ISA). The complete boot-up sequence, the automated execution of a set of commands emulating interactive user input at the console, followed by the full shut-down sequence is simulated on a standard quad-core Intel Xeon machine. This example includes rare events such as boot-up and shut-down comparable to the initialisation phase in an application, but it also includes very frequent events occurring after the boot-up sequence during interactive user mode. In a full-system OS simulation there are frequent calls to interrupt service routines that must be simulated. Our generalised tracing approach can easily identify frequently executed regions including interrupt service routines that would otherwise be

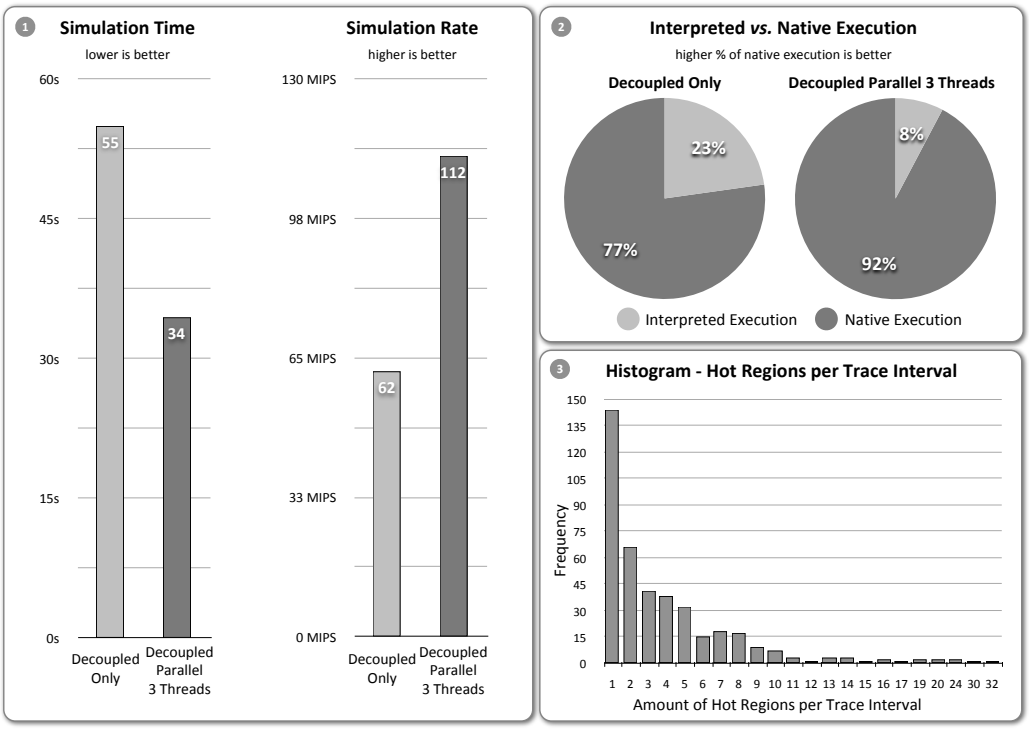

Figure 4.1: Full-system Linux OS simulation - comparison of ① simulation *time* in seconds, *rate* in MIPS, and ② interpreted *vs.* natively executed instructions in %, between *decoupled only* and *decoupled parallel* dynamic compilation using three compilation worker threads. Histogram ③ demonstrates how often more than one *hot* region per trace interval is found.

missed if we would restrict tracing to loop or function boundaries.

Our dynamic binary translator speeds up the simulation by identifying and translating hot dynamically discovered regions to native `x86` code during simulation. As simulation runs concurrently with dynamic compilation we continue to discover and dispatch more hot regions to the dynamic compiler. In fact, it is possible that dynamic compilation of the previous regions has not yet been completed by the time new regions are discovered. In this case, work is distributed over several dynamic compilation threads. To ensure that the most profitable regions are compiled first, a dynamic work scheduling strategy that dynamically prioritises compilation tasks according to their temperature and recency is implemented.

For the purpose of this motivating example a simulation using only one decoupled dynamic compiler thread (current state-of-the-art) is compared with a simulation using multiple decoupled dynamic compiler threads in parallel

(see ⑤ in Figure 2.3). In Chart ① of Figure 4.1 the overall simulation time in seconds and the simulation rate in MIPS is compared for both approaches. The new approach using three decoupled dynamic compilers in parallel completes the previously outlined sample application 21 seconds earlier. This results in an improvement of 38% and, thus, achieves a speedup of **1.6** when compared to using only one decoupled dynamic compiler. The overall simulation rate (in MIPS) improves from 62 to 112 MIPS by using the new approach. As several dynamic compilers work on hot regions in parallel, native translations are available much earlier than using a single decoupled dynamic compiler (see ④ in Figure 2.3), leading to a substantial increase from 77% to 92% of natively executed target instructions (see Chart ② of Figure 4.1).

The obvious question to ask is where does the speedup come from? Histogram ③ in Figure 4.1 shows how often a certain amount of frequently executed regions is found per trace interval. The fact that 65% of the time there are at least two or more hot regions discovered per interval clearly demonstrates the benefits of having more than one dynamic compiler available on today's multicore machines. Even if only one hot region per interval is discovered, the dynamic compilation of the previous hot region might not have finished. Having several dynamic compilers that can already start working on the newly discovered hot regions before others have finished helps to effectively hide most of the dynamic compilation latency (see Box ① in Figure 4.2).

4.2 Trace Based Dynamic Region Detection

Typically dynamic compilation schemes [39, 40, 47, 77, 80, 106] are based on compilation units reaching threshold frequencies of execution to determine when to trigger compilation. The approach proposed in this thesis is based around the concept of trace intervals during which regions are dynamically discovered and execution frequencies are maintained. An interval based scheme was selected as it generates more uniformly sized compilation units, resulting in better and more predictable load balance between compilation and execution. It also avoids the problem of profiling counters eventually overflowing the size of their count field if the trace interval length is chosen appropriately, eliminating the need for overflow checks. In the next sections the process of dynamically constructing and partitioning regions is explained.

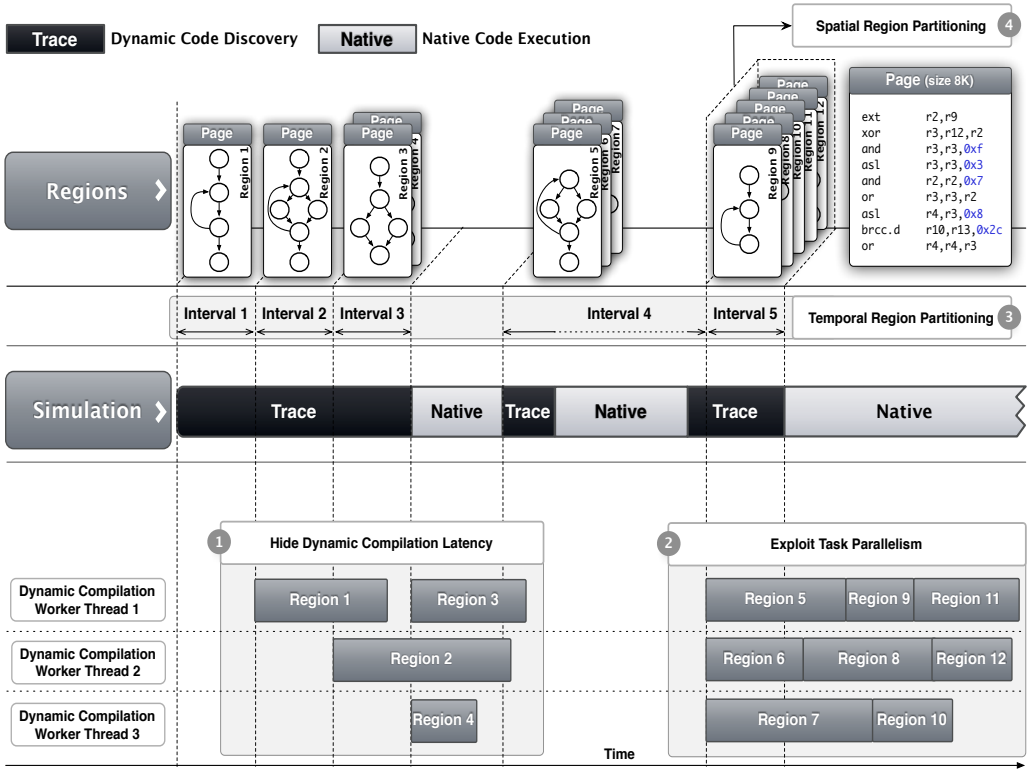

Figure 4.2: Concurrent dynamic compilation of hot regions - ② demonstrates how to exploit *task parallelism* by dynamically compiling independent hot regions in parallel. ① shows how several concurrent dynamic compilation threads can effectively hide most of the dynamic compilation latency by overlapping compilation of hot regions. ③ and ④ highlight interval based *temporal*, and memory space *spatial* region partitioning.

4.2.1 Tracing Regions

The term region refers to a compilation unit, which results from dynamically collecting executed code from the original program but excluding all never or rarely executed portions [96]. This enables a repartitioning of the original program into desirable compilation units [71]. Box ③ in Figure 4.2 shows the partitioning of simulation time into trace intervals during which dynamic code discovery is performed. The length of a trace interval is user definable and determines the number of interpreted instructions. During a trace interval a region is constructed incrementally from a sequence of interpreted basic blocks as demonstrated in Figure 4.3. The back-bone data structure behind a region is a directed graph whose vertices and edges represent dynamically discovered basic blocks and control flow edges.

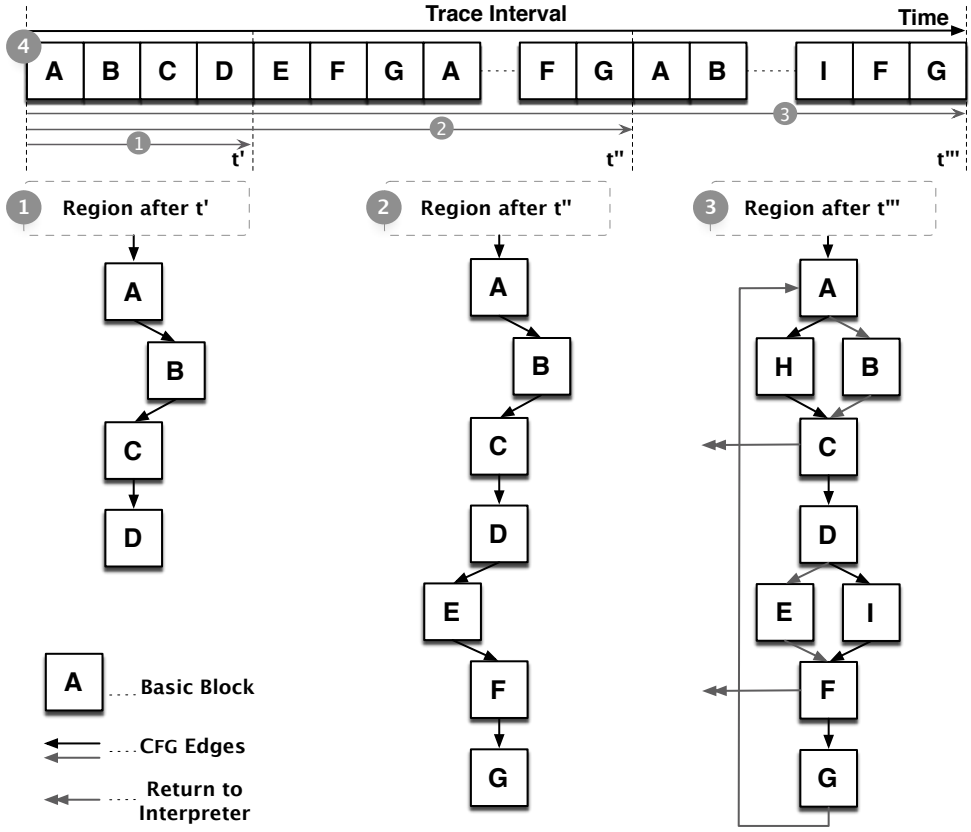

Figure 4.3: Incremental ①②③ region construction from sequence of interpreted basic blocks ④.

At the end of a trace interval a traced region is scheduled for dynamic compilation if it is considered to be hot. The temperature of a region is defined as the sum of the execution frequencies of its constituent basic blocks. As an optimisation, counters for interpreted basic blocks of regions that have already been dispatched for dynamic compilation but not yet compiled, are not maintained. It has already been determined that such basic blocks are worthy of compilation so further profiling is not necessary.

4.2.2 Partitioning Regions

The absence of static structural information (e.g. classes, functions, modules) in binary program representations poses a problem for region partitioning. A good strategy is needed to efficiently partition dynamically discovered code into uniformly sized and compilation worthy regions to enable quick transla-

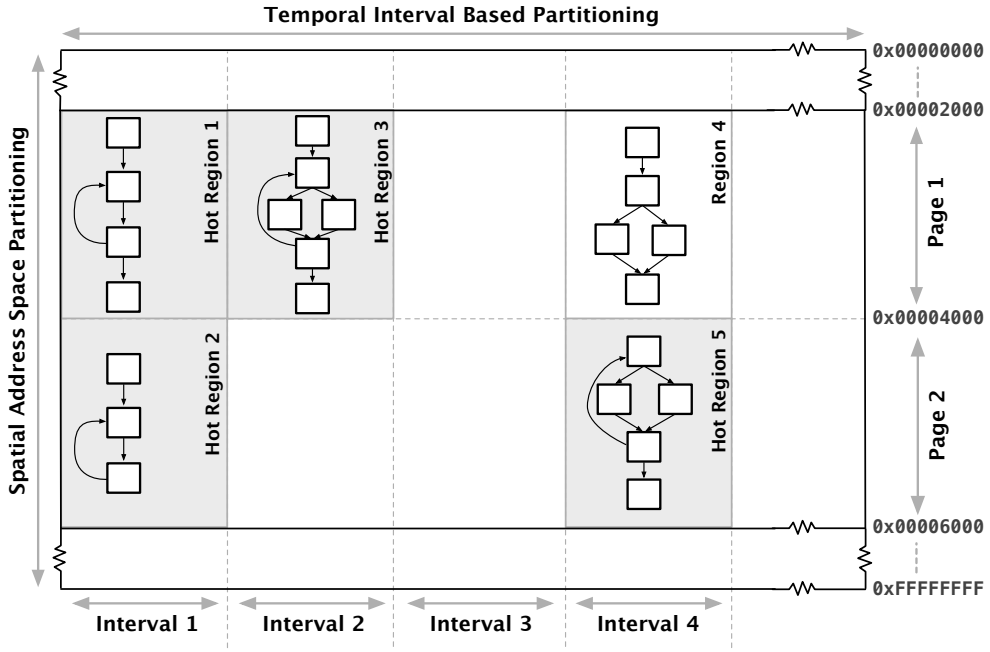

Figure 4.4: Spatial and temporal partitioning of regions. Regions 1, 2, 3, and 5 are hot regions selected for dynamic compilation.

tions and exploit parallel dynamic compilation. To solve this a novel incremental partitioning scheme that is simple, yet very effective at partitioning dynamically discovered code into regions is proposed.

Interval based tracing results in a *temporal* partitioning (see ③ in Figure 4.2 and Figure 4.4) where a region is bounded by the start and end of a trace interval. In addition, the notion of *spatial* partitioning where a region is bounded by address ranges within the target address space is introduced (see ④ in Figure 4.2 and Figure 4.4). Spatial partitioning is necessary to enable efficient architectural simulation of target memory components such as a memory management unit [93]. For this reason regions are partitioned at target page boundaries (see Figure 4.2 and 4.4). Spatial partitioning is inherent in non binary program representations in the form of classes, methods, or functions. Together, temporal and spatial region partitioning are the key to structured and more uniformly sized dynamic compilation units.

Tracing regions is light-weight because only basic block entry points (i.e. target memory addresses) are recorded as vertices, and pairs of source and target entry points as edges in the per-page regions. At the end of a trace interval all regions that have been touched during that interval are analysed.

For the hottest regions compilation units are constructed and dispatched to a translation priority queue for dynamic compilation (see Figure 4.5).

4.3 Concurrent and Parallel Dynamic Compilation

The main contribution of this thesis is the demonstration, through practical implementation, of the effectiveness of a concurrent and parallel dynamic compilation task farm based on the LLVM [69] compiler infrastructure. By parallelising dynamic compilation we effectively hide dynamic compilation latency and exploit the available parallelism exposed by our generalised region construction scheme. Executing parallel dynamic compilation concurrently with target program execution benefits highly interactive applications as it further reduces pause times. The following sections outline our dynamic compilation flow focusing on the most relevant components of our concurrent and parallel dynamic compilation system.

4.3.1 Dynamic Compilation Flow

The interpreter loop is responsible for tracing regions during dynamic code discovery. At the end of a trace interval recorded regions are analysed and frequently executed regions are dispatched to a translation priority queue for dynamic compilation before interpretation continues (see ①② in Figure 4.5). The translation priority queue is a concurrent and shared data-structure acting as the main interface between the interpreter loop and multiple dynamic compilation workers (see ③ in Figure 4.5). Dynamic compilation worker threads dequeue regions and compile them to native code in parallel (see ④ in Figure 4.5).

After dequeing a region a dynamic compilation worker generates its corresponding LLVM intermediate representation (IR). During LLVM IR generation a very effective control flow optimisation called software indirect jump prediction via inline caching is applied [34,53,93]. This optimisation is based on the observation that an indirect jump target never or very seldom changes. So for each indirect jump site an inline cache including one or more cached lookups is generated to speed up the transition from the simulated program counter to its corresponding dynamically compiled block (see ① in Figure 4.6).

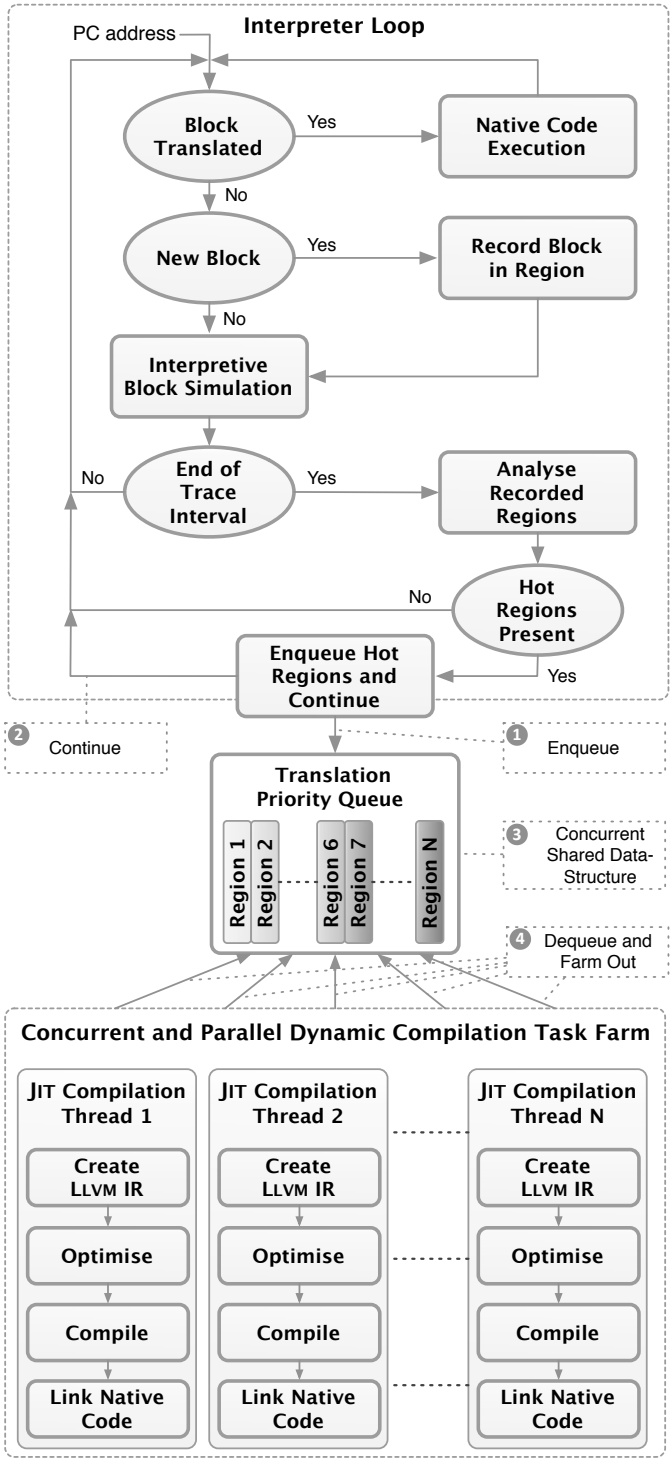

Figure 4.5: Concurrent and parallel dynamic compilation flow.

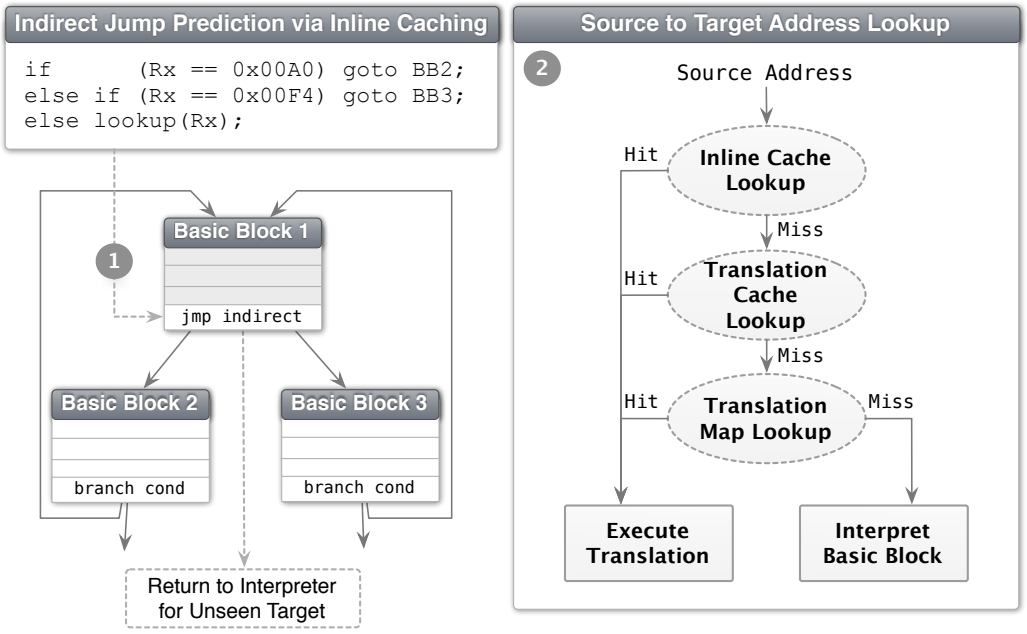

Figure 4.6: ① Demonstrates an example for software indirect branch prediction via inline caching. ② Shows a flow diagram for source to target program counter address translation.

In case of an inline cache miss a second level translation cache is used to find the translated target block. If the lookup in the second level translation cache fails we search for the translated target block in a map abstract data type, using a balanced binary search tree as its backbone data-structure. This search will fail for transitions to target basic blocks that have not been seen or translated yet, causing a return of control to the interpreter (see ② in Figure 4.6).

After applying software indirect jump prediction via inline caching, a sequence of LLVM optimisation passes such as instruction combining, jump threading, commutative expression reassociation, constant propagation, followed by corresponding clean up passes (i.e. CFG simplification pass) are applied to further optimise the generated LLVM IR. Finally we use LLVM's `ExecutionEngine` to dynamically compile and link the generated LLVM IR code. Each dynamic compilation worker thread receives its own private `ExecutionEngine` instance upon thread creation to minimise the amount of synchronisation. The next section outlines how the interpreter loop is notified about the availability of native region translations.

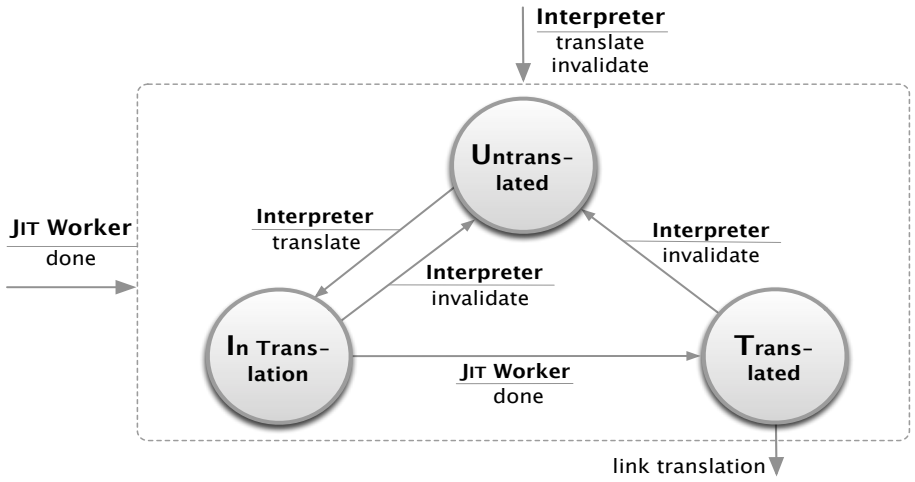

Figure 4.7: Basic block translation state transitions.

4.3.2 Interpretive - Native Execution Mode Transition

In our concurrent and parallel dynamic compiler the interpreter uses a translation state variable to mark unseen basic blocks for dynamic compilation and to determine if a native translation for a block already exists. While regions are dynamically compiled, the interpreter continues target program execution. As soon as dynamic compilation of a previously dispatched region is finished, the dynamic compilation worker immediately modifies the translation states of its constituent basic blocks. The interpreter is thereby notified about available translations and can immediately transition from interpretive to native execution mode on a basic block boundary. A basic block can be in one of the following three translation states:

- UNTRANSLATED - When a basic block is seen for the first time, its state is initialised to *untranslated* by the interpreter. Self-modifying code (see Section 4.4) affecting basic blocks that are either *in translation* or *translated* causes the state of a basic block to be reverted to *untranslated* by the interpreter.
- IN TRANSLATION - When a basic block is dispatched for dynamic compilation the interpreter changes its state from *untranslated* to *in translation*.
- TRANSLATED - When a dynamic compilation worker is finished with the translation of a region it changes the state of its constituent basic blocks from *in translation* to *translated*.

```
1  // Container class for frequently executed region
2  //
3  class CompilationUnit {
4    friend struct PrioritizeCompilationUnits;
5    uint64_t Timestamp_;
6    uint64_t ExecFreq_;
7    // ...
8  };
9  // Comparator for CompilationUnits
10 //
11 struct PrioritizeCompilationUnits {
12  bool operator() (CompilationUnit const * x,
13                   CompilationUnit const * y ) {
14    if (x->Timestamp_ == y->Timestamp_)
15      return x->ExecFreq_  < y->ExecFreq_;
16    else
17      return x->Timestamp_ < y->Timestamp_;
18  }
19 };
20 // Priority Queue abstract data type
21 //
22 std::priority_queue<CompilationUnit*,
23                     std::vector<CompilationUnit*>,
24                     PrioritizeCompilationUnits>     queue;
```

Listing 4.1: Dynamic work scheduling based on recency and frequency.

This particular use of a state variable indicating the translation state of a basic block allows for lock-free reading of basic block translation states by the interpreter loop (see Figure 4.7). Synchronisation is only required for invalidations caused by self-modifying code (see Section 4.4). This approach is similar to the compiled state variable concept implemented in [47].

4.3.3 Scheduling Compilation Workload

In any kind of dynamic compilation environment it is paramount to translate hot code regions as fast as possible into highly efficient native code. Thus having discovered multiple regions across several trace intervals we would like to dynamically compile the most recently and frequently executed regions first.

To efficiently implement a dynamic work scheduling scheme based on both recency and frequency of interpreted regions, a priority queue is used as an abstract data type using a binary heap as the backbone data-structure.

```
1  // Compute hotspot Threshold based on queue size.
2  //
3  uint64_t THRESHOLD = ComputeHotspotThreshold(queue.size());
4
5  // Extract CompilationUnits from dynamically discovered
6  // Regions based on execution Threshold.
7  //
8  list<CompilationUnit*> COMPUNITS = Extract(REGIONS,
9                                             THRESHOLD);
10 // Dispatch CompilationUnits for compilation and continue
11 // interpretation of target program.
12 //
13 DispatchHotspotsForCompilation(COMPUNITS);
```

Listing 4.2: Adaptive hotspot threshold selection and compilation unit dispatch.

We chose a binary heap because of its worst case complexity of $O(log(n))$ for inserts and removals. Our sorting criteria insert the most frequently executed region from the most recent trace interval at the front of the priority queue (see Listing 4.1).

4.3.4 Adaptive Hotspot Threshold Selection

It is possible that the proposed region-selection and dispatch system can produce more tasks than the dynamic compilation workers can reasonably handle. Dynamic workload scheduling mitigates this problem by ensuring that the hottest and most recent regions are compiled first, leaving relatively colder and older tasks waiting until the important work has been completed. However, it would also be beneficial to reduce the number of tasks actually being dispatched to the translation queue in the event of an overloaded dynamic compilation task farm.

In order to avoid large amounts of waiting translation tasks, we implemented an adaptive hotspot threshold scheme. Initially, the hotspot threshold is set to a constant value based on the number of dynamic workers available – as the number of workers increases, the threshold can be set more aggressively. This threshold is then adjusted based on the priority queue's current length, where a longer queue raises the threshold at which new potential regions are considered to be hot enough (see Listing 4.2). The threshold can either be tied directly to the length of the queue, or a certain queue length can trigger an increase in the hotspot threshold.

4.4 Self-Referencing and Self-Modifying Code

There are cases where an application program either refers to itself by reading from its code regions, or attempts to modify itself by writing to its code regions. This presents a problem for dynamic binary translators when the code that is actually executed is translated code. In the context of concurrent and parallel dynamic compilation this problem is exacerbated by the fact that regions that are *in translation* can be invalidated by concurrently executing self-modifying code in the interpreter. It is important that self-referencing and self-modifying code is handeled accurately and efficiently as such code occurs frequently for certain application domains such as full system OS simulation.

The basis for the solution is the same for both, self-referencing and self-modifying code. In particular, an accurate memory image of program code must be maintained at all times and all load and store addresses in the translated version must be mapped into the simulated memory region, regardless of whether code or data is being addressed. "Consequently, the self-referencing code case is automatically implemented correctly" [93].

There are several solutions to handle self-modifying code [93]. One possible solution is to write-protect memory regions that contain code. This can be done by the runtime system via appropriate operating system specific calls. Any attempt to write to a page that contains regions of dynamically translated code results in a protection trap and the delivery of a signal to the runtime. Unfortunately this method has relatively high overhead [93], portability issues, and complicates the integration of the runtime into other tools that may wish to register custom signal handlers for memory protection traps. More importantly it is not applicable in our context as the systems we simulate have configurable page sizes that do not necessarily correspond to simulation host page sizes. Therefore we implement a portable yet efficient scheme using a combination of write barriers together with block based translation caches as proposed in [58].

The proposed scheme uses direct mapped block translation caches for read, write, and instruction memory operations. These caches serve two purposes: (1) to speedup the mapping from simulated addresses onto underlying host memory addresses, and (2) to enable efficient permission checking and self-modifying code detection. A block translation cache is indexed by simulated addresses and stores base pointers to blocks of host memory. See Figure 4.8 for an example of how the simulated memory address `0x00004002` is translated to the appropriate host memory location.

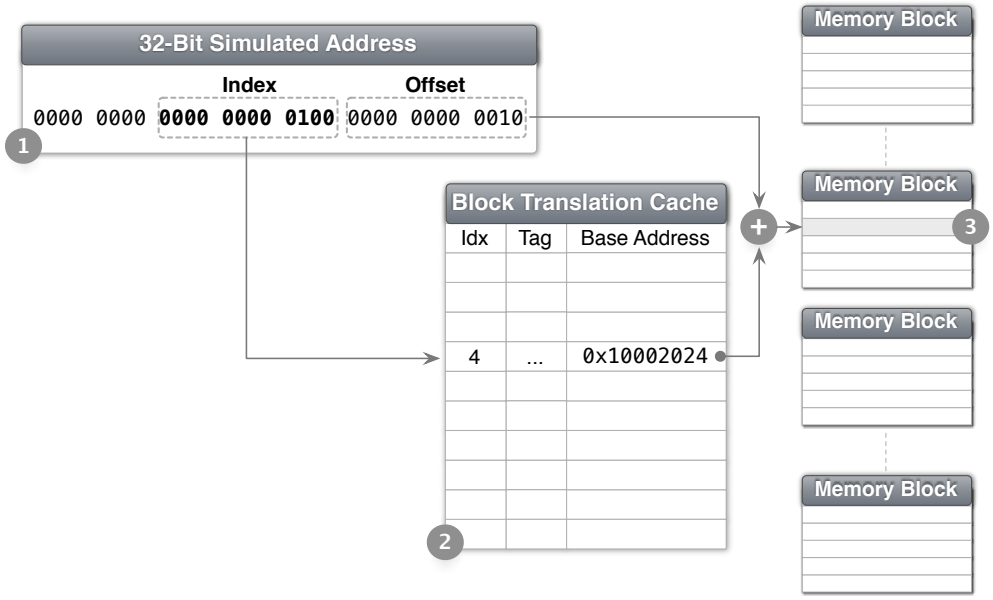

Figure 4.8: Mapping simulated address ① `0x00004002` onto host memory block entries ③ using a block translation cache ②.

When a write block translation cache miss occurs, the underlying host memory block is looked up. The resulting host memory block base address is entered into the write block translation cache. Before allowing the write to succeed the block translation cache containing instruction fetch addresses is checked to determine if it contains an entry for the same simulated write address. If that is the case a write to a block containing executable code has occurred and a further lookup is made to determine whether the write hit a code address for which dynamically compiled code exists. Writes to addresses containing dynamically compiled code cause an invalidation of the corresponding native translation. If a write from dynamically compiled code invalidates itself, the compiled code is deoptimised and invalidated.

This design is efficient as it is optimised for the common case where simulated address translations frequently hit into block translation caches. It also enables fast detection of writes to simulated memory regions that contain executable code in which case an additional lookup is performed to check if dynamically compiled code has been generated for the given simulated address.

4.5 Scaling for Multi-threaded Execution Environments

In single threaded execution environments frequently executed areas of code are identified for dynamic compilation and, therefore, native execution. When a single core is being simulated, discovering, dynamically compiling, and storing the natively translated code sections is straightforward. In the multi-core case, however, a number of issues arise about how individual simulator threads will cooperate in the compilation process. If and how they will share translated code sections, who will be responsible for performing the translations, and how the costs of any required synchronisations will be mitigated, are crucial questions that must be resolved to achieve high performance. In this section we demonstrate how to effectively scale our dynamic code discovery and concurrent and parallel dynamic compilation task farm for multi-threaded execution environments.

In multi-threaded execution environments that rely on dynamic code discovery, a naïve approach would operate on a single, global trace or region data structure, which would need to be protected from concurrent updates using an expensive locking mechanism. This approach has been taken in e.g. [57]. An alternative approach that avoids this problem is to maintain thread-private traces [41,49], i.e. for each application thread a separate trace data structure is maintained. While this approach does not suffer from excessive synchronisation cost, it introduces a new problem. Multiple threads, especially in data parallel applications, may produce nearly identical traces that differ only in thread-specific constants and accesses to thread-private variables. Clearly, this approach increases pressure on the dynamic compilation subsystem and does not scale.

In region-based multi-threaded execution environments using dynamic code discovery, no work has investigated different methods to parallelise the simulation environment. The work closest to ours is not region-based but trace-based. In [20] the benefits of sharing linear traces over keeping private traces has been considered. The compilation of traces is performed on the execution thread, rather than asynchronously in the background as in our case, resulting in progress on that thread stalling during dynamic compilation. In addition, because the compilation units are traces rather than regions, only one linear trace can exist per trace head.

As a consequence, if the next executed tail happens to be non representative of the general behaviour of the current thread, or indeed the many other

threads in the system, that poorly chosen trace will be forever attached to the trace head for all threads. In contrast, our system builds up regions of hot basic blocks. Each thread individually discovers which regions are important to it and shares identical regions with other threads. Our system permits threads to have overlapping regions between threads. In this way, each thread is not penalised by the occasional poor choices of other threads but benefits from sharing amongst threads which exhibit the same behaviour.

In this thesis, we propose a novel and scalable scheme for region-based dynamic compilation of multi-threaded applications. The key idea is to extend the thread-private region compilation model with the capability for sharing of regions between threads. Central to this idea is the generation of thread-agnostic regions, i.e. regions that do not contain thread-specific constants or data accesses, but are generic enough to be executed in the contexts of different threads. With these two features in place, we demonstrate that region sharing is both effective, and scalable.

4.5.1 Code Discovery in Multi-threaded Applications

Before we take a more detailed look at our approach to scaling region-based concurrent and parallel dynamic compilation for multi-threaded applications, we provide a comparison of state-of-the-art code discovery schemes to highlight our key concepts (see Figure 4.9). Recording of execution paths in terms of traces or regions is either triggered by detecting a special construct (e.g. loop header, method entry) [11, 40, 47, 57, 109], or always enabled when interpreting code [16].

Various backbone data structures have been suggested to capture dynamically discovered execution traces such as trace-trees [39], control-flow-graphs (CFG) of traced basic blocks [16], or hybrids between trace-trees and CFGs called trace-regions [11] or trace-graphs [49]. In general, dynamic code discovery approaches for multi-threaded execution environments can be categorized based on how they manipulate their underlying data-structures:

- *Global Recording Structure* - Most [11,39,40,47,49,57,109] trace-recording dynamic compilation systems use one shared global recording structure (① in Figure 4.9) to incrementally build a trace. This scheme works well for single-threaded execution environments but does not scale to multi-threaded applications as additional synchronisation is required when recording traces in parallel for multiple threads.

- *Local Recording Structures* - Another approach is to use private local recording structures for each traced thread (② in Figure 4.9). While this approach avoids synchronisation altogether, it causes threads executing data-parallel sections of code to independently trace nearly identical code paths and compile multiple thread-specific traces specialised for each thread.

Our approach builds on ② and extends it to compile thread-agnostic regions for data-parallel sections of code once and share the compiled region with all threads that traced the code region (see ③ in Figure 4.9). Consequently, the pressure on the underlying dynamic compiler is reduced and translations become available instantaneously to all threads that execute the same region as soon as the first of a number of identical regions has been compiled. We take the idea of region-based dynamic compilation for multi-threaded execution environments a step further. Using a lock free region recording strategy and a dynamic code generation approach enabling the sharing of compiled code for regions recorded by threads executing data-parallel sections of code, we demonstrate that our multi-threaded region-based dynamic compilation scheme is highly scalable.

4.5.2 Motivating Example

Consider the multi-threaded benchmark `water-spatial` from the SPLASH-2 benchmark suite, when executed with 128 threads. Of all the regions handled by the concurrent and parallel dynamic compilation subsystem, 79% were actually similar to previously requested regions (see ② in Figure 4.10). This shows that there is a large potential saving to be made in sharing regions between threads for data-parallel applications. We demonstrate that our approach leads to a speedup of 2.4x for this benchmark (see ① in Figure 4.10). This is made possible through the use of regions that can be executed by any thread – these are said to be *thread-agnostic* (see ② in Figure 4.11). These regions then allow us to develop a scheme where commonality between traced regions is identified as they are dispatched for dynamic compilation, allowing multiple threads to use the result of a region translation. This will reduce the amount of time threads have to continue executing in interpreted mode until a region is dynamically compiled. It enables earlier execution of native code and reduces the pressure on the dynamic compilation subsystem. As a result, the total dynamic compilation time for this benchmark is reduced by 73%.

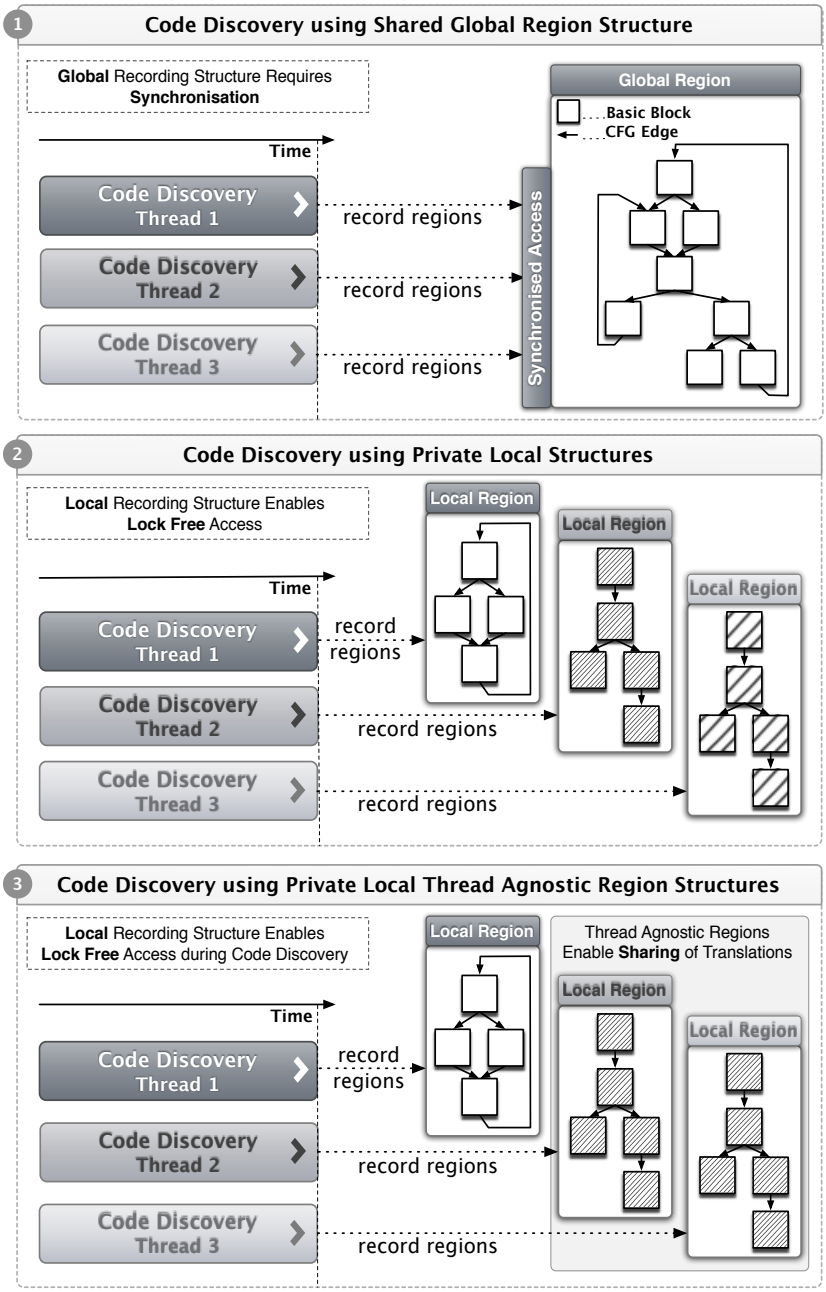

Figure 4.9: Code discovery approaches for multithreaded applications.

4.5.3 Thread Agnostic Code Generation

To enable the sharing of regions between threads, native code must access the thread state structure in a manner that will work for any thread that

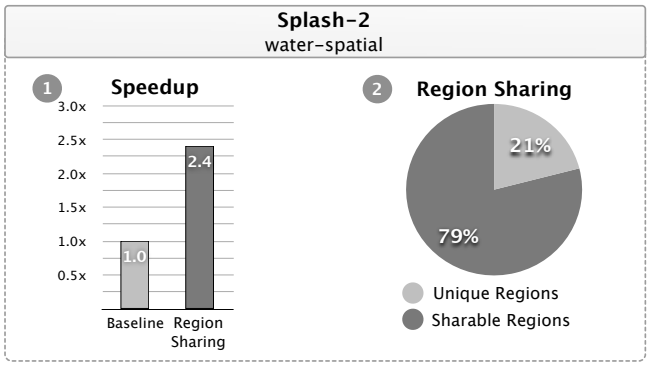

Figure 4.10: Splash-2 water-spatial benchmark ① demonstrating the achievable speedup using our novel region sharing optimisation, while ② shows the scope of region sharing for this benchmark.

may execute the compiled region. As this native code is generated at run time, the memory location of each thread state structure is known, and it would be obvious to reference the structure directly using the known, constant memory addresses. We call this a thread-specific region (see ① in Figure 4.11). This code generation scheme, however, prohibits region sharing between threads. Constant references to thread-private data make it impossible to reuse the translated region for any other thread other than the one it has been generated for,

Instead, we propose a scheme whereby the generated native code accesses the thread state structure indirectly through a base pointer. Each interpreter must then provide the base pointer to its own thread state structure when switching over to executing native code. We call these regions thread-agnostic (see ② in Figure 4.11). In this case, both threads can use the same region as thread-specific constants and thread-private variables are accessed via base pointer indirections.

It would be natural to assume that thread-agnostic regions are likely to be slower than thread-specific ones, due to the additional memory accesses and offset calculations. Surprisingly, we can see that the use of thread-agnostic regions is often faster than using thread-specific regions - a speedup of 1.09x on average. One would expect that having to obtain the thread state pointer and calculate an offset would be more expensive than simply accessing a constant known at runtime. However, this does not take into account the issue of code size. On the `x86` host architecture used throughout this thesis, an instruction that accesses a memory location using register + offset calculations should not require more than 4 bytes to encode. On the

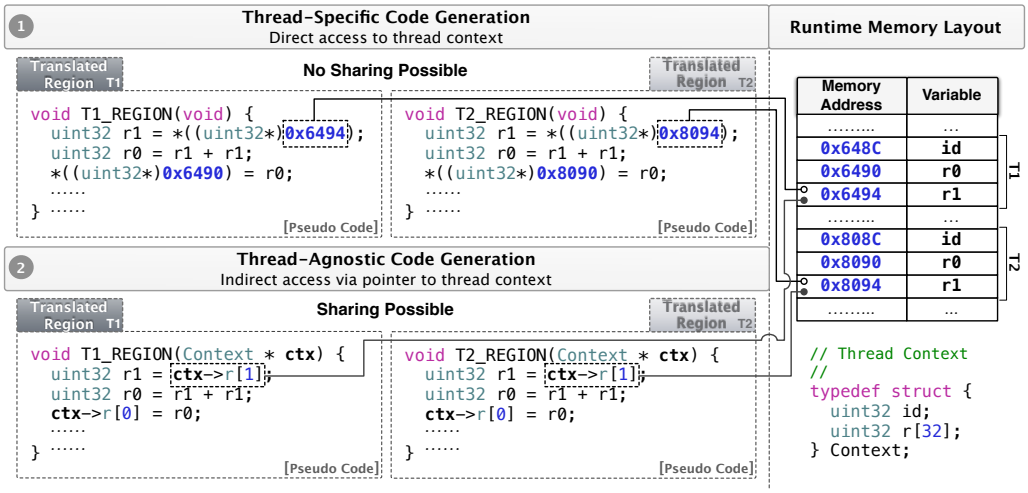

Figure 4.11: ① and ② highlight the key difference between *thread-specific* and *thread-agnostic* code generation, namely thread context independent code generation, *enabling* the sharing of dynamically compiled regions.

other hand, encoding a 32 or 64-bit immediate constant requires at least 4 or 8 bytes to encode the constant alone, ignoring the rest of the instruction.

We have observed that the use of thread-specific regions leads to an increase in overall code size, even if the number of instructions generated may decrease. This larger code may lead to slower execution, for instance, if sections of code can no longer reside completely in the simulation host instruction cache. These results show that the use of thread-agnostic regions actually results in faster execution of threads, even before enabling the sharing of regions between threads.

4.5.4 Inter-thread Region Sharing

As regions are dispatched to a translation priority queue for concurrent and parallel dynamic compilation, it would be beneficial to identify which regions cover identical code paths, and can therefore be shared between threads of execution. To quickly determine if two regions can be shared, a signature for each region is computed incrementally as it is constructed. The signature is the 32-bit result of a hash function applied to the physical addresses of all basic blocks in the trace. Only if two signatures are equal, a more expensive check for equality is to be performed to establish beyond doubt that the regions indeed cover the same code paths and rule out hash collisions.

A hash table is maintained alongside the translation priority queue. For a given key signature, this table stores all threads that are interested in the associated region that is currently waiting for dynamic compilation. Adding any region that matches signatures with a region already in the queue will result in this region being added to the hash table, instead of the priority queue itself. These regions are bundled in a manner which includes a reference to the requesting thread, so that dynamic compilation workers can update the requesting thread with the result of the compilation (see ① Figure 4.12).

Dynamic compilation worker threads continue to fetch regions from the queue. After dynamically compiling a region, the worker checks the hash table for all threads that are registered for the result of the compilation, allowing them to be updated with the native code generated from the region. See Figure 4.12 for an example demonstrating how regions are discovered, dispatched, and shared between threads in a multi-threaded execution environment. This technique has the dual effect of reducing the period many threads must wait between region dispatch and availability time of a translation. It also reduces the number of similar regions in the priority queue, thereby reducing the pressure on the underlying dynamic compilation subsystem.

4.5.5 Region Translation Caching

The previous section described how to share regions if a thread attempts to dispatch a region while a similar region is currently waiting for dynamic compilation. What if a particular thread dynamically constructs a hot region much later than other threads? In this particular case, there are no similar regions currently in translation, so a dynamic compilation worker would need to compile the region again.

One possible solution for this problem would be to register a translated region with all threads once it has been dynamically compiled. The drawback of this solution is that translations might be registered for threads that have not yet traced that specific code path, thereby adding complexity to the region recording interpreter loop. Furthermore, for task-parallel workloads most if not all translated regions cannot be shared and will only be used by one thread.

Instead we use a software cache for region translations and add a reference to each dynamically compiled region to this region translation cache. When a region is dispatched for dynamic compilation we first check if a translation

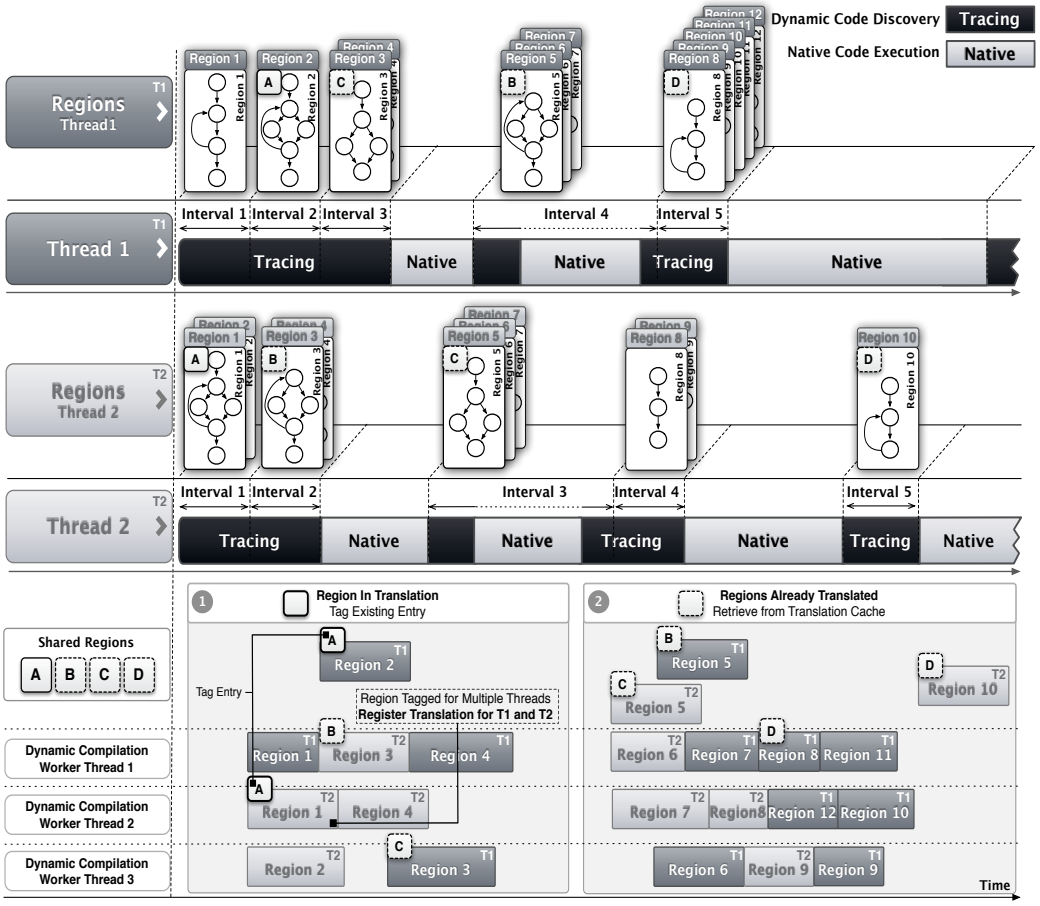

Figure 4.12: Region sharing for sample multi-threaded application - Frequently executed program regions from two threads `T1` and `T2` are recorded and dispatched for dynamic compilation. As soon as a region is compiled it is cached and its availability is registered with the thread responsible for its dispatching. ① Shows how `T1` records *Region 2* that is equal to a previously dispatched *Region 1* by `T2`. The previously dispatched region is still in translation, hence it is tagged to record the fact that an additional execution thread, namely`T1` is interested in its translation. ② When a thread records a new region that has already been translated by another thread, its translation is immediately retrieved from the translation cache, enabling almost instant availability of native code.

is already present in the region translation cache. If so, all native code generation is skipped, and the translation can be registered straight away for the thread that dispatched the region (see ② Figure 4.12). Region translation caching thereby removes redundant re-compilation from the critical path of execution for many threads, improving overall performance.

Caching and sharing of dynamically compiled regions in the contex of multi-threaded self-modifying code requires additional book keeping and infrastructure. In principle the detection and invalidation of self-modifying code remains as outlined in Section 4.4. Self-modifying code affecting translations present in the translation cache must be purged from the cache. Additionally other threads must be notified about invalidations due to self-modifying code to enable them to synchronise and update their thread local translation mappings, potentially causing deoptimisation of code executing in native mode, before any code modification can occur.

It is difficult to provide support for self-modifying multi-threaded code without a cost to efficiency. Fortunately multi-threaded self-modifying code is relatively uncommon and mainly restricted to language runtimes [4]. Therefore we have decided to implement a simple scheme that synchronises and halts all execution threads before modifying dynamically compiled code at runtime based on [97]. Further performance optimisations based on research done by [4, 97] are certainly possible and subject to future work.

4.6 Evaluation and Analysis of Results

We have evaluated our region-based concurrent and parallel dynamic compilation and work scheduling approach across more than 60 industry standard benchmarks, including BioPerf, Spec, Eembc, and CoreMark, from various domains. We also evaluate the performance benefits resulting from the use of region sharing to improve region-based concurrent and parallel dynamic compilation in multi-threaded execution environments using the Splash-2 [110] parallel benchmark suite. In this section we describe our experimental setup and methodology before we present and discuss our results.

4.6.1 Benchmarks and Experimental Setup

We have evaluated our region-based concurrent and parallel dynamic compilation approach using the BioPerf benchmark suite that comprises a comprehensive set of computationally-intensive life science applications [8]. It is well suited for evaluating our concurrent and paralle region-based dynamic compiler as it exhibits many different potential hotspots per application for most of its benchmarks. We also used the industry standard Eembc 1.1, and CoreMark [36] embedded benchmark suites. These benchmarks represent small and relatively short running applications with complex algorithmic ker-

Vendor & Model	DELL™ POWEREDGE™ 1950
Number CPUs	4 (quad-core)
Processor Type	Intel©Xeon™ processor E5430
Clock/FSB Frequency	2.66/1.33 GHz
L1-Cache	32K Instruction/Data caches
L2-Cache	12 MB
Main Memory	8 GB
Operating System	Scientific Linux 5.5 (64-bit)

Table 4.1: Simulation Host Configuration.

nels. An evaluation using SPEC CPU 2006 benchmarks [52] is included as they are widely used and considered to be representative of a broad spectrum of application domains.

To evaluate the performance benefits resulting from the use of region sharing to improve concurrent and parallel dynamic compilation in multi-threaded execution environments we use the SPLASH-2 benchmark suite [110]. SPLASH-2 is a set of 12 parallel benchmarks covering a range of application domains such as linear algebra, complex fluid dynamics and graphics rendering. In cases where both contiguous and non-contiguous versions of a benchmark are provided, we have used the contiguous version.

The BIOPERF benchmarks were run with "class-A" input data-sets available from the BIOPERF web site. The EEMBC 1.1 benchmarks were run for the default number of iterations and COREMARK was run for 1000 iterations. For practical reasons we used the largest possible data set for each of the SPEC CPU 2006 benchmarks such that simulation time does not become excessive.

Our main focus has been on simulation speedup by reducing the overall simulation time. Therefore we have measured the elapsed real time between invocation and termination of our simulator using the UNIX `time` command. We used the average elapsed wall clock time across 10 runs for each benchmark and configuration (i.e. interpreted-only, concurrent dynamic compilation, concurrent and parallel dynamic compilation) in order to calculate speedups. Additionally, we provide error bars for each benchmark result denoting the standard deviation to show how much variation there is between different program runs.

We use a strong and competitive baseline for our comparisons, namely a concurrent region-based dynamic compiler using one asynchronous thread

for dynamic compilation [47]. Relative to that baseline we plot the speedups achieved by our concurrent and parallel region-based dynamic compiler using three asynchronous dynamic compilation threads. Furthermore, we also present speedups (i.e. slowdowns) relative to our baseline when using interpreted simulation only (i.e. disabling region-based dynamic compilation).

All measurements were performed on a standard `x86` DELL™ POWEREDGE™ quad-core outlined in Table 4.1 under conditions of low system load. To evaluate the scalability of our approach, when adding more cores, we performed additional measurements on a parallel symmetric multiprocessing machine with 16 2.6 GHz AMD Opteron™ (AMD64e) processors running Scientific Linux 5.0 (see Section 4.6.3).

4.6.2 Speedup

Key Results

Our novel concurrent and parallel region-based dynamic compilation approach is *always* faster than the baseline decoupled dynamic compiler and achieves an average speedup of **1.38** equivalent to an average execution time reduction of **22.8**% for the BIOPERF benchmark suite. This corresponds directly to an average increase of **14.7**% in the number of natively executed instructions compared to the baseline.

For some benchmarks (e.g. `blastp`, `clustalw`) our proposed scheme is more than *twice* as fast as the baseline. This can be explained by the fact that 59% of the time we find more than one hot region per trace interval for `blastp`. From this it follows that `blastp` exhibits a large amount of task parallelism (see Box ② in Figure 4.2). `Clustalw` mainly benefits from hiding dynamic compilation latency by using several dynamic compilation worker threads (see Box ① in Figure 4.2).

Shorter running BIOPERF benchmarks perform particularly well with our scheme (e.g. `tcoffee`, `hmmsearch`, `clustalw`) because more dynamic compilation workers can deliver translations much quicker as they can split the workload (i.e. hide compilation latency). Especially for `hmmsearch` where the baseline decoupled dynamic compiler performs worse than the interpreted-only version, our scheme can significantly boost execution speed and reduce overall simulation time by 34.4%. Even for very long running BIOPERF benchmarks (e.g. `fasta-ssearch`, `promlk`, `hmmer-hmmpfam`, `ce`), where dynamic compilation time typically represents only a small fraction of the over-

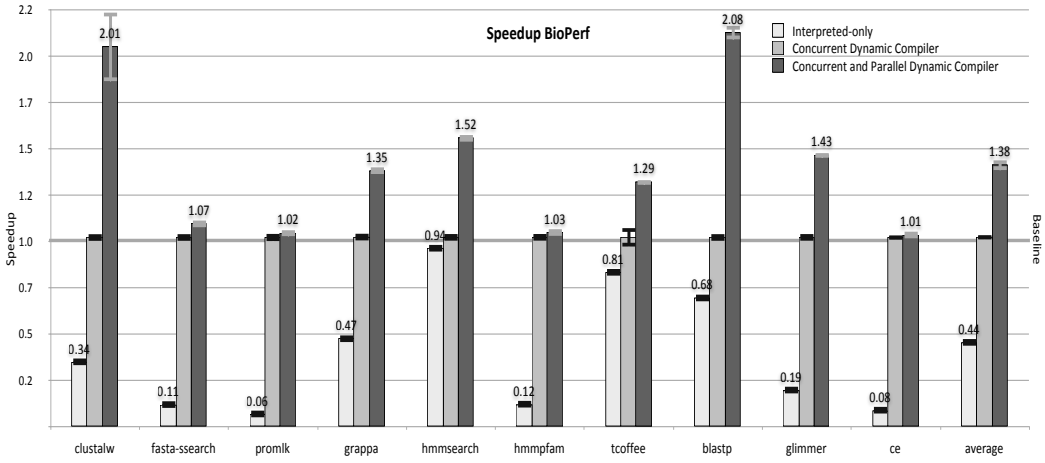

Figure 4.13: Speedups for BioPerf benchmark suite comparing (a) interpreted-only simulation, (b) simulation using a concurrent dynamic compiler, and (c) simulation using our novel, concurrent and parallel dynamic compiler with three dynamic compilation worker threads including dynamic work scheduling.

all execution time, our scheme achieves a reduction of execution times of up to 6.8%.

Worst-Case Scenarios

The BioPerf benchmark suite is well suited to show the efficacy of our concurrent and parallel region-based dynamic compiler. Additionally we also demonstrate its favorable impact on benchmarks where we would not expect to see significant speedups from our technique - so called *worst-case scenarios*.

For this analysis we have considered short running embedded benchmarks (Eembc, CoreMark) containing few application hotspots (i.e. algorithmic kernels). Some of these benchmarks are so short that interpreted only execution takes less than two seconds, leaving very little scope for improvement by a dynamic compiler. At the other end of the scale are very long running and Cpu intensive benchmarks (Spec Cpu 2006) where dynamic compilation time contributes only a marginal fraction to the overall execution time.

Across the Spec Cpu 2006 benchmarks our concurrent and parallel region-based dynamic compiler is *never* slower than the baseline and achieves an average speedup of **1.15**, corresponding to an average increase of **4.2**% in the number of natively executed instructions. The best speedups are achieved for `gcc` (2.04x), `xalancbmk` (1.47x), `povray` (1.2x), and `perlbench`

(1.18x). For `perlbench` and `gcc` the number of natively executed instructions improves by 19.5% and 38.0%, respectively, when using our concurrent and parallel region-based dynamic compiler. The `gcc` benchmark greatly benefits from our approach as it runs a compiler with many optimization flags enabled resulting in a multitude of application hotspots representing compilation phases.

`Xalancbmk` is one of the shorter running SPEC CPU benchmarks performing XML transformations. Due to its short runtime and abundance of application hotspots the code discovery and region construction overhead causes the baseline concurrent dynamic compiler to be slower than the interpreted-only version. Our concurrent and parallel dynamic compiler can easily recover this overhead resulting in a speedup of 1.47 when compared to the baseline, and a speedup of 1.21 when compared to interpreted-only simulation. `Povray` represents a long running benchmark where we achieve a speedup of 1.2. This is again due to an abundance of application hotspots across the runtime of the `povray` benchmark.

For the EEMBC and COREMARK benchmark suites our approach achieves an average speedup of 1.12 over the baseline, and an average improvement of 3.8% in the number of natively executed instructions. Small embedded benchmarks do not often benefit from task parallelism because they rarely exhibit more than one hot region per trace interval. The speedups are mostly due to the fact that dynamic compilation workers can already start working on newly discovered regions while previous regions are being translated by other workers. Also profiling and and region construction must be lightweight, causing only very little overhead, to enable speedups for small benchmarks like EEMBC and COREMARK.

For all benchmarks performing Fast Fourier Transforms (i.e. `aifftr01`, `aiifft01`, `fft00`) speedups ranging from 1.46 to 1.7 are achieved by our concurrent and parallel dynamic compiler. The bit manipulation (`bitmnp01`) and infinite impulse response filter (`iirflt01`) benchmarks also show speedups of 1.57 and 1.46 using our scheme. Three EEMBC benchmarks (`cacheb01`, `puwmod01`, `ospf`) yield very short runtimes using interpreted-only mode (i.e. below one second) causing a small slowdown of the baseline concurrent dynamic compiler and our concurrent and parallel dynamic compiler. This is entirely due to the fact that for very short benchmarks a dynamic compiler has almost no chance to speed up execution, actually causing a slight slowdown due to the overheads caused by profiling and dynamic compilation worker thread creation.

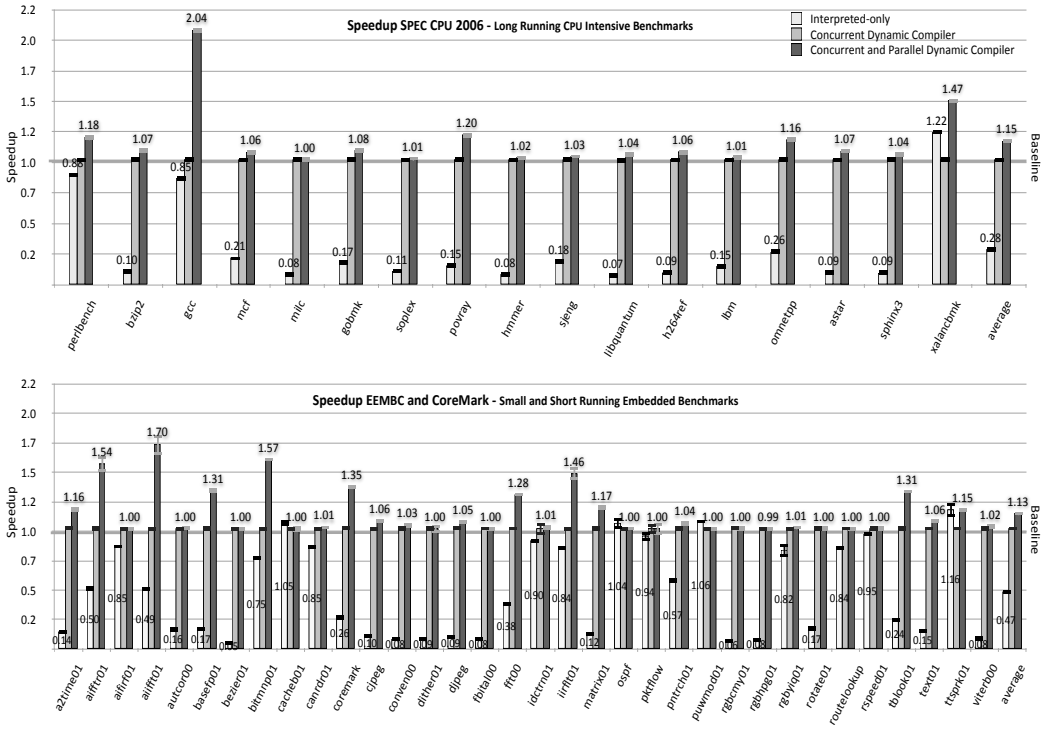

Figure 4.14: Speedups for SPEC CPU 2006, EEMBC and CoreMark benchmark suites comparing (a) interpreted-only simulation, (b) simulation using a concurrent dynamic compiler, and (c) simulation using our novel, concurrent and parallel dynamic compiler with three dynamic compilation worker threads including dynamic work scheduling.

We have also evaluated the impact of our novel dynamic work scheduling scheme in comparison to a simpler approach that schedules regions based on their order of creation for concurrent and parallel dynamic compilation. For BioPerf, we measured an average improvement of 8.9%, which is equivalent to an average speedup of 1.13. For the SPEC CPU 2006 benchmarks, the average improvement and speedup are 3.5% and 1.04, respectively.

4.6.3 Scalability

According to Amdahl's law we expected only marginal improvements with increasing numbers of dynamic compilation worker threads, so it is remarkable how well some benchmarks scale (see Figure 4.15) on a 16-core system. For `blastp` ① from BioPerf the maximum speedup of 3.1 is reached with 14 dynamic compilation workers. The `gcc` ④ and `perlbench` ⑤ benchmarks

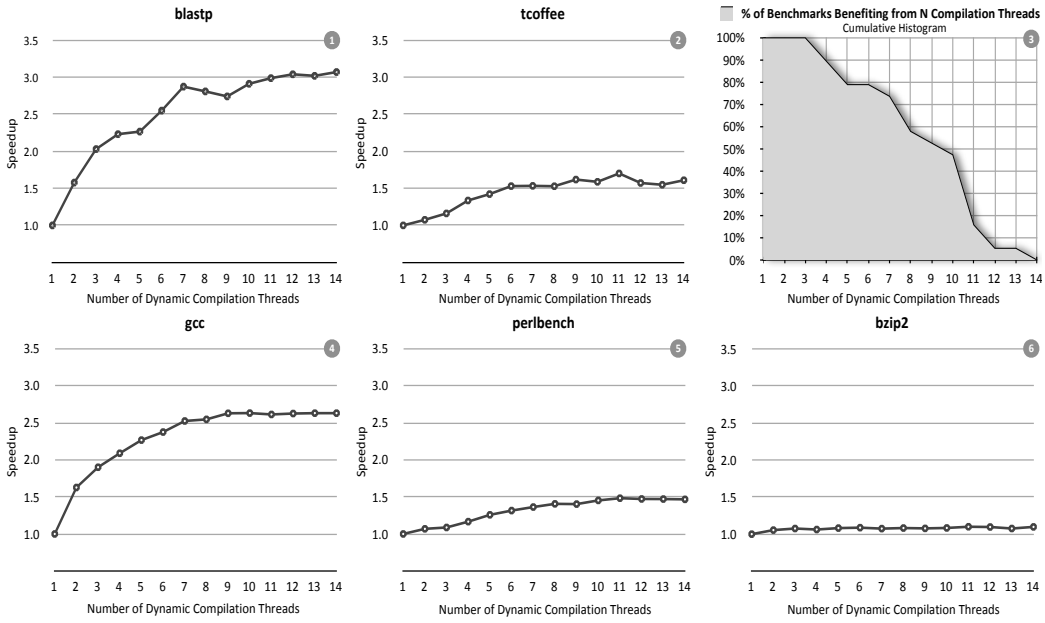

Figure 4.15: Scalability charts for selected benchmarks from BioPerf ①② and Spec Cpu 2006 ④⑤⑥ demonstrating the effect of additional dynamic compilation workers on speedup. Top right cumulative histogram ③ shows % of benchmarks benefiting from given number of dynamic compilation threads.

from Spec Cpu reach their maximum speedup of 2.6 and 1.5 with 10 and 11 dynamic compilation workers, respectively. Not all benchmarks show benefits from adding more dynamic compilation threads. For `bzip2` ⑥ from Spec Cpu the peak speedup is reached with 3 dynamic compilation threads, thus adding more threads does not improve execution time.

As shown in the scalability charts in Figure 4.15, the peak speedup is reached with different numbers of dynamic compilation worker threads. So what is the maximum number of dynamic compilation threads such that 100% of all benchmarks show speedup, i.e. how far does it scale? The cumulative histogram ③ in Figure 4.15 answers this question by depicting the number of benchmarks (in %) that show an improvement by adding more dynamic compilation threads. From the histogram we can see *all* benchmarks show speedups with 3 dynamic compilation worker threads and 50% of all benchmarks benefit from 9 or more dynamic compilation threads.

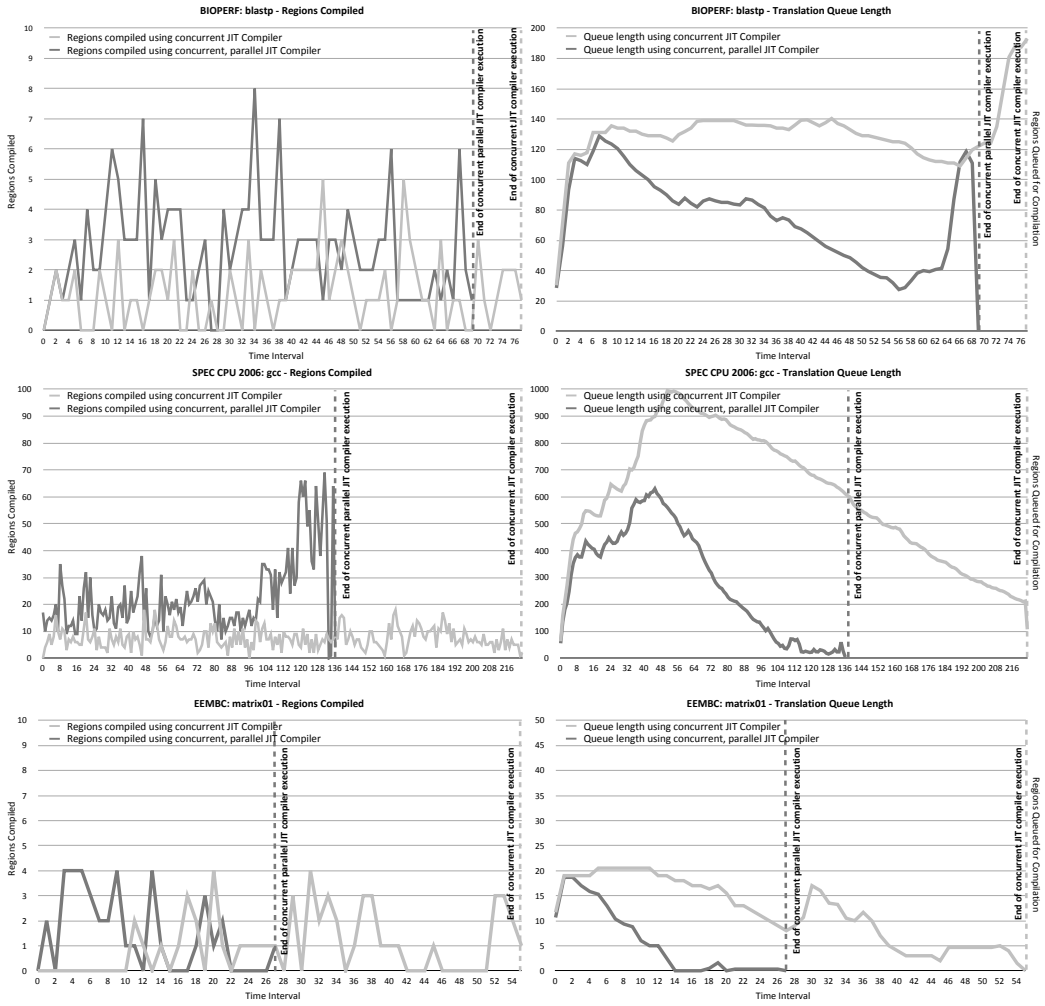

Figure 4.16: Compiled regions per time interval and translation queue length per time interval using an *aggressive* threshold for hot region selection using (a) a concurrent dynamic compiler, and (b) our novel concurrent and parallel dynamic compiler with three compilation worker threads.

4.6.4 Throughput

The key performance indicator of our concurrent and parallel region based compilation scheme is the reduction of overall simulation time. In this section we give a more detailed overview of two additional performance indicators, namely (1) the number of compiled regions per time interval (i.e. rate at which work is completed), and (2) the resulting translation queue lengths over the runtime of three selected benchmarks (see Figure 4.16). We chose one benchmark from each benchmark suite exhibiting a variety of application

hotspots over time, and used a rather aggressive threshold for hot region selection to highlight some of the main advantages of using more than one dynamic compilation thread.

A comparison of the number of compiled regions and translation queue lengths over time indicates that our concurrent and parallel dynamic compilation task farm is able to translate regions significantly faster than the decoupled scheme which relies on a single dynamic compilation thread. On average, three parallel dynamic compilation threads can translate 2.1, 3.1 and 1.5 times as many regions per time interval than the decoupled dynamic compiler for `blastp`, `gcc` and `matrix01`, respectively. Consequently, the *average* translation queue length is 43%, 52% and 51% shorter, and the *maximum* observed queue sizes are 33%, 37%, and 9% shorter for the same three benchmarks, respectively. At the same time, the queue length grows at a noticeably slower rate.

Using a very aggressive initial hotspot threshold, the amount of hot regions identified per trace interval and the resulting translation queue lengths quickly exceed the number of available dynamic compilation workers, even for a short benchmark such as `matrix01`. This demonstrates the need for our dynamic work scheduling and adaptive hotspot threshold selection schemes. Dynamic work scheduling ensures that the most recent and hottest regions are scheduled for translation first, whereas adapting the hotspot threshold based on the translation queue length aims at improving the utilisation of available resources.

4.6.5 Region Sharing

Key Results

For multi-threaded execution environments we evaluated the runtime performance improvements gained by applying the proposed region sharing optimisation. Figure 4.17 presents speedups obtained for multi-threaded data parallel Splash-2 benchmarks when executing 4, 32 and 128 threads on the 4-core host machine outlined in Table 4.1. The baseline is the same private region dynamic binary translator – still with thread-agnostic regions, but without region sharing. As the number of threads increases, so too should the potential for sharing regions between threads resulting in improved performance.

We observe that in all cases the use of region sharing improves over-

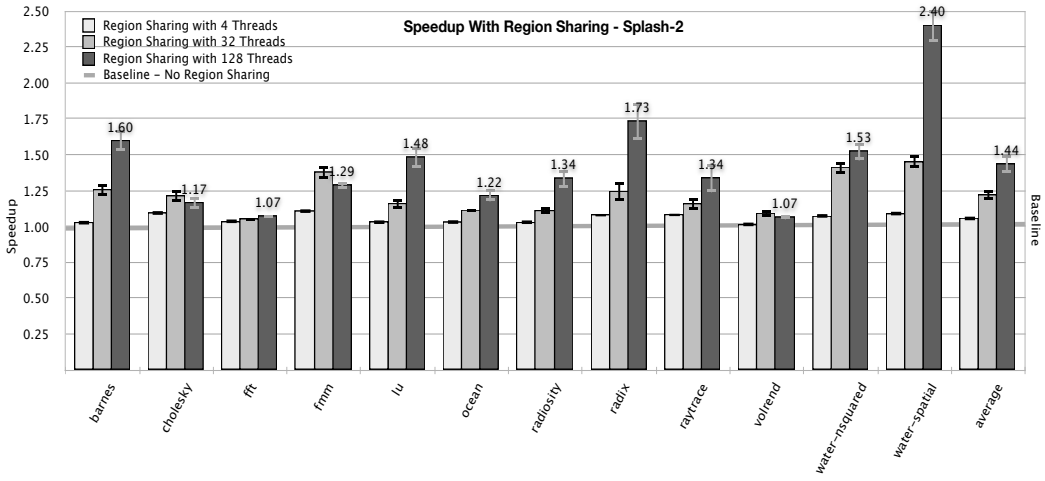

Figure 4.17: Speedups achieved through the use of region sharing, over a baseline execution where region sharing is not used. Speedups are presented when executing 4, 32, and 128 application threads with all benchmarks from the SPLASH-2 suite.

all execution time. The average improvement for 4 threads is 1.06, 1.22 for 32 threads, and for 128 threads, 1.44. The highest speedup of 2.4 is obtained for `water-spatial` with 128 threads. On average, we can see that the speedup obtained when sharing regions increases as the number of executed threads increases. However, three benchmarks do not follow this average trend: `cholesky`, `fmm`, and `volrend`. This is due to a lower potential for region sharing resulting from non-homogeneous compute patterns in these benchmarks. Fewer shared regions lead to reduced savings in dynamic compilation time in relation to the overheads added by increasing the number of threads.

Our results clearly highlight the benefits of region sharing between application threads. Extending our concurrent and parallel dynamic compilation system with a scheme to tag sharable regions ensures that multiple threads can be served simultaneously from just a single region translation. Additional caching of recently handled regions supports this sharing concept further. More threads reach native code execution sooner. This is demonstrated by an overall larger percentage of natively executed code.

Region Sharing Optimisation Scope

The speed-ups presented in the previous section are due to the ability to share regions between threads, reducing the time that threads need to spend

interpreting until they can execute native code, and reducing pressure on the concurrent and parallel dynamic compilation subsystem. For each benchmark, it would be interesting to know how many regions that are handled by the dynamic compilation subsystem are similar to regions that have already been handled for the case where region sharing is disabled. This would imply that in a perfect situation, all the time spent compiling sharable regions could be saved. Figure 4.18 shows the percentage of unique and sharable regions for each benchmark, when executed across 128 application threads.

With the exception of `radiosity`, all benchmarks have over 65% of their regions marked as sharable, with an average of 76%. The largest percentage seen here was the 94% of regions generated during the execution of `radix`. The low 47% of `radiosity` are explained by the fact that the benchmark was built without parallel preprocessing enabled, increasing the total percentage of the program that was executed sequentially. These percentages demonstrate the great potential that exists to speed up execution of data-parallel multi-threaded programs when using region based concurrent and parallel dynamic compilation. On average 76% of all regions are sharable, this means that 76% of all time spent dynamically compiling regions could be saved if region translations are shared between threads using thread-agnostic regions, inter-thread region sharing, and region translation caching.

Thread-Specific vs. Thread-Agnostic Regions

Section 4.5.3 discussed the use of thread-agnostic and thread-specific regions, explaining that thread-agnostic regions enable sharing of dynamically compiled regions between threads. Code generated for thread-agnostic regions requires the use of indirection via a pointer to access a thread's state, versus direct access to runtime addresses known at dynamic compilation time. It would be natural to presume that this indirection imposes a penalty on the overall performance of execution, so we present the effect this code generation approach has on the runtime of the SPLASH-2 benchmarks - when executing only one thread - in Figure 4.18.

Surprisingly, the use of thread-agnostic regions is often faster than using thread-specific regions - a speed-up of 1.09 on average. One would expect that having to obtain the thread state pointer and calculate an offset would be more expensive than simply accessing a constant known at runtime. However, this does not take into account the issue of code size. On the x86 host architecture used throughout this thesis, an instruction that accesses a memory location using register + offset calculations should not require more than

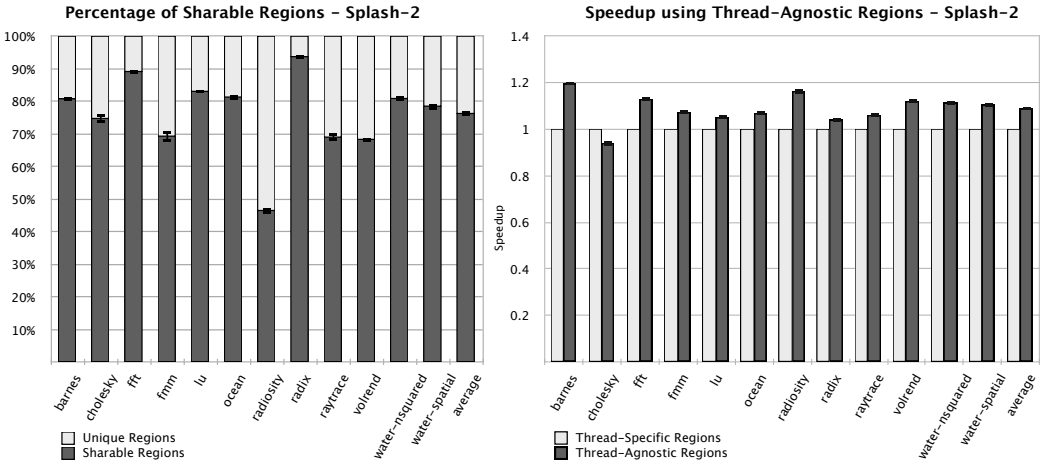

Figure 4.18: Left hand chart shows the percentage of sharable regions for the Splash-2 benchmark suite when executing 128 threads. The right hand chart demonstrates the relative performance when using thread-agnostic over thread-specific regions, when executing one thread for the Splash-2.

4 bytes to encode. On the other hand, encoding a 32 or 64-bit immediate constant requires at least 4 or 8 bytes to encode the constant alone, ignoring the rest of the instruction.

Therefore the use of thread-specific regions leads to an increase in overall code size, even if the total number of instructions generated may decrease. This larger code may lead to slower execution, for instance, if sections of code can no longer reside completely in the instruction cache of the simulation host. While these results may differ on other architectures, they show that on the x86 platform the use of thread-agnostic regions actually results in faster execution, even before enabling the sharing of regions between threads.

4.7 Summary

This chapter presented an incremental region construction scheme enabling concurrent and parallel dynamic compilation based on the task farm design pattern. By combining concurrent and parallel dynamic compilation with light-weight dynamic code discovery, region-based translation units and dynamic work scheduling, we not only minimize and hide dynamic compilation overhead, but fully exploit the available hardware parallelism in standard multi-core desktop PCs. Across three full benchmark suites comprising non-trivial and long-running applications from various domains we achieve an

average reduction in total execution time of 11.5% – and up to 51.9% – for four processors. Our innovative, concurrent and parallel dynamic compilation scheme is robust and never results in a slowdown. Given that only a small fraction of the overall execution time is spent on dynamic compilation, and the majority of time is spent executing natively-compiled code, these results are more than remarkable.

Furthermore, we have demonstrated that our scheme also scales for multi-threaded execution environments. Using thread-agnostic regions – as well as the sharing of regions that this enables – leads to faster execution of multi-threaded programs when using region-based dynamic compilation to enhance performance. These effects are often more pronounced as the number of cores increases – with an average speedup of 1.44x with 128 threads. For the SPLASH-2 benchmark suite, 76% of all regions produced could be shared on average, when executing 128 threads.

While primarily developed for dynamic binary translation, the concept of region-based concurrent and parallel dynamic compilation may also be exploited elsewhere to effectively reduce dynamic compilation overheads and speedup execution. Prime examples are JIT-compiled Java Virtual Machines (JVM) or JavaScript engines. The next Chapter discusses how to successfully exploit this technology in the context of architectural and micro-architectural performance modeling.

5

Architectural and Microarchitectural Modelling

Achieving accurate architectural and micro-architectural observability for microprocessor simulation is in tension with high speed simulation - accuracy vs. performance. In this chapter we propose the application of dynamic compilation techniques to enable ultra-high speed, software only, micro-architectural simulation. The technology surpasses FPGA based simulation in terms of performance whilst maintaining micro-architectural observability.

Simulators play an important role in the design of today's high performance microprocessors. They support design-space exploration, where processor characteristics such as speed and power consumption are accurately predicted for different architectural and micro-architectural models. The information gathered enables designers to select the most efficient processor designs for fabrication. On a slightly higher level instruction set simulators provide a platform on which experimental instruction set architectures can be tested, and new compilers and applications may be developed and verified. They help to reduce the overall development time for new microprocessors by allowing concurrent engineering during the design phase. This is especially important for embedded system-on-chip (SoC) designs, where processors may be extended to support specific applications. However, increasing size and complexity of embedded applications challenges current Iss

technology. For example, the JPEG encode and decode EEMBC benchmarks execute between $10 * 10^9$ and $16 * 10^9$ instructions. Similarly, AAC (Advanced Audio Coding) decoding and playback of a six minute excerpt of Mozart's *Requiem* using a sample rate of 44.1 kHz and a bit rate of 128 kbps results in $\approx 38 * 10^9$ executed instructions. These figures clearly demonstrate the need for fast ISS technology to keep up with performance demands of real-world embedded applications. The broad introduction of multi-core systems, e.g. in the form of multi-processor systems-on-chip (MPSoC), has exacerbated the strain on simulation technology and it is widely acknowledged that improved single-core simulation performance is key to making the simulation of larger multi-core systems a viable option [6].

In this chapter we extend the dynamic compilation architecture outlined in chapter 4 to enable ultra-fast ISS. Dynamic binary translation (DBT) combines interpretive and compiled simulation techniques in order to maintain high speed, observability and flexibility. However, achieving accurate state and even more so microarchitectural observability remains in tension with high speed simulation. In fact, none of the existing ISS [18, 59, 92, 101] based on dynamic binary translation maintains a detailed microarchitectural model. This thesis presents a novel methodology for fast and cycle-accurate performance modelling of the processor pipeline, instruction and data caches, and memory within a DBT ISS. The main contribution is a simple, yet powerful software pipeline model together with an instruction operand dependency and side-effect analysis pass that allows to retain an ultra-fast *instruction-by-instruction* execution model without compromising micro-architectural observability. The essential idea is to reconstruct the microarchitectural pipeline state *after* executing an instruction. This is less complex in terms of runtime and implementation than a *cycle-by-cycle* execution model and reduces the work for pipeline state updates by more than an order of magnitude.

In our ISS we maintain additional data structures relating to the processor pipeline and the caches and dynamically compile highly optimised code that maintains the processors micro-architectural state. In order to maintain flexibility and to achieve high simulation speed our approach decouples the microarchitectural model in the ISS from the architectural model, thereby eliminating the need for extensive rewrites of the simulation framework to accommodate micro-architectural changes. In fact, the strict separation of concerns (functional simulation *vs.* microarchitectural performance modelling) enables the automatic generation of a pipeline performance model from a processor specification written in an architecture description language (ADL) such as LISA [90]. This is, however, beyond the scope of this thesis.

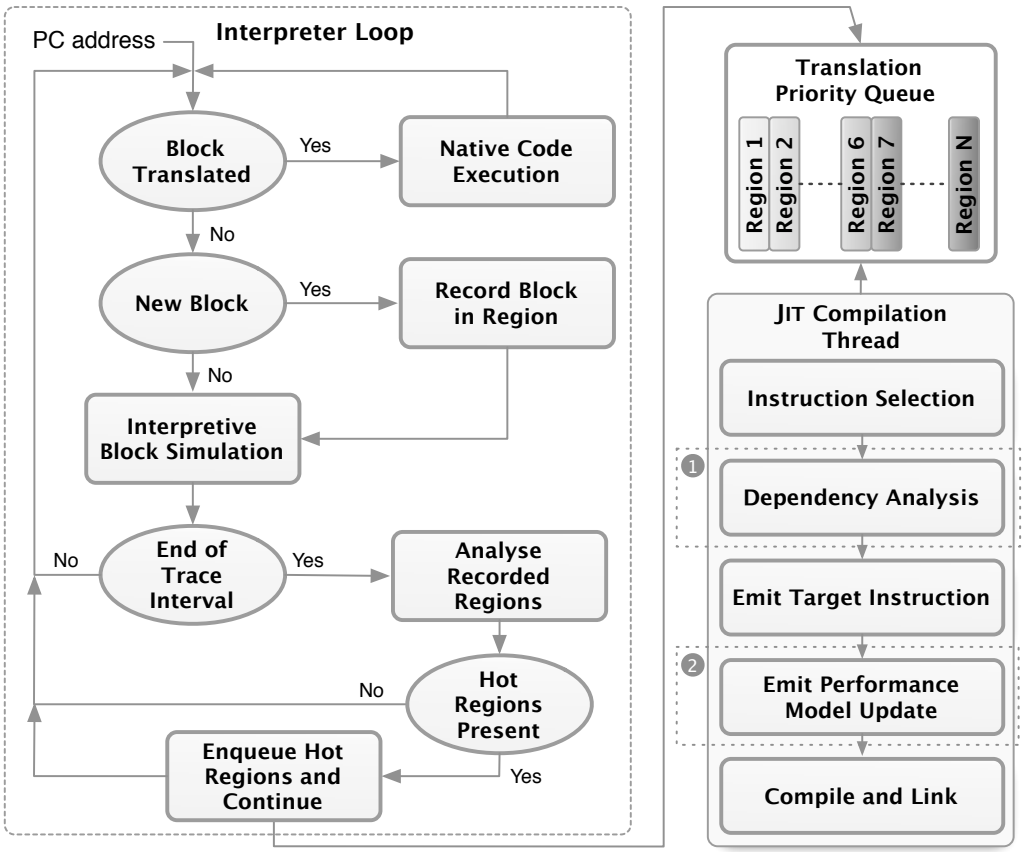

Figure 5.1: Dynamic compilation flow for micro-architectural simulation.

The microarchitectural modelling methodology presented in this thesis is suitable for any interlocked processor pipeline implementation. It has been evaluated using the industry standard EEMBC, CORE MARK, and BIOPERF benchmark suites, against the silicon proven ENCORE embedded processor, implementing the ARCompact [98] ISA. The ENCORE processor was selected to verify the model due to the availability of a hardware implementation. The microarchitectural model faithfully models the 5-stage interlocked ENCORE processor pipeline (see Figure 5.4) with forwarding logic, its mixed-mode 16/32-bit instruction set, zero overhead loops, static and dynamic branch prediction, branch delay slots, and four-way set associative data and instruction caches. To demonstrate its portability we also provide results for the 7-stage ENCORE processor pipeline variant. Across all 44 benchmarks from EEMBC, CORE MARK, and BIOPERF the speed of simulation reaches up to 88 MIPS on a standard `x86` desktop computer and outperforms that of a speed-optimised FPGA implementation of the ENCORE processor.

5.1 Motivating Example

Before taking a more detailed look at how to dynamically compile code to faithfully model microarchitectural components, a motivating example is provided to highlight the key concepts. Consider the block of ARCompact instructions in Figure 5.2 taken from the CoreMark benchmark. Our Iss ArcSim identifies this region of code as a hotspot and compiles it to native machine code using the sequence of steps illustrated in Figure 5.1. Each region maps onto a function denoted by its address (see label ① in Figure 5.2), and each instruction is translated into semantically equivalent native code faithfully modelling the processors architectural state (see labels ②, ③, and ⑥ in Figure 5.2). In order to correctly track microarchitectural state, each translated ARCompact instruction is augmented with calls to specialised functions (see labels ③ and ⑦ in Figure 5.2) responsible for updating the underlying microarchitectural model (see Figure 5.4).

Figure 5.4 demonstrates how the hardware pipeline microarchitecture is mapped onto a software model capturing its behaviour. To improve the performance of microarchitectural state updates we emit several versions of performance model update functions tailored to each instruction *kind* (i.e. arithmetic and logical instructions, load/store instructions, branch instructions). Section 5.2.2 describes the microarchitectural software model in more detail. After code has been emitted for a region of executed basic blocks, it is dynamically compiled and linked by a concurrent and parallel dynamic compiler as outlined in Section 4.3.

5.2 Dynamic Compilation of Microarchitectural Components

In common with the EnCore processor, the ArcSim Iss is highly configurable. Architectural features such as register file size, instruction set extensions, the set of branch conditions, the auxiliary register set, as well as memory mapped Io extensions can be specified via a set of well defined Apis and configuration settings. Furthermore, microarchitectural features such as pipeline depth, per instruction execution latencies, cache size and associativity, cache block replacement policies, memory subsystem layout, branch prediction strategies, as well as bus and memory access latencies are fully configurable. The microarchitectural configurations used for all experiments are listed in Table 5.1.

Figure 5.2: Dynamic binary translation of ARCompact basic block with `CpuState` structure representing architectural ⑥ and microarchitectural state ⑦. See Figure 5.4 for an implementation of the micro-architectural state update function `pipeline()`.

The following sections outline the processor pipeline model and describe how to account for instruction operand availability and side-effect visibility timing. Additionally cache and memory models and the integration of control flow and branch prediction into the microarchitectural model is discussed.

Processor Microarchitecture	EnCore
Pipeline	5-Stage and 7-Stage Interlocked
Execution Order	In-Order
Branch Prediction	Yes
Isa	ARCompact
Register Set	32 baseline registers
Instruction Set Extensions	None
Floating-Point	Hardware
Memory System	
L1-Cache	
Instruction	32k/4-way associative
Data	32k/4-way associative
Replacement Policy	Pseudo-random
L2-Cache	None
Bus Width/Latency/Clock Divisor	32-bit/16 cycles/2
Instruction Set Simulator	ArcSim
Simulator	Full-system, cycle-accurate
Jit Compiler	Llvm
I/O & System Calls	Emulated

Table 5.1: Configuration and setup of the simulated target microarchitectures together with the instruction set simulator. Fpga and Asip implementations of the outlined microarchitectures were used for verification.

5.2.1 Microarchitecture

To demonstrate the effectiveness of our approach we use a state-of-the-art processor implementing the ARCompact Isa, namely the EnCore [83]. Its microarchitecture is based on an interlocked pipeline with forwarding logic, supporting zero overhead loops (Zol), freely intermixable 16- and 32-bit instruction encodings, static and dynamic branch prediction, branch delay slots, and predicated instructions. There exist two pipeline variants of the EnCore processor, namely a 5-stage (see Figure 5.3) variant and a 7-stage variant which has an additional Align stage between the Fetch and Decode stages, and an additional Register stage between the Decode and Execute stages.

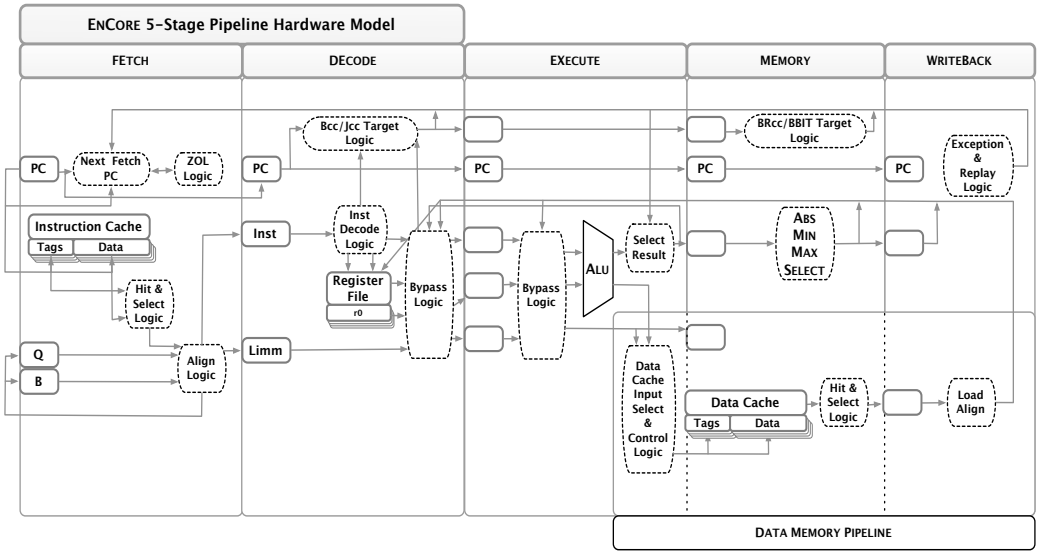

Figure 5.3: EnCore processor 5-Stage pipeline microarchitecture.

The processor was configured using 32K 4-way set associative instruction and data caches with a pseudo-random block replacement policy. Because cache misses are expensive, a pseudo-random replacement policy requires *accurate* modelling of cache behaviour to avoid large deviations in cycle count. Although the above configuration was used for this work, the processor is highly configurable. Pipeline depth, cache sizes, associativity, cache block replacement policies, byte order (i.e. big endian, little endian), bus widths, register-file size, and many other instruction set specific options are configurable. The processor is fully synthesisable onto an Fpga and fully working Asip silicon implementations have been taped-out.

5.2.2 Pipeline Model

The granularity of execution on hardware and Rtl simulation is cycle based —*cycle-by-cycle*. If the designer wants to find out how many cycles it took to execute an instruction or program, all that is necessary is to simply count the number of cycles. While this execution model works well for hardware it is too detailed and slow for Iss purposes. Therefore fast functional Iss have an *instruction-by-instruction* execution model. While this execution model yields faster simulation speeds it usually compromises microarchitectural observability and detail. Our dynamically compiled software pipeline model together with an instruction operand dependency and side-effect analysis

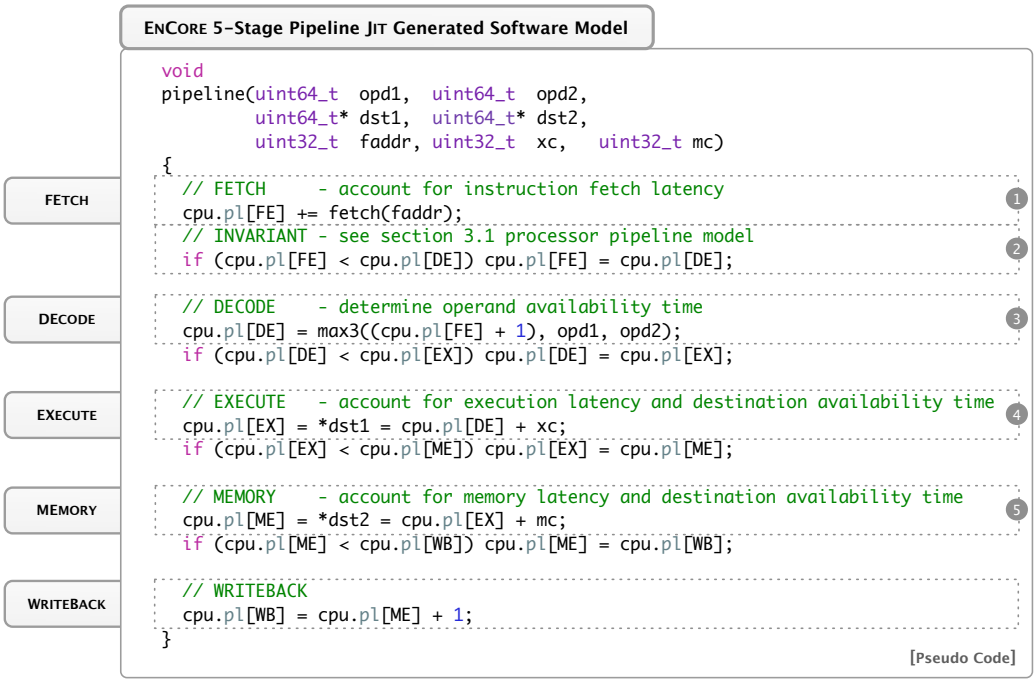

Figure 5.4: Jit generated EnCore 5-Stage pipeline microachitectural software model.

pass allows to retain an *instruction-by-instruction* execution model without compromising microarchitectural observability. The essential idea is to reconstruct the microarchitectural pipeline state *after* executing an instruction. This approach is also known as *functional-first* simulation [28, 29].

Thus the processor pipeline is modelled as an array with as many elements as there are pipeline stages (see definition of `pl[STAGES]` at label ⑦ in Figure 5.2). For each pipeline stage the corresponding latencies are added up, and the cycle-count at which the instruction is ready to *leave* the respective stage is stored. The line with label ① in Figure 5.4 demonstrates this for the fetch stage `cpu.pl[FE]` by adding the amount of cycles it takes to fetch the corresponding instruction to the current cycle count at that stage.

The next line in Figure 5.4 with the label ② is an *invariant* ensuring that an instruction cannot leave its pipeline stage *before* the instruction in the immediately following stage is ready to proceed. Figure 5.5 contains a detailed example of the microarchitectural performance model determining the cycle count for a sample ARCompact instruction. It demonstrates the reconstruction of microarchitectural pipeline state *after* the instruction has been executed using the final instruction of the basic block depicted in Figure 5.2.

	FE	DE	EX	ME	WB	
Instruction	or r4,r4,r3					Pipeline Model
FETCH	100	102	103	110	115	`// INITIAL STATE AT FETCH`
	101	102	103	110	115	`cpu.pl[FE] += fetch(0x00000868);`
	102	102	103	110	115	`if (cpu.pl[FE] < cpu.pl[DE])` `  cpu.pl[FE] = cpu.pl[DE];`
DECODE	102	102	103	110	115	`// INITIAL STATE AT DECODE`
	102	105	103	110	115	`cpu.pl[DE] = max3((cpu.pl[FE]+1),` `  opd1, opd2);`
	102	105	103	110	115	`if (cpu.pl[DE] < cpu.pl[EX])` `  cpu.pl[DE] = cpu.pl[EX];`
EXECUTE	102	105	103	110	115	`// INITIAL STATE AT EXECUTE`
	102	105	106	110	115	`cpu.pl[EX] = cpu.pl[DE] + 1;` `*dst1  = cpu.pl[EX];`
	102	105	110	110	115	`if (cpu.pl[EX] < cpu.pl[ME])` `  cpu.pl[EX] = cpu.pl[ME];`
MEMORY	102	105	110	110	115	`// INITIAL STATE AT MEMORY`
	102	105	110	111	115	`cpu.pl[ME] = cpu.pl[EX] + 0;` `*dst2  = cpu.pl[ME];`
	102	105	110	115	115	`if (cpu.pl[ME] < cpu.pl[WB])` `  cpu.pl[ME] = cpu.pl[WB];`
WRITEBACK	102	105	110	115	115	`// INITIAL STATE AT WRITEBACK`
	102	105	110	115	116	`cpu.pl[WB] = cpu.pl[ME] + 1;`
	102	105	110	115	116	`// FINAL PIPELINE STATE`
	Per Pipeline Stage Cycle Count					

Figure 5.5: Detailed example of microarchitectural performance model determining the cycle count for a sample ARCompact instruction. Bold red numbers denote changes to cycle-counts for the respective pipeline stages, bold green numbers denote already committed cycle-counts.

5.3 Instruction Operand Dependencies and Side Effects

To determine when an instruction is ready to leave the decode stage it is necessary to know when its operands become available. For instructions with side-effects (i.e. modification of register contents) it must be recorded when side-effects will become visible. This operand availability timing information is encoded and maintained in the `avail[GPRS]` array (see label ⑦ in Figure 5.2) for each operand. Instruction execution latencies are configurable and it is also possible to have variable execution latencies for specific instructions that depend on operand values or the state of processor flags. Figure 5.4 demonstrates how variable execution latencies are passed as a parameter and accounted for in the execute stage of the processor pipeline model.

Microarchitectural update functions are parameterised with source operand availability times and pointers to destination operand availability memory locations determined during dependency analysis (see label ③ in Figure 5.2). This information is then used to compute when an instruction can leave the decode stage (see label ③ in Figure 5.4) and to record when side-effects become visible in the execute and memory stage (see labels ④ and ⑤ in Figure 5.4). Because not all instructions modify general purpose registers or have two source operands, there exist several highly optimised versions of microarchitectural state update functions, and the function outlined in Figure 5.4 demonstrates only one of several possible variants.

5.3.1 Control Flow and Branch Prediction

When dealing with explicit and implicit control flow instructions (e.g. jump, branch, branch on compare, zero overhead loops) special care must be taken to account for various types of penalties and speculative execution. The ARCompact ISA allows for delay slot instructions and the ENCORE processor and ARCSIM ISS support various static and dynamic branch prediction schemes.

The code highlighted by label ④ in Figure 5.2 demonstrates how a branch penalty is applied for a mis-predicted branch using a static branch prediction scheme. The pipeline penalty depends on the pipeline stage when the branch outcome and target address are known, and the availability of a delay slot instruction. These latencies can typically be derived from a microarchitectural hardware model such as the one depicted in Figure 5.3, by looking

up at which stages target addresses for various control flow instructions such as Bcc/Jcc[1] and Brcc/Bbit[2] are available.

It is also necessary to account for speculatively fetched and executed instructions in case of a mis-predicted branch. For the example in Figure 5.2 this means that the instruction *after* the delay slot instruction must be fetched to maintain an accurate microarchitectural memory model state for a taken branch.

5.3.2 Memory Model

Because cache misses and off-chip memory access latencies significantly contribute towards the final cycle count, ArcSim maintains an accurate cache and memory model. In its default configuration the EnCore processor implements a pseudo-random block replacement policy where the content of a shift register is used in order to determine a *victim* block for eviction. The rotation of the shift register must be triggered at the same time and by the same events as in hardware, requiring a faithful microarchitectural model.

The ARCompact Isa offers very flexible and powerful `load` and `store` operations. Therefore memory access simulation is a critical aspect of high-speed full system simulation. Section 4.4 describes in detail how simulated memory addresses are mapped onto host memory address locations enabling the architectural simulation of `load` and `store` instructions at the highest possible rate. To enable high speed microarchitectural memory timing simulation, different timing models can be plugged in by implementing a predefined interface. Then, for each load, store, and instruction fetch event, the pipeline model calls specific methods implemented by memory timing model plugins to determine memory access latencies. For instruction fetch latencies such a call is demonstrated in ④ Figure 5.2 and ① Figure 5.4.

Our primary objective was to enable high-speed microarchitectural simulation of the processor pipeline. Cache and memory interconnect timing models faithfully model the silicon behaviour of the EnCore processor using state-of-the-art memory modelling approaches [13, 89]. Improving state-of-the-art of microarchitectural memory model simulation is a research topic of its own and is being actively worked on by other members of the research group.

[1] Bcc/Jcc - Branch/Jump conditionally instructions.
[2] Brcc/Bbit - Compare and Branch, Branch on Bit test instructions.

Vendor & Model	HP™ COMPAQ™ dc7900 SFF
Number CPUs	1 (dual-core)
Processor Type	Intel© Core™ 2 Duo processor E8400
Clock Frequency	3 GHz
L1-Cache	32K Instruction/Data caches
L2-Cache	6 MB
FSB Frequency	1333 MHz

Table 5.2: Simulation Host Configuration.

5.4 Evaluation and Analysis of Results

We have extensively evaluated the proposed dynamically compiled microarchitectural modelling approach and in this section we describe the experimental setup and methodology before presenting and discussing the results.

5.4.1 Benchmarks and Experimental Setup

We have evaluated our dynamically compiled microarchitectural modelling approach using the BIOPERF benchmark suite that comprises a comprehensive set of computationally-intensive life science applications [8]. We also used the industry standard EEMBC 1.1, and COREMARK [36] embedded benchmark suites comprising applications from the automotive, consumer, networking, office, and telecom domains.

All codes have been built with the ARC port of the GCC 4.2.1 compiler with full optimisation enabled (i.e. `-O3 -mA7`). Each benchmark has been simulated in a stand-alone manner, without an underlying operating system, to isolate benchmark behaviour from background interrupts and virtual memory exceptions. Such system-related effects are measured by including a Linux full-system simulation in the benchmarks.

The BIOPERF benchmarks were run with "class-A" input data-sets available from the BIOPERF web site. The EEMBC 1.1 and COREMARK benchmarks were configured using large iteration counts to execute at least 10^9 instructions. All benchmarks were simulated until completion. The Linux benchmark consisted of simulating the boot-up and shut-down sequence of a Linux kernel configured to run on a typical embedded ARC700 system with two interrupting timers, a console UART, and a paged virtual memory system.

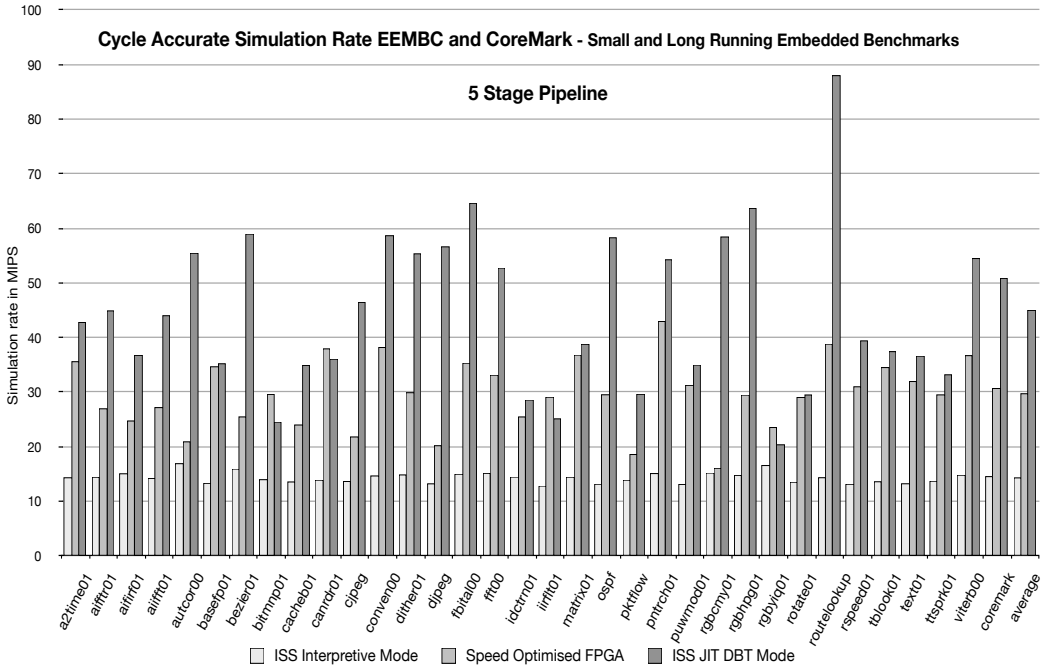

Figure 5.6: 5-Stage Pipeline - Simulation rate (in MIPS) using EEMBC and CoreMark benchmarks comparing (a) ISS interpretive cycle-accurate simulation mode, (b) speed-optimised FPGA implementation, and (c) our novel ISS DBT cycle-accurate simulation mode.

Our main interest has been on simulation *speed*, therefore we have measured the maximum possible simulation speed in MIPS using various simulation modes (FPGA speed *vs.* cycle-accurate interpretive mode *vs.* cycle-accurate DBT mode - see Figures 5.6, 5.7, 5.8 and 5.9). Table 5.1 lists the configuration details of our simulator and target processor. All measurements were performed on a x86 desktop computer detailed in Table 5.2 under conditions of low system load. When comparing ArcSim simulation speeds to FPGA implementations shown in Figures 5.6, 5.7, 5.8 and 5.9, we used a Xilinx Virtex5 XC5 VFX70T (speed grade 1) FPGA clocked at 50 MHz.

5.4.2 Speedup

We initially discuss the simulation speed-up achieved by our novel cycle-accurate DBT microarchitectural simulation mode compared to a verified cycle-accurate interpretive simulation mode for a 5-stage processor pipeline variant. A comparison against a speed-optimised FPGA implementation is also provided as this has been the primary motivation of our work. Finally,

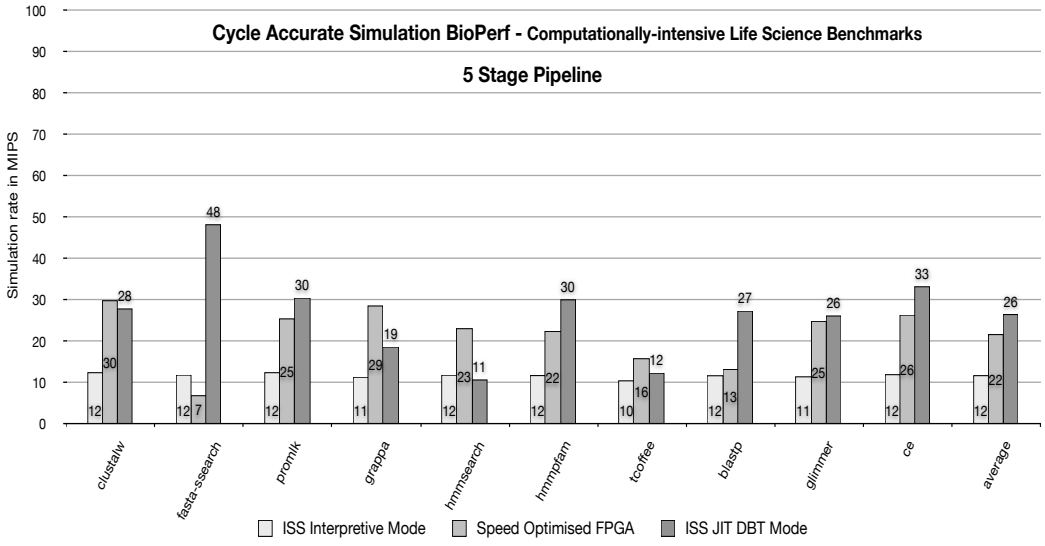

Figure 5.7: 5-Stage Pipeline - Simulation rate (in Mips) using the BioPerf benchmarks comparing (a) Iss interpretive cycle-accurate simulation mode, (b) speed-optimised Fpga implementation, and (c) our novel Iss Dbt cycle-accurate simulation mode.

results for a different pipeline variant, namely the 7-stage pipeline version of the EnCore, are presented. A summary of all results is shown in Figures 5.6, 5.7, 5.8, and 5.9.

For Eembc and CoreMark benchmarks (Figure 5.6) the proposed cycle-accurate Dbt simulation mode for the 5-stage pipeline variant is more than *three* times faster on average (45 Mips) than the verified cycle-accurate interpretive mode (14 Mips). It even outperforms a speed-optimised Fpga implementation of the EnCore processor (30 Mips) clocked at 50 Mhz. For some benchmarks (e.g. `autcor00`, `bezier01`, `cjpeg`, `djpeg`, `rgbcmy01`, `rgbhpg01`, `routelookup`) the new cycle-accurate Dbt mode is more than *twice* as fast as the speed-optimised Fpga implementation. This can be explained by the fact that those benchmarks contain sequences of instructions that map particularly well onto the simulation host Isa. Furthermore, frequently executed blocks in these benchmarks contain instructions with fewer dependencies resulting in the generation and execution of simpler microarchitectural state update code.

The new cycle-accurate Dbt simulation achieves an average simulation rate of 26 Mips for the computationally-intensive life science application programs from the BioPerf benchmark suite (Figure 5.7), again outper-

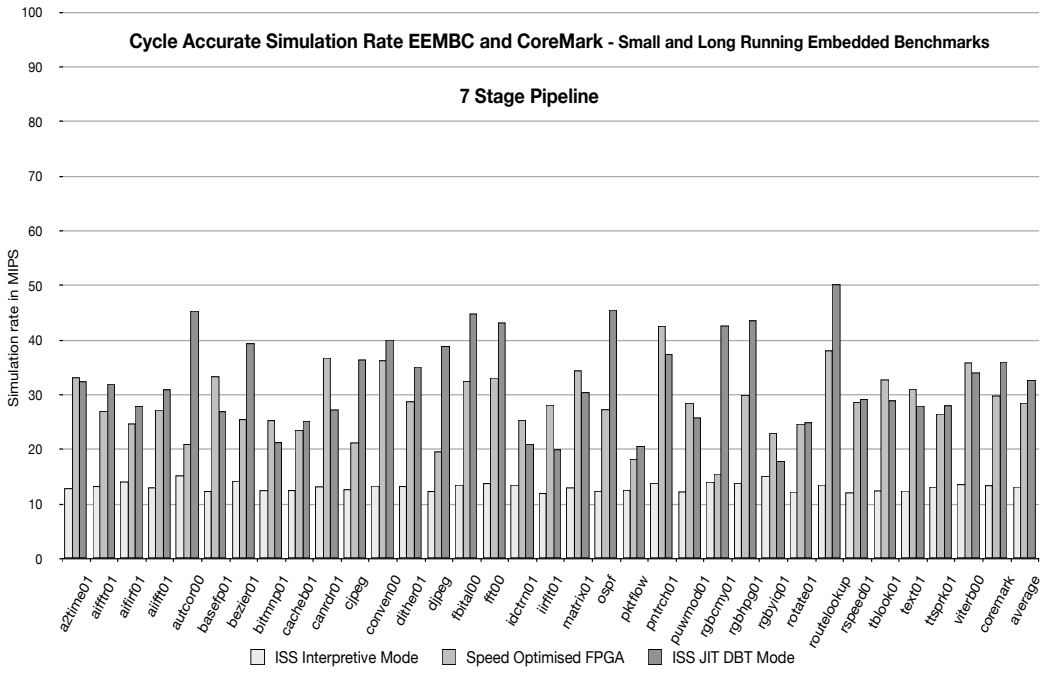

Figure 5.8: 7-Stage Pipeline - Simulation rate (in Mips) using Eembc and CoreMark benchmarks comparing (a) Iss interpretive cycle-accurate simulation mode, (b) speed-optimised Fpga implementation, and (c) our novel Iss Dbt cycle-accurate simulation mode.

forming the previously outlined speed-optimised Fpga implementation (22 Mips). Due to a relatively high cycles per instruction (Cpi) metric of 7, the speed-optimised Fpga is more than 6 times slower than the cycle-accurate Dbt for the `fasta-ssearch` benchmark. For the `hmmsearch` benchmark the Dbt cycle accurate simulation is slightly slower than interpretive cycle accurate simulation. This is entirely due to the shorter runtime and abundance of application hotspots keeping the concurrent and parallel Dbt engine busy, resulting in a slowdown due to dynamic code discovery and dynamic compilation overheads. A similar effect for the same reasons has been observed for the `xalancmbk` benchmark from Spec Cpu 2006 (see Section 4.6.2). The solution is to use more dynamic compilation worker threads to improve the throughput of the dynamic compilation subsystem.

For Eembc and CoreMark benchmarks the cycle-accurate Dbt simulation mode for the 7-stage pipeline variant (Figure 5.8) is more than *twice* as fast on average (33 Mips) than the verified cycle-accurate interpretive mode (13 Mips). Again it outperforms the speed-optimised Fpga implementation of the 7-stage EnCore processor variant (28 Mips) clocked at 50 Mhz. For

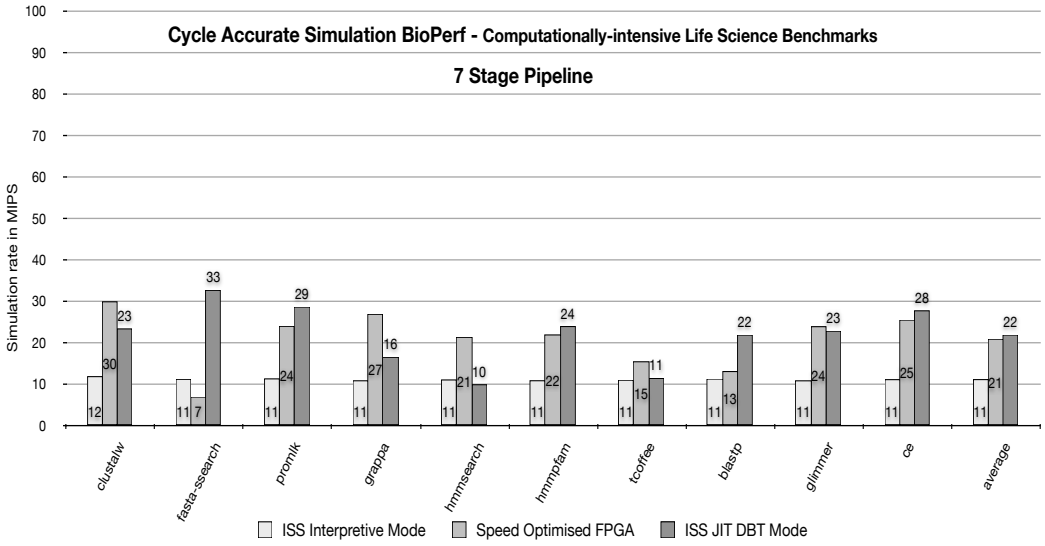

Figure 5.9: 7-Stage Pipeline - Simulation rate (in MIPS) using the BIOPERF benchmarks comparing (a) ISS interpretive cycle-accurate simulation mode, (b) speed-optimised FPGA implementation, and (c) our novel ISS DBT cycle-accurate simulation mode.

some benchmarks (e.g. `autcor00`, `djpeg`, `rgbcmy01`) the novel cycle-accurate DBT mode is almost *twice* as fast as the speed-optimised FPGA implementation. Average BIOPERF benchmark simulation rate figures for the 7-stage pipeline (Figure 5.9) demonstrate that our cycle-accurate DBT (22 MIPS) once more outperforms a speed-optimised FPGA implementation (21 MIPS) and is *twice* as fast as cycle-accurate interpretive simulation (11 MIPS).

For the introductory sample application performing AAC decoding and playback of Mozart's *Requiem* outlined in Section 5.1, the cycle-accurate DBT mode is capable of simulating at a sustained rate of 31 MIPS (7-stage pipeline) and 36 MIPS (5-stage pipeline), enabling real-time simulation. For the boot-up and shutdown sequence of a Linux kernel cycle-accurate DBT simulation mode achieves 12 MIPS for both pipeline variants resulting in a highly responsive interactive environment. These examples clearly demonstrate that the ISS used in these experiments is capable of simulating system-related effects such as interrupts and virtual memory exceptions efficiently and still provide full microarchitectural observability.

5.5 Summary

We have demonstrated that our approach to microarchitectural ISS easily surpasses speed-optimised FPGA implementations whilst providing detailed architectural and microarchitectural profiling feedback and statistics. The main contribution is a simple yet powerful software pipeline model in conjunction with an instruction operand dependency and side-effect analysis pass integrated into a DBT ISS enabling ultra-fast simulation speeds without compromising microarchitectural observability. Our cycle-accurate microarchitectural modelling approach is portable and independent of the implementation of a functional ISS. More importantly, it is capable of capturing even complex interlocked processor pipelines. Because our novel pipeline modelling approach is microarchitecture adaptable and decouples the performance model in the ISS from functional simulation, it can be automatically generated from ADL specifications. Research into auto generation of microarchitectural ISS using DBT given an ADL is actively explored within our research group by another student.

This Chapter investigated fast microarchitectural modelling of interlocked in order processors. One important area of future research is to improve microarchitectural simulation of out-of-order, superscalar, and multi-core processors. Improving the simulation performance of microarchitectural interconnect and memory models is another important research area. Currently several students within our research group are actively working on problems in these areas, using microarchitectural modelling, runtime information and dynamic compilation techniques developed in this thesis in novel ways to improve performance whilst maintaining accuracy.

6 Conclusions

This chapter summarizes the techniques and contributions of this thesis. It provides a critical anlysis and discusses future work in the area of concurrent and parallel dynamic compilation, and architectural and microarchitectural performance modelling.

This thesis has investigated techniques and approaches for dynamic code discovery and dynamic compilation exploiting parallel design patterns to improve dynamic compilation throughput, and to enable the design of more responsive and interactive virtual machines. In particular, Chapter 4 has presented a concurrent and parallel dynamic compilation system that can speed up the execution of target programs. The design is generic and has been shown to work in the context of dynamic binary translation, architectural, and microarchitectural simulation. Finally, Chapter 5 has presented a novel application of the concurrent and parallel dynamic compiler design for microarchitectural simulation, where code updating microarchitectural models is compiled on the fly.

6.1 Contributions

This section summarises the main contributions of this thesis for dynamic compilation system, virtual machine, and simulation tool design and imple-

mentation, exploiting the concept of concurrent and parallel dynamic compilation.

6.1.1 Concurrent and Parallel Dynamic Compilation

Chapter 4 has explored a novel approach to dynamic code discovery and dynamic compilation. The dynamic compiler runs concurrently with an interpreter that performs region based incremental code discovery. To hide compilation latency and improve dynamic compilation throughput the dynamic compiler compiles detected program hotspots in parallel. Adaptive and automatically adjusting heuristics for selecting and scheduling units of compilation give priority to the most recent, and frequently executed regions to further reduce time spent in interpreted or unoptimised execution mode. Compared to previously proposed schemes [20, 21, 67, 87], this design of a concurrent and parallel dynamic compiler is the first end-to-end solution applicable in contexts where code discovery must be dynamic and self-modifying code is possible.

Across three full industry standard benchmark suites (Spec Cpu 2006, BioPerf, Eembc and CoreMark) comprising non-trivial and long-running applications from various domains, the proposed concurrent and parallel dynamic compilation system achieves an average reduction in total execution time of 11.5% - and up to 51.9% - on a standard quad-core `x86` processor. The proposed scheme is robust and never results in a slowdown. Given that only a small fraction of the overall execution time is spent on dynamic compilation, and the majority of time is spent executing natively-compiled code, these results are more than remarkable.

While the proposed dynamic code discovery is optimised for dynamic binary translation, the concurrent and parallel dynamic compiler design is a practical solution applicable in any dynamic compilation scenario to reduce dynamic compilation latency, consequently improving target application execution performance

6.1.2 Architectural and Microarchitectural Modelling

It has been shown in Chapter 5 how to apply the proposed dynamic compilation scheme in the context of architectural and microarchitectural simulation. Program fragments that simulate the behaviour of architectural and microarchitectural components such as a memory management unit, a processor

pipeline, caches, and a branch predictor, are optimised and dynamically compiled to speedup simulation and enable effective design space exploration. The presence and complexity of additional dynamically generated code has a negative impact on compilation latency that can be recovered by using the previously proposed concurrent and parallel dynamic compilation scheme.

This is the first time dynamic compilation has been explored in the context of microarchitectural simulation. The main contribution is a simple yet powerful software processor pipeline model designed to work with the dynamic compiler to enable runtime optimisations based on instruction kinds (i.e. arithmetic, memory, logic, and control flow instructions). The model captures enough microarchitectural detail to enable cycle accurate simulation. Finally, the proposed model is microarchitecture adaptable and decouples the performance model in the ISS from functional simulation.

The performance of the proposed dynamically compiled microarchitectural modelling technique has been evaluated using industry standard benchmarks from various domains (BIOPERF, EEMBC and COREMARK). Model accuracy has been verified using FPGA implementations of the modelled microarchitectures. On average the proposed microarchitectural modelling scheme is faster than FPGA implementations of the simulated microprocessors, reducing engineering efforts by an order of magnitude, and enabling effective, software only, early design space exploration and application performance evaluation.

6.2 Critical Analysis

This thesis has investigated the design of a concurrent and parallel dynamic compilation system using region based incremental code discovery. This section now conducts a critical analysis of this work.

6.2.1 Dynamic Compilation Methodology

All experimental results presented in this thesis have been obtained through the use of ARCSIM, an instruction set simulator for the ARCOMPACT platform. Apart from the commercial XISS simulator [99], ARCSIM is the only instruction set simulator with full support for the ARCOMPACT ISA. Under the hood XISS uses dynamic binary translation to speed up instruction set simulation. According to the XISS product website [99] ARCSIM is more

than one order of magnitude faster than xISS. This performance advantage has led to its licensing and productisation by the worlds second largest Ip provider. The concurrent and parallel dynamic compilation engine is now part of a variety of production tools enabling high speed simulation.

A direct comparison with Jit-based simulators such as Qemu [12] or Simit-ARM [92] that target the Arm Isa has not been performed due to the differences of instruction set architectures, compilers targeting those architectures, and the level of modelling detail - it would be like a direct comparison of the Microsoft Common Language Runtime [75] with a Java virtual machine [60]. The fact that researchers and industry [2, 56, 65] cite and use our concurrent and parallel dynamic compilation technology to improve the runtime performance of Iss and Java virtual machines, is hopefully sufficient proof for the general applicability of the presented technology.

Once frequently executed program regions have been discovered and dynamically compiled, the generated native code does not include any further profiling code. As a side-effect of this design it is not possible to drive further optimisations of compiled code (i.e. perform tiered-compilation [9,30,45]) as no further profiling information during native code execution is collected. In an experiment we have extended the proposed dynamic compilation infrastructure to support tiered compilation. While performance gains for some benchmarks could be observed, the additional book keeping of runtime information and profiling logic together with the overhead of dynamic re-compilation caused the average performance across all benchmarks to degrade.

6.2.2 Microarchitectural Modelling

Chapter 5 has shown how to combine the proposed concurrent and parallel dynamic compilation architecture with a novel microarchitectural processor pipeline model to enable high-speed microarchitectural simulation. This was the first time ever that microarchitectural model update code is optimised and dynamically compiled with the goal to speed up microarchitectural processor simulation. The lack of verified microarchitectural simulators that use dynamic compilation to speedup microarchitectural modelling made a direct comparison infeasible. A comparison to interpretive microarchitectural simulators such as Gem5 [13] is unfair because Gem5 only supports interpretive microarchitectural simulation that is orders of magnitude slower. Therefore we have chosen state-of-the-art Fpga based simulation of the modelled microarchitectures as a baseline for a direct performance comparison.

6.3 Future Work

This thesis has investigated the design and implementation of a concurrent and parallel dynamic compiler to improve the performance of dynamic binary translators and instruction set simulators. The dynamic compilation scheme was then used together with a novel microarchitectural processor modelling approach that enables dynamic compilation of optimised microarchitectural state update logic. This combination resulted in significant speedups over state-of-the-art FPGA based simulation solutions.

One area of future research in the realm of dynamic code discovery and concurrent and parallel dynamic compilation is that of tiered compilation. The main challenge in this context is to minimise the overheads of prolonged profiling, bookkeeping and analysis of additional profiling data together with dynamic re-compilation. The ideal solution has to outweigh the overheads of additional runtime profiling and dynamic re-compilation with the potential performance improvements for a given application hotspot based on dynamic runtime information.

Another direction of future research is in the area of fast microarchitectural modelling as it has the potential to realise high gains over state-of-the-art simulation solutions. Improving the design of processor pipeline models that support high-speed out-of-order, superscalar, and multi-core microarchitectural processor simulation has the potential to significantly reduce the design time and risk of future architectures. Advancing the simulation performance of microarchitectural interconnect and memory models is another important research area. The ideal solutions in the context of microarchitectural simulation will combine dynamic compilation techniques together with runtime information driving optimisation and code generation decisions for code manipulating microarchitectural state.

Finally, the high engineering effort that is necessary to create and verify a highly optimised instruction set simulator that is also capable of fast microarchitectural simulation, raises the question whether such tools can be constructed automatically from an architecture description language. The dynamic code discovery and compilation framework that has been designed in this thesis is modular and mostly independent of the simulated instruction set architecture. Therefore it can be integrated into automatically generated instruction set simulators.

List of Figures

1.1 Motivating Example - Concurrent and Parallel Dynamic Compilation 5

2.1 Classification of Dynamic Compilation Approaches 10
2.2 Typical Dynamic Compilation System Infrastructure 13
2.3 Dynamic Compilation Strategies 15

4.1 Motivating Example - Full-system Linux OS Simulation 31
4.2 Concurrent and Parallel Dynamic Compilation in Action . . . 33
4.3 Incremental Interval Based Region Construction 34
4.4 Spatial and Temporal Region Partitioning 35
4.5 Concurrent and Parallel Dynamic Compilation Flow 37
4.6 Software Indirect Branch Prediction via Inline Caching 38
4.7 Basic Block Translation State Transitions 39
4.8 Simulated Address onto Host Memory Address Mapping . . . 43
4.9 Code Discovery Approache for Multithreaded Applications . . 47
4.10 Motivating Example - Region Sharing Optimisation 48
4.11 Tread-specific vs. Thread-agnostic Code Generation 49
4.12 Region Sharing in Action for Multi-threaded Applications . . . 51
4.13 BioPerf - Concurrent and Parallel Dynamic Compiler Performance Results 55
4.14 Spec Cpu 2006, Eembc and CoreMark - Concurrent and Parallel Dynamic Compiler Performance Results 57
4.15 Concurrent and Parallel Dynamic Compilation Scalability Results 58
4.16 Concurrent and Parallel Dynamic Compiler Throughput . . . 59
4.17 Splash-2 - Region Sharing Optimisation Results 61
4.18 Splash-2 - Region Sharing Optimisation Scope, Thread-agnostic vs. Thread-specific Performance Comparison 63

5.1 Concurrent and Parallel Dynamic Compilation Flow for Microarchitectural Simulation 67
5.2 Dynamic Translation of Basic Block using Microarchitectural State Updates 69
5.3 EnCore Processor Pipeline Microarchitecture 71

5.4 Dynamically Compiled Microarchitectural Software Model . . 72
5.5 Microarchitectural Pipeline Model in Action 73
5.6 5-Stage Pipeline - EEMBC and CoreMark - Microarchitectural Performance Modelling Results 77
5.7 5-Stage Pipeline - BioPerf - Microarchitectural Performance Modelling Results 78
5.8 7-Stage Pipeline - EEMBC and CoreMark - Microarchitectural Performance Modelling Results 79
5.9 7-Stage Pipeline - BioPerf - Microarchitectural Performance Modelling Results 80

List of Tables

4.1 Simulation Host Configuration - Concurrent and Parallel Dynamic Compilation Experiment 53

5.1 Simulated Architecture and Microarchitecture Configuration - Microarchitectural Simulation Experiment 70
5.2 Simulation Host Configuration - Microarchitectural Simulation Experiment 76

List of Listings

4.1 Dynamic Work Scheduling Based on Recency and Frequency . 40
4.2 Adaptive Hotspot Threshold Selection and Compilation Unit Dispatch 41

Bibliography

[1] *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. Dec. 2007. URL: http://dl.acm.org/citation.cfm?id=1557467.

[2] B. Alexander, S. Donnellan, A. Jeffries, and T. Olds. Boosting Instruction Set Simulator Performance with Parallel Block Optimisation and Replacement. In *ACSC'12: Proceedings of the 35 Australasion Computer Science Conference*, 2012. URL: http://crpit.com/confpapers/CRPITV122Alexander.pdf.

[3] O. Almer, I. Böhm, T. von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham. Scalable multi-core simulation using parallel dynamic binary translation. In *SAMOS'11: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 190–199, 2011. URL: http://dx.doi.org/10.1109/SAMOS.2011.6045461.

[4] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI'11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, June 2011. URL: http://dx.doi.org/10.1145/1993498.1993540.

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Architecture and Policy for Adaptive Optimization in Virtual Machines. Technical Report RC23429 (W0411-125), Nov. 2004. URL: http://researchweb.watson.ibm.com/people/h/hind/RC23429.pdf.

[6] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. In *Computer Architecture Letters*, pages 45–48, 2007. URL: http://dx.doi.org/10.1109/L-CA.2007.12.

[7] J. Aycock. A brief history of just-in-time. In *ACM Computing Surveys (CSUR)*, June 2003. URL: http://dx.doi.org/10.1145/857076.857077.

[8] D. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *IISWC'05: Proceedings of the IEEE International Symposium on Workload Characterization*, pages 163–173, 2005. URL: http://dx.doi.org/10.1109/IISWC.2005.1526013.

[9] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI'00: Proceedings of the ACM Conference on Programming Language Design and Implementation*, Aug. 2000. URL: http://dx.doi.org/10.1145/349299.349303.

[10] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *CODES'00: Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, pages 82–86, May 2000. URL: http://dx.doi.org/10.1145/334012.334028.

[11] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-Based Compilation in Execution Environments without Interpreters. In *PPPJ'21: International Conference on the Principles and Practice of Programming in Java*, pages 1–10, Sept. 2010. URL: http://dx.doi.org/10.1145/1852761.1852771.

[12] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATEC'05: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005. URL: http://www.usenix.org/event/usenix05/tech/freenix/full_papers/bellard/bellard_html/.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. In *SIGARCH Computer Architecture News*, Aug. 2011. URL: http://dx.doi.org/10.1145/2024716.2024718.

[14] I. Böhm, B. Franke, and N. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *SAMOS'10: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 1–10, 2010. URL: http://dx.doi.org/10.1109/ICSAMOS.2010.5642102.

[15] I. Böhm, B. Franke, and N. Topham. Cycle-Accurate Performance Modelling in an Ultra-Fast Just-In-Time Dynamic Binary Translation Instruction Set Simulator. *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC'11).*, 5(4), 2011. URL: http://www.hipeac.net/system/files?file=main_0.pdf.

[16] I. Böhm, T. J. K. E. von Koch, S. C. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *PLDI'11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, June 2011. URL: http://dx.doi.org/10.1145/1993498.1993508.

[17] G. Bontempi and W. Kruijtzer. A Data Analysis Method for Software Performance Prediction. In *DATE'02: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 971–976, Mar. 2002. URL: http://dx.doi.org/10.1109/DATE.2002.998417.

[18] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler. Fast and Accurate Simulation using the LLVM Compiler Framework . In *RAPIDO'09: 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pages 1–6, Dec. 2009.

[19] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1625–1639, 2004. URL: http://dx.doi.org/10.1109/TCAD.2004.836734.

[20] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, Massachusetts Institute of Technology, 2003. URL: http://www.burningcutlery.com/derek/docs/phd.pdf.

[21] D. Bruening. Thread-shared software code caches. In *CGO'06: Proceedings of the 4th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2006. URL: http://dx.doi.org/10.1109/CGO.2006.36.

[22] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO'03: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 265–275, Mar. 2003. URL: http://dx.doi.org/10.1109/CGO.2003.1191551.

[23] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. In *SIGARCH Computer Architecture News*, June 1997. URL: http://dx.doi.org/10.1145/268806.268810.

[24] H. W. Cain, K. M. Lepak, and M. H. Lipasti. A dynamic binary translation approach to architectural simulation. *SIGARCH Computer Architecture News*, 29(1), Mar. 2001. URL: http://dx.doi.org/10.1145/373574.373586.

[25] S. Campanoni, G. Agosta, and S. Reghizzi. A parallel dynamic compiler for CIL bytecode. In *SIGPLAN Notices*, Apr. 2008. URL: http://dx.doi.org/10.1145/1374752.1374754.

[26] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA'89: Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 49–70, New Orleans, LA, Oct. 1989. URL: http://dx.doi.org/10.1145/74877.74884.

[27] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates. FX!32 a profile-directed binary translator. *IEEE MICRO*, 18(2):56–64, 1998. URL: http://dx.doi.org/10.1109/40.671403.

[28] D. Chiou, H. Angepat, N. Patil, and D. Sunwoo. Accurate Functional-First Multicore Simulators. In *Computer Architecture Letters*, pages 64–67, 2009. URL: http://dx.doi.org/10.1109/L-CA.2009.44, doi:10.1109/L-CA.2009.44.

[29] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *MICRO'07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007. URL: http://dx.doi.org/10.1109/MICRO.2007.36.

[30] C. Click. Tiered Compilation. *jayView - Passion for Software*, 22(2):18–21, 2010. URL: http://www.jayway.com/media/28083/jayview22.pdf.

[31] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS'94: Proceedings of the ACM*

SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1994. URL: http://dx.doi.org/10.1145/183018.183032.

[32] A. Cohen and E. Rohou. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *DAC'10: Proceedings of the 47th Design Automation Conference*, June 2010. URL: http://dx.doi.org/10.1145/1837274.1837303.

[33] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1996. URL: http://dx.doi.org/10.1145/237721.237767.

[34] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL'84: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Jan. 1984. URL: http://dx.doi.org/10.1145/800017.800542.

[35] K. Ebcioğlu and E. R. Altman. DAISY: dynamic compilation for 100 In *ISCA'97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997. URL: http://dx.doi.org/10.1145/264107.264126.

[36] Embedded Microprocessor Benchmark Consortium. EEMBC - The Embedded Microprocessor Benchmark Consortium [online]. Dec. 2011. URL: http://www.eembc.org/.

[37] S. Farfeleder, A. Krall, and N. Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *Journal of Systems Architecture*, 53(8):501–510, Aug. 2007. URL: http://dx.doi.org/10.1016/j.sysarc.2006.11.003, doi:10.1016/j.sysarc.2006.11.003.

[38] B. Franke. Fast cycle-approximate instruction set simulation. In *SCOPES'08: Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems*, Mar. 2008. URL: http://doi.acm.org/10.1145/1361096.1361109.

[39] A. Gal, M. Bebenita, M. Chang, and M. Franz. Making the Compilation "Pipeline" Explicit: Dynamic Compilation Using Trace

Tree Serialization. Technical Report ICS-TR-07-12, Computer Science Department University of California, Irvine Irvine, CA, 92697, USA, Irvine, 2007. URL: http://www.ics.uci.edu/~franz/Site/pubs-pdf/ICS-TR-07-12.pdf.

[40] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI'09: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009. URL: http://dx.doi.org/10.1145/1542476.1542528.

[41] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE'06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, June 2006. URL: http://dx.doi.org/10.1145/1134760.1134780.

[42] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Multiprocessor performance estimation using hybrid simulation. In *DAC'08: Proceedings of the 45th annual Design Automation Conference*, June 2008. URL: http://dx.doi.org/10.1145/1391469.1391552.

[43] L. Gao, S. Kraemer, K. Karuri, R. Leupers, G. Ascheid, and H. Meyr. An Integrated Performance Estimation Approach in a Hybrid Simulation Framework. In *MOBS'08: Annual Workshop on Modeling, Benchmarking and Simulation*, 2008. URL: http://www-mount.ece.umn.edu/~jjyi/MoBS/2008/program/02D-Gao.pdf.

[44] L. Gao, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. A fast and generic hybrid simulation approach using C virtual machine. In *CASES'07: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Sept. 2007. URL: http://dx.doi.org/10.1145/1289881.1289885.

[45] Google Inc. Design Elements - V8 JavaScript Engine [online]. 2012. URL: http://code.google.com/intl/sv/apis/v8/design.html.

[46] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *PLDI'99: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999. URL: http://dx.doi.org/10.1145/301618.301683.

[47] J. Ha, M. R. Haghighat, S. Cong, and K. S. Mckinley. A Concurrent Trace-based Just-In-Time Compiler for Single-threaded JavaScript. In *PESPMA'01: Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, 2009. URL: http://www.east.isi.edu/~jha/papers/cjit-pespma09.pdf.

[48] G. Hamerly, E. Perelman, and J. Lau. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction-Level Parallelism*, 2005. URL: http://www.jilp.org/vol7/v7paper14.pdf.

[49] C. Häubl and H. Mössenböck. Trace-based Compilation for the Java HotSpot Virtual Machine. In *PPPJ'11: Proceedings of the International Conference on Principles and Practice of Programming in Java*, pages 1–10, Kongens Lyngby, Denmark, Aug. 2011. URL: http://dx.doi.org/10.1145/2093157.2093176.

[50] C. Häubl, C. Wimmer, and H. Mössenböck. Evaluation of Trace Inlining Heuristics for Java. In *SAC'12: Symposium on Applied Computing*, pages 1–6, Mar. 2012. URL: http://dx.doi.org/10.1145/2245276.2232084.

[51] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-priented applications. In *OOPSLA'04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2004. URL: http://dx.doi.org/10.1145/1028976.1028998.

[52] J. L. Henning. SPEC CPU2006 benchmark descriptions. In *SIGARCH Computer Architecture News*, Sept. 2006. URL: http://dx.doi.org/10.1145/1186736.1186737.

[53] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Lecture Notes In Computer Science*, pages 21–38, Berlin/Heidelberg, 1991. URL: http://www.springerlink.com/index/10.1007/BFb0057013.

[54] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI'92: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, July 1992. URL: http://dx.doi.org/10.1145/143095.143114.

[55] N. Horspool and N. Marovac. An Approach to the Problem of Detranslation of Computer Programs. *The Computer Journal*, 23:223–229, Aug. 1980. URL: http://dx.doi.org/10.1093/comjnl/23.3.223.

[56] C.-C. Hsu, P. Liu, J.-J. Wu, C.-M. Wang, D.-Y. Hong, and W.-C. Hsu. Improving Region Selection Through Early-Exit Detection. Technical Report TR-IIS-12-003, Taipei, Apr. 2012. URL: http://www.iis.sinica.edu.tw/page/library/TechReport/tr2012/tr12003.pdf.

[57] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *CGO'11: Proceedings of the 9th International Symposium on Code Generation and Optimization*, pages 246–256, 2011. URL: http://dx.doi.org/10.1109/CGO.2011.5764692.

[58] D. Jones. *High Speed Simulation of Microprocessor Systems Using LTU Dynamic Binary Translation.* PhD thesis, University of Edinburgh: U.K, Jan. 2010. URL: http://hdl.handle.net/1842/4609.

[59] D. Jones and N. Topham. High Speed CPU Simulation Using LTU Dynamic Binary Translation. In *HiPEAC'09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 50–64. Institute for Computing Systems Architecture School of Informatics, University of Edinburgh, Great Britain, 2009. URL: http://dx.doi.org/10.1007/978-3-540-92990-1_6.

[60] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1), May 2008. URL: http://dx.doi.org/10.1145/1369396.1370017.

[61] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr. HySim: a fast simulation framework for embedded software development. In *CODES+ISSS'07: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, Sept. 2007. URL: http://dx.doi.org/10.1145/1289816.1289837.

[62] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *PLDI'01: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001. URL: http://dx.doi.org/10.1145/378795.378831.

[63] C. J. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, 2001. URL: `http://doi.wiley.com/10.1002/spe.384`.

[64] P. Kulkarni, M. Arnold, and M. Hind. Dynamic compilation: the benefits of early investing. In *VEE'07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, June 2007. URL: `http://dx.doi.org/10.1145/1254810.1254824`.

[65] P. A. Kulkarni. JIT compilation policy for modern machines. In *OOPSLA'11: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Oct. 2011. URL: `http://dx.doi.org/10.1145/2048066.2048126`.

[66] S. Kyle, I. Böhm, B. Franke, H. Leather, and N. Topham. Efficiently parallelizing instruction set simulation of embedded multi-core processors using region-based just-in-time dynamic binary translation. In *LCTES'12: Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, pages 1–11, June 2012. URL: `http://dx.doi.org/10.1145/2248418.2248422`, `doi:10.1145/2248418.2248422`.

[67] M. Lagergren and M. Hirt. *Oracle JRockit - The Definitive Guide*. 2010.

[68] R. E. Lantz. Fast Functional Simulation with Parallel Embra. In *MOBS'08: Annual Workshop on Modeling, Benchmarking and Simulation*. Computer Systems Laboratory Stanford University, 2008. URL: `http://www-cs-students.stanford.edu/~rlantz//papers/lantz-mobs08-final.pdf`.

[69] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04: Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2004. URL: `http://dx.doi.org/10.1109/CGO.2004.1281665`.

[70] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *PLDI'96: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1996. URL: `http://dx.doi.org/10.1145/231379.231407`.

[71] P. Lowney, S. Freudenberger, and T. Karzes. The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7:51–142, 1993. URL: http://www.springerlink.com/index/R567618268550493.pdf.

[72] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005. URL: http://dx.doi.org/10.1145/1065010.1065034.

[73] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002. URL: http://dx.doi.org/10.1109/2.982916.

[74] C. May. Mimic: a fast system/370 simulator. In *SIGPLAN'87: Papers of the Symposium on Interpreters and Interpretive Techniques*, July 1987. URL: http://dx.doi.org/10.1145/29650.29651.

[75] Microsoft Corporation. The .NET Common Language Runtime [online]. May 2012. URL: http://msdn.microsoft.com/en-us/netframework/.

[76] C. Mills, S. C. Ahalt, and J. Fowler. Compiled instruction set simulation. In *Software: Practice and Experience*, pages 877–889, Aug. 1991. URL: http://doi.wiley.com/10.1002/spe.4380210807.

[77] Mozilla Inc. Tamarin JavaScript Engine [online]. 2012. URL: https://developer.mozilla.org/en/Tamarin.

[78] A. Muttreja, A. Raghunathan, S. Ravi, and N. Jha. Hybrid simulation for embedded software energy estimation. In *DAC'05: Proceeding of the 42nd Design Automation Conference*, pages 23–26, 2005. URL: http://dx.doi.org/10.1145/1065579.1065590, doi:10.1109/DAC.2005.245623.

[79] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha. Hybrid simulation for energy estimation of embedded software. In *DAC'05: Proceedings of the 42nd Annual Design Automation Conference*, pages 23–26, 2005. URL: http://dx.doi.org/10.1145/1065579.1065590.

[80] Oracle Corporation. Oracle HotSpot JVM [online]. Dec. 2011. URL: http://www.oracle.com/technetwork/java/javase/tech/hotspot-138757.html.

[81] M. S. Oyamada, F. Zschornack, and F. R. Wagner. Accurate software performance estimation using domain classification and neural networks. In *SBCCI'04: Proceedings of the 17th Symposium on Integrated Circuits and System Design*, Sept. 2004. URL: http://dx.doi.org/10.1145/1016568.1016617.

[82] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *JVM'01: Proceedings of the Symposium on Java Virtual Machine Research and Technology Symposium*, 2001. URL: http://dl.acm.org/citation.cfm?id=1267848.

[83] PASTA Research Group. PASTA: EnCore CPU [online]. 2011. URL: http://groups.inf.ed.ac.uk/pasta/hw_encore.html.

[84] M. P. Plezbert and R. K. Cytron. Does "just in time" = "better late than never"? In *POPL'97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1997. URL: http://dx.doi.org/10.1145/263699.263713.

[85] M. Poletto, D. R. Engler, M. F. Kaashoek, M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: a system for fast, flexible, and high-level dynamic code generation. In *PLDI'97: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 109–121, 1997. URL: http://dx.doi.org/10.1145/258915.258926.

[86] D. Powell and B. Franke. Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators. In *CODES+ISSS'09: Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, Oct. 2009. URL: http://dx.doi.org/10.1145/1629435.1629478.

[87] W. Qin, J. D'Errico, and X. Zhu. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *CODES+ISSS'06: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 193–198, Oct. 2006. URL: http://dx.doi.org/10.1145/1176254.1176302.

[88] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *DAC'03: Proceedings of the Design Automation Conference*, pages 758–763, 2003. URL: http://dx.doi.org/10.1109/DAC.2003.1219121.

[89] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. In *Computer Architecture Letters*, pages 16–19, 2011. URL: http://dx.doi.org/10.1109/L-CA.2011.4, doi:10.1109/L-CA.2011.4.

[90] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture implementation using the machine description language LISA. In *ASP-DAC'02: Proceedings of 7th Asia and South Pacific Design Automation and Test and the 15th International Conference on VLSI Design*, pages 239–244, 2002. URL: http://dx.doi.org/10.1109/ASPDAC.2002.994928.

[91] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. Technical report, 2001. URL: http://investigacion.ac.upc.es/conferencies/PACT01/wbt/davidson.pdf.

[92] SimIt-ARM Inc. SimIt-ARM [online]. 2007. URL: http://simit-arm.sourceforge.net/.

[93] J. Smith and R. Nair. *Virtual Machines*. Versatile Platforms for Systems and Processes. San Francisco, 1 edition, June 2005.

[94] E. Stahl. A comparison of PowerVM and x86-based virtualization performance. Technical Report WP101574, 2009. URL: ftp://dispsd-40-www3.boulder.ibm.com/common/ssi/sa/wh/n/pow03029usen/POW03029USEN.PDF.

[95] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Transactions on Programming Languages and Systems*, 27(4), July 2005. URL: http://dx.doi.org/10.1145/1075382.1075386.

[96] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems*, 28(1), Jan. 2006. URL: http://dx.doi.org/10.1145/1111596.1111600.

[97] V. Sundaresan, D. Maier, and P. Ramarao. Experiences with multi-threading and dynamic class loading in a Java just-in-time compiler. In *CGO'06: International Symposium on Code Generation and Optimization*, 2006. URL: http://dx.doi.org/10.1109/CGO.2006.16.

[98] Synopsys Inc. DesignWare ARC Processor Cores [online]. 2011. URL: http://www.synopsys.com/IP/ProcessorIP/ARCProcessors/.

[99] Synopsys Inc. DesignWare ARC xISS [online]. 2012. URL: http://www.synopsys.com/dw/ipdir.php?ds=sim_xiss.

[100] V. Tan. Asynchronous just-in-time compilation. Patent Number G06F 9/45E9:Patent Number G06F 9/45–11, June 2008.

[101] N. Topham and D. Jones. High Speed CPU Simulation using JIT Binary Translation. In *MOBS'07: Annual Workshop on Modeling, Benchmarking and Simulation*, pages 1–9, 2007.

[102] D. Ung and C. Cifuentes. Machine-adaptable dynamic binary translation. In *DYNAMO'00: Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Jan. 2000. URL: http://dx.doi.org/10.1145/351397.351414.

[103] P. Unnikrishnan, M. Kandemir, and E. Li. ASP-DAC'06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference. In *ASP-DAC'06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, 2006. URL: http://dx.doi.org/10.1109/ASPDAC.2006.1594805.

[104] H. G. Vélez. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 2010. URL: http://dx.doi.org/10.1002/spe.v40:12.

[105] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic. RAMP: Research Accelerator for Multiple Processors. *IEEE MICRO*, 27(2):46–57, 2007. URL: http://dx.doi.org/10.1109/MM.2007.39.

[106] WebKit Inc. SquirrelFish JavaScript Engine [online]. 2009. URL: http://trac.webkit.org/wiki/SquirrelFish.

[107] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *JAVA'00: Proceedings of the ACM 2000 Conference on Java Grande*, June 2000. URL: http://dx.doi.org/10.1145/337449.337483.

[108] J. Whaley. Partial method compilation using dynamic profile information. In *OOPSLA'01: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nov. 2001. URL: http://dx.doi.org/10.1145/504282.504295.

[109] C. Wimmer, M. S. Cintra, M. Bebenita, M. Chang, A. Gal, and M. Franz. Phase detection using trace compilation. In *PPPJ'09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, page 172, New York, New York, USA, 2009. URL: http://dx.doi.org/10.1145/1596655.1596683.

[110] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA'95: Proceedings on the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995. URL: http://dx.doi.org/10.1145/223982.223990.

[111] D. B. Wortman and M. D. Junkin. A concurrent compiler for Modula-2+. In *PLDI'92: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, July 1992. URL: http://dx.doi.org/10.1145/143095.120025.

[112] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA'03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–95, May 2003. URL: http://dx.doi.org/10.1109/ISCA.2003.1206991.

[113] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009. URL: http://dx.doi.org/10.1109/SP.2009.25, doi:10.1109/SP.2009.25.

[114] M. Zaleski, A. D. Brown, and K. Stoodley. YETI: a graduallY extensible trace interpreter. In *VEE'07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, June 2007. URL: http://dx.doi.org/10.1145/1254810.1254823.

[115] C. Zheng and C. Thompson. PA-RISC to IA-64: transparent execution, no recompilation. *Computer*, 33(3):47–52, 2000. URL: http://dx.doi.org/10.1109/2.825695.

[116] J. Zhu and D. Gajski. A retargetable, ultra-fast instruction set simulator. In *DAC'99: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 298–302, 1999. URL: `http://dx.doi.org/10.1109/DATE.1999.761137`.