Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-Core Processors Using Region-based Just-in-Time Dynamic Binary Translation

Stephen Kyle Igor Böhm Björn Franke Hugh Leather Nigel Topham

Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom
s.kyle@ed.ac.uk, i.bohm@sms.ed.ac.uk, bfranke@inf.ed.ac.uk, hleather@inf.ed.ac.uk, npt@inf.ed.ac.uk

Abstract

Embedded systems, as typified by modern mobile phones, are already seeing a drive toward using multi-core processors. The number of cores will likely increase rapidly in the future. Engineers and researchers need to be able to simulate systems, as they are expected to be in a few generations time, running simulations of many-core devices on today's multi-core machines. These requirements place heavy demands on the scalability of simulation engines, the fastest of which have typically evolved from just-in-time (JIT) dynamic binary translators (DBT).

Existing work aimed at parallelizing DBT simulators has focused exclusively on trace-based DBT, wherein linear execution traces or perhaps trees thereof are the units of translation. Region-based DBT simulators have not received the same attention and require different techniques than their trace-based cousins.

In this paper we develop an innovative approach to scaling multi-core, embedded simulation through region-based DBT. We initially modify the JIT code generator of such a simulator to emit code that does not depend on a particular thread with its thread-specific context and is, therefore, *thread-agnostic*. We then demonstrate that this thread-agnostic code generation is comparable to thread-specific code with respect to performance, but also enables the *sharing* of JIT-compiled regions between different threads. This sharing optimisation, in turn, leads to significant performance improvements for multi-threaded applications. In fact, our results confirm that an average of 76% of all JIT-compiled regions can be shared between 128 threads in representative, parallel workloads. We demonstrate that this translates into an overall performance improvement by 1.44x on average and up to 2.40x across 12 multi-threaded benchmarks taken from the SPLASH-2 benchmark suite, targeting our high-performance multi-core DBT simulator for embedded ARC processors running on a 4-core Intel host machine.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Incremental Compilers, code generation

General Terms Design, experimentation, performance

Keywords Dynamic binary translation, parallelization

LCTES'12 12-13 June 2012, Beijing, China.

1. Introduction

Dynamic Binary Translation (DBT) is an efficient technique to achieve high performance instruction set simulation of one target architecture on another host architecture with possibly different instruction sets. Such simulators might be aimed at only functional simulation as in QEMU[3] or cycle accurate or both[5].

In DBT, a thread is interpreted and frequently executed (hot) areas of code are identified for JIT compilation and, therefore, native execution. When a single core is being simulated, 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. This paper examines the optimal strategies for just such a parallelization of one of the two forms of DBT simulators. The two forms of DBT compilation are *trace-based* and *region-based*, and while some work has investigated parallelization schemes for trace-based systems, the same cannot be said for the potentially more powerful region-based ones. This paper is the first paper to do such an investigation.

The *trace-based* variants of DBT gather linear traces of basic blocks from a deemed hot entry point, called the trace head. These linear traces are sometimes collected into trees, with branches compiled separately and patched together. Due to the linearization of control flow in which basic blocks may be replicated in many different traces, trace-based systems may suffer from code explosion[2]. The great advantage of trace-based methods is that the linearization of control flow makes compilation and data-flow analysis particularly simple, allowing the compilers to be easy to write, and to have extremely small footprints. However, the benefit of complex optimizations working over larger control flow graphs (CFG) is lost.

On the other hand, *region-based* systems operate on hot, arbitrarily shaped sub-graphs of the CFG. Their compilers are necessarily more complicated and difficult to write, but may yield additional optimisation opportunities that are unavailable to their trace-based cousins. The advent of retargetable, reusable JIT systems such as LLVM, has meant that compiler complexity is no longer detrimental to the development time of region-based systems. Indeed, even trace-based systems are being retrofitted with such off the shelf compilers[5].

In trace-based systems, a number of approaches to sharing translations and efficient synchronization have been tried. A naïve

approach to trace-based JIT compilation would operate on a single, global trace data structure, which would need to be protected from concurrent updates using an expensive locking mechanism. This approach has been taken in e.g. [19]. An alternative approach that avoids this problem is to maintain thread-private traces [13, 18], 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 JIT subsystem and does not scale.

In region-based DBT, no work has investigated different methods to parallelize the simulation environment. The work closest to ours is not region-based but trace-based. Bruening et al.[6] considered the benefits of sharing linear traces over keeping private traces. In their work, 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 the compilation. In addition, because their 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, potentially with multiple entry points per region. 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 paper, we develop a novel and scalable scheme for region-based JIT 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 effective and scalable.

We have implemented and evaluated our proposed technique using the ARCSIM dynamic binary translator targeting a multi-core platform comprising up to 128 embedded cores, where each core is capable of supporting a single application thread using the POSIX threading API. We show that thread-agnostic region code generation does not incur any performance penalty, although it may prohibit some minor opportunities for code optimisation. We demonstrate that region sharing, however, provides great potential for performance improvement. An average 76% of all JIT-compiled regions can be shared in the SPLASH-2 benchmarks when run with 128 threads. This results in an overall performance improvement of 1.44x averaged across all benchmarks and over a state-of-the-art thread-private region compilation approach without region sharing.

1.1 Region-Recording for Multi-Threaded Applications

Before we take a more detailed look at our approach to scaling region-based JIT compilation for multi-threaded applications, we provide a comparison of state-of-the-art recording schemes used in existing DBT systems, both tracing and region based, to highlight our key concepts (see Figure 1).

Recording of execution paths in terms of traces is either triggered by detecting a special construct (e.g. loop header, method entry) [2, 15, 17, 19, 36], or always enabled when interpreting code [4]. Various backbone data structures have been suggested to capture recorded traces such as trace-trees [14], control-flow-graphs (CFG) of traced basic blocks [4], or hybrids between trace-trees and CFGs called trace-regions [2] or trace-graphs [18]. In general, recording approaches for multi-threaded applications can be categorized based on how CFG recordings are accessed and constructed:

- *Global Recording Structure* - Most [2, 14, 15, 17–19, 36] JIT compilation systems use one shared global recording structure (① in Figure 1) to incrementally build CFG sections. This scheme works well for single-threaded execution environments but does not scale to multi-threaded applications as additional synchronisation is required when recording in parallel for multiple threads.
- *Local Recording Structures* - Another approach is to use private, local structures for each thread (② in Figure 1). While this approach avoids synchronisation altogether, it causes threads executing data-parallel sections of code to independently construct nearly identical code paths and compile multiple *thread-specific* CFG sections specialised for each thread.

Our approach builds on ② and extends it to compile *thread-agnostic* regoins for data-parallel sections of code *once* and share the compiled region with all threads that profiled the code region (see ③ in Figure 1). Consequently, the pressure on the underlying JIT 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 JIT compilation for multi-threaded execution environments a step further. Using a lock free region recording strategy and a JIT 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 JIT scheme is highly scalable.

1.2 Motivating Example

Consider the multi-threaded benchmark `water-spatial` from the SPLASH-2 benchmark suite, when executed with 128 threads in our dynamic binary translator. Of all the regions handled by the asynchronous JIT subsystem, 79% were actually similar to previously requested regions (see ② in Figure 2). This shows that there is a large potential saving to be made in sharing regions between threads. We demonstrate that our approach leads to a speed-up of 2.4x for this benchmark (see ① in Figure 2). This is made possible through the use of regions that can be executed by any thread – these are said to be *thread-agnostic* (see ④ in Figure 2). These regions then allow us to develop a scheme where commonality between regions is identified as they are dispatched for JIT compilation, allowing multiple threads to use the result of the region translation. This will reduce the amount of time threads have to continue executing in interpreted mode until a region is JIT compiled. It enables earlier execution of native code and reduces the pressure on the JIT compilation subsystem. As a result, the total JIT compilation time for this benchmark is reduced by 73%.

1.3 Contributions

In this paper, we present the following contributions to improving the performance of multi-threaded DBT when using region-based JIT compilation:

1. The use of *thread-agnostic* regions, that are sections of JIT-compiled code using indirection to access an executing thread's state. This region can then be executed by multiple threads wishing to execute the same region of code,
2. a strategy for sharing translated regions between application threads, to both reduce pressure on the JIT subsystem, and

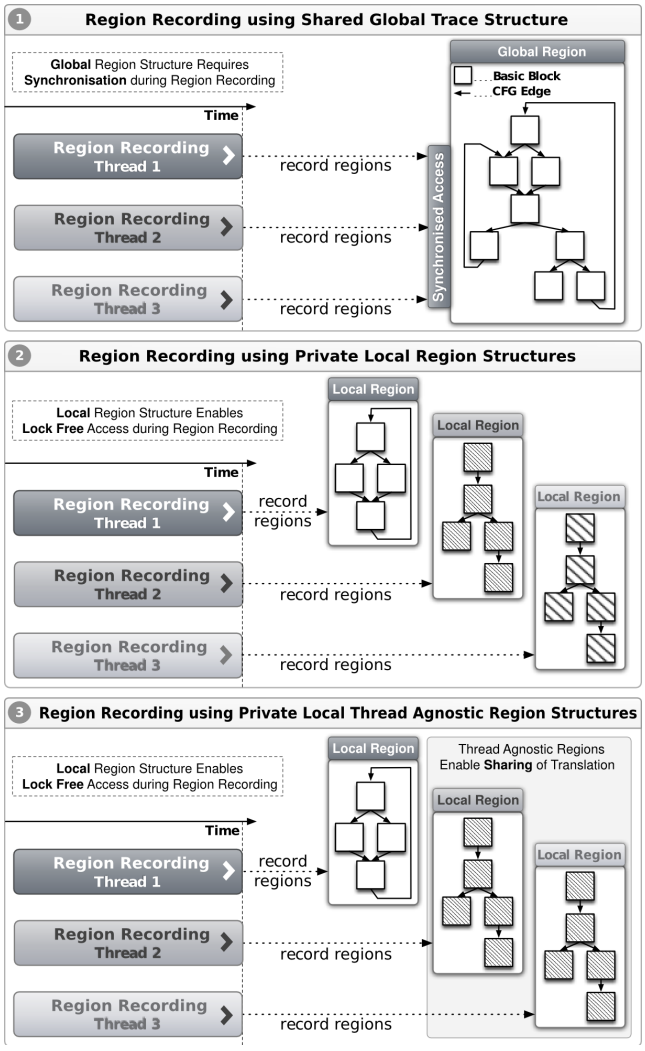

Figure 1. Comparison of recording approaches for multithreaded applications: ① recording using a shared global structure [2, 15, 17, 19, 36] requiring synchronisation, ② lock free recording using private local structures. Finally, ③ shows our lock free region recording approach using private local thread-agnostic region structures.

ensure that more threads start executing native code earlier. This is done through the identification of common regions between threads, as well as caching of region translations, and

3. an extensive evaluation of performance improvements gained from using this technique using the SPLASH-2 benchmark suite.

1.4 Overview

The rest of this paper is outlined as follows. In Section 2, the design of our dynamic binary translator and the paper's contributions are explained in detail, with the performance improvements resulting from these contributions presented in Section 3. In Section 4, related work in dynamic binary translation, trace-based and region-based JIT compilation is discussed, and in Section 5 we summarise and conclude.

2. Methodology

We begin this section with a description of our multi-core DBT system, focusing on its parallel task farm based JIT subsystem. The design of *thread-specific* and *thread-agnostic* regions is then contrasted, followed by a description of how the DBT has been extended to enable sharing of regions between threads of execution.

2.1 Multi-core DBT Design

In this paper, we have modified the operation of our dynamic binary translator, called ARCSIM. This DBT allows the execution of binaries built for the ARCOMPACT instruction set on a host machine implementing a different host ISA. ARCSIM combines interpreted execution and native execution following JIT compilation of the most frequently executed regions of code. JIT compilation is performed asynchronously to the interpretation of the program, as a parallel task farm based JIT subsystem has been added to the DBT.

Light-weight tracing is always enabled when code is interpreted, recording the control flow exhibited by the source program at run-time. Interpretation is partitioned into intervals (see Figure 3), the length of which is determined by a user-defined number of interpreted instructions. At the end of each interval frequently executed regions are dispatched to a region translation *priority queue*, before continuing to interpret the program. A detailed description of the priority function used to order regions can be found in [4]. JIT workers operating in parallel threads can then claim a region from the queue, JIT compile it, and update the region translation state, thereby indicating the availability of native code for that recorded region to the interpreter loop.

ARCSIM has the ability to emulate multiple processors executing concurrently in a shared-memory environment. Each processor runs in a separate thread on the host machine, and records its own internal representation of the target binary, dispatching hot regions to a global priority queue. The JIT subsystem operates as before, except each work unit dispatched to the queue is tagged with the requesting processor. Upon translating the region to native code, the JIT worker updates the region translation state of the processor that dispatched the region.

2.2 Thread-Agnostic Regions

In order 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 may execute the 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 ③ in Figure 2). 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 ④ in Figure 2). In this case, both threads can use the same native code 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. In Section 3.5 we compare the performance of these two approaches and demonstrate that – contrary to intuition – thread-agnostic code generation does not incur any performance penalty.

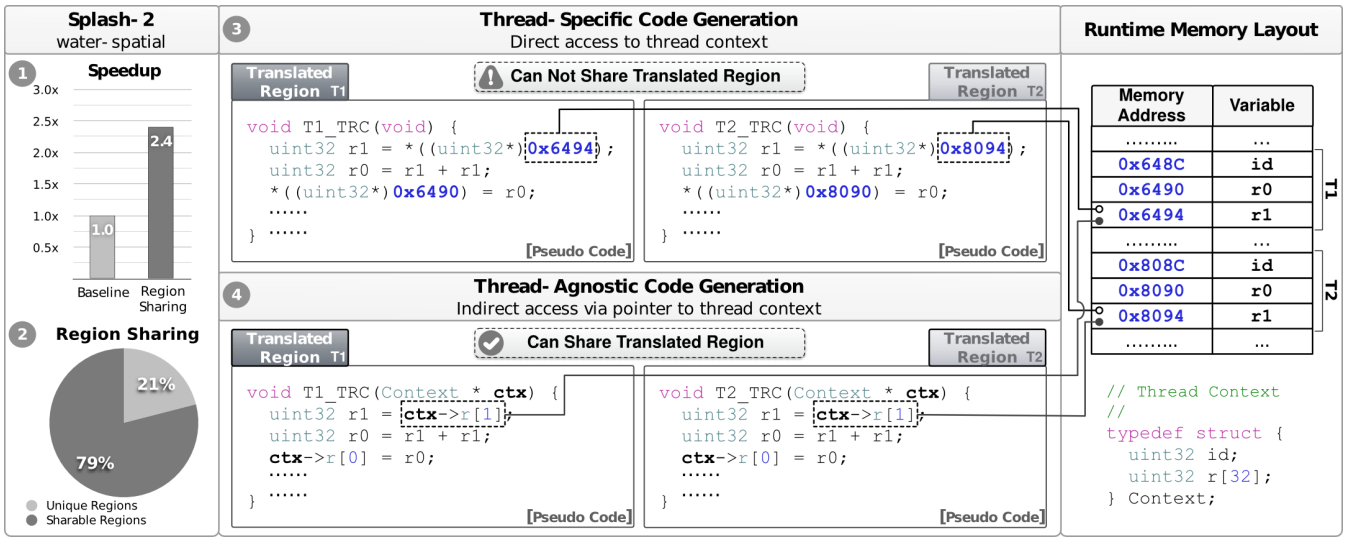

Figure 2. SPLASH-2 water-spatial benchmark - ① demonstrates the achievable speed-up using our novel region sharing optimisation, while ② shows the scope of region sharing for this benchmark. The key difference between between ③ *thread-specific* and ④ *thread-agnostic* code generation is highlighted, namely code generation independent of thread context, *enabling* our region sharing optimisation.

2.3 Inter-thread Region Sharing

As regions are dispatched to the priority queue of the asynchronous JIT subsystem, 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, we generate signatures for each region as it is constructed. The signature is the 32-bit result of a hash function applied to the physical addresses of all basic block entry points in the region. Only if two signatures match a more expensive check for equality needs 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 priority queue. For a given key signature, this table stores all threads that are interested in the associated region that is currently waiting to be handled by the JIT subsystem. Adding any region that matches signatures with a region already in the JIT subsystem 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 JIT workers can update the requesting thread with the result of the JIT compilation (see ① Figure 3).

The JIT compilation workers continue to fetch regions from the queue. When handling a region, the worker checks the hash table for the threads that are interested in this region, allowing them to be updated with the native code generated from the region. See Figure 3 for an example of an overview of our DBT design, as well as how regions can be shared between threads.

This technique has the dual effect of reducing the period many threads must wait between the dispatch of a region and the receipt of a translation. It also reduces the number of similar regions in the priority queue, thereby reducing the pressure on the underlying JIT subsystem.

2.4 Region Translation Caching

The previous section describes how we can share regions if a thread attempts to dispatch a region while a similar region is currently in the JIT subsystem. What if a particular thread reaches a hot region of code much later than other threads? In this case, there are no similar regions currently in the JIT subsystem, so a JIT 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 JIT compiled. The drawback of this solution is that translations might be registered for threads that have not yet executed that specific code path, thereby adding complexity to the tracing 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 each JIT compiled region to this region translation cache. When a region is dispatched for JIT compilation we first check if a translation is already present in the region translation cache. If so, all native code generation can be skipped, and the translation can be registered for the thread that dispatched the region (see ② Figure 3). Region translation caching thereby removes redundant re-compilation from the critical path of execution for many threads, improving overall performance.

3. Empirical Evaluation

In this paper, we use the SPLASH-2 benchmark suite [38] to evaluate the performance benefits resulting from the use of region sharing to improve region-based JIT compilation in multi-threaded execution environments. 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.

In the remainder of this section, we describe our experimental set-up, before presenting performance improvements as well as throughput measurements and data relating to the potential for region sharing in each benchmark.

3.1 Experimental Set-up

A machine running the operating system Scientific Linux 5.5 with the following specifications was used to perform all experiments: a 4-core Intel Xeon E5430 running at 2.66Ghz, with 8GB of memory available.

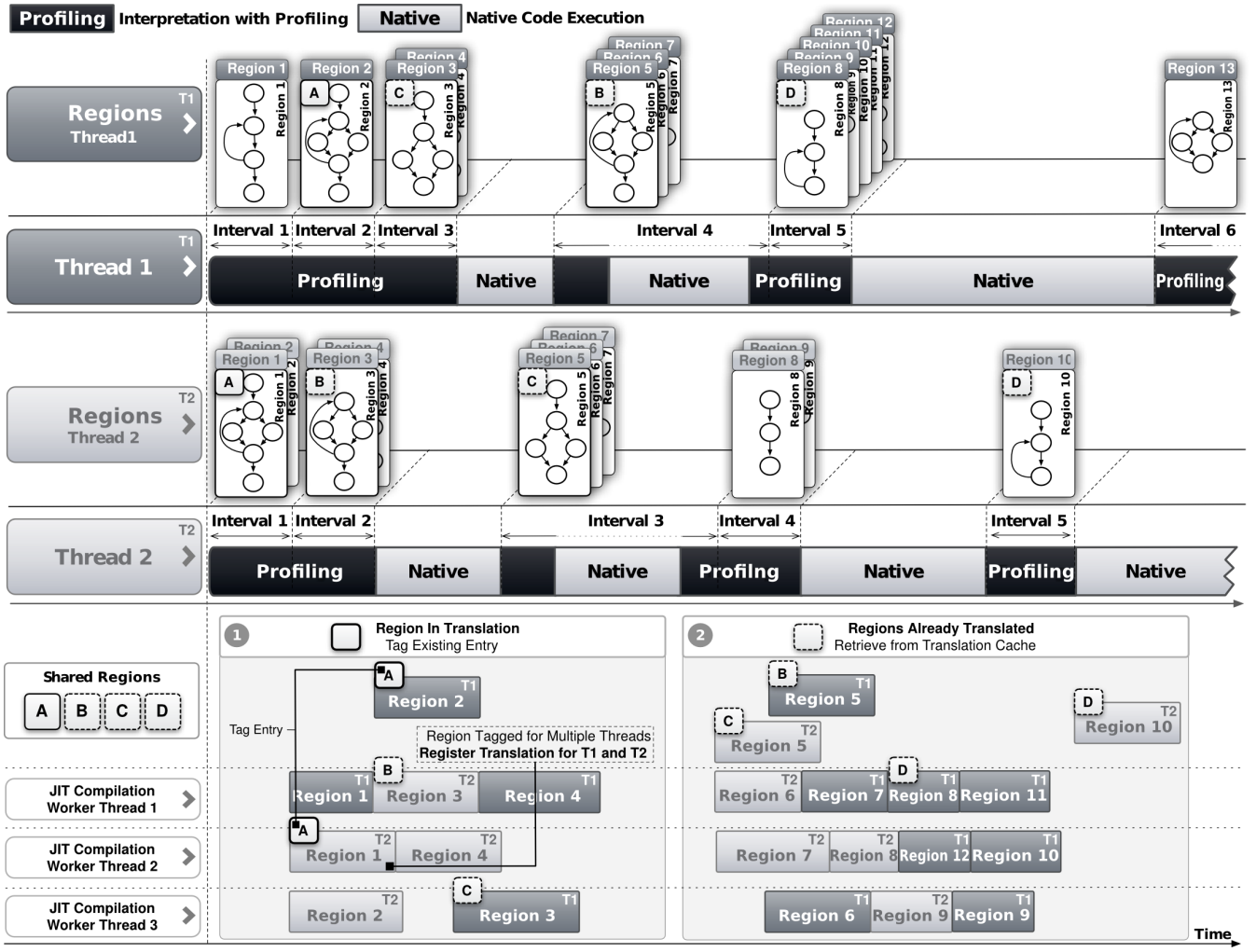

Figure 3. Region sharing for sample multi-threaded application - Frequently executed program regions from two threads `T1` and `T2` are recorded as regions and dispatched for JIT compilation. As soon as a region is compiled it is cached in a translation cache and its availability is registered with the thread responsible for dispatching the region. ① Shows how `T1` records a region (Region 2) that is equal to a previously dispatched region (Region 1) by `T2`. The previously dispatched region is still in translation, hence it is tagged to record the fact that its translation must be registered with `T2` and `T1`. 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.

All experiments were performed under low system load and each experiment was run at least 15 times. In all results, error bars represent the standard error. The number of JIT compilation workers used in each experiment is fixed at $3 + \lfloor log_2(n) \rfloor$, where n is the number of threads being executed.

3.2 Performance Improvements

Our first experiment was to measure the runtime improvements gained through the use of region sharing. In Figure 4 we present speed-ups obtained when executing 4, 32 and 128 threads on our 4-core host machine. Our baseline here is the same private tracing DBT – 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 the performance of the DBT. The average improvement for 4 threads is 1.06x, 1.22x for 32 threads, and for 128 threads, 1.44x. The highest speed-up of 2.40x is obtained for `water-spatial` with 128 threads.

On average, we can see that the speed-up 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. As Section 3.4 will show, these three benchmarks have a smaller percentage of sharable regions than the average. Fewer shared regions lead to reduced savings in JIT 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 parallel task-farm JIT subsystem 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

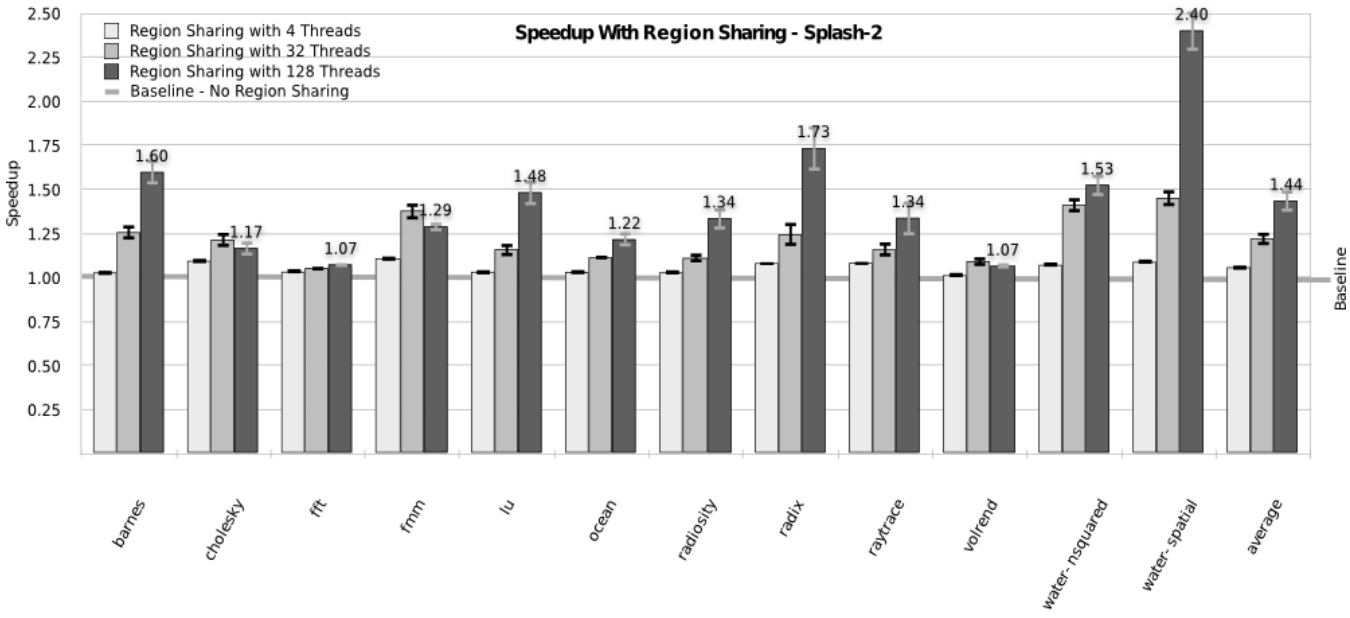

Figure 4. Speed-ups achieved through the use of region sharing, over a baseline execution where region sharing is not used. Speed-ups are presented when executing 4, 32, and 128 threads with all benchmarks from the SPLASH-2 suite.

code execution sooner. This is demonstrated by an overall larger percentage of natively executed code.

3.3 JIT Subsystem Service Throughput

The JIT subsystem used in our DBT is essentially a decoupled service that the DBT can utilise to perform JIT compilation. As such, we can look at the service throughput of the JIT subsystem - how many threads can it provide with native code in any given unit of time? In the region sharing case, this means that if a region is shared among e.g. three threads, then the worker will have served three threads after performing the translation to native code (rather than just one if no regions are shared). Figure 5 shows how the throughput differs when using and not using region sharing, when executing 128 threads.

We note that for every benchmark, the use of region sharing increases the throughput of the JIT subsystem. The highest throughput seen is 10.3 regions/time unit when executing `radix` with region sharing, but the greatest actual improvement obtained is an increase of 4.39x in `water-spatial`. The generally low throughput of `fft` may be due to particularly large regions being generated, or the total number of regions being generated in the benchmark being relatively small in comparison to its total runtime. The important result to note is that region sharing roughly doubles the throughput of the system on average, increasing from 2.1 to 4.6 regions/time unit.

3.4 Percentage of Potentially Shareable Regions

The speed-ups we present in Section 3.2 are due to our 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 JIT subsystem. For each benchmark, it would be interesting to know what percentage of the regions that are handled by the JIT subsystem are similar to regions that have already been handled when not sharing regions. This would imply that in a perfect situation, all the time spent compiling these sharable regions could be saved. To measure this, we modified the non-sharing version to output signatures of regions that are handled. Figure 6 shows

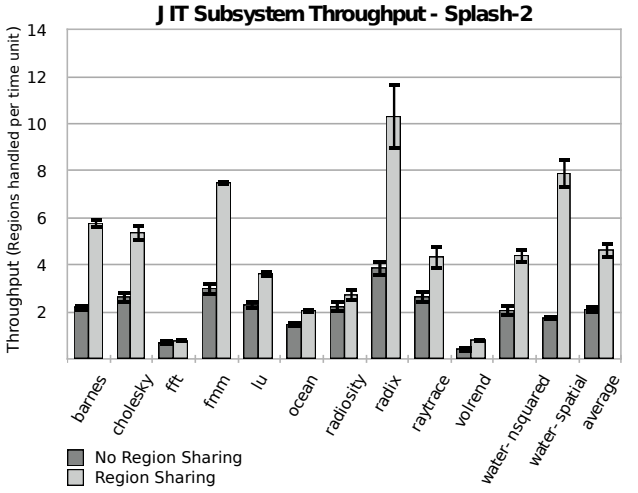

Figure 5. Comparison of JIT subsystem service throughput, with and without region sharing enabled, when executing 128 threads.

the percentage of unique and sharable regions for each benchmark, when executed across 128 cores.

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` could be explained by the fact that the benchmark had to be built without parallel preprocessing, increasing the total percentage of the program that was executed sequentially.

These percentages demonstrate the great potential that exists to speed up execution of multi-threaded programs when using region-based JIT compilation. 94% of all regions are sharable, this means that 94% of all time spent compiling regions could be saved if we share region translations between threads using *thread-agnostic* regions, inter-thread region sharing, and region translation caching.

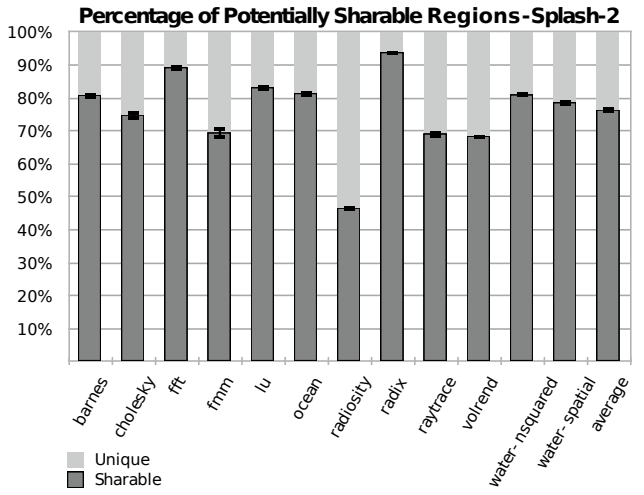

Figure 6. Percentage of sharable regions for the SPLASH-2 benchmark suite when executing 128 threads.

3.5 Thread-Specific vs. Thread-Agnostic Regions

In Section 2.2, we discussed the use of thread-agnostic and thread-specific regions, explaining that thread-agnostic regions allow us to share region translations between threads. Code generated for thread-agnostic regions requires the use of indirection via a pointer to access the thread's state, versus direct access to addresses known at JIT 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 tracing style has on the runtime of the SPLASH-2 benchmarks – when executing only one thread – in Figure 7.

Surprisingly, we can see that the use of thread-agnostic regions is often faster than using thread-specific regions - a speed-up 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 paper, an instruction that accesses a memory location using register + offset calculations should not require more than 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.

We have observed that the use of thread-specific tracing 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 cache. Of course, these results may differ on other architectures. These results show that the use of thread-agnostic tracing actually results in faster execution of threads, even before enabling the sharing of regions between threads.

3.6 JIT Compilation Time Saved

One aim of sharing regions is to reduce the amount of time we spend JIT compiling regions. If 128 threads will all request that the same region be JIT compiled, then it should be possible to remove 127 compilations, allowing JIT workers to move onto other regions of code that are enqueued. For each of the SPLASH-2 benchmarks, we present the percentage of time that workers spent compiling regions when sharing them, in comparison to when not. Figure 8 presents these results, when executing 128 threads. These baseline here of 100% represents the time spent JIT compiling when not sharing regions.

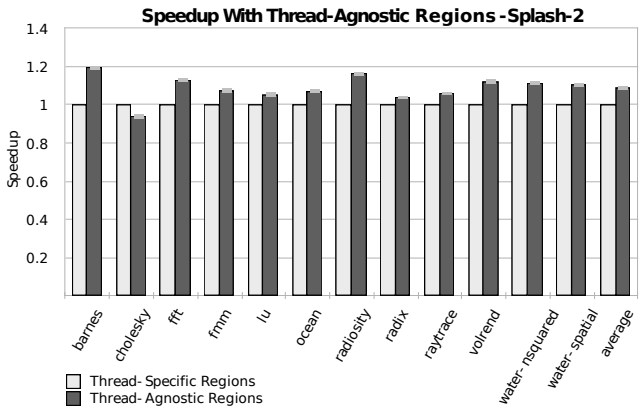

Figure 7. Relative performance when using thread-agnostic over thread-specific regions, when executing one thread for the SPLASH-2 benchmark suite.

On average, JIT compilation time is reduced by 56% when sharing regions. These results also reinforce which benchmarks have less potential for sharing available – the compilation time for `radiosity` is reduced by only 15%, and previously (in Figure 6) only exhibited 47% of its regions as sharable. The best reductions can be seen in `ocean` and `radix`, which are reduced by 92% and 88%, respectively. These results confirm that the use of thread-agnostic regions and region sharing can greatly reduce JIT compilation time.

3.7 Summary of Key Results

In summary, we have seen that the use of thread-agnostic tracing – as well as the sharing of regions that this enables – leads to faster execution of multi-threaded programs when using region-based JIT compilation to enhance performance. These effects are often more pronounced as the number of cores increases – with an average speed-up 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.

4. Related Work

In this section, we discuss the previous work done in the areas of static binary translation, dynamic binary translation, tracing and region-based JIT compilation, and tracing and non-tracing parallel JIT compilation.

4.1 Static Binary Translation

Simulating instruction sets was performed by static compilation in [26]. Programs are compiled by mapping instructions through macros in C switch statements. In [11], source binaries are initially translated to an intermediate representation before converting to C or assembly; portability is much enhanced. SYNTSIM [7] only statically translates those instructions determined to be hot by a prior profiling run and interprets the remainder. Such static approaches can not easily handle self modifying code and must determine instruction targets even in the presence of indirect branches.

4.2 Dynamic Binary Translation

The MIMIC simulator[24] simulates IBM System/370 instructions on the IBM RT PC and translates groups of target basic blocks into host instructions. SHADE[10] and later EMBRA[37] cache translations to speed execution. Both pause execution during translation. DAISY[12] translates PowerPC code onto a VLIW machine. They use page faults to trap necessary translation tasks. The CRUSOE

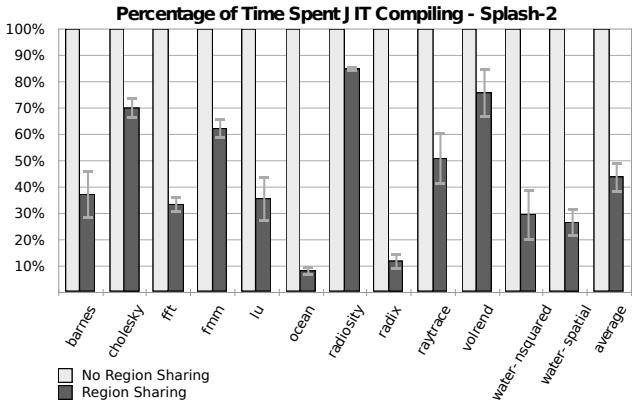

Figure 8. Percentage reduction in time JIT workers spend compiling when using region sharing.

processor from Transmeta[20] also translates to a VLIW in software but from source x86 code.

Just-In-Time Cache Compiled Simulation (JIT-CCS)[27] executes and caches pre-compiled instruction-operation functions for each function fetched. The Instruction Set Compiled Simulation (IC-CS) simulator [31] precompiles the source binary but notices changes to the instructions and invokes a runtime recompilation as needed. [5] merges hot basic blocks from MIPS source binaries into regions and JIT compiles them using LLVM. These approaches target a single threaded environment for execution and compilation.

4.3 Tracing and Region-based JIT Compilation

Tracing is a well established technique for dynamic profile guided optimization of native binaries. DYNAMO[1] introduced tracing as a method for runtime optimization of native program binaries. YETI[39] uses tracing in a mixed in-line, subroutine threading interpreter to avoid additional virtual method calls and unnecessary branches.

Partial method compilation is used in [35] uses profile information to detect never or rarely executed parts of a method and to ignore them during compilation. If such a part gets executed later, execution continues in the interpreter. Region-based compilation is used in [33] to overcome the limitations of method-based compilation. They eliminate rarely executed sections of code, but rely on expensive runtime code instrumentation for trace identification.

Dynamically profiled trace trees are the compilation units for [13, 15]. Traces are aggregated into trees. Trace trees 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. The need for interpretation is removed in [2] by first compiling the method with instrumentation but no optimisation. When suitable thresholds are reached, a execution switches to compiled tracing version of the code. Finally, the selected traces are aggressively optimised.

4.4 Non-tracing Parallel JIT Compilation

Krintz et al. [21] 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 profling. The ability of background compilation to be more aggressive is exploited by [34] to choose the best way to compile a program on an embedded device as its battery energy changes. Again, only single execution and compiler threads are used. Kulkarni et al.[23] 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. The potential to extend their work to multiple compiler threads is considered in [22], wherein the impacts of iteration count thresholds and the number of compiler threads are explored. The Java Hotspot(tm) Server Compiler[28], allows for the creation of multiple compiler threads via a command line option. Azul VM may use as many as fifty compiler threads for large programs, as reported in [23]. These techniques all focus on compiling whole methods, not traces.

The Ultra-fast Instruction Set Simulator is presented in [30]. This paper pioneered the concept of concurrent JIT compilation workers to speed up DBT, but suffers from a number of flaws. First, rather than taking a trace-based compilation approach entire pages are translated – this is unnecessarily wasteful in a time-critical JIT environment. Second, there are no provisions for a dynamic work scheduling scheme that prioritizes compilation of hot traces – this may defer compilation of critical traces and lower overall efficiency. Third, JIT compilers reside in separate processes on remote machines – this significantly increases the communication overhead and limits scalability. This last point is critical, as results shown in [30] 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. A parallel JIT for C is demonstrated in [29]. Whole translation units are compiled at once by one background compiler thread. Execution is able to jump from iterpretation to native whenever the compilation is complete. A form of region based parallel JIT for DBT appears in JPSX, a PlayStation emulator written entirely in Java[32]. Multiple compilation threads translate R3000 code into Java byte-codes with the different threads managing progressively higher optimization levels. Code is initially interpreted or translated via an in-line threading interpreter or, when suitably hot, translated with the most optimising compiler thread. loading mechanism to initiate compilation of unvisited code. The HotSpot JVM then provides additional optimisation, being able to make use of the fact that all code blocks are translated to static methods, leading to a system which is so fast that it requires careful synchronisation, not present in the original binaries, to ensure that video frames are not produced faster than the screen can display.

4.5 Tracing Parallel JIT Compilation

Ha et al.[17] attach a 'Compiled State Variable' (CSV) to each trace anchor. The CSV is allows a tracing JavaScript interpreter and background compiler to manage transitions from interpreted execution to native without locks. Their approach, however, is only applied to single execution and compilation threads. MOJO[9] is system very like dynamo but extends it to permit proper exception handling. Individual blocks are compiled and then when hot traces are identified, those are translated. MOJO also offers support for multi-threading; threads maintain a private list of individual blocks but share the set of translated paths. Inoue[19] implement tracing in the IBM J9/TR JVM. They use a global trace cache and note that the time spent searching the cache is significant. Only one background compilation thread is used. A similar global trace cache is used by Häubl[18] for the Hotspot JVM. Wimmer et al.[36] 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.

Some approaches [8, 14] have 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 [16]. Small translation units and long compilation pipelines, however, increase the relative synchronization 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.

Mehrara et al. take a different tack [25]. They note that a considerable time is spent executing trace guards, which would trigger a side exit but are rarely taken. They offload the checking to another, background thread, enabling the main execution thread to speculatively continue along the trace. If the background thread discovers a guard violation, the main thread is halted and execution resumes at the appropriate side exit.

5. Summary and Conclusions

In this paper we have developed an approach to region-based JIT compilation that enables sharing of common regions between multiple application threads. For this we first need to modify the code generator to generate regions that do not depend on any thread-specific context, but are generic enough to be executed in any thread context. Surprisingly, this thread-agnostic code generation does not only enable region sharing, but also facilitates a minor performance improvement when applied in isolation. Sharing of regions between application threads then reduces the pressure on the JIT subsystem and improves performance further. In our experiments we found that on average 76% of all regions can be shared in the SPLASH-2 benchmarks when run with 128 threads in our ARCSIM dynamic binary translator. This results in an overall performance improvement of 1.44x averaged across all benchmarks and over a state-of-the-art thread-private region compilation approach without region sharing.

Future work will focus on region sharing in other multi-threaded and JIT compilation based execution environments such as JVMs.

REFERENCES

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, *Dynamo: a transparent dynamic optimization system*, Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (New York, NY, USA), PLDI '00, ACM, 2000, pp. 1–12.

[2] Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz, *Trace-based compilation in execution environments without interpreters*, Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (New York, NY, USA), PPPJ '10, ACM, 2010, pp. 59–68.

[3] Fabrice Bellard, *QEMU, a fast and portable dynamic translator*, Proceedings of the annual conference on USENIX Annual Technical Conference (Berkeley, CA, USA), ATEC '05, USENIX Association, 2005, pp. 41–41.

[4] Igor Böhm, Tobias J.K. Edler von Koch, Stephen Kyle, Björn Franke, and Nigel Topham, *Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'11, ACM, 2011.

[5] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler, *Fast and accurate simulation using the LLVM compiler framework*, Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO'09, 2009, pp. 1–6.

[6] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji, *Thread-shared software code caches*, Proceedings of the International Symposium on Code Generation and Optimization (Washington, DC, USA), CGO '06, IEEE Computer Society, 2006, pp. 28–38.

[7] Martin Burtscher and et al., *Automatic synthesis of high-speed processor simulators*, Proceedings of the 37th annual International Symposium on Microarchitecture, MICRO'04, 2004.

[8] Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi, *A parallel dynamic compiler for CIL bytecode*, SIGPLAN Not. **43** (2008), 11–20.

[9] W.-K. Chen, S. Lerner, R. Chaiken, and D.M. Gillies, *Mojo: a dynamic optimization system*, Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization, FDDO'00, 2000.

[10] Bob Cmelik and David Keppel, *Shade: A fast instruction-set simulator for execution profiling*, Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems, SIGMETRICS'94, 1994, pp. 128–137.

[11] Jianwen Zhu Daniel and Daniel D. Gajski, *A retargetable, ultra-fast instruction set simulator*, Proceedings of the Design Automation and Test Conference In Europe, DATE'95, 1995, pp. 363–373.

[12] Kemal Ebcioglu and Erik R. Altman, *Daisy: Dynamic compilation for 100% architectural compatibility*, 1997.

[13] A. Gal, C. W. Probst, and M. Franz, *HotpathVM: an effective JIT compiler for resource-constrained devices*, Proceedings of the 2nd International Conference on Virtual Execution Environments, VEE'06, ACM, 2006, pp. 144–153.

[14] Andreas Gal, Michael Bebenita, Mason Chang, and Michael Franz, *Making the compilation "pipeline" explicit: Dynamic compilation using trace tree serialization*, Tech. Report 07-12, University of California, Irvine, 2007.

[15] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz, *Trace-based just-in-time type specialization for dynamic languages*, Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA), PLDI '09, ACM, 2009, pp. 465–478.

[16] Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, and David Appenzeller, *Dynamic and transparent binary translation*, Computer **33** (2000), 54–59.

[17] Jungwoo Ha, Mohammad R. Haghighat, Shengnan Cong, and Kathryn S. McKinley, *A concurrent trace-based just-in-time compiler for single-threaded Javascript*, Workshop on Parallel Execution of Sequential Programs on Multicore Architectures, PESPMA'09, June 2009, in conjunction with ISCA 09.

[18] Christian Häubl and Hanspeter Mössenböck, *Trace-based compilation for the Java HotSpot virtual machine*, Proceedings of the International Conference on Principles and Practice of Programming in Java (Kongens Lyngby, Denmark), PPPJ'11, August 2011.

[19] H. Inoue, H. Hayashizaki, Peng Wu, and T. Nakatani, *A trace-based Java JIT compiler retrofitted from a method-based compiler*, Code Generation and Optimization, 2011 9th Annual IEEE/ACM International Symposium on, CGO'11, april 2011, pp. 246 –256.

[20] Alexander Klaiber, *The Technology Behind Crusoe Processors*, Tech. report, Transmeta Corporation, January 2000.

[21] Chandra Krintz, David Grove, Derek Lieber, Vivek Sarkar, and Brad Calder, *Reducing the overhead of dynamic compilation*, Software: Practice And Experience **31** (2000), 200–1.

[22] P.A. Kulkarni and J. Fuller, *JIT compilation policy on single-core and multi-core machines*, Interaction between Compil-

ers and Computer Architectures, 2011 15th Workshop on, INTERACT'11, feb. 2011, pp. 54 –62.

[23] Prasad Kulkarni, Matthew Arnold, and Michael Hind, *Dynamic compilation: the benefits of early investing*, Proceedings of the 3rd international conference on Virtual execution environments (New York, NY, USA), VEE '07, ACM, 2007, pp. 94–104.

[24] C. May, *Mimic: a fast System/370 simulator*, Papers of the Symposium on Interpreters and interpretive techniques (New York, NY, USA), SIGPLAN '87, ACM, 1987, pp. 1–13.

[25] M. Mehrara and S. Mahlke, *Dynamically accelerating client-side web applications through decoupled execution*, Code Generation and Optimization, 2011 9th Annual IEEE/ACM International Symposium on, CGO'09, april 2011, pp. 74 – 84.

[26] Christopher Mills, Stanley C. Ahalt, and Jim Fowler, *Compiled instruction set simulation*, 1991.

[27] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann, *A universal technique for fast and flexible instruction-set architecture simulation*, Proceedings of the 39th annual Design Automation Conference (New York, NY, USA), DAC '02, ACM, 2002, pp. 22–27.

[28] Michael Paleczny, Christopher Vick, and Cliff Click, *The Java Hotspot^{TM} server compiler*, USENIX Java Virtual Machine Research and Technology Symposium, USENIX-JVM'01, 2001, pp. 1–12.

[29] Michael Plezbert and Ron K. Cytron, *Does "just in time" = "better late than never"?*, In Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, ACM Press, 1997, pp. 120–131.

[30] Wei Qin, Joseph D'Errico, and Xinping Zhu, *A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation*, Proceedings of the 4th international conference on Hardware/software codesign and system synthesis (New York, NY, USA), CODES+ISSS '06, ACM, 2006, pp. 193–198.

[31] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt, *Instruction set compiled simulation: a technique for fast and flexible instruction set simulation*, Proceedings of the 40th annual Design Automation Conference (New York, NY, USA), DAC '03, ACM, 2003, pp. 758–763.

[32] Graham Sanderson, *High performance: Writing a Sony PlayStation emulator using Java^{TM} technology*, 2006.

[33] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani, *A region-based compilation technique for dynamic compilers*, ACM Trans. Program. Lang. Syst. **28** (2006), 134–174.

[34] P. Unnikrishnan, M. Kandemir, and F. Li, *Reducing dynamic compilation overhead by overlapping compilation and execution*, Proceedings of the 2006 Asia and South Pacific Design Automation Conference (Piscataway, NJ, USA), ASP-DAC '06, IEEE Press, 2006, pp. 929–934.

[35] John Whaley, *Partial method compilation using dynamic profile information*, Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '01, ACM, 2001, pp. 166–179.

[36] Christian Wimmer, Marcelo S. Cintra, Michael Bebenita, Mason Chang, Andreas Gal, and Michael Franz, *Phase detection using trace compilation*, Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (New York, NY, USA), PPPJ '09, ACM, 2009, pp. 172–181.

[37] Emmett Witchel and Mendel Rosenblum, *Embra: Fast and flexible machine simulation*, Measurement and Modeling of Computer Systems, SIGMETRICS'96, 1996, pp. 68–79.

[38] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta, *The SPLASH-2 programs: characterization and methodological considerations*, Proceedings of the 22nd annual international symposium on Computer architecture (New York, NY, USA), ISCA '95, ACM, 1995, pp. 24–36.

[39] Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley, *Yeti: a gradually extensible trace interpreter*, Proceedings of the 3rd international conference on Virtual execution environments (New York, NY, USA), VEE '07, ACM, 2007, pp. 83–93.