Mitigating JIT Compilation Latency in Virtual Execution Environments

Martin Kristien
School of Informatics
University of Edinburgh, UK
m.kristien@sms.ed.ac.uk

Tom Spink
School of Informatics
University of Edinburgh, UK
tspink@inf.ed.ac.uk

Harry Wagstaff
School of Informatics
University of Edinburgh, UK
hwagstaf@inf.ed.ac.uk

Björn Franke
School of Informatics
University of Edinburgh, UK
bfranke@inf.ed.ac.uk

Igor Böhm
Synopsys Inc.
igor.boehm@synopsys.com

Nigel Topham
School of Informatics
University of Edinburgh, UK
npt@inf.ed.ac.uk

Abstract

Many Virtual Execution Environments (VEEs) rely on Just-in-time (JIT) compilation technology for code generation at runtime, e.g. in Dynamic Binary Translation (DBT) systems or language Virtual Machines (VMs). While JIT compilation improves native execution performance as opposed to e.g. interpretive execution, the JIT compilation process itself introduces latency. In fact, for highly optimizing JIT compilers or compilers not specifically designed for JIT compilation, e.g. LLVM, this latency can cause a substantial overhead. While existing work has introduced asynchronously decoupled JIT compilation task farms to hide this JIT compilation latency, we show that this on its own is not sufficient to mitigate the impact of JIT compilation latency on overall performance. In this paper, we introduce a novel JIT compilation scheduling policy, which performs continuous low-cost profiling of code regions already dispatched for JIT compilation, right up to the point where compilation commences. We have integrated our novel JIT compilation scheduling approach into a commercial LLVM-based DBT system and demonstrate speedups of 1.32× on average, and up to 2.31×, over its state-of-the-art concurrent task-farm based JIT compilation scheme across the SPEC CPU2006 and BioPerf benchmark suites.

CCS Concepts • **Hardware → Simulation and emulation**; Hardware description languages and compilation; • **Software and its engineering →** *Simulator / interpreter*; **Just-in-time compilers**;

ACM Reference format:
Martin Kristien, Tom Spink, Harry Wagstaff, Björn Franke, Igor Böhm, and Nigel Topham. 2019. Mitigating JIT Compilation Latency in Virtual Execution Environments. In *Proceedings of Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Providence, RI, USA, April 14, 2019 (VEE '19),* 7 pages.
https://doi.org/10.1145/3313808.3313818

1 Introduction

Many VEEs, ranging from DBT systems [1, 3, 12] through language VMs [4, 10, 25] to specialized accelerator runtime environments [16], rely on JIT compilation as a key performance enabler. While some of these VEEs use JIT compilers specifically designed for this purpose, the LLVM [19] compiler framework finds increasing use in both academic projects, and commercial products as a JIT compiler. The list of examples is long: From Julia [2] and R [22] to Facebook's HHVM [21], Azul's Falcon Java JIT [23] and Apple's Nitro JavaScript engine, and virtually every OpenCL compiler (e.g. Intel, Apple, Arm), LLVM is used as a JIT compiler.

However, while actively supported and equipped with strong optimizations, the LLVM compiler was not originally designed as a fast JIT compiler. In fact, it is well known that the use of LLVM as a JIT compiler can introduce a large compilation overhead [5]. This is because some traditional algorithms used in static compilers are too slow to be used in JIT compilers [6].

As a way of hiding this JIT compilation latency, and to increase the throughput of the JIT compiler, parallel and concurrent JIT compilation task farms have been developed [3, 18] and, later, adopted by industry, e.g. to drive data center software infrastructure like in HHVM [21], to improve the start-up times of managed application as in Microsoft's .Net Multicore JIT [7], or to power browser based applications like in Google's Chrome v8 JavaScript engine. While effective, we show how this decoupled approach to JIT compilation introduces a new problem: Maintaining the right balance between the selection of code regions for JIT compilation, and scheduling regions for compilation is non-trivial. We show how state-of-the-art scheduling approaches [3, 18] can arrive at non-optimal decisions resulting in poor application performance as hot code regions are compiled too late.

In this paper we propose a novel scheme for scheduling JIT compilation units in a VEE. The **key idea** is to separate the *dispatch* of a compilation unit into a compilation queue from *scheduling* the next unit for compilation. When the compilation threshold is met, compilation units are dispatched to a compilation queue, but their heat is continually updated as execution progresses. When the JIT compiler fetches the next compilation unit from the queue, we select the hottest unit to be compiled. This scheme enables us to make scheduling decisions based on accurate and up-to-date *heat* information, leading to better runtime performance.

We have extended a commercial multi-threaded and LLVM-based DBT system (Synopsys DesignWare ARC nSIM [14]) with our new JIT scheduling technique and demonstrate its viability. In our evaluation against the SPEC CPU2006 and BioPerf benchmark suites we demonstrate overall average speedups of 1.32× and 1.2×, respectively, and up to 2.31× over the default scheme.

1.1 Contributions

In this paper, we make the following contributions:

- We show that existing *FIFO* and heat-and-recency based JIT scheduling policies lead to sub-optimal compilation schedules;
- We introduce a novel JIT compilation queue scheduling policy, *Dynamic Heat*;
- We perform a detailed analysis of this new scheduling policy, and compare it to existing policies across a range of industry standard benchmarks.

2 Background and Motivating Example

Hybrid interpreter/DBT systems [17] offload the expensive JIT compilation of work-units to threads [3], whilst still making *forward progress* in the interpreter, and thus hiding the latency of JIT compilation. Such set-ups (described as *Asynchronous Mixed-mode Translation* by [24]) have a greater scope for implementation, and raise questions such as *what*, *when*, and *how* guest code should be translated.

Figure 1 contrasts a typical configuration for a hybrid interpreter/DBT-based VEE against our novel scheme. In this example, the main execution loop starts by checking to see if the code to be executed (i.e. the code residing at the current program counter (PC)) has already been translated. If a translation exists, it is used for execution. Otherwise, the guest code is executed by the interpreter. Following this, in typical asynchronous implementations, if the code has been marked for translation (i.e. it has been *dispatched*), then the main execution loop continues as normal in the interpreter, until eventually the translation is available. If the code has not been *dispatched*, its profile is updated, and its *heat* (the demand for this code to be executed) is measured. Code that passes a heat threshold is *dispatched*, and thus scheduled for compilation.

Typically, the code to be translated (the *work unit*) will be added to a queue, and a compilation worker thread will remove and process the work unit, usually in FIFO order. We refer to a particular ordering as the *compilation schedule*, and the *policy* dictates how this schedule is formed.

2.1 Existing Scheduling Policies

In this paper, we make reference to the *Default* scheduling policy as the one supplied with the *Synopsys DesignWare ARC nSIM* product. This policy relies on both *heat* and *recency* for determining the compilation schedule [3]. The most prevalent policy found in literature is *FIFO* [9, 15, 21], which although does not prioritize compilation units in any way, maintains a strong sense of fairness, and prevents a compilation unit from being stuck in the queue indefinitely.

2.2 Motivating Example

To motivate the research of compilation scheduling, we demonstrate the effect of two different scheduling policies in Figure 2. This figure depicts the execution of **perlbench** from the SPEC CPU2006 [11] suite as heatmaps showing execution in different regions of the application's address space over time. The horizontal axis represents the number of executed instructions, as this is the time seen by the executed application. Note, instruction time is not skewed by different execution speeds resulting from different scheduling policies.

A red color represents execution in *interpretive* mode. A blue color represents execution in *native* mode. The intensity of the color indicates the amount of execution in the corresponding space-time region. A good policy should produce heatmaps that are more blue overall, by turning high-intensity red regions into blue quickly.

The *Default* compilation scheduling policy (Figure 2a) produces a heatmap that contains long horizontal interpretation lines (red). These indicate the policy has failed to recognize the "importance" of the corresponding code regions. On the other hand, the *Dynamic Heat* policy (Figure 2b) turns high intensity interpretation into native execution more quickly. This indicates that the policy selects important code regions for compilation with a relatively short delay.

The two heatmaps in Figure 2 demonstrate the effect of the compilation queue scheduling policy on the performance of the whole system. In the case of perlbench, *Dynamic Heat* results in a speedup of 1.99× relative to *Default*.

3 Methodology

The motivating example has shown clear sub-optimality of the *Default* policy for a particular work-load. The visualization (Figure 2a) shows code regions being interpreted

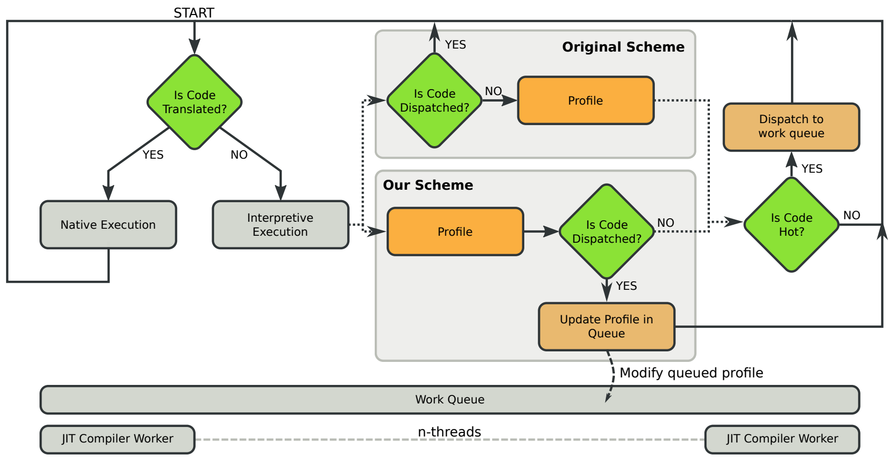
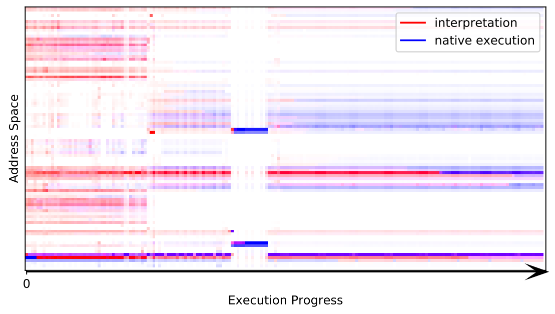
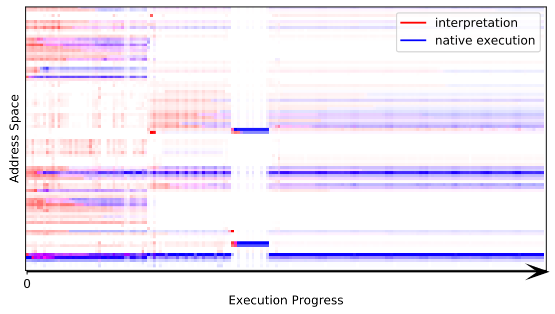


Figure 1. Operation of an asynchronous DBT system, with the original profiling scheme, and our proposed dynamic scheme.



(a) *Default* scheduling policy.



(b) *Dynamic Heat* scheduling policy.

Figure 2. Execution of the SPEC CPU2006 *perlbench* benchmark with different scheduling policies. Red indicates *interpretive* and blue indicates *native* execution. The intensity indicates the amount of execution in the corresponding space-time region.

for a long time, without the policy recognizing the regions' importance to the application. Although the same code regions were dispatched for compilation, the compilation order differed, as dictated by the respective scheduling policy.

To tackle the issue of suboptimal compilation scheduling, we introduce a novel scheduling policy focused on the changing demands of applications. The policy relies on continuous profiling of already dispatched code regions, resulting in dynamic updates of heat of compilation units already present in the queue. The compilation units are then prioritized based on the values of this dynamically updated heat. Note, dynamic here means *after-dispatch*, rather than the conventional meaning of *at-runtime*. We distinguish from a typical *heat* policy as using static heat, i.e. *at-dispatch*.

The *Dynamic Heat* policy targets the long red interpretation lines by allowing all compilation units in the queue that are being interpreted to increase in priority. Such dynamic priority updates can fast-track previously moderately hot units to the front of the compilation queue, preventing any code region from being interpreted for a long time.

3.1 Implementation

We implement our novel policy in a state-of-the-art commercial DBT system, Synopsys DesignWare ARC nSIM. This DBT system implements the asynchronous compilation technique introduced previously. We make several changes to the JIT compilation system to implement our new policy. These were mainly in the profiling subsystem (to perform dynamic heat updates), and in the compilation queue organization (to take dynamic heat updates into consideration).

3.1.1 Profiler

In the original DBT system, profiling of application's basic blocks stopped after dispatch of the corresponding compilation units. During dispatch, a handle to contain the produced native code was registered with each basic block corresponding to a particular compilation unit.

To allow dynamic updates to the heat of the compilation units, a handle containing the compilation unit was also registered with each dispatched basic block. Now, when the profiler reaches a basic block already dispatched but not yet compiled, the corresponding compilation unit's heat is

Table 1. System Configuration

Host System			
System	Supermicro		
Architecture	x86-64	*Model*	Intel Xeon
Sockets/Cores	2/10	*Frequency*	2.4 GHz
L1 Cache	I$32 kB/D$32 kB	*L2 Cache*	256 kB
Guest System			
System	Synopsys DesignWAre ARC nSIM		
Architecture	ARC700		
Tracing Scheme	Region-based	*Interval*	1,024 blocks
JIT Compiler	LLVM	*# Threads*	1
JIT Optimization	-O3	*Threshold*	Adaptive

incremented through the handle on the basic block. During this update no synchronization is involved, as the heat is only an approximate metric and no error can arise from the data races in accessing this metric.

Continued profiling of dispatched but not yet compiled basic blocks resulted in no observable performance penalty. Although more computation is being performed, it is done only for blocks which are being interpreted, and the actual interpretation is more costly than the counter increment.

3.1.2 Compilation Queue

All previous compilation queue implementations ordered units during dispatch, as the priority metrics were fixed. This led to an efficient implementation using standard C++ libraries, e.g. `std::queue` or `std::priority_queue`. However, no standard data structure allows for ordering elements when the ordering metric is not fixed at insertion.

For simplicity of implementation, the compilation units ordering point was moved from dispatch to selection (i.e. pop-time). This allowed the use of an unstructured data structure, in particular a `std::vector`. At the point of selection, a linear scan through this data structure is performed, finding a compilation unit with the maximal current (dynamic) heat.

Although this increases the complexity of the compilation unit selection from $O(1)$ (for `std::queue`) or $O(ln(n))$ (for `std::priority_queue`) to $O(n)$ in size of the compilation queue, no performance penalty was observed. Since the ordering has been moved from the application thread to the compilation thread, the application thread can dispatch faster and continue executing the application. On the other hand, the compilation threads do not suffer from the increased complexity, as the linear scan is negligible compared to the computational cost involved in the compilation itself and the synchronization overheads already present.

4 Evaluation

Our novel compilation scheduling policy was evaluated using SPEC CPU2006 and Bioperf benchmarking suites, using the host machine and DBT configuration described in Table 1. For SPEC CPU2006, due to compilation issues and long runtimes, we only use the *integer* benchmarks with the *test* input set. Benchmarks from the Bioperf suite are run with the class-A workloads. The benchmarks have been compiled with gcc 4.2.1, with `-O3` optimizations. Arithmetic mean and standard deviation of 15 runs of each experiment are depicted.

4.1 Key Results

In our results, the baseline policy (used for comparison) is the *Default* policy, as implemented in the reference DBT system, and described in subsection 2.1.

The results show speedups for both SPEC CPU2006 and Bioperf in Figure 3. Some benchmarks achieve up to 2.31× speedup compared to the baseline policy. On average, speedups for SPEC and Bioperf are 1.32× and 1.2×, respectively.

Previous research [3] (from an older version of the reference DBT system) suggests the *Default* policy improves over *FIFO* with a speedup of 1.04× and 1.13×, for SPEC and Bioperf, respectively. Our *Dynamic Heat* policy achieves further improvements, with speedups over *FIFO* of 1.29× and 1.54× for SPEC and Bioperf, respectively.

Furthermore, while both the *Default* and the *FIFO* can be outperformed by the *Random* policy for some benchmarks, *Dynamic Heat* is never outperformed by the *Random* policy.

4.2 Comparison to Parallel JIT

The scheduling policy itself only aims at reducing the amount of interpretation by selecting the most important code regions. Another way to reduce the amount of interpretation is to increase compilation throughput by using multiple parallel JIT workers. Figure 4 compares the speedups achieved from parallelism, to the speedup achieved by using *Dynamic Heat*, for the SPEC and Bioperf benchmark suites. All speedups are relative to the *Default* policy with one JIT worker.

As expected, introducing multiple workers improves performance on average. Interestingly, for most benchmarks, our novel scheduling policy results in better performance improvement than one additional JIT worker. The graphs clearly indicate the benchmarks that benefit the most from concurrent compilation, and our policy follows this trend.

4.3 Reduction of Interpretation

Figure 5 shows the relative reduction in the number of interpreted instructions, when using the *Dynamic Heat* policy, compared to the *Default* policy. A smaller number of interpreted instructions indicates a larger proportion of native instructions, which is the desirable outcome. Using the *Dynamic Heat* policy results in a reduction of instructions being interpreted by more than 48% relative to the *Default* policy, on average.

However, due to the long runtime, execution of some benchmarks (e.g., `bzip2`, `hmmer`) is dominated by native execution for all policies (see Figure 6). In these cases, further reductions in the proportion of interpreted instructions do

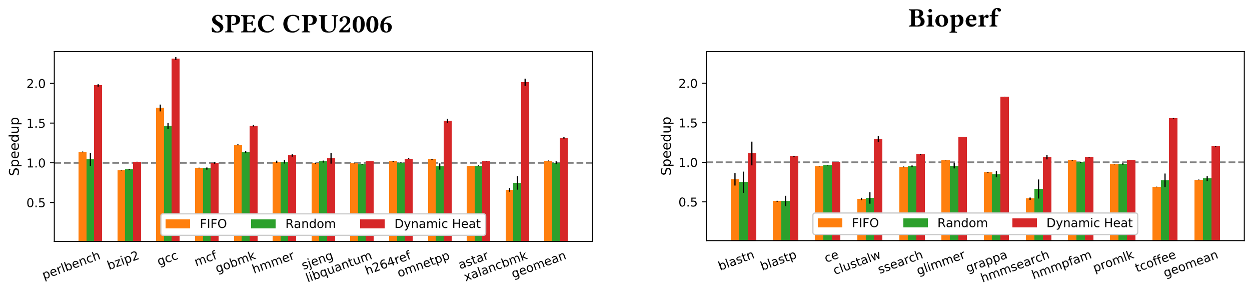


Figure 3. Speedups over the baseline policy for both the SPEC CPU2006 and Bioperf benchmark suites. Higher is better.

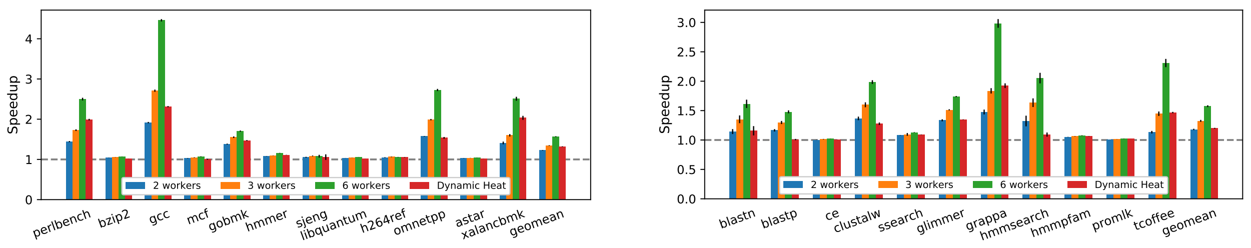


Figure 4. Speedups over the baseline policy, when used with different numbers of JIT worker threads. Higher is better.

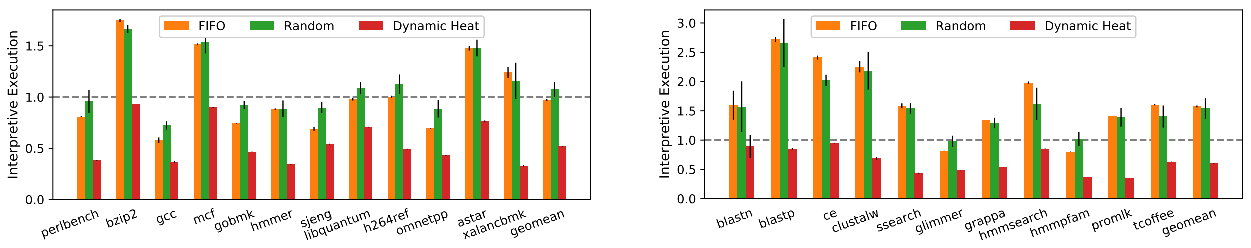


Figure 5. Proportion of interpreted instructions in each benchmark suite, relative to the *Default* policy. Lower is better.

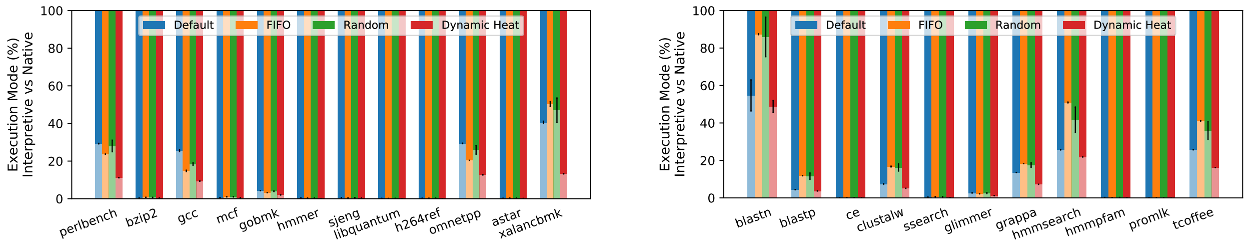


Figure 6. Proportion of interpreted instructions in each benchmark suite, relative to total instructions, for each policy. Shaded areas are *interpretive*, solid areas are *native* execution. Larger solid area is better.

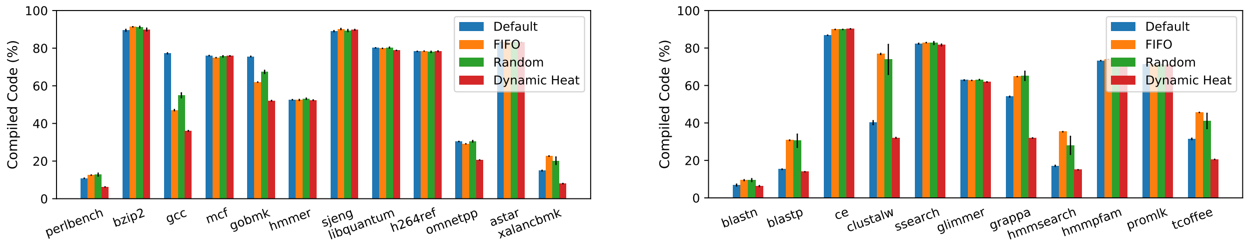


Figure 7. The percentage of *static* code JIT-compiled during execution. Lower is better.

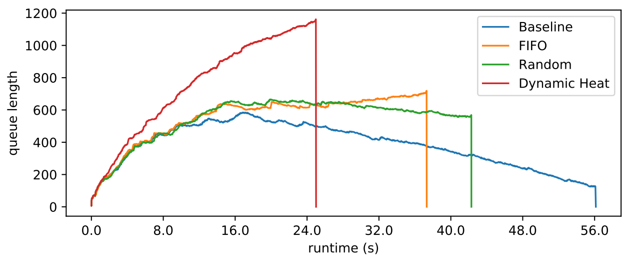


Figure 8. Compilation queue length over run-time for gcc.

not translate to significant speedups. This also explains negligible effect of compilation scheduling policy on the runtime of some benchmarks.

4.4 Quantity of Translated Code

Figure 7 shows the total quantity of translated code for each benchmark suite. For several benchmarks, the *Dynamic Heat* policy results in a significant reduction in the amount of translated code (e.g., `gcc`, `gobmk`, `grappa`) compared to the *Default* policy. Although less code is compiled with *Dynamic Heat*, more native execution is observed, indicating *Dynamic Heat* is better at selecting "important" code.

4.5 Compilation Queue Length

The choice of scheduling policy also affects the length of the compilation queue. Counter-intuitively, speedups are associated with longer compilation queues. Since the queue consumption rate is fixed by the compilation throughput, different scheduling policies can only affect the queue production rate. New compilation units are added to the compilation queue when newly discovered code regions become hot. Therefore, fast native execution results in less time elapsed before new code is discovered, increasing the effective queue production rate. In other words, good policy speeds up application execution leaving less time for the JIT workers to consume the compilation queue. We observe this effect in Figure 8 for `gcc`, where the compilation queue is observed to be significantly longer for *Dynamic Heat* (nearly 1200) than for *Default* (peaking at approximately 600).

5 Related Work

QEMU [1] is a widely-supported, portable binary translation tool, which uses its own block-based JIT compiler (TCG). QEMU performs translation synchronously with execution, and so does not need to perform any compilation scheduling. On the other hand, it is unable to extensively optimize the translated code, since it is only considering a small code region at a time. MAMBO-X86 [8] uses a tracing and translation scheme which maps guest call/return instructions to equivalent host instruction sequences in order to take advantage of the host system's return address prediction mechanisms. Asynchronous DBT systems have been presented in a variety of contexts. These include HQEMU [12], an extension of QEMU which introduces an asynchronous trace-based optimizer based on LLVM. HQEMU profiles and translates traces, rather than the large code region which nSIM uses. [3] presents another asynchronous DBT system. Here, a parallel task farm is used to translate large code regions using LLVM. They use an interpreter for the profiling/tracing step, unlike HQEMU that uses tiered compilation, and traces continuously rather than only when the possibility of a hot region is encountered. [13] extends HQEMU with Intel Processor Trace, to reduce the overheads involved in trace forming. Here, modern tracing and profiling hardware built in to the CPU is used to reduce the cost of hot regions detection and selection for further optimization. However, the paper does not discuss the compilation scheduling policy. Ha et al.[10] present an asynchronous, trace-based JavaScript JIT compiler. This uses a simple FIFO queue to order the traces to be compiled. They also suggest that a lock-free queue could be used, although they also admit that this is unlikely to have a significant effect on performance. In [20], Namjoshi et al present a method for online predictive profiling of Java applications. This method attempts to detect the iteration count for each loop, and prioritize the compilation of loops that are predicted to become hot, as well as any methods called from such loops.

6 Summary & Conclusion

In this paper we have developed a novel JIT compilation scheduling policy mitigating the negative impact of compilation latency in an asynchronous JIT system. Our *Dynamic Heat* policy represents a significant improvement over a state-of-the-art combined heat/recency policy used in a commercial LLVM based DBT system. We demonstrated that our novel policy consistently performs as good or better than the default policy and integrates well with the multi-threaded JIT compilation task farm system of the DBT system. In addition to average speedups of 1.32× and 1.2× for SPEC 2006 and Bioperf benchmark suites respectively, and up to 2.31×, we found that the use of our novel scheduling policy provides a performance boost roughly equivalent to that of adding one further JIT compilation thread. For systems with a small or modest number of host machine cores this is of particular significance as our *Dynamic Heat* policy is able to outperform the *Default* policy using two workers on every benchmark, and with three workers in many cases. Larger systems still benefit from lower machine utilization while delivering the same performance with fewer JIT compilation threads.

References

[1] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2014. Julia: A Fresh Approach to Numerical Computing. *CoRR* abs/1411.1607 (2014). arXiv:1411.1607 http://arxiv.org/abs/1411.1607

[3] Igor Böhm, Tobias JK Edler von Koch, Stephen C Kyle, Björn Franke, and Nigel Topham. 2011. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 74–85.

[4] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 18–25.

[5] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. 2009. Fast and Accurate Simulation using the LLVM Compiler Framework. In *1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*.

[6] M. Cierniak and W. Li. 1997. Just-in-time optimization for high-performance Java programs. In *Java for Computational Science and Engineering - Simulation and Modeling II*.

[7] Microsoft Corporation. 2012. An easy solution for improving app launch performance. https://blogs.msdn.microsoft.com/dotnet/2012/10/18/an-easy-solution-for-improving-app-launch-performance/. [accessed 13-December-2018].

[8] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. 2017. Low Overhead Dynamic Binary Translation on ARM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 333–346. https://doi.org/10.1145/3062341.3062371

[9] Google. 2018. Chrome V8. https://developers.google.com/v8/. [accessed 8-August-2018].

[10] Jungwoo Ha, Mohammad Haghighat, Shengnan Cong, and Kathryn McKinley. 2009. A concurrent trace-based just-in-time compiler for JavaScript. *University of Texas, Austin, Tech. Rep. TR-09-06* (2009).

[11] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[12] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 104–113.

[13] Ding-Yong Hong, Jan-Jan Wu, Yu-Ping Liu, Sheng-Yu Fu, and Wei-Chung Hsu. 2018. Processor-Tracing Guided Region Formation in Dynamic Binary Translation. *ACM Trans. Archit. Code Optim.* 15, 4, Article 52 (Nov. 2018), 25 pages. https://doi.org/10.1145/3281664

[14] Synopsys Inc. 2018. DesignWare ARC nSIM. https://www.synopsys.com/dw/ipdir.php?ds=sim_nsim. [accessed 8-August-2018].

[15] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. 1999. Design, Implementation, and Evaluation of Optimizations in a Just-in-time Compiler. In *Proceedings of the ACM 1999 Conference on Java Grande (JAVA '99)*. ACM, New York, NY, USA, 119–128. https://doi.org/10.1145/304065.304111

[16] Pekka Jääskeläinen, Carlos Sánchez Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. Pocl: A Performance-Portable OpenCL Implementation. *Int. J. Parallel Program.* 43, 5 (Oct. 2015), 752–785. https://doi.org/10.1007/s10766-014-0320-y

[17] Daniel Jones and Nigel Topham. 2009. High speed CPU simulation using LTU dynamic binary translation. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 50–64.

[18] Prasad A Kulkarni. 2011. JIT compilation policy for modern machines. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 773–788.

[19] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[20] Manjiri A Namjoshi and Prasad A Kulkarni. 2010. Novel online profiling for virtual machines. In *ACM Sigplan Notices*, Vol. 45. ACM, 133–144.

[21] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 151–165. https://doi.org/10.1145/3192366.3192374

[22] R Development Core Team. 2008. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org ISBN 3-900051-07-0.

[23] Philip Reames. 2017. Falcon: An Optimising Java JIT. In *LLVM Developers Meeting*. Azul Systems. https://llvm.org/devmtg/2017-10/slides/Reames-FalconKeynote.pdf

[24] Tom Spink, Harry Wagstaff, Bjoern Franke, and Nigel Topham. 2015. Efficient Dual-ISA Support in a Retargetable, Asynchronous Dynamic Binary Translator. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. IEEE, 103 – 112. https://doi.org/10.1109/SAMOS.2015.7363665

[25] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. 2003. A region-based compilation technique for a Java just-in-time compiler. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 312–323.