Compilation Scheduling Policy for JIT-Based Systems

Martin Kristien

Master of Science by Research
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2018

Abstract

To enable fast platform virtualisation, parts of the target applications are compiled Just-In-Time (JIT) to enable fast native execution, as opposed to interpretation. As the compilation is performed while the application is being executed, it contributes to the end-to-end runtime. To offset the compilation cost, only frequently executed (i.e. hot) code regions are compiled. Further speedups can be achieved by decoupling the application execution from the compilation sub-system. A compilation queue facilitates the transfer of code regions selected for compilation between the two sub-systems. Although several researchers pointed out the importance of policy used for scheduling the code regions in the compilation queue for the actual compilation, little research of practical policies has been done.

This project explores new compilation scheduling policies. The most significant innovation is the *Dynamic Heat* policy. The policy schedules code regions based on the value of execution frequency (i.e. heat). This heat is dynamically derived by profiling before and after the code regions are dispatched to the compilation queue. The novel policy results in the maximum speedup of 2.14X over the state-of-the-art policies. Average speedup reaches 1.24X. The *Dynamic Heat* policy is further improved by two more policy iterations. Although these two policies resulted in significant speedups for some benchmarks, only marginal improvement was achieved on average.

The importance of compilation scheduling policy was also evaluated. Highest policy-related performance sensitivity was observed for benchmarks that have high percentage of time spent in the interpretation mode. Furthermore, compilation scheduling policy was determined to be a more effective mechanism for achieving speedups that increasing compilation throughput by employing more compilation workers.

Acknowledgements

I would like to thank my supervisor, Björn Frake, for academic guidance throughout the course of the project and writing this thesis.

I would also like to express thanks to Igor Böhm, who helped me with many implementation issues and with making good design decisions.

Finally, I am grateful to Zuz Olsinova for emotional support during the whole project and for proof-reading this thesis.

This work was supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh.

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.

(*Martin Kristien*)

Table of Contents

Chapter 1

Introduction

We can see many examples of systems that defer machine specific code generation until runtime. These systems allow applications to run in environments that are different from their original target platforms by running in virtual machines rather than on actual physical hardware. Such machine virtualisation can be seen across the whole computing spectrum.

Starting with cloud computing, machine virtualisation enables hardware consolidation, creating and allocating virtual machines on demand [36]. Furthermore, virtualisation provides a secure environment through separation by the virtualisation layer [35]. In desktop computing, machine virtualisation can be utilised to run several operating systems side-by-side without the need of rebooting [13]. Furthermore, everyday users encounter virtualisation when using applications distributed in platform agnostic representation. These include all web-based applications (e.g. Google Maps) and JAVA based applications. Machine virtualisation is also present in mobile computing. For example, Android Studio can use virtualisation to emulate a full Android system, allowing developers to write mobile application from the comfort of their desktops [10]. Lastly, in embedded world, virtualisation allows for early-stage software development and debugging of plethora of exotic hardware and even architectures that possibly do not exist yet [20, 24, 8].

In all of these scenarios, a program compiled for one architecture is to be executed on a different architecture. To execute code in this platform-agnostic manner, runtime systems must either resort to interpretation or they can dynamically compile application code and execute it natively. Since interpretation is orders of magnitude slower than

native execution, it is generally agreed that high performance virtualisation cannot be achieved without dynamic (i.e. Just-In-Time JIT [1]) compilation.

However, JIT compilation is performed during application's execution and thus contributes to the overall end-to-end runtime. Therefore, all JIT systems must carefully consider which parts of the application are worth compiling. Specifically, the fast native execution of compiled code must outweigh the initial compilation cost. This issue boils down to three fundamental questions: *What* to compile, *When* to compile, and *How* to compile.

The question of *What* to compile concerns the formations of compilation units that correspond to parts of the application's code base. *When* to compile allows systems to make decision about when a compilation unit is considered worthy of compilation. Finally, the question of *How* to compile decides the overall orchestration of the compilation subsystem and its integration into the rest of the virtualisation system.

1.1 Problem Statement

The primary concern of this thesis is to expand the question of *When* to compile compilation units. In particular, the thesis explores how to order compilation units that are already deemed worthy of compiling for the actual compilation. We abstract the collection of compilation units into a compilation queue. The question of *When* is then transformed into a scheduling problem over this queue.

We define an optimal scheduling policy as one that maximises the amount of native execution, and consequently reduces the overall runtime of application. Even with perfect information about application's behavioural pattern, finding the optimal schedule is shown to be an NP-complete problem [12]. A simple heuristic-based policy could, for example, prioritise the compilation units in the compilation queue based on the total future execution counts of the code regions corresponding to the compilation units. However, real JIT systems do not have the information about application's future code requirements. Therefore, real JIT systems can only estimate future application requirements by using past observations of the application's execution.

The current state-of-the-art compilation scheduling policies are found to be sub-optimal in minimising the overall runtime of applications. The scheduling policies in most JIT systems manifest as either *First-In-First-Out* (FIFO) queues [14, 31, 25] or priority

queues based on execution frequency (*heat*) of the code regions observed during profiling [26]. The *FIFO* queues use the profiling information only to answer the question of *When* a compilation unit becomes worthy of compiling. Although priority queues use the profiling information also to schedule units in the compilation queue for the actual compilation, the profiling stops when a code region is dispatched to the compilation queue. Therefore, no scheduling policy can take into account changes in the application behaviour that occur during queueing. In the presence of long queueing delays, this renders all current compilation scheduling policies sub-optimal.

1.2 Goal

The primary goal of this project is to improve performance of JIT systems by designing a new compilation queue scheduling scheme. The new scheme should decrease end-to-end runtime of most applications. The scheme should be robust so that no applications suffers a significant slowdown.

1.3 Motivating Example

To motivate the research of compilation scheduling, the effect of different scheduling policies is demonstrated below. The figures depict the execution of *perlbench* benchmark from *SpecCPU2006* [16] benchmarking suite in the form of a heatmap. The heatmaps show execution in different regions of the application's address space over time. Note, the time axis (horizontal) represents the number of executed instructions as this is the time seen by the executed application. Since the wall clock time is skewed by different speeds of interpretive and native execution, it is affected by the choices made by the compilation scheduling policy.

Red colour represents execution in interpretive mode. Blue colour represents execution in native mode. Colour intensity represents 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. Compilation scheduling policy *A* produces a heatmap that contains long horizontal red lines. These lines indicate the policy has failed to recognise the importance of the corresponding code regions. On the other hand, policy *B* does not result in the horizontal red lines but

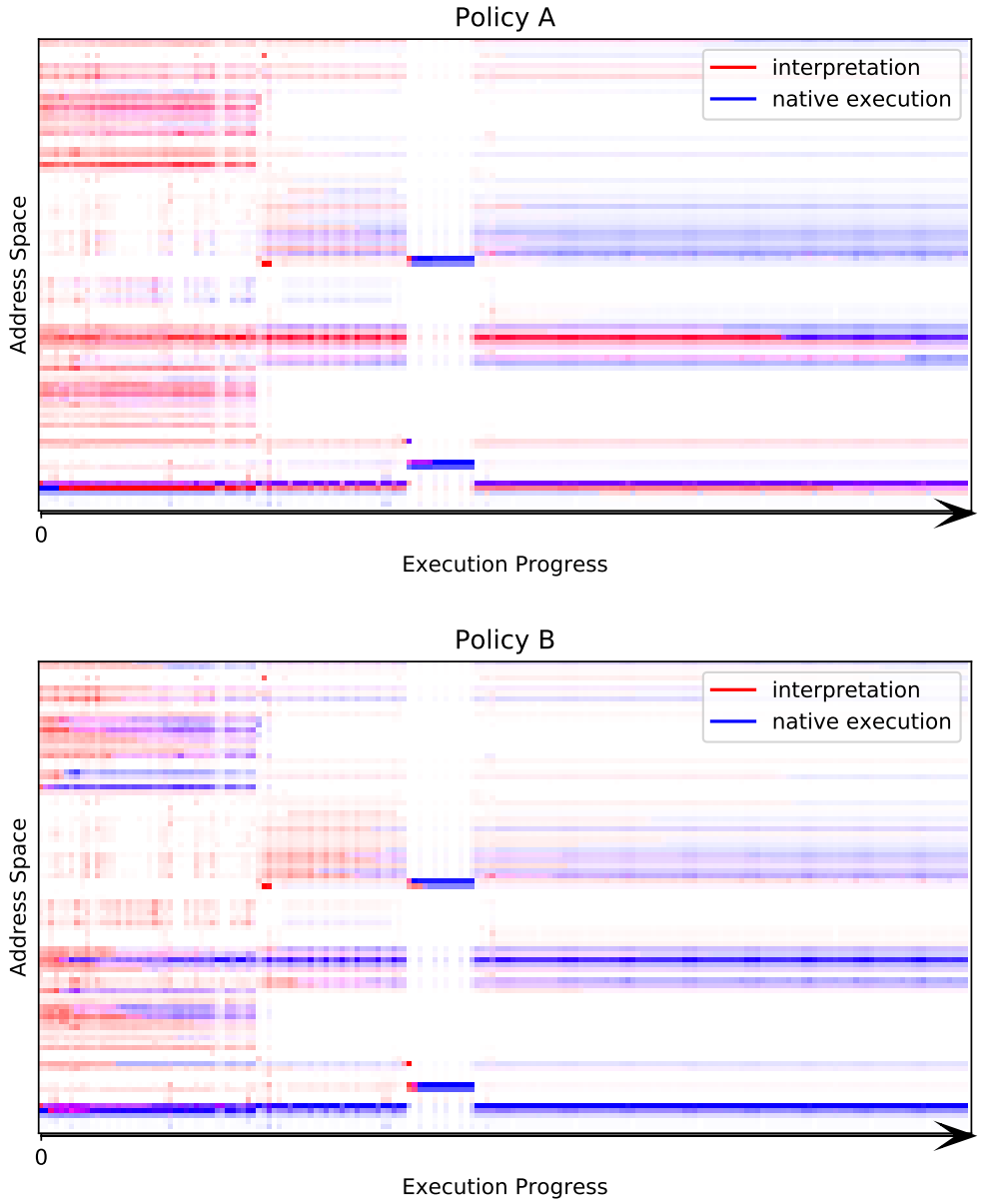


Figure 1.1: Execution of *perlbench* with different scheduling policies.

turns high intensity interpretation into native execution relatively fast. This indicates that the policy accurately recognises the important code regions and selects them for compilation with relatively short delay.

The two heatmaps in Figure 1.1 demonstrate the effect of the compilation queue scheduling policy on the performance of the whole system. In fact, policy B results in a speedup of 2.1X relative to policy A.

1.4 Contributions

These are the main contributions of this thesis:

- Novel compilation queue scheduling policies:
  - *Dynamic Heat* - profiling continues after compilation units are dispatched to the compilation queue
  - *Momentum* - like *Dynamic Heat* but with addition of recency aspect
  - *Compilation Time* - like *Momentum* but with addition of service time (i.e. compilation time)
- Addition of service time (i.e. compilation time) to scheduling policy
  - training a machine learning model for compilation time prediction
  - integration of the prediction model into a JIT system
- Analysis of importance of compilation queue scheduling policy in a JIT system
  - the effect of multiple compilation threads
  - the effect of application characteristic
- Visualisation of application execution in a JIT system (heatmaps)

1.5 Thesis Structure

The rest of the thesis is structured as follows. Chapter 2 introduces terms and concepts used throughout the rest of the thesis. It also puts the thesis into a perspective of broader research in the field. Chapter 3 introduces the particular JIT system used in this research project. The chapter also describes the infrastructure used for evaluation.

Chapters 4 to 7 explore various scheduling policies. Chapter 4 compares several standard scheduling policies found in the literature and industry. Chapter 5 introduces a novel scheduling policy based on dynamic updates of heat of the compilation units already present in the compilation queue. Chapter 6 builds on this approach by adding a recency aspect to the scheduling policy. Chapter 7 introduces compilation time as

a feature to be used for compilation scheduling. This is equivalent to service time in classical OS scheduling context.

After introducing various scheduling policies, Chapter 8 evaluates the given JIT system from other perspectives. First, all new scheduling policies are compared against the default policy and a baseline policy. Then, the importance of good scheduling policy with relation to application characteristics is explored. Finally, the importance of good scheduling policy in the presence of multiple compilation threads is investigated.

Chapter 9 concludes the thesis by critically evaluating the research and stating the future work of the project.

Chapter 2

Background and Related Work

This chapter provides the basic background necessary for this project. First, interpretation and native execution are described as two approaches to virtualisation. Benefits and drawbacks of each are presented. Then, several aspects of systems combining these two approaches are analysed. Finally, particular issues present in *Dynamic Binary Translation* are stated.

2.1 Virtual Execution Environments

Execution in a virtual environment requires two components, namely a model of the virtual machine and a transformation function that faithfully modifies the model according to the semantics of the application being executed. The virtual system is often referred to as guest system, while the underlying virtualisation platform is referred to as host. In the classical non-virtualised execution, code is executed on a processor in a *fetch-decode-execute* cycle. *Fetch* stage retrieves a code segment from memory. In a virtual environment, depending on the application, this can be a machine instruction or even a line of *JavaScript* code. In the rest of this thesis, we will use machine instruction as the code segment in question. *Decode* parses the instruction to derive a formal representation of the instruction's semantics, and *execute* performs the semantics by accurately modifying the state of the machine.

Interpretation achieves machine virtualisation by a direct software implementation of this *fetch-decode-execute* cycle. *Fetch* need loads an instruction from the model of the virtualised memory, *decode* performs software parsing, and *execute* modifies the

underlying model of the virtualised machine. Although this approach is straightforward to implement, it suffers a performance penalty as each stage usually results in many host instructions. Furthermore, interpretation operates only on one instruction at a time, which prohibits possible optimisations based on data flows among several instructions.

An alternative approach to machine virtualisation is to construct a small sequence of host instructions for each guest instruction performing the same modification to the machine state as the guest instruction would. This effectively translates the guest code into a corresponding host code. All the translations can be cached and reused. After the translation is performed, the resultant instruction sequence can be run on the host machine directly, achieving the same behaviour as interpretation but with fewer instructions, resulting in improved performance. In the literature, this approach is referred to as dynamic translation and consecutive native execution. Furthermore, the translation and native execution does not need to operate on only one guest instruction at a time. Translating several guest instructions as a single unit allows for optimisations to be performed, resulting in higher native code quality and better performance.

Although dynamic translation and consequent native execution result in fast execution of guest code, this approach incurs a significant computational cost in the translation step. To achieve the best of both interpretation and native execution, many systems combine the approaches by interleaving both modes of execution. The next chapter discusses several aspects of these systems.

2.2 Just-In-Time Compilation

Just-In-Time (JIT) compilation is a synonym for dynamic translation. JIT-based systems provide machine virtualisation by interleaving interpretation and native execution, generating native code on the fly. To take the best of both worlds, these systems must answer three important questions: *What*, *When*, and *How* to compile guest code into host native code. These questions are not formally defined and generally there is a lot of overlap between them. The following sections try to describe these questions as they are usually presented in the literature.

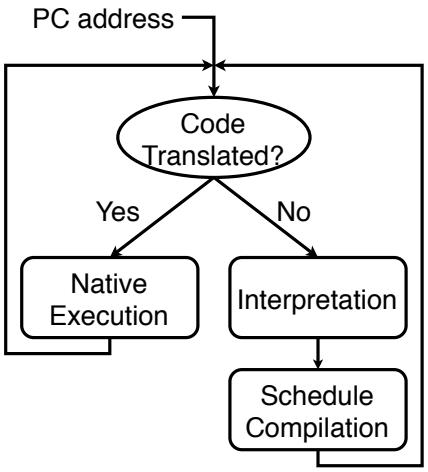


Figure 2.1: Execution flow in a typical JIT system.

2.2.1 What to compile

The question of *What* to compile concerns the formation of the compilation units. As discussed in the previous section, the simplest approach can compile one guest code segment at a time. Resultant native code is often of low quality as only few optimisations can be performed on a single code segment. The compilation is, however, relatively fast, reducing the latency to native execution.

A bigger compilation unit can be formed by compiling a basic block of guest code. Basic block is a linear sequence of instructions with a single entry point and a single exit point (e.g branch or jump instruction). Such a compilation unit allows for more optimisations to be performed, resulting in a better native code quality. Furthermore, as several guest instructions are executed at a time, fewer jumps between application native execution and the virtualisation system are required. This approach can be exemplified by QEMU's *tiny code generator* [5].

Control flow graph (CFG) is a graph where nodes represent basic blocks and edges represent jumps between the basic blocks. This can be considered as a full formal representation of the guest application. A linear trace is a subset of the control flow graph. It is formed as a sequence of basic blocks that only jumps forward without any cycles. Some systems construct the compilation units in the form of traces [3, 15, 7]. If a good trace is selected, the whole sequence of basic blocks can be executed natively without any interaction with the executing environment. What constitutes a good trace is discussed in the next section.

Code region is the coarsest granularity of compilation present in the literature. A region is a sub-graph of the whole CFG with multiple entry points and multiple exit points. It can contain both forward and backward edges, forming cycles. This form of compilation unit also allows for loop-based compiler optimisations, producing a high quality native code. Furthermore, the guest application can potentially execute only in the produced native code by running a high iteration count loop. Although this approach is common in JVM [34], binary translators adopted this approach only recently [28] due to increased compilation latency.

Compilation granularity results in a trade-off between compilation latency and native code quality. The next sections will discuss how the system can be tuned in the presence of this trade-off.

2.2.2 When to compile

As the dynamic compilation introduces a significant computational cost, compiling the whole application would result in sub-optimal performance. For example, if a particular function in an application is called only once, then compiling the function can introduce greater cost than what is the benefit of executing the function natively in the future. In other words, compiled parts of the application must be executed often enough in the future so that the fast native execution offsets the initial compilation cost. In a JIT system, this is determined by a model predicting the future behaviour of the guest application. Note, some successful systems avoid the question of *When* completely by compiling all code with a fast non-optimising compiler [5].

In general, the creation of this prediction model can be categorised into offline and online approaches. Offline approach derives the prediction data from past runs of the target application. For example, offline profiling collects execution counters for various interesting code structures, e.g. start of basic blocks, loop headers, or functions. These counters can be then used in future runs of the application to reliably predict how often a particular function will be called. Although this approach can gain perfect knowledge of the application's execution, it cannot be applied to unseen applications. Furthermore, the predictions might be inaccurate if the application's behaviour changes with different inputs. Since practical JIT systems cannot rely on application re-runs, offline profiling is mostly used in the research of other aspects of JIT systems [12, 29, 26].

Another offline technique splits the whole application into representative phases [38]. Then, the result of behavioural analysis in these phases can be extrapolated to the whole application. This technique, however, is not used in system virtualisation as it specifically avoids running the whole application which prohibits achieving the full functional behaviour.

Online techniques derive predictions dynamically, as the application is being executed, by analysing code regions where the application spends most of its time. For example, online profiling collects execution counts similarly to offline profiling [5, 17, 3, 15, 34, 6]. Sampling approach periodically samples the application thread to increment sample counts for the corresponding code regions [39]. These online techniques, however, can only observe the past execution of the application and thus do not have any information about the future behaviour. Such systems must assume the application's behaviour is relatively stable, i.e. the future behaviour will be similar to the observed past behaviour.

An alternative approach can derive the future behaviour dynamically by estimating loop iteration count [30]. If loop iteration count can be known at the first encounter of the loop, functions in the loop's body are bound to be executed at least as many times as the loop iteration count. This technique, however, relies on well formed programs, so that a loop structure can be indiscriminately recognised.

Besides the future execution counts, the compilation worth model can consider other metrics for each code segment. In particular, compilation time, execution time in interpretive mode, and execution time in native mode can be compared to analytically determine if the native execution can offset the compilation cost. However, this technique is not used in practice, as these metrics are generally unknown or hard to obtain dynamically. In practice, approximations typically based on execution counters must be used.

Once a prediction model determines the worth of code regions, threshold-based comparison can be made to decide if the code regions should be compiled. In most systems, the threshold value can be parametrised to control the aggressiveness of compilation [5, 17, 31, 14, 22, 15]. High threshold values would permit only the highest worth code to be compiled, resulting in a conservative compilation strategy and potentially a small fraction of native execution. On the other hand, low threshold values would try to compile almost all the code, dramatically increasing the load on the compilation sub-system. This could result in a high compilation latency, delaying delivery of the

native code, and postponing the native execution. Several studies have investigated the right setting of the threshold on modern machines [29, 26].

Others have partially avoided the threshold setting by using a dynamic threshold value derived from the current load on the compilation sub-system [6]. In particular, if the compilation queue is already congested, a high threshold value is used, and the other way around. Although the optimal threshold scaling based on the compilation queue length must still be tuned, this technique can keep the compilation sub-system busy while maintaining the compilation latency reasonably small.

2.2.3 How to compile

The question of *How* to compile concerns orchestrating the compilation within the virtualisation system. The simplest approach stops interpretation once the system encounters a code region deemed worthy of compiling [5, 19]. In the literature, this is referred to as foreground compilation, as it happens on the same thread as the application execution. Since foreground compilation introduces pauses to the application execution, this approach is highly sensitive to the compilation latency [15].

An alternative approach is background compilation, which offloads the actual compilation to a helper thread [15, 17]. As the compilation latency no longer causes the application pauses, this approach is preferred for highly responsive systems. However, the application throughput (i.e. the execution speed) can only be improved if the host platform has enough hardware parallelism to support both the application and the compilation threads without contention. If enough hardware parallelism is present, the approach can be further extended to concurrent and parallel compilation, employing several compilation threads [6]. Such systems significantly reduce the compilation latency and thus improve the application throughput.

The question of *How* to compile also concerns the actual compilation, in particular the optimisation level. The system always needs to make a trade-off between the compilation time and the quality of the produced native code. The simplest solution is to select a fixed optimisation level and design the rest of the system accordingly. For example, selecting a high optimisation level might motivate larger compilation units and a more conservative threshold to avoid long compilation delays caused by the long compilation time.

Alternatively, researchers tried predicting the best optimisation levels for individual compilation units [27]. This approach can selectively turn on particular optimisation passes based on features of the compilation unit. Others use tiered compilation, where a compilation unit is first compiled fast with few optimisations and based on the further profiling the same compilation unit can be recompiled at a higher optimisation level [17].

2.2.4 When to compile - revisited

The original question of *When* concerned deciding only when a code region is worth compiling. However, the question avoids deciding when to actually compile the created compilation units. Note, the latter issue is present only if the JIT system in question allows for existence of multiple compilation units at a time, i.e. the system must use background compilation and the compilation queue. The issue can be then understood as ordering/scheduling of the compilation units in the compilation queue.

Several researchers have pointed out the magnitude of the effect of the compilation queue scheduling on the performance of the JIT system [26, 29, 6]. A theoretical analysis of JVM using a perfect information of the application workload, the execution times, and the compilation times shows a good scheduling policy can produce up to 2X speedup [12]. Furthermore, is has been shown that finding an optimal schedule is an NP-complete problem and thus is impractical even with perfect information. However, a good heuristic can be obtained in polynomial time.

Practical explorations of the compilation queue scheduling are limited. Most state-of-the-art systems use the FIFO queues with the assumption that the compilation queue never grows into length that would cause a significant compilation delay [15, 14, 22, 17]. Others use priority queues based on some notion of heat [26]. The most sophisticated queue scheduling policy uses notions of both heat and recency to order compilation units in the queue [6].

All of the scheduling policies found in the literature order the compilation queue at insertion point, i.e. when new compilation units are added to the queue. Thus, no policy is able to capture changing application behaviour after the compilation units are dispatched.

2.3 Dynamic Binary Translation

Dynamic Binary Translation (DBT) specifically focuses on the virtualisation of machine code. This manifests into a direct translation of instructions from the guest Instruction Set Architecture (ISA) to the host ISA. ISA virtualisation is often used for early stage software development and debugging targeting embedded platforms from the comfort of desktop machines [5, 8]. ISA virtualisation, however, can also be seen directly in desktop machines. For example, Rosetta [21] is DBT developed by Apple Inc. to support legacy applications after the company changed the underlying ISA from PowerPC to x86. Since ISA virtualisation operates directly on machine code, DBTs need to face a new set of issues usually not present in the higher-level (e.g. language) virtual machines.

For programs in binary representation, ”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” [18]. This means that it is hard for DBTs to discover application’s code, as it cannot be done statically. Instead, application’s code is discovered incrementally, as the application executes. Such restriction prevents prediction of heat of undiscovered code regions. In other words, DBTs can answer the question of *When* to compile only by online profiling, looking into the past.

Furthermore, binary programs permit unstructured control flow. Therefore, DBT cannot rely on method or loop based counters during profiling. Instead, lower-level counters must be employed that rely on structures directly present in the Control Flow Graphs. For example, basic block or backward jumps profiling counters can be used.

Another challenging issue prevalent in binary programs is self-modifying code. In particular, interpretive execution can write into an application’s memory region corresponding to an already translated code. Furthermore, it is possible for already translated code to modify itself during native execution. DBTs must take special care for these cases, invalidating old translations and making sure no stale code is executed.

DBT systems also deal with issues similar to higher level JIT systems, such as online/offline profiling, foreground/background compilation, and compilation unit formation. However, DBT systems must often solve further low-level issues, such as efficient virtualisation of Memory Management Unit (MMU), support of interrupts, and Input/Output device emulation [32]. Therefore, DBT systems come at different

levels of integration with the host platform. Most common DBT systems are user-level applications that need to virtualise the full guest system [5, 6]. Other DBTs can be integrated with the host OS as kernel modules, taking advantage of the greater control over the host hardware and software resources [4, 37]. At the end of the spectrum, there are also DBT systems deeply embedded into the processor firmware, translating x86 instructions into energy efficient Very-Long-Instruction-Word (VLIW) host ISA [11].

2.4 Summary

Virtualisation can be achieved by either interpretation or native execution. Interpretation is inherently slow, and thus systems try to use as much native execution as possible by employing Just-In-Time compilation. Since JIT compilation is performed at runtime, it adds to the overall execution time. To make the best of both approaches, virtualisation systems must carefully consider *What*, *When*, and *How* to compile regions of application code.

JIT systems can compile the application code at various levels of granularity, starting with individual instructions, through basic block, and traces, up to compiling large code regions corresponding to the subset of the whole control flow graph.

Whatever code granularity is selected, only some code regions should be compiled. In particular, the future native execution of the compiled code regions must offset the initial compilation cost. Several techniques for predicting the future application loads have been developed as either offline or online analyses. The most prevalent technique is online profiling based on execution counters incremented on the fly every time a corresponding code region is executed. The execution counters are then compared to a particular threshold value to identify code regions worthy of compilation. This technique assumes that the past execution of the application indicates the future loads.

Compilation sub-system and its relation to the rest of the system must also be carefully designed. The possible design choices are between foreground and background compilation, selecting an optimisation level, and potentially employing a multi-tiered compilation.

Dynamic Binary Translation comes with another set of issues, namely dynamic code discovery, unstructured control flow, and self-modifying code. Furthermore, DBT

needs to simulate low-level details of the target system, such as MMU, interrupts, and IO.

The work presented in the thesis is set in the context of a DBT system. The system is a particular instance of all the issues discussed above. The next chapter discusses how the system resolves these issues.

Chapter 3

Research Infrastructure

This chapter describes the system in the context of which this project is being performed. The system is characterised using the features of JIT systems discussed in the previous Chapter 2.

As the project has iterative nature, each iteration is compared against the previous iteration. This evaluation, unless stated otherwise, is performed using the experimental setup described here.

3.1 Just-In-Time Compilation System

The JIT system used in this project, nSIM [23], is a direct descendant of a research Dynamic Binary Translator [6] adopted by Synopsys Inc. nSIM is a full system simulator capable of emulating ARCompact ISA on a x86 architecture. For this emulation, nSIM employs both interpretation and native execution. A flow chart representing the logical design of the execution of nSIM is depicted in Figure 3.1.

nSIM decouples the application threads from the compilation threads using parallel task-farm design pattern. The application threads communicate with the compilation threads using a compilation queue. The presence of the queue allows for spawning any number of the application threads for multi-core simulation and any number of the compilation threads for a greater compilation throughput on parallel hardware.

Compilation units take form of non-linear code regions representing large control flow sub-graphs. The code regions are always limited to already discovered code and are

partitioned at page granularity. This results in the formation of relatively large code regions allowing more compiler optimisations acting across several basic blocks. As the compilation engine performs these optimisations, the compilation time becomes relatively high compared to other DBTs (e.g. QEMU [5]). However, the compilation produces a high quality native code.

Code to be compiled is selected based on online profiling. nSIM employs a mixed mode of sampling and counter based profiling. During interpretation a counter associated with each basic block is incremented at every entry to the basic block. Then, every 10000 instructions a sampling point is generated. At this point, all accessed pages are inspected for the code to be compiled. If a page contains basic blocks whose counters exceed a certain threshold, a compilation unit is formed and added to the compilation queue. The sum of basic blocks' counters in a code region is taken as the heat of the resulting compilation unit. The sequence number of the sampling point is taken as the recency of the compilation unit. The threshold is dynamically re-evaluated based on the current compilation queue length. The longer the queue the more conservative (i.e. higher) threshold is used, scaling linearly with the queue length.

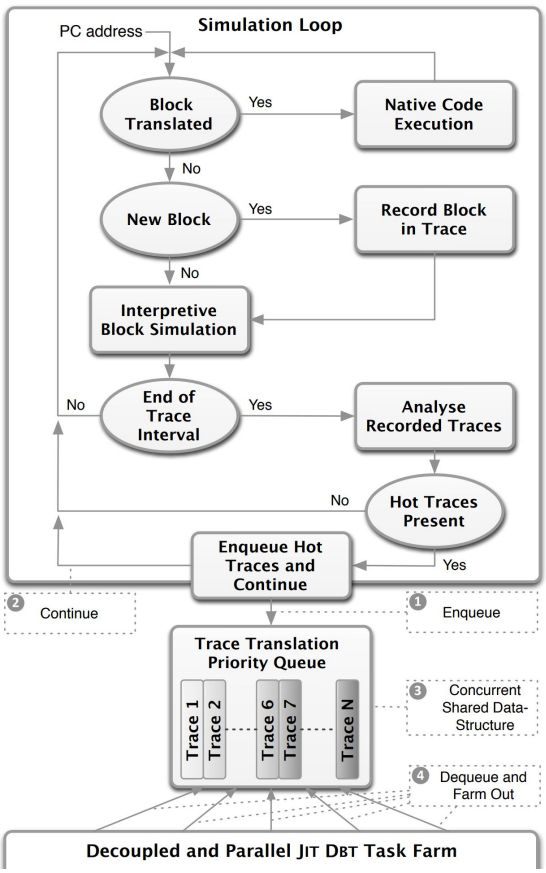


Figure 3.1: Execution flow in nSIM [6]

As the application thread only operates on basic blocks, a handle containing the future native code is attached to all basic blocks corresponding to the compilation unit during dispatch. If a basic block is waiting for the compilation (i.e. the corresponding compilation unit has been dispatched but not yet compiled) the application thread checks the handle to poll the status of the pending compilation. When the compilation finishes, the handle is updated with the native code, the application thread retrieves the

compiled binary, and the application can switch to native execution.

The compilation queue is organised as `std::priority_queue` with added synchronisation for concurrent access. The queue is ordered using a simple function based on recency and heat of the compilation units. The ordering always prefers more recent compilation units. In case of a matching recency (i.e. the compilation units were added at the same dispatch point), the units are ordered according to their heat. The exact function used for comparing compilation units in the queue is depicted in Listing 3.1.

```
bool less (CompilationUnit x, CompilaitonUnit y) {
  if (x.dispatch_time == y.dispatch_time)
    return x.heat < y.heat;                    // compare heat
  else
    return x.dispatch_time < y.dispatch_time;  // compare recency
}
```

Listing 3.1: Comparator function used for ordering compilation units. This corresponds to the *Default* scheduling policy used in nSIM

3.2 Experimental Setup

The project was evaluated using three benchmark suites. *BioPerf* [2] comprises applications of medium code size from the life science domain. The *EEMBC* [9] benchmark suite is an industry standard suite for embedded applications. It mostly consists of applications with small computational kernels. *SpecCPU2006* [16] is another widely used benchmark suite consisting of CPU-based applications from a broad spectrum of domains.

All experiments were performed on the benchmarking machine specifies in Table 3.1.

All experiments, unless stated otherwise, were obtained by running each test in 15 iterations. The most important evaluation metric is the end-to-end runtime. Values reported correspond to the arithmetic average of the measured runtimes of the 15 iterations, error bars correspond to the standard deviation. To enhance the effect of the compilation queue scheduling policy on the resultant runtime, the compilation throughput was intentionally limited by using only one compilation thread. This restriction is deemed realistic as well, as an average user might not have a large hardware parallelism available or they might want to use the parallelism for simulating multi-core

CPU model	Intel(R) Core(TM) i7-4770
number of CPU	4
CPU frequency	3.9GHz
L1-Cache	32K instruction/data
L2-Cache	256K
L3-Cache	8M
Main memory	16GB
OS	Linux 4.4 (x86_64)

Table 3.1: Specification of the benchmarking machine.

systems.

Initial experiments showed the *EEMBC* benchmark suite is not significantly sensitive to the compilation queue scheduling policy. Most benchmarks in the suite produced relatively short compilation queues (sometimes the maximal queue length was no longer than one compilation unit) giving little scope to the scheduling policy to affect the runtime. Furthermore, due to the small code size of the benchmarks in the suite, the runtimes were mostly affected by the produced code quality rather than the compilation scheduling strategy. Therefore, the *EEMBC* benchmark was not used to drive the development on novel scheduling policies. All benchmarking suites, however, are used for the final comparison of the new scheduling policies in Chapter 8.1.

Using the benchmarking setup above, all benchmarks were initially evaluated using the default scheduling policy to produce the workload characterisation metrics. The metrics are presented in Appendix A.

Chapter 4

Standard Scheduling Policies

As an initial evaluation of the given JIT system, several standard scheduling policies were experimented with. In particular, the given policy (*Default*) is compared against a *FIFO* policy and a heat-priority queue (*Heat*) policy. Furthermore, the (*Stack*) and the *Random* policies are also experimented with.

4.1 Motivation

The *Default* policy is a relatively sophisticated scheduling policy among what can be found in the literature. It relies on both heat and recency to order compilation units, and thus also encourages temporal locality.

The *FIFO* policy is the most prevalent in the literature and the industry. Although it does not prioritise compilation units in any way, it has a strong sense of fairness and prevents a compilation unit from being indefinitely stuck in the queue. Note, in many systems all compilation units can have the same heat, which would be exactly the heat equal to the value of the set threshold for compilation (see question of *When* in 2.2.2). Systems using the *FIFO* policy often assume the compilation queue never grows significantly long, i.e. there is no significant compilation delay in the system, thus there is no need for further compilation unit ordering. Since our system can produce relatively long compilation queues due to the low compilation throughput caused by the highly optimising compiler, the *FIFO* policy is not likely to perform well in our system.

The *Heat* and the *Stack* policies represent the two intuitive metrics for prioritising com-

pilation units, namely execution frequency (i.e. heat) and recency, respectively. Both policies are predicted to perform better than *FIFO*. However, neither should perform as well as *Default*, as the policies represent a limited view of the compilation units and it is assumed combination heat and recency gives the best performance.

The *Random* policy orders the compilation queue randomly. Note, all compilation units in the queue must already have had heat above some threshold to be added to the compilation queue, thus a random schedule might still result in a decent performance. In fact, the performance of the *Random* policy is expected to be comparable to the *FIFO* policy. The *Random* policy mainly serves as a sanity check to verify other policies are doing smart and useful decisions.

4.2 Implementation

The first step necessary for experimentation with different scheduling policies was to remove the hard-coded compilation queue based on `std::priority_queue`. For this purpose, `CompilationQueueInterface` was created, abstracting the accessor functions expected from the compilation queue. Using this interface, the underlying data structures and algorithms supporting the compilation queue were abstracted from the rest of the system.

The *Default* policy was implemented the same way as in the original system, using `std::priority_queue` hidden behind the `CompilationQueueInterface`. The *FIFO* policy used `std::queue` as the underlying data structure. The *Heat* policy also used `std::priority_queue`, this time with the ordering function using only heat of the compilation units. The *Stack* policy used `std::stack` data structure. *Random* used `std::vector`. In this case, the compilation queue was not structured in any way. The selection was performed at pop-time by generating a random index into the compilation queue and taking out the corresponding compilation unit.

4.3 Comparison and Analysis

Experimental results comparing several standard scheduling policies are depicted in Figure 4.1. The figure compares speedups when using different compilation scheduling

policies relative to using the *Default* policy.

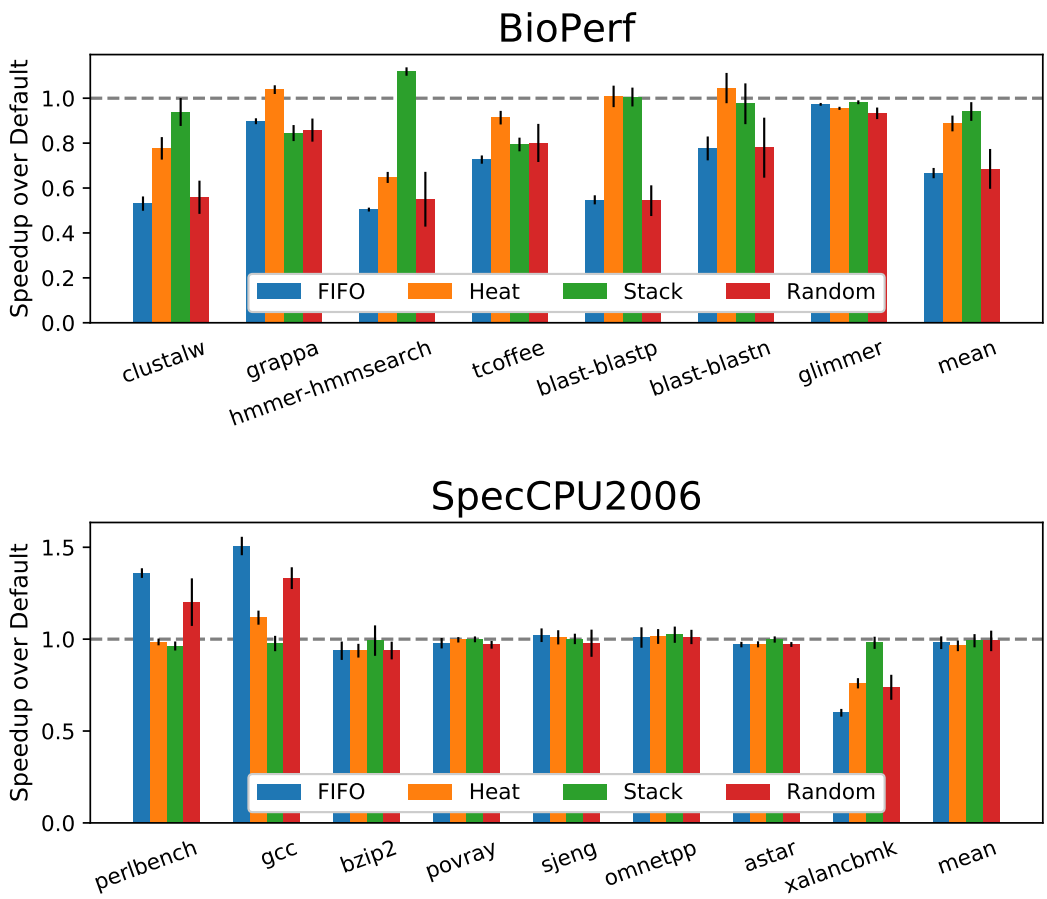


Figure 4.1: Speedup of several standard compilation scheduling policies relative to the *Default* policy. Higher means better.

First, we observe that the *Default* policy is the best policy on average for the *BioPerf* benchmarking suite. For the *SpecCPU2006* suite, all scheduling policies result in similar performance on average. This finding gives justification to the *Default* policy being used by our JIT system instead of any other standard policy. Furthermore, the *Default* policy can sometimes outperform the standard the *FIFO* policy by speedup of up to 1.98X (e.g. *hmmer-hmmsearch* in *BioPerf*). On average, the *Default* policy outperforms *FIFO* by 1.5X in case of the *BioPerf* benchmarking suite and is on-par (1.02X) in case of the *SpecCPU2006* benchmark suite. On the other hand, for some benchmarks *FIFO* outperforms the *Default* policy by 1.51X (e.g. *gcc* in *SpecCPU2006*).

Next, we observe that for each scheduling policy there exists a benchmark on which this policy outperforms *Default*. This finding suggests the sub-optimality of the *Default* policy and justifies research of better scheduling policies. For example, a meta-

policy which would select among several standard policies based on some application behaviour could result in the best performance for all benchmarks.

Finally, we observe that the *Random* policy outperforms *Default* for *perlbench* and *gcc* in the *SpecCPU2006* benchmarking suite, with speedups of 1.2X and 1.33X, respectively. This finding strongly undermines the viability of the *Default* policy, as it suggests a systematic flaw in how the policy predicts future code requirements of applications. For these benchmarks the *FIFO* policy seems like a better candidate. However, on average the *FIFO* policy performs sub-par to *Default* and it is also outperformed by *Random* for some benchmarks (e.g. *xalancbmk*). Note, for many benchmarks the *FIFO* and the *Random* scheduling policies perform on-par.

To better understand the paradox of the good performance of the *Random* policy, the execution of *perlbench* is visualised in Figure 4.2. The diagram shows heatmaps of executions for the *Default* and the *Random* policies. The horizontal axis represents the execution time in terms of executed instruction. At each time point the system only sees past behaviour of the application (to the left). Based on this execution observation, the system can make choices about which of the code regions (vertical axis) should be compiled next. The effects of these decisions are represented by the colour of the heatmap. Red colour represents interpretation while blue represents native execution. The colour intensity represents the number of instructions executed in the corresponding time-space region.

The better the policy the more of native execution should be observed, producing more blue heatmaps . It is apparent that the heatmap for *Random* is truly more blue than the heatmap for *Default*, which corresponds to the better performance of the *Random* scheduling policy. For both policies, the regions of bad performance are highlighted.

In case of the *Default* policy, we can see signs of a bad policy. In particular, long red horizontal lines indicate the policy failed to recognise the corresponding code regions as important for the application, resulting in long-term interpretation of the code regions. We hypothesize this is due to the policy putting too much relevance on recency. If an important code region is not compiled quickly, it may be ”overshadowed” by code regions from future dispatches, which would have higher recency. Putting more relevance on heat, however, does not improve the performance for *perlbench*. We further hypothesize that another factor causing sub-optimal performance of the *Default* policy is limited view of heat. Since at each dispatch point only the past behaviour

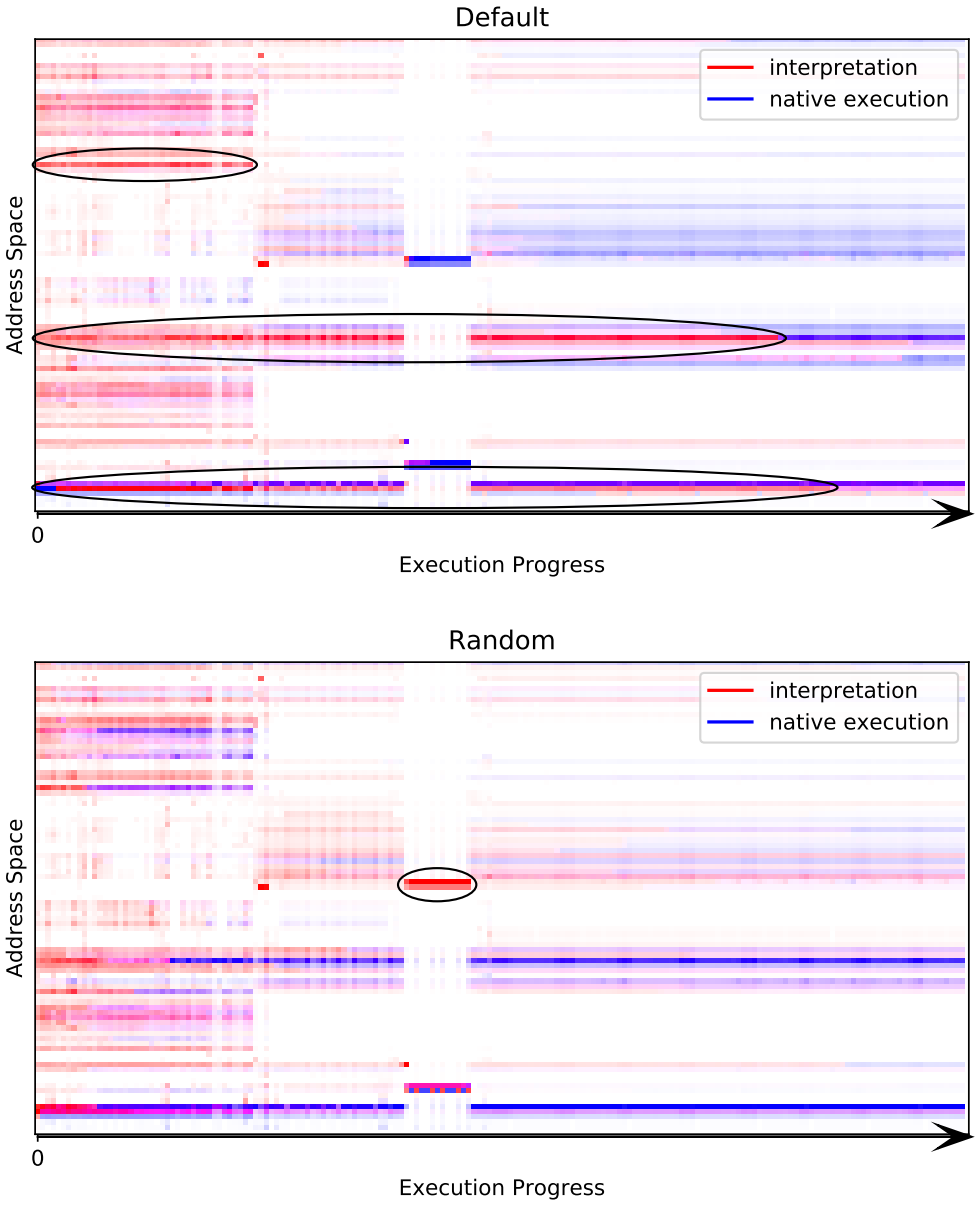


Figure 4.2: Execution of *perlbench* with the *Default* and the *Random* compilation scheduling policies.

is observer, if the application changes its behaviour after a dispatch point, the heat of already dispatched code regions is not able to reflect this changing behaviour.

4.4 Summary

After making changes to the JIT system to allow different implementation of the scheduling policy, several standard policies were evaluated.

The *FIFO* policy, the most prevalent policy found in the literature, resulted in the best performance for some benchmarks. However, our JIT system's *Default* policy (based on heat and recency) resulted in the best performance on average across all benchmarks.

The most surprising finding is that there are benchmarks for which the *Random* policy outperforms *Default* and *FIFO*. Note, there is no benchmark for which *Random* is observed to be the best policy. To better understand the problem, application execution was visualised in the form of a heatmap, showing interpretive and native execution over time and address space. The heatmap for *Default* clearly indicates misjudgement of the policy in terms of selecting the relevant code region for the compilation. In particular, the policy allowed interpretation of a code region for a long time.

We hypothesized this is due to the *Default* policy putting too much relevance on recency and having inaccurate view of heat of code regions. This hypothesis will be explored in the next chapter.

Chapter 5

Dynamic Heat

5.1 Motivation

The previous chapter showed clear sub-optimality of the *Default* policy for some benchmarks. In particular, the *Random* policy outperformed *Default* in terms of the end-to-end runtime. Furthermore, visualisation of application execution showed code regions that are being interpreted for a long time without the policy recognising the regions' importance for the application. For convenience, this visualisation is also depicted here in Figure 5.1.

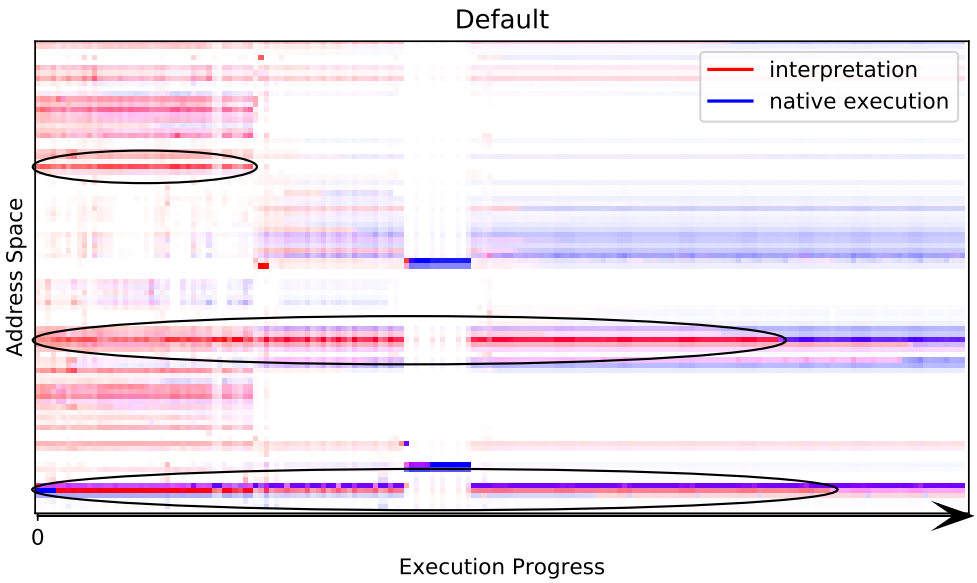


Figure 5.1: Execution of *perlbench* with the *Default* scheduling policy.

We hypothesise the long red lines were caused by the *Default* policy putting too much weight on recency. If a compilation unit was dispatched to the queue but not selected for the actual compilation before another dispatch point, all new compilation units would have higher priority due to being more recent. Unless all more recent compilation units have been compiled, the original unit would never be selected for compilation. This effect is similar to what can be observed with the *Stack* policy.

Detrimental effect of the indefinite delay of some compilation unit becomes more pronounce when the application becomes more dependant on the code region corresponding to the delayed compilation units. This seems to be the case for *perlbench*, as evident by the high intensity red interpretation lines.

5.2 Approach

A naive solution would use the *Heat* scheduling priority instead of *Default* to reduce the stack-like behaviour. However, as seen in Figure 4.1 in the previous chapter, the *Heat* policy does not perform any better than *Default* for *perlbench*. This can be explained by an inaccurate view of the heat metric. Note, heat of a compilation unit is set and fixed at the dispatch point. Since our JIT system uses dynamic threshold based on the current compilation queue size, early dispatched compilation units might have a relatively small heat due to the small threshold value at the beginning of the application. Even if the corresponding code regions are used by the application later, the compilation unit heat remains fixed to the value set during dispatch.

Instead of using the *Heat* policy, our system implemented a novel policy focused on reflecting the changing demands of applications. The policy relies on continued profiling of already dispatched code regions, resulting in dynamic updates of heat of the compilation units already present in the queue. The compilation units are then prioritised based on the values of this dynamically updated heat. Note, dynamic here means *after-dispatch*, rather than the conventional meaning *at-runtime*. At this point, we can further distinguish the previous *Heat* policy as using static heat, i.e. *at-dispatch*.

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

5.3 Implementation

Several changes for the JIT system were necessary to implement the *Dynamic Heat* policy. The main changes were in profiler, to produce the dynamic heat updates, and in the compilation queue organisation, to take those dynamic heat updates into consideration.

5.3.1 Profiler

In the original JIT system, profiling of basic blocks stopped after dispatch of the compilation units corresponding to those basic blocks. For each basic block a handle to contain the produced native code was registered with the basic block during dispatch. Note, a compilation unit can correspond to multiple basic blocks and originally the heat of a compilation unit was computed as a sum of visit counters of the corresponding basic blocks.

To allow the dynamic updates of 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 can be incremented through the handle on the basic block. During this update no synchronisation 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 basic blocks which are being interpreted. Note, interpretation is a much more costly operation in comparison to a single counter increment.

5.3.2 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 unstructured data structure, in particular `std::vector`. At selection point, a linear scan through this data structure is performed, finding a compilation unit with the maximal current (dynamic) heat. The precise selection algorithm is depicted in Listing 5.1.

```
CompilationUnit* CompilationQueueInterface::Pop() {
  CompilationUnit* max_unit = 0;

  // find compilation units with maximal dynamic heat
  for (CompilationUnit* unit : compilation_queue_) {
    if (!max_unit || unit->heat > max_unit->heat)
      max_unit = unit;
  }

  // remove selected compilation unit from the compilation queue
  compilation_queue_.Pop(max_unit);
  return max_unit;
}
```

Listing 5.1: Pop-time ordering by the *Dynamic Heat* scheduling policy

Although this change 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 with smaller delay. On the other hand, the compilation thread 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 synchronisation overheads already present.

5.4 Comparison and Analysis

Experimental results using the *Dynamic Heat* policy are compared to the *Default*, the *FIFO*, and the *Random* policies in Figure 5.2. The figure reports speedups relative to the *Default* policy.

We observe the *Dynamic Heat* policy is the best policy in case of *perlbench* by a significant margin. Achieved speedup of the *Dynamic Heat* relative to the *Default* policy is 2.14X.

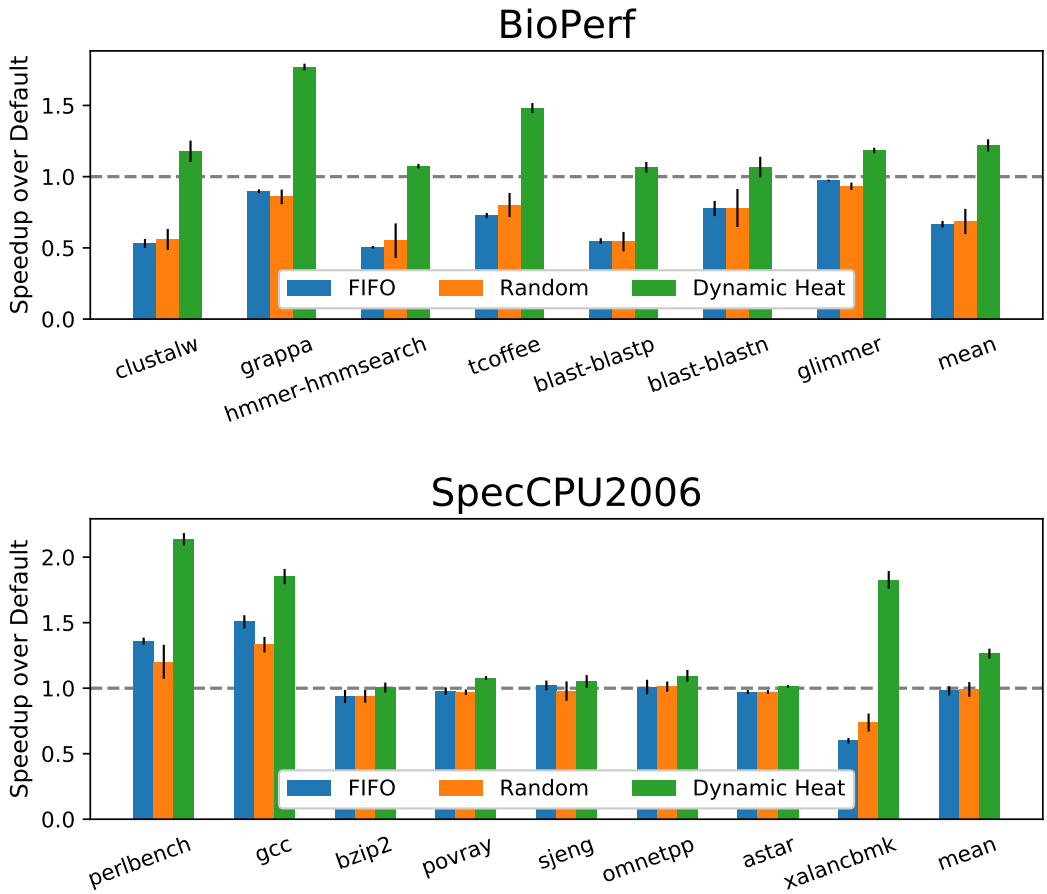


Figure 5.2: Speedups of the *FIFO*, the *Random*, and the *Dynamic Heat* scheduling policies relative to the *Default* policy. The higher the better.

More importantly, *Dynamic Heat* outperform or is on-par with other policies for all benchmarks experimented with. This result strongly supports our approach. On average, *Dynamic Heat* results in speedups relative to the *Default* policy of 1.22X in case of the *BioPerf* and 1.26X in case of the *SpecCPU2006* benchmarking suites.

For some benchmarks, using *Dynamic Heat* produces only marginal speedups. We believe this is due to the behavioural patterns of the individual benchmarks. The behavioural patterns of some benchmarks might be already optimally captured by the *Default* policy and some benchmarks might not be sensitive to the scheduling policy at all. The relationship between benchmark characteristics and the potential of the *Dynamic Heat* policy resulting in a speedup is explored in Chapter 8.2.

Heatmap visualisation for *Dynamic Heat* was also generated. Comparison of the heatmaps of the *Default* and the *Dynamic Heat* policies is depicted in Figure 5.3. The

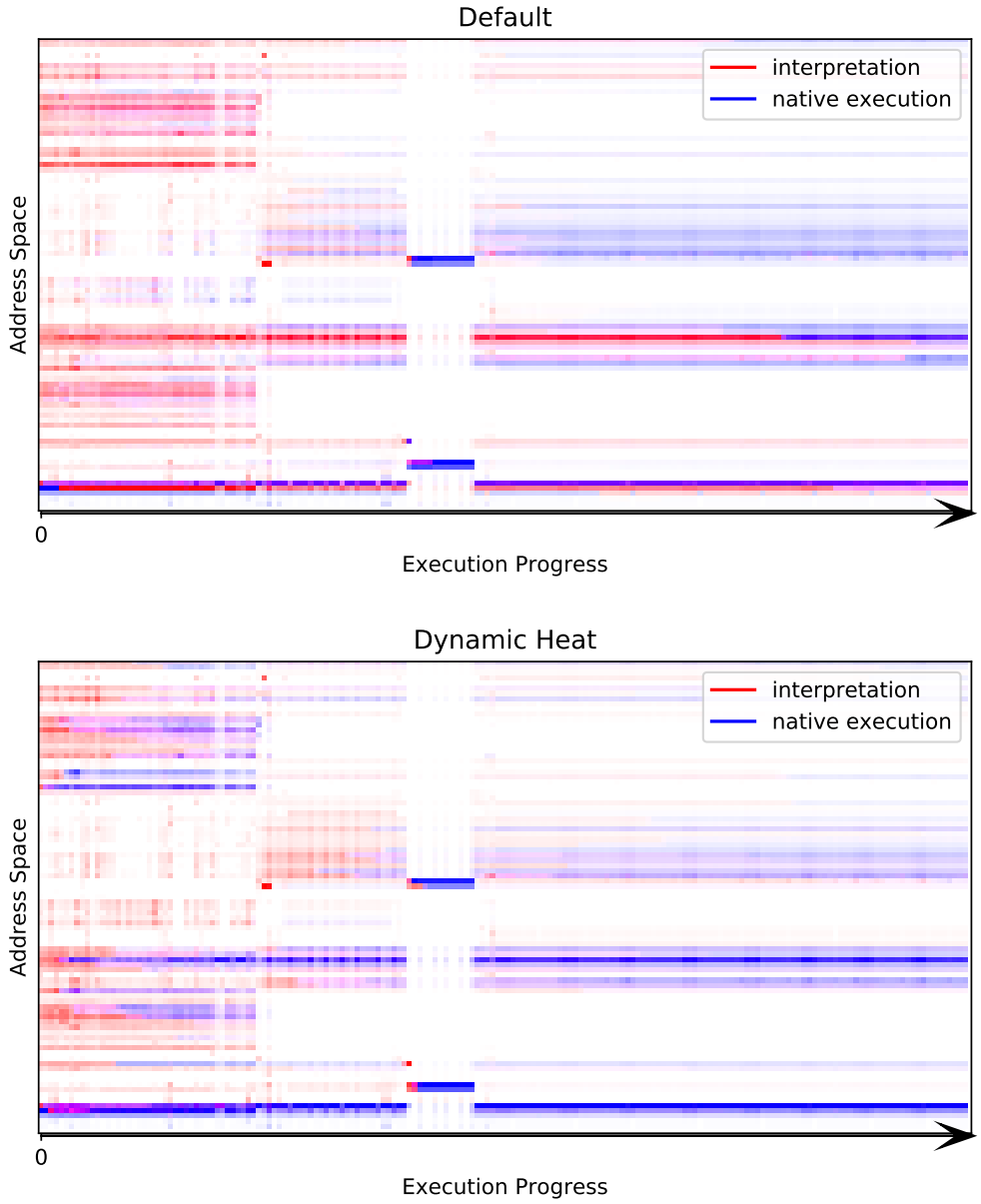


Figure 5.3: Execution of *perlbench* with the *Default* and the *Dynamic Heat* policies.

heatmap of the *Dynamic Heat* policy clearly shows the policy has the intended effect. Not only does the heatmap have much more blue colour (corresponding to native execution) but all high-intensity red regions are turned blue relatively quickly, preventing presence of the long red interpretation lines.

5.5 Summary

As the *Default* policy was shown to be sub-optimal, we hypothesised it was due to putting too much weight on recency and having an inaccurate view of execution frequency (i.e. heat). To mitigate this problem, a new policy, *Dynamic Heat*, was designed. The policy is based on continued profiling after dispatch of compilation units, resulting in dynamic (after dispatch) updates to the units' heat.

The *Dynamic Heat* policy resulted in significant speedups relative to the *Default* policy. For some benchmarks, the speedup reached 2.14X. On average, observed speedups reached 1.22X for the *BioPerf* and 1.26X for the *SpecCPU2006* benchmarking suites. The *Dynamic Heat* policy shows slowdown for no benchmark.

The success of the *Dynamic Heat* is deemed the most important finding of this thesis. Not only does the policy result in significant speedups, it is also simple, its implementation incurs insignificant costs, and it does not have any hyper-parameters that would require prior training/tuning. However, *Dynamic Heat* only considers execution frequency (i.e. heat) of the compilation units for ordering. Therefore, there is a room for improvement by adding consideration of a recency aspect to the policy. This addition is explored in the next chapter.

Chapter 6

Momentum

6.1 Motivation

Although the *Dynamic Heat* scheduling policy shows substantial improvements relative to the *Default* policy, it utilises only heat for ordering compilation units. Recency is another intuitive metric that can be used to make predictions about the future application behaviour. At any point in the application execution, the most recently accessed code regions are the most likely to be accessed in the near future, and thus are good candidates for compilation.

By ignoring recency, the *Dynamic Heat* policy fails to recognise when the application moves to a different set of hotspots. This behaviour can be referred to as phased behaviour. Many application might exhibit this phased behaviour, e.g. initialisation code in an application can be a different phase from the main workload code, or different stages in a compiler can be regarded as individual phases.

In the presence of such behaviour, right after application enters a new phase, the compilation queue might still contain compilation units dispatched from the previous phase. Note, these remaining units are likely to have a relatively high value of heat just by being in the compilation queue for a long time. New compilation units dispatched to the queue corresponding to the code regions from the new phase cannot initially compete on heat with the past-phase units. Although the new units are clearly more important for the application at that point, their compilation is delayed by the presence of units remaining from the previous phase.

Since heat of compilation units in the *Dynamic Heat* policy is being updated dynamically while the compilation units are already in the queue, we were able to monitor these heat updates per individual units to demonstrate the phasing problem described above. Heat evolution per compilation unit for benchmark *hmmer-hmmsearch* from the *BioPerf* suite is depicted in Figure 6.1.

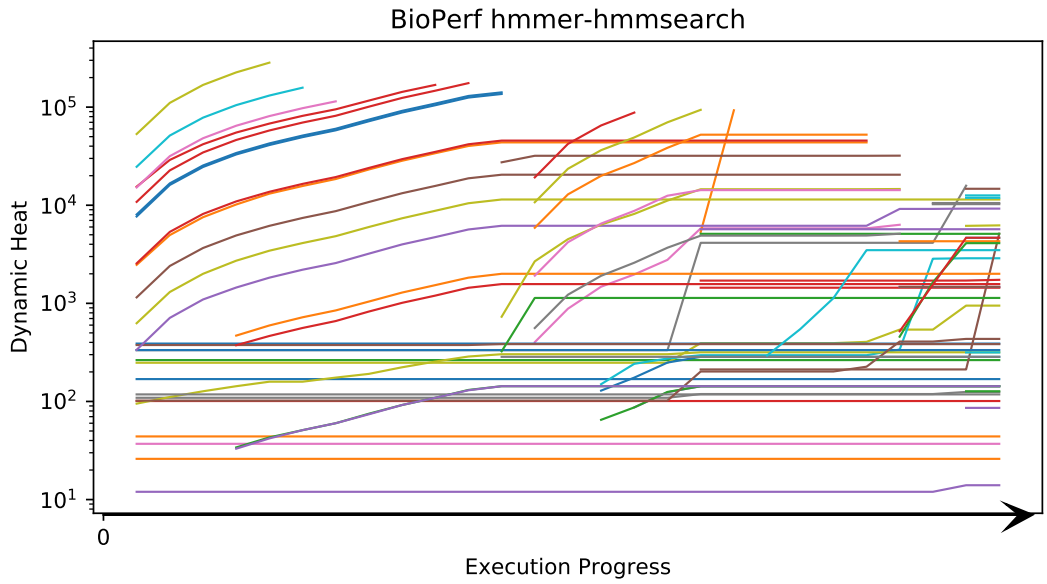


Figure 6.1: Evolution of heat of individual compilation units in the compilation queue over time.

Several observations can be made. Firstly, the evolution of heat indicates phases in the application. A phase can be identified by a set of compilation units that are being used, i.e. that increase in heat. If the set of the heat-increasing units changes a new phase is detected. Secondly, for several compilation units, their heat increases only for some time and then remains constant for the rest of the application execution. This can result in the phase-problem described above if the constant-heat units have higher heat value than the increasing-heat units. Indeed, such an instance of the problem can be observed in Figure 6.1. This chapter aims at tacking this particular problem.

6.2 Approach

To capture the recency aspect of compilation units, we took inspiration from the physical notion of heat. A hot object, if left unaffected, will gradually dissipate its heat to

the environment. To maintain the heat, the object must receive heat from an external source. In case of the compilation units, if the corresponding code is not being used by the application their priority should gradually decrease over time. For a compilation unit to maintain high priority its corresponding code must be continually used by the application.

This new priority metric is equivalent to using slope instead of value of heat from Figure 6.1 for ordering compilation units. Therefore, we call the new priority metric momentum. The *Momentum* policy then uses the momentum priority to schedule compilation units for the actual compilation. By the cooling and heating mechanism, the *Momentum* policy is capable of capturing the recency aspect of the compilation units as well as heat. The momentum priority can be interpreted as updates of heat of compilation units in the recent past.

6.3 Implementation

As there might be potentially many small heat updates to compilation units (one per interpreted basic block), monitoring all dynamic updates would not be computationally feasible. Instead, only cumulative heat updates propagate to the updates of momentum priority. This mechanism is based on sampling of the values of dynamic heat. Every 10000 interpreted instruction a sampling point is generated, a new value of dynamic heat is observed for each compilation unit in the queue, and the momentum priority for each unit is updated. This sampling frequency was empirically determined to result in the optimal trade-off between granularity of momentum updates and incurred computational cost.

Since the momentum priority is closely related to slope of dynamic heat, only difference between the current heat and the heat at the previous sampling point (i.e. heat derivative) is considered for the momentum updates. The heating mechanism is implemented by direct addition of the heat derivative to the previous value of momentum. The cooling mechanism is implemented using an exponential decay with parameter $0 \leq \alpha \leq 1$. At each sampling point, the previous value of the momentum is multiplied by α to produce the new momentum value. Note, values of α close to 1 result in relatively slow cooling (i.e. decrease of priority) while values close to 0 decay the momentum priority quickly. The exact momentum update is presented in Listing 6.1.

```
void CompilationUnit::UpdateMomentum(uint32 current_heat) {
  uint32 heat_addition;
  heat_addition = current_heat - past_heat_;
  past_heat_ = current_heat;

  momentum_ = momentum_ * alpha_ + heat_addition;
}
```

Listing 6.1: Update of momentum metric for individual compilation unit

6.4 Comparison and Analysis

The *Momentum* policy was evaluated for several values of α. The resultant speedups of using the *Momentum* relative to the *Dynamic Heat* priority for different benchmarks are presented in Figure 6.2.

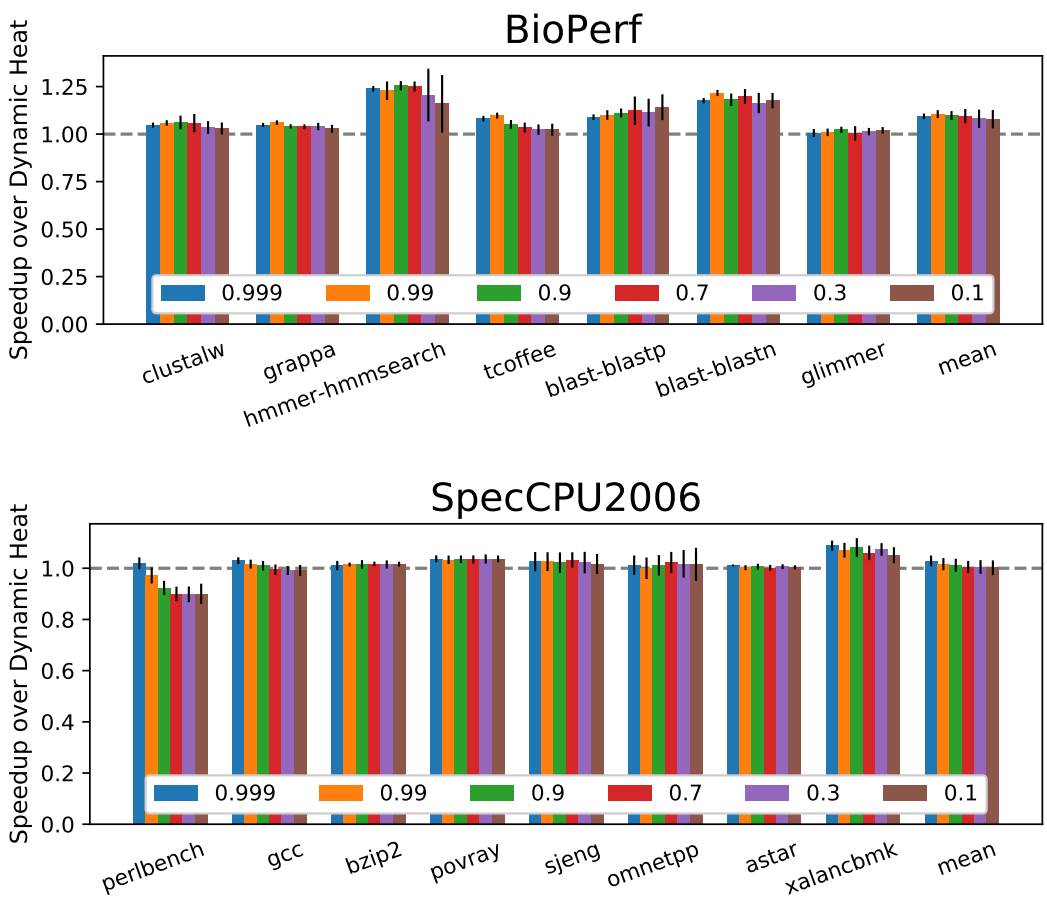


Figure 6.2: Speedup of the *Momentum* scheduling policy relative to the *Dynamic Heat* policy for several values of the α parameter. The higher the better.

Using the *Momentum* scheduling policy results in significant speedup compared to *Dynamic Heat* for *hmmer-hmmsearch* form the *BioPerf* suite, reaching 1.25X. This was expected, as this particular benchmark was used to motivate the *Momentum* policy.

Some other benchmarks also benefit from using the *Momentum* policy. However, using low value of α can degrade performance (e.g. *perlbench* with slowdown of 0.9X for $\alpha = 0.3$) by cooling compilation units too fast, resulting in the momentum priority being dependant on the very recent heat updates only.

The value of the parameter $\alpha = 0.99$ was empirically determined as the optimal value of the cooling parameter, as the value results in significant speedups for some benchmarks and produces significant slowdown for no benchmark. For this value of α, average speedup across all benchmarks in the *BioPerf* and the *SpecCPU2006* benchmarking suites were 1.1X and 1.02X, respectively.

6.5 Summary

To consider the recency aspect into scheduling of the compilation units in the compilation queue, a new *Momentum* policy was implemented. The policy relies on dynamic updates of heat of individual compilation units, similarly to the *Dynamic Heat* policy. However, the *Momentum* policy does not use the heat directly as the priority metric for unit ordering. Instead, it uses momentum priority that increases proportionally to periodic increases of dynamic heat. The priority also decreases (i.e. cools down) gradually if there is no increase of heat, using exponential decay with parameter α. The resultant priority can be interpreted as smoothed heat derivatives in the recent past, thus resulting in a policy that prioritises hot and recent compilation units.

Some benchmarks show significant speedups of using the *Momentum* policy relative to the *Dynamic Heat*, reaching 1.25X. Other benchmarks are on-par with *Dynamic Heat* for relatively high values of α. Low values of α can result in degraded performance, down to 0.9X. The value of $\alpha = 0.99$ was determined to be the optimal value of the cooling parameter, with average speedups of 1.1X for the *BioPerf* and 1.02X for the *SpecCPU2006* benchmarking suites.

The next chapter will explore addition of another priority metric often used for scheduling in other domains.

Chapter 7

Compilation Time

7.1 Motivation

In many scheduling problems, service time is considered an important metric. For example, this manifests in the OS scheduling as the *Shortest Job First* scheduling algorithm [33]. Furthermore, in embedded systems, scheduling algorithms must often satisfy hard or soft deadlines, which utilises the notion of service time as well.

Since interpretation is always available, there are no hard deadlines in our JIT system. However, service time can still be taken advantage of. Intuitively, in the compilation queue scheduling, if two compilation units have similar priorities, it should always be preferable to schedule the compilation unit with the shorter service time first. Note, in this scenario, service time is the compilation time. Such mechanism can further reduce the compilation delay by a small margin by preferentially compiling units that take less time to compile, thus increasing the compilation throughput in terms of the number of units compiled per unit time.

7.2 Approach

The first step towards using compilation time for scheduling is to instrument a mechanism for acquiring this metric for each compilation unit. For this purpose, a machine learning model can be used. The model, given a compilation unit, can predict the compilation time of the unit. Note, the success of the resultant scheduling policy largely

depends on the accuracy of this prediction model. The model training can be done offline. The trained model can be then integrated into our JIT system.

As the model's predictions only need to give accurate ordering of compilations units, the model is not specific to the performance of the machine used for collecting the training data. Therefore, the model is usable by other machines as long as the true compilation times can be correlated with the predicted compilation times. This restriction makes the model specific to the compilation algorithm rather than a particular machine used for training of the model.

Using compilation time as a sole feature for scheduling of compilation units would be counter-productive. Heat and recency are already established as important metrics for compilation scheduling, and thus scheduling based only on compilation time would likely result in degraded performance. Instead, predicted compilation time shall be combined with the already established priority of compilation units.

7.3 Prediction Model

7.3.1 Feature Selection

Selecting the input features for the prediction model is an important step in machine learning. As the model will be used in a JIT system, the features will be extracted from the inputs at runtime. Therefore, the features must be cheap to collect. Furthermore, not too many features should be used, so that the prediction model can remain simple and its evaluation will not be computationally intensive. The selected features should also be good indicators of the compilation time. This is a prerequisite for the model to be able to learn how to predict the compilation time.

Since the compilation units represent subsets of Control Flow Graphs, we selected model features that capture the complexity and size of the corresponding CFGs. Note, time complexities of many algorithms in compilers are expressed in these metrics, thus the features should be able to indicate the corresponding compilation time. The following features were selected:

- total number of instructions
- number of Basic Blocks

- number of edges in the CFG
- number of backward edges in the CFG

The total number of instruction is computed as the sum of instructions in individual Basic Blocks. This is the first-order measure of size of a compilation unit. The number of Basic Blocks and edges can be directly extracted from the CFG representation and they collectively capture the complexity of CFG. Backward edges further indicate compilation time, as they often correspond to loops in the CFG and several compiler passes specifically target loop-based optimisations.

7.3.2 Training

We selected linear regression as the model to be used for the compilation time prediction. The model is simple to implement, cheap to evaluate, and produces relatively accurate predictions. The four selected input features where transformed using a quadratic transformation function, to capture co-occurrences of the input features.

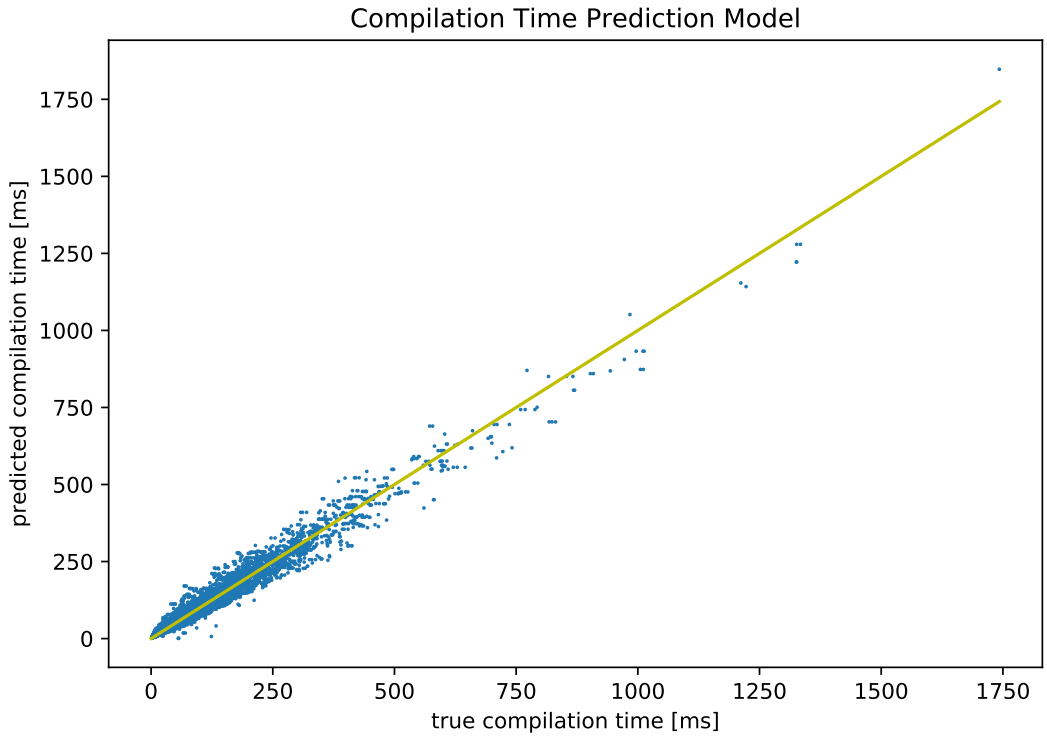
Data points for training were generated using all benchmarks in the three benchmarking suites used in this thesis. For each benchmark, the four input feature and the true compilation time for all produced and compiled compilation units were recorded. The resultant dataset contained 35877 data points.

Using 10-fold cross validation, the predictions of compilation time for all 35877 data points were generated. Since linear regression is known to be sensitive to outliers, these predictions were compared to the true compilation times to identify 35 outliers. The outliers were identified by having the largest values of prediction errors.

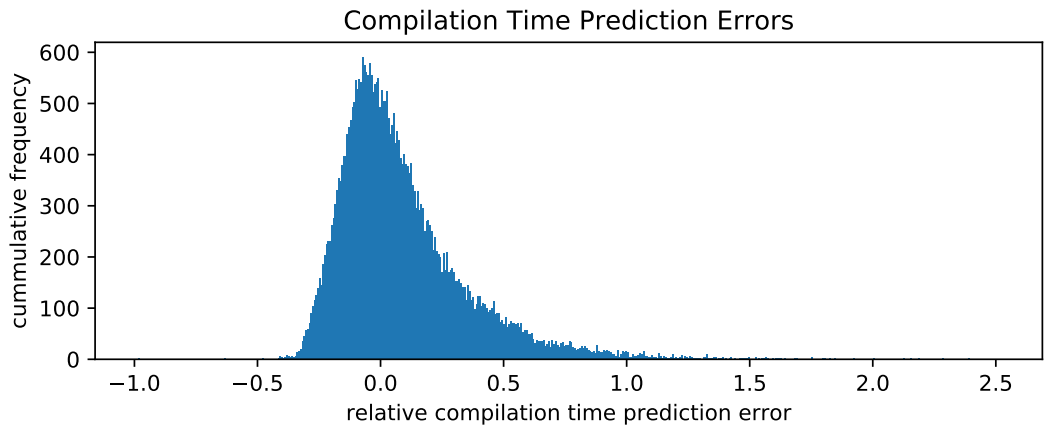
After removing the outliers from the dataset, 10-fold cross validation was used again to validate the model. Resultant fit can be seen in Figure 7.1. The model shows high degree of correlation of the predicted and the true compilation time, with the correlation coefficient of $R = 0.986$.

The relative prediction errors exhibit near normal distribution. The distribution is skewed towards overestimate of compilation time. We attribute this skew to the compilation times not following a simple polynomial distribution due to the non-polynomial time complexities of algorithms in our compiler engine. The maximum relative error reaches value of 9.13 standard deviation. Although the relative errors are significant,

the prediction model can still be useful to discriminate between compilation units with significantly different compilation times.



(a) Predicted vs true compilation times.



(b) Relative prediction errors.

Figure 7.1: Compilation time predictions produced using 10-fold cross-validation.

To finally train the model, the whole dataset without the outliers was used. Although we were unable to test the model using a separate testing data set, due to the simplicity of the model and the large number of data points in the training dataset, the model is

unlikely to overfit to the training dataset. In fact, using the trained model to predict the compilation times of all data point in the training set improves the correlation coefficient only to $R = 0.987$.

7.4 Implementation

The prediction model was integrated into our JIT system. During dispatching of the compilation units into the compilation queue, all new compilation units were given to the compilation time prediction model. The predicted times were stored with the compilation units to be readily available for the scheduling policy.

To use the predicted compilation time together with other scheduling metric, a combination metric and a combination function were selected. In the spirit of iterative development of policies, we took the momentum priority from the previous chapter as the combination metric, in particular the momentum priority with the cooling parameter $\alpha = 0.99$. For the combination function, we took weighted average for its simplicity. Note, weighted average introduces another hyper-parameter, namely the weight $0 \leq \beta \leq 1$. Large values of β indicate more weight is given to momentum priority while small values give more priority to compilation time. To meaningfully combine momentum priority and compilation time, both measures were scaled so that they take values in similar orders of magnitude.

7.5 Comparison and Analysis

The *Compilation Time* scheduling policy was evaluated with several value of the weight parameter β. Speedups of combining momentum priority with predicted compilation times relative to using momentum priority alone are depicted in Figure 7.2.

Data from Figure 7.2 demonstrates that compilation time can be used to improve the performance of our JIT system. Combining momentum priority and predicted compilation time with the values of β close to 1 result in speedups for several benchmarks and produced significant slowdown for no benchmark. The maximal speedup of 1.14X was observed for *blast-blastp* in the *BioPerf* suite for $\beta = 0.7$. Average speedup over all benchmarks also shows improvements for high values of β. The average speedup

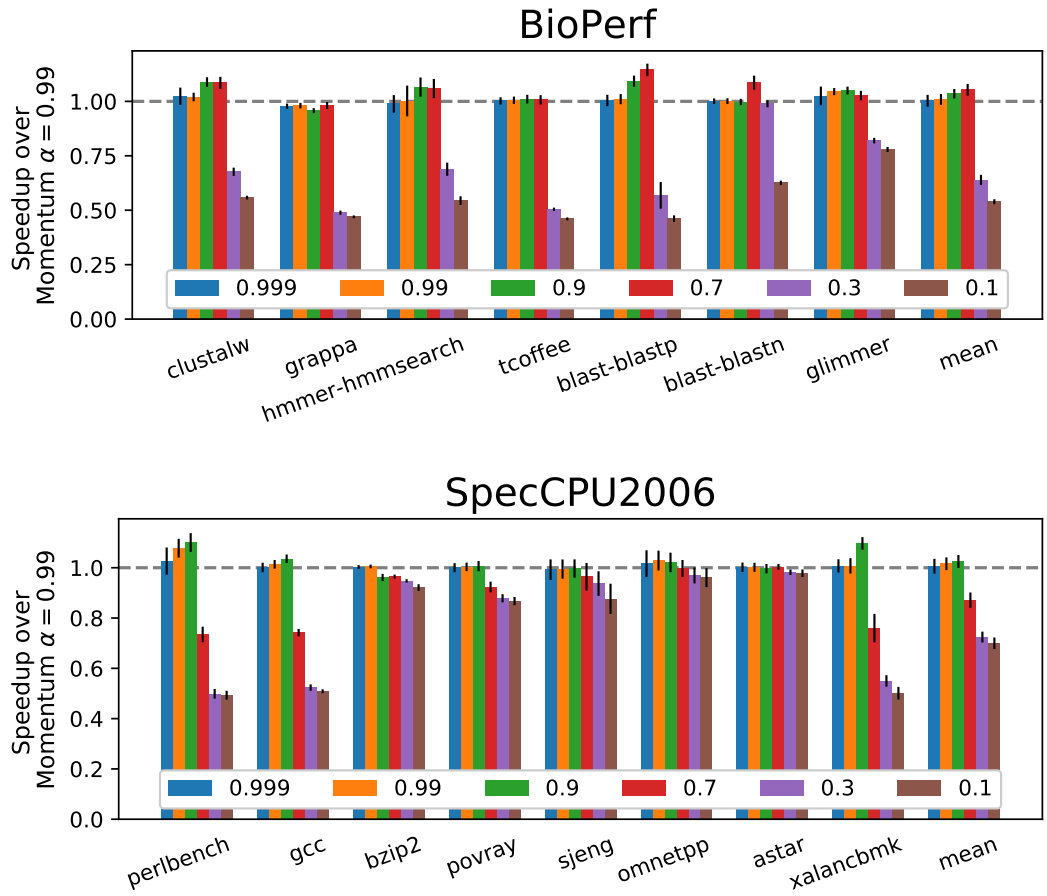


Figure 7.2: Speedup of the *Compilation Time* over the *Momentum* compilation scheduling policies for several values of the combination weigth β. The higher the better.

peaks at 1.05X for $\beta = 0.7$ in case of *BioPerf*, and at 1.02X for $\beta = 0.9$ in case of the *SpecCPU2006* benchmarking suite.

Decreasing the value of β (i.e. putting more and more weight on compilation time) seems to improve performance only to some optimal point. Further decrease of β degrades performance by prioritising compilation time instead of heat and recency, as expected. The optimal point, however, seems to be different for the two benchmarking suites.

To take a conservative approach and since $\beta = 0.7$ produces an average slowdown for *SpecCPU2006* of 0.87X, value of the combination parameter $\beta = 0.9$ was determined to be the optimal value. For this value of β, most benchmarks show a speedup or are on-par relative to using the *Momentum* priority. Only *grappa* in the *BioPerf* and *bzip2* in the *SpecCPU2006* suites show marginal slowdowns of 0.96X and 0.96X, respectively.

7.6 Summary

To further improve the compilation queue scheduling policy, compilation time was added and combined with momentum priority from the previous chapter. Compilation time is deemed important for scheduling, as it acts as service time, which is often used in scheduling problems in other domains. This is the first time compilation time is being modelled and used in a JIT context.

To derive compilation time for individual compilation units, a machine learning model was trained. The input features to the model were selected so that they are easy to obtain and they capture the complexity of units' Control Flow Graphs. Four features were selected, namely the number of instructions, the number of Basic Blocks, the number of edges and backward edges. Before feeding the input features into the model, they were transformed into a quadratic feature space to capture co-occurrences (i.e. to generate interaction features).

Dataset was generated using the compilation units observed in the *EEMBC*, the *BioPerf*, and the *SpecCPU2006* benchmarking suites. The dataset was used to train a linear regression model. Resultant model was integrated into out JIT system to predict compilation times for all compilation units dispatched to the queue. These predictions were combined with momentum priority from the previous chapter using weighted average with the combination parameter β.

The *Compilation Time* scheduling policy showed speedups for several benchmarks, provided more weight was given to momentum priority than predicted compilation time. If too much weight was given to the compilation time performance rapidly degraded due to not using heat and recency to make compilation scheduling decisions. Finally, combination parameter of $\beta = 0.9$ was empirically determined as the optimal value. For this value most benchmarks showed improvements in performance or were on-par with the *Momentum* policy, and only two benchmarks showed minor slowdowns.

This was the last compilation scheduling policy, and thus it concludes the exploration of scheduling policies. The next chapter evaluates the whole JIT system and the role of scheduling policies in the JIT system.

Chapter 8

Evaluation

This chapter evaluates compilation scheduling policies in our JIT system form a different perspective. First, the policy exploration is evaluated in terms of achieved speedup of the novel policies relative to the original *Default* policy. Then, benchmark sensitivity to scheduling policy will be explored by matching benchmark characteristics with observed policy related speedups. Finally, the importance of scheduling policy as a whole will be evaluated by analysis of relationship between policy related speedups and the number of compilation threads (i.e. compilation throughput).

8.1 Final Policy Comparison

To compare all explored scheduling policies, data corresponding to *Default*, *Dynamic Heat*, *Momentum*, and *Compilation Time* were collected. Note, the *Momentum* policy used the heat cooling parameter $\alpha = 0.99$ and *Compilation Time* used the combination weight parameter $\beta = 0.9$. These are the optimal values found previously.

All three benchmarking suites, namely *BioPerf*, *SpecCPU2006*, and *EEMBC*, were used during this evaluation. The resultant speedups for individual benchmarking suites relative to the *Default* policy can be seen in Figures 8.1, 8.2, and 8.3. The *Random* policy is also included in the comparison as baseline policy that does not reason about the data. Note, all compilation units in the queue must have already achieved relatively high heat in order to be dispatched into the queue, thus all units are deemed worthy of compiling and the policy only shows the effect of compilation scheduling. This means that even the *Random* policy might perform relatively well for some benchmarks if

the benchmark is not ”sensitive” to scheduling policy. Some benchmarks in *BioPerf*

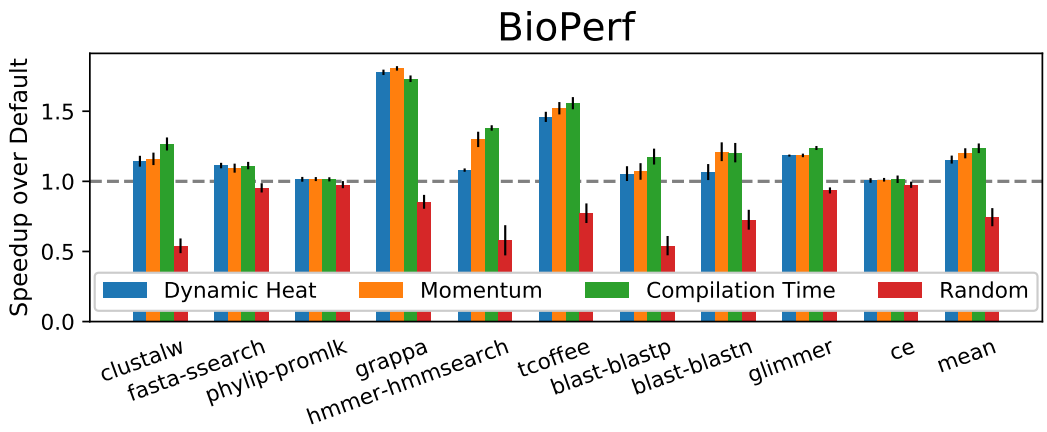


Figure 8.1: Speedups of all policies explored in this thesis for *BioPerf* benchmark suite. The higher the better.

benchmark suite show significant improvements of using the *Dynamic Heat* over the *Default* compilation scheduling policy. The greatest speedup of 1.78X is observed for *grappa* benchmark. Other policies introduced in this thesis performed even better. On average across all benchmarks in *BioPerf* suite, observed speedups are 1.16X for the *Dynamic Heat*, 1.2X for the *Momentum*, and 1.24X for the *Compilation Time* scheduling policy. The total best speedup is achieved for *grappa* benchmark using the *Momentum* policy. The observed speedup is 1.81X.

No policy introduced in this thesis results in a slowdown relative to the *Defaul* policy. In case of some benchmarks (e.g. *phylip-promlk* and *ce*), the scheduling policy does not seem to affect the performance. For these policies, even the *Random* policy performs on-par with the other policies.

SpecCPU2006 benchmarking suite also shows potential of our scheduling policies. *Dynamic Heat* achieves the best speedup relative to *Default* of 2.14X for *perlbench* benchmark. However, for this benchmark even the *Random* policy managed to outperform *Default*. The maximum speedup of the *Dynamic Heat* relative to the *Random* policy is actually achieved for *xalancbmk* benchmark with value of 2.48X. On average across all benchmarks in *SpecCPU2006* suite, observed speedups relative to the *Default* policy are 1.2X for the *Dynamic Heat*, 1.2X for the *Momentum*, and 1.21X for the *Compilation Time* scheduling policies.

The *Dynamic Heat* and the *Momentum* policies result in slowdown for no benchmark.

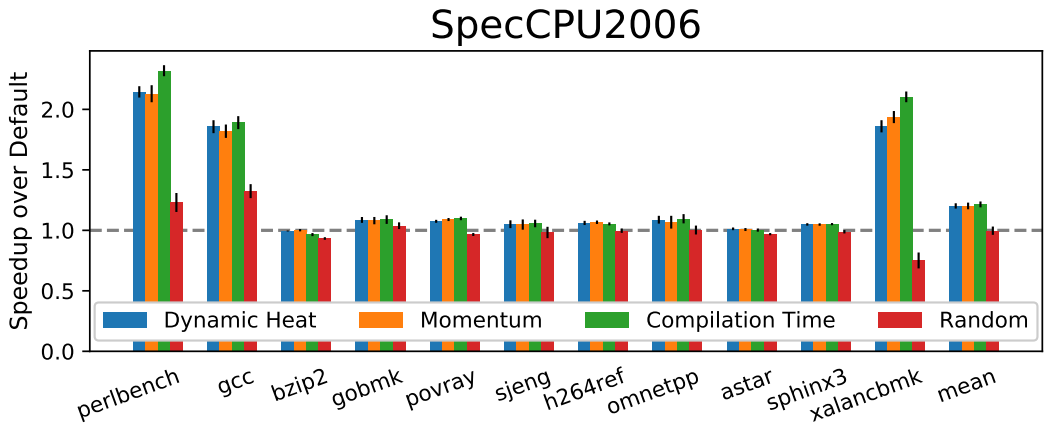


Figure 8.2: Speedups of all policies explored in this thesis for *SpecCPU2006* benchmark suite. The higher the better.

The *Compilation Time* policy shows a marginal slowdown of 0.96X for *bzip2* benchmark. As observed for *BioPerf* benchmarking suite, *SpecCPU2006* suit also contains several benchmarks that do not seem to be sensitive to compilation scheduling policy. For these benchmarks, all policies introduced in this thesis perform on-par with both the *Default* and the *Random* policies.

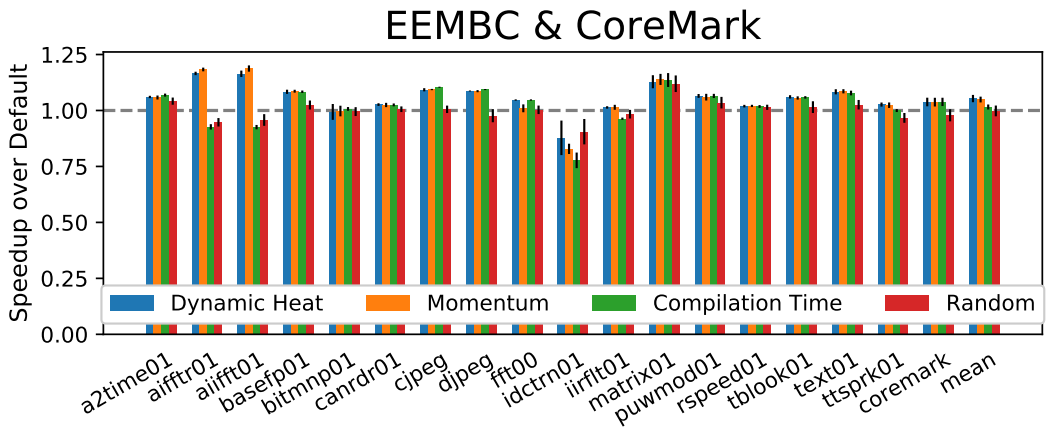


Figure 8.3: Speedups of all policies explored in this thesis for *EEMBC* benchmark suite. The higher the better.

Most benchmarks in *EEMBC* benchmarking suite also benefit from using *Dynamic Heat*, achieving maximum speedup relative to the *Default* policy of 1.17X for *aifftr01* benchmark. With the exception of *idctrn01*, the policy does not produce any slowdown. For *idctrn01*, slowdown of 0.88X is observed. The *Momentum* scheduling policy performed on-par with the *Dynamic Heat* policy for most benchmarks, achiev-

ing the best speedup of 1.19X for *aiifft01* benchmark. The *Compilation Time* policy, however, resulted in degraded performance for several benchmarks, sometimes performing even worse than the *Random* policy. On average across all benchmarks in *EEMBC* suite, observed speedups relative to the *Default* policy are 1.05X for the *Dynamic Heat*, 1.04 for the *Momentum*, and 1.02X for the *Compilation Time* scheduling policies.

The *Dynamic Heat* scheduling policy performed well in all three benchmarking suites. With the exception of *idctrn01* benchmark, *Dynamic Heat* never resulted in a slowdown. In *BioPerf*, *SpecCPU2006*, and *EEMBC*, average speedups relative to the *Defaul* policy reached 1.16X, 1.2X, and 1.05X, respectively, while maximum speedups were 1.78X, 2.14X, and 1.17X, respectively.

The *Momentum* and the *Compilation Time* scheduling performed similarly to the *Dynamic Heat* policy or slightly better for most benchmarks. These policies, however, rely on tuned hyper-parameters. Since the optimal values of the hyper-parameters might be context dependant (e.g. be specific to compilation throughput of the system or distribution of sizes of the compilation units), they are not considered as portable to new systems and applications as the *Dynamic Heat* policy. This tuning problem can be observed in *EEMBC* benchmarking suite, where *Compilation Time* resulted in degraded performance for several benchmarks. In this suite, benchmarks usually consist of small computational kernel, producing small compilation units. As most units in the compilation queue might exhibit similar compilation time, compilation time ceases being a good scheduling metric, and thus putting too much weight on compilation time might degrade performance.

8.2 Application Characteristics

In the previous section, some benchmarks seemed to be insensitive to compilation scheduling policy. This section tries to predict the sensitivity from other metrics that can be easily collected about individual applications. We do this by collecting several metrics from all available benchmarks and trying to correlate them with speedups observed by changing compilation scheduling policy from *Random* to *Dynamic Heat*. The *Dynamic Heat* policy serves as an example to an intelligent policy that is simple and performs consistently well. The *Random* policy serves as a baseline for the affect

of scheduling policy on runtimes, as it does not reason about the data in any way. Note, all compilation units in the queue must have already be deemed worthy of compilation to be dispatched into the queue, thus even the *Random* policy might perform relatively well. In fact, the *Random* policy reasons about the data as much as the *FIFO* policy, which is the most prevalent policy used in the literature. Even in our system, the *Random* policy results in comparative runtimes to the *FIFO* policy (see Chapter 4).

The application metrics collected are similar to the metrics used for benchmark workload characterisation in Appendix A. The metrics were correlated with observe policy related speedup. The individual correlations are depicted in Table 8.1. The metrics that have weak correlation with the speedup are greyed out. The strongest predictor of

#	Metric name	Correlation with speedup
1	Total number of instructions	-0.15
2	% of interpreted instructions	0.68
3	Total runtime	-0.04
4	% of runtime spent interpreting	0.81
5	Simulation rate (MIPS)	-0.59
6	Mean queue length	0.53
7	Maximum queue length	0.5

Table 8.1: Characterisation metrics and their correlation with policy related speedup

speedup based on correlation is the percentage of time spent interpreting. This results matches our intuition, as compilation scheduling aims at making good code selection to maximise native execution and thus reduce runtime. Similar argument can explain strong correlations of percentage of instructions interpreted and simulation rate. All three of these metrics indirectly measure performance of the whole system, and thus are expected to be correlated with the speedup as well as with each other.

Metrics related to queue length also show correlation with the speedup. Compilation scheduling policy directly operates on the compilation queue, trying to minimise compilation queueing delay for the most important code. Since long queues result in long queueing delays, long queues give more scope for a good scheduling policy to manifest into a runtime speedup. Therefore, both mean and maximum queue length metrics are correlated with observed speedups. Furthermore, they are expected to be correlated with each other.

Several metrics that are strongly correlated with observed speedups are expected to be also correlated with each other. To find these correlation, metrics covariance matrix is presented in Table 8.2. Furthermore, all features were used to fit a linear regression model, trying to predict the speedups. Since a lot of covariance among metrics is expected, *Principal Component Analysis* was performed to reduce dimensionality of the metric space. Resultant correlations of observed speedups and speedup predicted by the regression model for different degrees of PCA (i.e. number of dimensions used) are depicted in Table 8.3.

#	1	2	3	4	5	6	7
1	1.0	−0.17	0.9	−0.32	0.03	−0.08	0.08
2	−0.17	1.0	−0.08	0.74	−0.52	0.49	0.44
3	0.9	−0.08	1.0	−0.19	−0.15	0.21	0.39
4	−0.32	0.74	−0.19	1.0	−0.61	0.52	0.47
5	0.03	−0.52	−0.15	−0.61	1.0	−0.43	−0.48
6	−0.08	0.49	0.21	0.52	−0.44	1.0	0.97
7	0.08	0.44	0.39	0.47	−0.48	0.97	1.0

Table 8.2: Covariance matrix of collected metrics

PCA degree	1	2	3	4	5	6	7
prediction correlation	0.819	0.819	0.820	0.830	0.831	0.834	0.834
preserved variance (%)	52.4	79.8	88.7	95.6	99.3	99.9	100

Table 8.3: Correlations between speedups predicted by linear regression model and observed speedups using different number of dimensions in PCA.

The data above supports our expectations. Several application metrics are strongly correlated with each other. The strongest correlation is observed between mean and maximum compilation queue length (i.e. indices 6 and 7). Another example of a strong correlation is observed between the total number of executed instructions and the total runtime.

Furthermore, fitting a linear regression model after PCA transformation of the application metrics shows that constructing a single metric using a linear combination of the 7 original metric can still preserve as much as 52.4% of variance in the metrics

data, while resulting in correlation between predicted and observed speedups of 0.819. Note, using all 7 features results in the prediction correlation of 0.834 and using the best correlated metric, namely the percentage of runtime spent interpreting, directly givers correlation of 0.81.

This result indicates all metrics useful for prediction of policy related speedup are strongly correlated with each other and adding more metrics to the best predictor metric result in only marginal improvements of prediction accuracy.

8.3 Number of Compilation Threads

An argument can be made that compilation queue scheduling does not matter as one can always employ more compilation workers, thus increasing compilation throughput and reducing compilation latency, which is a more effective way of achieving speedup than changing scheduling policy. This section explores this argument by evaluating the importance of scheduling policy in presents of several compilation workers.

For the approach above to achieve the best potential speedups, hardware parallelism must support the increased number of compilation workers. Therefore, our experiments are limited to 1, 2, and 3 compilation workers. This is limited by 4 hardware cores on the CPU of our benchmarking machine. Note, 1 core is reserved by the application threads, which makes only 3 available for the compilation threads. Similarly to the previous section, we look at speedups using the *Dynamic Heat* scheduling policy and the *Random* policy under different number of compilation worker.

Researchers have already performed analysis of speedups for up to 14 compilation workers in a similar JIT system [6]. The results suggest the speedups achieved by using more compilation workers level off at different number of workers for different benchmarks. The number of compilation workers for which all benchmarks showed a speedup was 3. For some benchmarks, further increase of the number of workers resulted in degraded performance.

8.3.1 Effect of Multiple Compilation Workers on Runtime

This subsection explores the effect of employing several compilation workers on the resultant runtime. In particular, speedups using several compilation workers relative

to using only 1 worker are depicted. Separate experiments were performed for using the *Random* and the *Dynamic Heat* compilation scheduling policies. This is to find out if the effect of introducing more compilation workers on the runtime is different depending on what compilation scheduling policy is used. Figure 8.4 depicts speedups

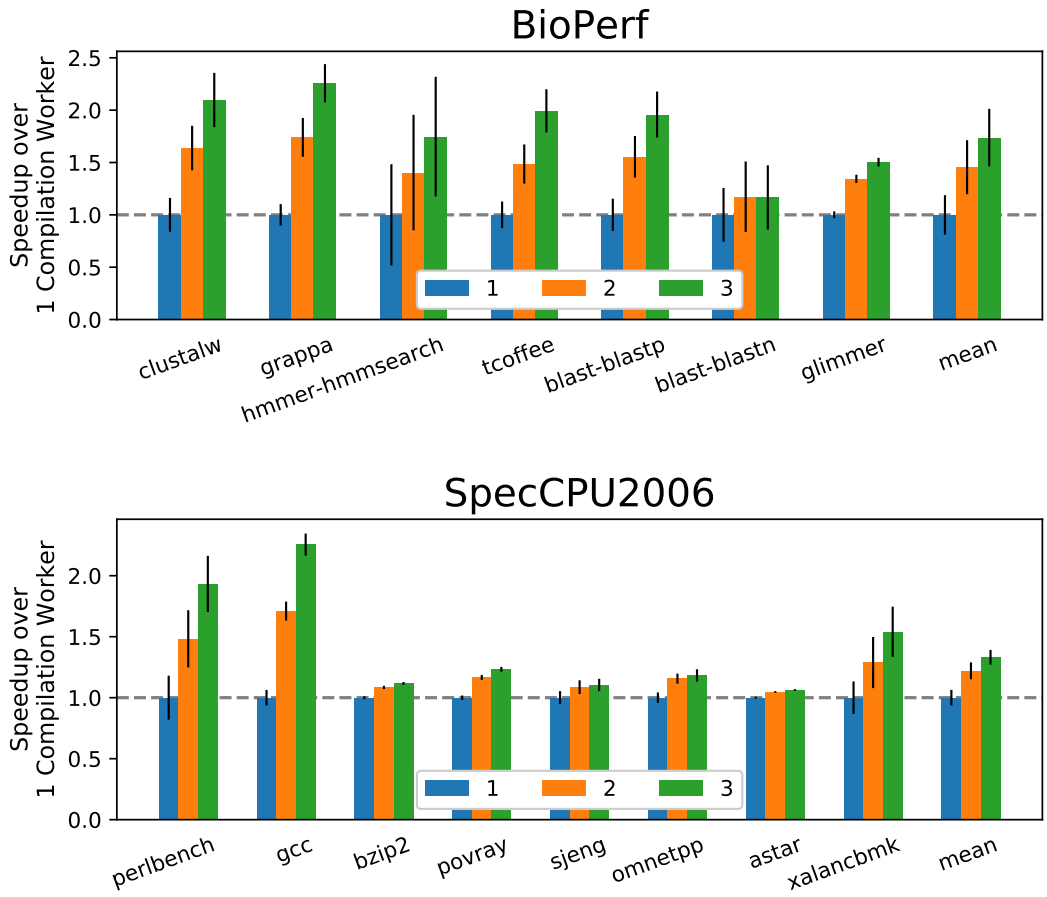


Figure 8.4: Speedup of using several compilation workers relative to using only 1 worker. The *Random* compilation scheduling policy is used.

achieved by using several compilation workers for the *Random* scheduling policy. As expected, using more compilation workers produced significant speedups by increasing the number of compilation units compiled per unit time. By compiling more code faster, the benchmarks could spend more time executing natively. The best speedups are achieved when using 3 compilation workers. The maximal speedups for *BioPerf* and *SpecCPU2006* benchmarking suites were 2.26X and 2.25X, respectively. Mean speedups across all benchmarks in the suites were 1.74X and 1.33X, respectively.

Interestingly, some benchmarks in *SpecCPU2006* benchmarking suites did not show significant speedups for higher numbers of compilation workers. For example, *astar*

benchmark exhibit speedup of only 1.06X. Such marginal speedup can be explained by the benchmark performing already well for with only 1 compilation workers. In fact, *astar* spends in native mode 99.8% of its instructions and 95.2% of its time when using 1 compilation workers and the *Defaul* compilation policy (see Appendix A). Since runtime of applications that are small in terms of code size or run for relatively long is not dominated by the compilation delay, reducing the compilation delay by employing several compilation workers does not produce significant speedups.

The same analysis was performed when using the *Dynamic Heat* scheduling policy. The speedups of using several compilation workers relative to using 1 worker for the *Dynamic Heat* scheduling policy are depicted in Figure 8.5.

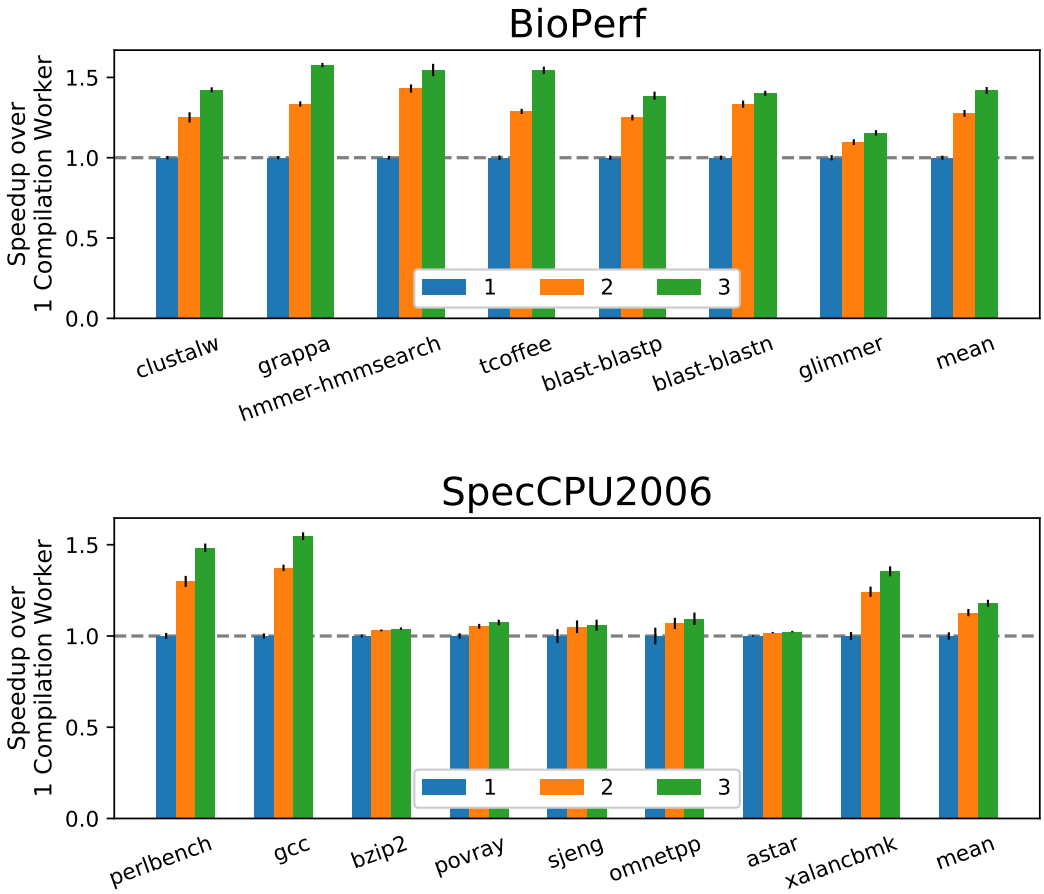


Figure 8.5: Speedup of using several compilation workers relative to using only 1 worker. The *Dynamic Heat* compilation scheduling policy is used.

Similar observations to the *Random* scheduling policy can be made for the *Dynamic Heat* policy. Higher number of compilation workers always produces a speedup but

for some benchmarks that are not dominated by the compilation delay the speedup is marginal. The best speedups are achieved when using 3 compilation workers. The maximal speedups for *BioPerf* and *SpecCPU2006* benchmarking suites were 1.58X and 1.56X, respectively. Mean speedups across all benchmarks in the suites were 1.42X and 1.18X, respectively.

The mean and maximum speedups per benchmarks are not as high when using the *Dynamic Heat* scheduling policy as when using the *Random* policy. In other words, when using the *Dynamic Heat* scheduling policy, adding more compilation units does not improve runtime of applications as much as when using the *Random* policy. This observation can be explained by the *Dynamic Heat* scheduling policy already performing relatively well when using only 1 compilation worker, leaving less room for speedup gained by parallelisation of the compilation subsystem.

$$speedup = \frac{1}{(1-p) + \frac{p}{\#\,of\ compilation\ workers}} \tag{8.1}$$

An analogy using the Amdahl's law can be made (Formula 8.1), where p corresponds to the proportion of the runtime caused by the compilation delay. Since the *Random* policy performs much worse than the *Dynamic Heat* policy, compilation delay contributes much more to the overall runtime for *Random* than for *Dynamic Heat*, i.e. $p_{random} > p_{dynamic\_heat}$. With greater value of p for the *Random* policy than for *Dynamic Heat*, speedups gained from compilation parallelisation are also greater for the *Random* than for the *Dynamic Heat* policy.

To sum up, useful speedups can be gained from employing multiple compilation workers regardless of the compilation scheduling policy. However, when using a good scheduling policy, the gain of parallelisation is smaller than when using a bad policy. This supports the investment into a good scheduling policy. If addition of further parallelism does not pay-off as much for a good scheduling policy the available parallelism can be used for other purposes, e.g. virtualisation of multi-core systems.

8.3.2 Effect of Multiple Compilation Workers on Speedup

This subsection explores the effect employing multiple compilation workers has on the speedup of using the *Dynamic Heat* scheduling policy relative to the *Random* policy. The speedups for several compilation workers are depicted in Figure 8.6.

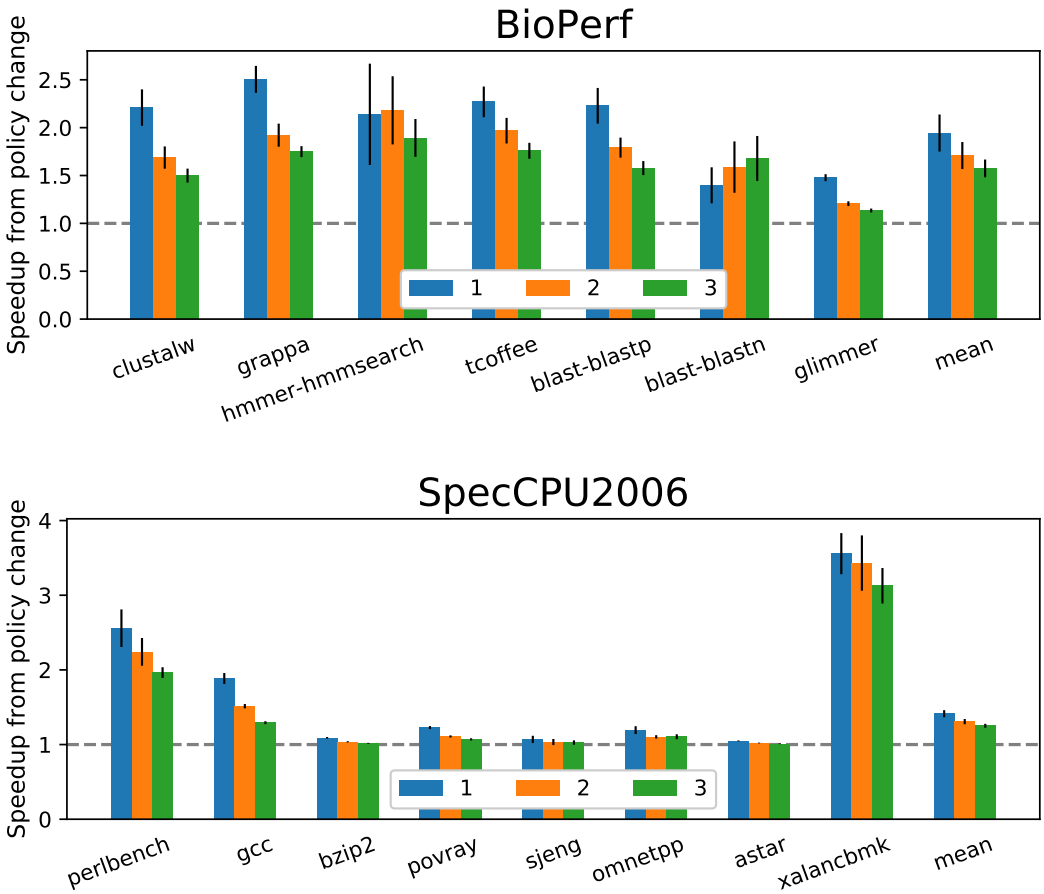


Figure 8.6: Speedup of using the *Dynamic Heat* scheduling policy relative to the *Random* policy for several compilation workers.

For most benchmarks, the best policy-related speedups are achieved when using only 1 compilation workers. However, for more compilation workers employed, changing the scheduling policy still results in significant speedups. This behaviours confirms the observations from the previous subsection. If the compilation subsystem is already performing relatively well by employing multiple compilation workers, further improvements related to scheduling policy have diminishing effect on the overall application runtime. In other words, the effect of scheduling policy on the resultant runtime is the strongest when only 1 compilation worker is used.

For 1 compilation workers, the maximum speedup from changing the scheduling policy for *BioPerf* and *SpecCPU2006* were 2.5X and 3.56X, respectively. The mean speedups over all benchmarks in the suites were, 1.94X and 1.41X, respectively. All four of these speedups are greater than the corresponding speedups achieved for the *Random* policy by switching from 1 to 3 compilation workers (see Figure 8.4). This

result clearly show superiority of investing into compilation scheduling policy over employing more compilation workers in achieving application speedup. Not only can the policy change produce greater speedups than tripling the number of compilation workers but policy change does not require additional computation power and is simple to implement.

To conclude, changing compilation scheduling policy is much more effective technique for achieving speedups than employing more compilation workers.

8.4 Summary

Compilation scheduling policies were evaluated from several different perspectives. Firstly, the explored policies were compared against the *Default* scheduling policy. The *Dynamic Heat* policy is considered to be the most significant improvement from the *Default* policy, showing speedup for most benchmarks and a slight slowdown for only one benchmark. Although the *Momentum* and the *Compilation Time* policies managed to achieve further speedups relative to *Dynamic Heat*, the observed speedups were less significant and only present for some benchmarks. Furthermore, the two policies require their hyper-parameters to be tuned.

Then, benchmark characteristics were correlated with the observed speedups of using *Dynamic Heat* relative to the *Random* scheduling policy. Several characterisation metrics were found to be correlated with each other. Therefore, using more metrics for predicting policy related speedups did not achieve much better predictions compared to using only the most speedup-indicative metric. The metric that can be best correlated with observed speedups is percentage of runtime spent interpreting, achieving correlation with policy-related speedup of 0.81X.

Finally, the importance of a good scheduling policy while utilising up to three compilation workers was evaluated. Regardless of the used scheduling policy, adding more compilation workers always resulted in a speedup. However, when an effective policy was used, the speedups achieved from adding more compilation workers were not as significant as when a less advanced policy was used. Alternatively, when more compilation workers were used, switching from less to more advanced policy resulted in smaller speedups than when only one compilation worker was used. In other words, both the policy and the number of compilation workers can improve performance of the

compilation sub-system, but once the sub-system is relatively optimal, further optimisations give diminishing returns. For achieving speedups, policy-related improvement was found to be more effective than adding more compilation workers.

Chapter 9

Conclusion

This project explored compilation scheduling in a JIT-based system. The primary aim was to explore policies used for selecting compilation units from a compilation queue for an actual compilation.

Chapter 4 compared several standard scheduling policies found in the literature. Chapters 5, 6, and 7 explored novel policies in an iterative manner with the aim of gaining better performance relative to the previous policy iteration. Altogether, three novel policies were devised.

Chapter 8 first evaluated the three novel policies against each other and the default policy of our JIT system. Then, policy-related speedups observed for all benchmarks used in this project were correlated with the several evaluation metrics of the benchmarks. The focus was on finding a model predicting an application performance sensitivity to compilation scheduling policy. Finally, the chapter explored the importance of scheduling policy on application speedup in the presence of several compilation workers.

9.1 Contributions

Evaluation of industry standard compilation scheduling revealed sub-optimality of the policies. Although our default policy outperformed all other policies on average, for some benchmarks the default policy performed worse than a random scheduling policy. Since the *Random* policy does not reason about the compilation units at all, outper-

forming the *Default* policy motivated further exploration of better scheduling policies. To analyse the policies' performance, execution visualisation of a benchmark for *Random* and *Default* was constructed. The visualisation depicted execution in interpretive and native mode over time and application's address space, forming a heatmap. Comparing resultant heatmaps for the *Default* and the *Random* policies revealed that the *Default* policy sometimes fails to recognise code regions that are important for the application. This was attributed to the strong bias of the policy towards recency, resulting in a stack-like behaviour.

A new policy, *Dynamic Heat*, was devised, targeting the shortcomings of the *Default* policy. The new policy employed continued profiling after compilation units were formed and dispatched to the compilation queue. This resulted in dynamic updates of the compilation units' heat. The policy scheduled the compilation units according to the value of this dynamic heat (the higher heat the higher priority). The *Dynamic Heat* policy resulted in the maximum speedup over the *Default* policy of 2.14X. The average speedups were 1.22X for *BioPerf* and 1.26X for *SpecCPU2006* benchmarking suites. This policy is considered the primary result of this project. The policy achieves the greatest speedup relative to *Default*, it always performs better than the *Random* policy, it is simple to implement, and it does not require tuning of any hyper-parameters.

The *Dynamic Heat* policy was further improved by adding a recency aspect to compilation scheduling. The *Momentum* policy used increase of the dynamic heat, cooling the compilation units over time if the corresponding code is not being used by the application and heating the units if the application continually executes the corresponding code. The new policy achieved further performance improvement for some benchmarks, resulting in the maximum speedup over *Dynamic Heat* of 1.25X. The average speedups were 1.1X for *BioPerf* and 1.02X for *SpecCPU2006* benchmarking suites.

The final policy, *Compilation Time*, built on the *Momentum* policy by also considering minimising compilation time for scheduling compilation units. By collecting compilation units produced by all benchmarks, a dataset of 35845 data points was formed. The dataset was used to train a linear regression model to predict compilation time for individual compilation units. After the resultant model was integrated into our JIT system, predicted compilation time was combined with momentum priority using weighed average. The *Compilation Time* policy showed the maximum speedup of 1.1X, averaging 1.04X for *BioPerf* and 1.02X for *SpecCPU2006* benchmarking suites.

The importance of compilation scheduling policy was also explored. By correlating speedups of *Dynamic Heat* over the *Random* policy with several application metrics, percentage of time spent interpreting was identified as the best indicator of policy-related speedup. It was also revealed that other application metrics (e.g. percentage of interpreted instruction or mean length of the compilation queue) were strongly correlated with each other. For this reason, using multiple metrics to predict policy-related speedup did not achieve much higher prediction accuracy than using only percentage of time spent interpreting alone.

Finally, policy-related speedups in the presence of up to three compilation workers were analysed. The results indicate that optimising the compilation sub-system can be achieved by either adding more compilation workers or improving scheduling policy. Between the two approaches, improving scheduling policy was found to be more effective at producing application speedups. More importantly, changing scheduling policy requires no additional computational resources, and thus is a viable option even in systems with little hardware parallelism available.

9.2 Critical Analysis

The project is deemed complete and successful. Nevertheless, it is important to critically evaluate the project and state any shortcomings and assumptions.

As the project was performed in the context of only one particular JIT system, its results might not be applicable in other contexts. The investigation of compilation scheduling policies can only be performed if the JIT system in question is based on background compilation using a compilation queue. Furthermore, for scheduling policy to affect performance, the compilation queueing delay needs to be significant. In our JIT system, significant queueing delay was caused by using a highly optimising compiler and setting compilation threshold to a relatively low value, resulting in production of many compilation units. If these assumptions do not hold in other JIT systems, improving the compilation scheduling policy might not result in any performance improvement.

Scheduling policy improvement might also not result in an application speedup if the application is already performing relatively well. In particular, since compilation scheduling policy tries to reduce the amount of interpretation, if the percentage of time

spent interpreting is low, a better scheduling policy might not result in any further speedups. Low amount of interpretation for an application can usually be seen when the current scheduling policy already captures the application's requirements. Moreover, long-running applications might have only short period of execution before all code is compiled. For these applications, the runtime is mostly limited by the speed of native execution rather than by the effectiveness of the compilation scheduling policy.

Another shortcoming of the project is the limited evaluation of the compilation time prediction model used in the *Compilation Time* scheduling policy. Although the model is relatively simple, and thus is unlikely to over-fit to the training set, the model is evaluated only using the compilation units that were used for the training of the model. In other words, there is no test set for the model. To verify the applicability of the model in a scheduling policy, the compilation time predictions should also be produced for unseen applications and unseen compilation units.

9.3 Future Work

The most interesting direction of future work is to test the new compilation scheduling policies in other systems. In particular, *Dynamic Heat* can be integrated into JVM [25] or V8 Javascript engine [14]. Positive results would indicate the robustness of the *Dynamic Heat* policy, leading to improvement of virtualisation systems across the whole computing spectrum.

Compilation scheduling policies can also be explored in the context of tiered compilation. In JIT systems that use tiered compilation, different compilation levels might need to handle different number of compilation units. Furthermore, depending on the optimisation levels, compilation time and consequently compilation queueing delay might vary as well. In the presence of such variety in the compilation subsystems, the research can focus on finding the optimal compilation schedules.

Another area of interest is multi-core simulation. The potential research can investigate how to combine dynamic heat updates if the updates are caused by execution of multiple application threads. Alternatively, new policies might be explored, targeting scenarios where a compilation unit is hot for only some application threads.

This project investigated scheduling of compilation units already present in the queue. The principles of the new policies, however, can also be applied for making decision

about the original question of *When* to compile. In particular, a policy derived priority can be directly compared against the compilation threshold to inform formation and dispatch of compilation units. For example, the momentum priority might be employed to dispatch only compilation units that have been used recently.

Appendix A

Benchmark Characterisation

To characterise the individual benchmarks, several metrics are reported for all benchmarks used in this project. The metrics were generated using the *Default* compilation queue scheduling policy and 1 compilation thread.

Due to convenience of running experiments in a timely manner, some benchmarks were excluded during the development of novel scheduling policies. Such benchmarks were either long running, had relatively short compilation queues on average, or already performed relatively well (in terms of % of time spend in native execution). The excluded benchmarks are greyed out in the tables below.

Note, all benchmarks reported here were used for the final system evaluation in Chapter 8.1.

A.1 EEMBC

Name	instruct. 10^6	instruct. native %	time sec	time native %	rate MIPS	queue len mean	queue len max
a2time01	1182	99.2	1.9	81.0	622	2.65	26
aifftr01	1097	97.6	1.84	49.6	597	9.13	26
aifirf01	1168	99.1	1.57	76.8	743	0.28	3
aiifft01	1051	97.6	1.74	49.9	603	8.44	26
autcor00	1169	99.9	1.42	96.6	822	1.88	3
basefp01	1168	99.0	1.88	75.4	621	1.78	17
bezier01	1149	99.8	0.68	85.6	1701	0.98	1
bitmnp01	1146	98.6	3.18	81.7	360	2.17	19
cacheb01	1342	99.0	2.16	76.5	622	0.42	5
canrdr01	1315	99.4	2.53	87.1	519	0.71	10
cjpeg	15379	99.6	19.0	88.0	809	3.43	34
conven00	1158	99.8	1.1	94.1	1049	0.99	2
dither01	1737	99.7	1.88	92.1	923	0.41	2
djpeg	13340	99.6	14.95	87.6	892	4.77	45
fbital00	1115	99.8	1.04	92.1	1074	0.06	2
fft00	1044	99.3	0.91	73.0	1147	3.17	17
idctrn01	1098	95.3	2.92	37.4	376	15.8	32
iirflt01	1047	98.7	2.1	73.3	497	0.98	9
matrix01	1675	96.4	5.19	55.9	322	11.41	47
ospf	1271	99.7	2.27	92.2	560	0.2	4
pktflow	1076	99.5	2.05	88.8	524	0.36	4
pntrch01	1058	99.2	2.28	86.7	463	1.6	11
puwmod01	1025	98.9	1.89	77.5	541	1.71	17
rgbcmy01	2073	99.8	2.0	91.1	1034	0.99	1
rgbhpg01	1274	99.8	0.78	92.4	1632	0.08	1
rgbyiq01	1597	99.6	0.97	78.8	1642	0.0	1
rotate01	1088	99.5	2.59	91.7	421	0.22	5
routelookup	1095	99.6	1.96	91.0	559	1.02	2
rspeed01	1008	99.5	1.44	85.7	702	0.88	7
tblook01	1308	98.8	2.56	76.1	510	3.36	28
text01	1135	99.2	2.47	84.7	460	2.74	20
ttsprk01	1126	98.4	2.28	69.4	494	4.26	19
viterb00	1100	99.7	1.43	89.8	772	0.22	3
coremark	1197	99.0	1.95	78.0	613	3.38	26

A.2 BIOPERF

Name	instruct. 10^6	instruct. native %	time sec	time native %	rate MIPS	queue len mean	queue len max
clustalw	833	93.7	3.37	43.3	247	64.79	96
fasta-ssearch	18390	99.5	29.71	87.2	618	9.65	72
phylip-promlk	46020	99.9	57.42	97.9	801	6.42	57
grappa	493	86.4	4.85	39.5	101	61.45	92
hmmer-hmmsearch	118	73.3	1.83	28.3	65	48.84	84
tcoffee	319	76.4	4.79	35.7	66	73.35	101
blast-blastp	1368	95.8	5.07	50.8	270	89.8	143
blast-blastn	60	58.2	2.0	32.6	30	141.34	202
glimmer	2447	98.0	6.36	69.4	385	16.21	53
ce	33897	99.9	57.13	98.2	593	7.2	44

A.3 SPECCPU 2006

Name	instruct. 10^6	instruct. native %	time sec	time native %	rate MIPS	queue len mean	queue len max
perlbench	536	60.1	17.06	33.3	31	110.41	165
gcc	5320	79.9	91.82	43.5	57	285.56	581
bzip2	20161	99.7	27.72	92.0	727	3.16	27
gobmk	45647	99.0	145.19	86.5	314	27.83	157
povray	20797	99.1	56.5	85.9	368	35.62	188
sjeng	15901	99.5	44.5	90.9	357	9.0	67
h264ref	101327	99.8	139.61	92.7	725	4.02	98
omnetpp	7532	98.7	33.8	85.4	222	21.05	138
astar	25581	99.8	37.44	95.2	683	2.75	35
sphinx3	47902	99.7	86.39	93.2	554	8.14	66
xalancbmk	541	59.9	15.41	30.0	35	345.23	480

Bibliography

[1] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[2] David A Bader, Yue Li, Tao Li, and Vipin Sachdeva. Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 163–173. IEEE, 2005.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *Acm Sigplan Notices*, 46(4):41–52, 2011.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

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

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

[7] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 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*, pages 18–25. ACM, 2009.

[8] Jianjiang Ceng, Weihua Sheng, Jeronimo Castrillon, Anastasia Stulova, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A high-level virtual platform for

early mpsoc software development. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 11–20. ACM, 2009.

[9] EDN Embedded Microprocessor Benchmark Consortium et al. The eembc benchmark suite. `http://www.eembc.org/coremark/index.php`. [accessed 6 Dec 2017].

[10] Clifton Craig and Adam Gerber. *Learn Android Studio: Build Android Apps Quickly and Effectively*. Apress, 2015.

[11] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing/spl trade/software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 15–24. IEEE, 2003.

[12] Yufei Ding, Mingzhou Zhou, Zhijia Zhao, Sarah Eisenstat, and Xipeng Shen. Finding the limit: examining the potential and complexity of compilation scheduling for jit-based runtime systems. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 607–622. ACM, 2014.

[13] Parallels IP Holdings GmbH. Parallels Desktop for Mac. https://www.parallels.com/uk/products/desktop/, 2018. [accessed 8-August-2018].

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

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

[16] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[17] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceed-*

ings of the Tenth International Symposium on Code Generation and Optimization, pages 104–113. ACM, 2012.

[18] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[19] Mingkai Huang, Xianhua Liu, Tingyu Zhang, and Xu Cheng. Exploration of the relationship between just-in-time compilation policy and number of cores. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 293–307. Springer, 2015.

[20] Imperas. DEV - Virtual Platform Development and Simulation. http://www.imperas.com/dev-virtual-platform-development-and-simulation, 2018. [accessed 8-August-2018].

[21] Apple Inc. Rosetta. `https://web.archive.org/web/20110107211041/http://www.apple.com/rosetta`, 2018. [accessed 8-August-2018].

[22] Mozilla Inc. Tamarin JavaScript Engine. https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Tamarin, 2018. [accessed 8-August-2018].

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

[24] Synopsys Inc. Virtual Prototyping. https://www.synopsys.com/verification/virtual-prototyping.html, 2018. [accessed 8-August-2018].

[25] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 Conference on Java Grande*, JAVA '99, pages 119–128, New York, NY, USA, 1999. ACM.

[26] Michael R Jantz and Prasad A Kulkarni. Exploring single and multilevel jit compilation policy for modern machines 1. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):22, 2013.

[27] Michael R Jantz and Prasad A Kulkarni. Performance potential of optimization phase selection during dynamic jit compilation. *ACM SIGPLAN Notices*, 48(7):131–142, 2013.

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

[29] Prasad A Kulkarni. Jit compilation policy for modern machines. In *ACM SIGPLAN Notices*, volume 46, pages 773–788. ACM, 2011.

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

[31] Jikes RVM. https://www.jikesrvm.org/, 2016. [accessed 8-August-2018].

[32] Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-accelerated cross-architecture full-system virtualization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):36, 2016.

[33] William Stallings and Goutam Kumar Paul. *Operating systems: internals and design principles*. Pearson, 2012.

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

[35] VMware. Compliance and Cyber Risk Solutions. https://www.vmware.com/uk/solutions/compliance-cyber-risk.html, 2018. [accessed 8-August-2018].

[36] VMware. Server Consolidation. `https://www.vmware.com/uk/solutions/consolidation.html`, 2018. [accessed 8-August-2018].

[37] VMware. VMware ESXi: The Purpose-Built Bare Metal Hypervisor. `https://www.vmware.com/products/esxi-and-esx.html`, 2018. [accessed 8-August-2018].

[38] Vincent M Weaver and Sally A McKee. Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 305–319. Springer, 2008.

[39] John Whaley. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87. ACM, 2000.