Finding the Limit: Examining the Potential and Complexity of Compilation Scheduling for JIT-Based Runtime Systems

Yufei Ding, Mingzhou Zhou, Zhijia Zhao, Sarah Eisenstat*, Xipeng Shen

Computer Science Department, The College of William and Mary
*Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology
{yding,mzhou,zzhao,xshen}@cs.wm.edu
*seisenst@csail.mit.edu

Abstract

This work aims to find out the full potential of compilation scheduling for JIT-based runtime systems. Compilation scheduling determines the order in which the compilation units (e.g., functions) in a program are to be compiled or recompiled. It decides when what versions of the units are ready to run, and hence affects performance. But it has been a largely overlooked direction in JIT-related research, with some fundamental questions left open: How significant compilation scheduling is for performance, how good the scheduling schemes employed by existing runtime systems are, and whether a great potential exists for improvement. This study proves the strong NP-completeness of the problem, proposes a heuristic algorithm that yields near optimal schedules, examines the potential of two current scheduling schemes empirically, and explores the relations with JIT designs. It provides the first principled understanding to the complexity and potential of compilation scheduling, shedding some insights for JIT-based runtime system improvement.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors

General Terms Performance

Keywords JIT, Compilation Scheduling, Compilation Order, NP-completeness, Heuristic Algorithm, Runtime System

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/2541940.2541945

1. Introduction

Just-In-Time (JIT) has been the major code translation paradigm in Java runtime, binary translation, and other settings. Enhancing the performance of programs compiled by JIT has received steady attentions in the last decade. The attentions have been reinforced recently as the runtime engines of some popular web scripting languages (e.g., Javascript) starts to embrace JIT as a replacement or complement of interpreters for better performance.

Enhancing the performance of JIT-based runtime systems is a multi-dimensional problem. So far, research efforts have been largely focused on the various facets of the internal design of JIT [9, 10, 13, 16], and the determination of suitable optimization levels for a function [5, 27, 32]. Few have been devoted to the compilation scheduling of JIT. It has remained preliminarily understood when a JIT should be invoked to compile which function, or in other words, how compilation order affects the performance of JIT-based runtime systems.

Formally, *a compilation order* refers to an order in which the compilation units in a program (e.g., functions or traces) are being compiled or recompiled[1]. Compilation order determines what is the earliest time the executable of a function (or trace) becomes available, which in turn affects the performance of the program. For instance, consider a call sequence "a b g g g g e g". Let $C_i(x)$ represent the compilation of function x at level i—a higher level means deeper optimizations and also larger compilation time overhead. A compilation order, $(C_1(a), C_1(b), C_1(g), C_1(e), C_2(g))$, dictates that the better optimized version of function g is not available before $C_1(e)$ finishes. If we switch $C_1(e)$ with $C_2(g)$, the better version of g can become available earlier, benefit more invocations of g, and hence possibly produce better performance. For mobile applications, better performance translates to shorter response time and often higher energy efficiency, which are key variables for user satisfaction in handheld systems [25, 30].

[1] It should not be confused with the order of the various optimization phases in a compilation, which is related with internal design rather than usage of a compiler.

Despite the natural connection between compilation orders and JIT performance, there has been little work concentrated on it. Some fundamental questions have never been systematically explored, such as how significant the connection is, how good the orders employed by existing runtime systems are, whether there is a great potential for improvement. Answers to these questions are directional. They will indicate whether enhancing compilation orders is valuable and promising, and guide research in this preliminarily understood direction.

This paper aims to answer those fundamental questions, and provide a principled understanding to the problem of compilation ordering in JIT-based runtime systems.

Specifically, we concentrate on two aspects. The first is on optimal compilation orders: For a given execution, are optimal orders feasible to compute? If so, how to compute them efficiently? If not, how to approximate them effectively? Answers to these open questions are essential for revealing the full impact of compilation orders on a runtime system.

Through a series of rigorous analysis, we provide a threefold answer to the questions (Sections 3 to 5.) First, we prove that a simple, linear algorithm can give optimal compilation orders when compilation and execution have to happen on the same core. Second, through a reduction from PARTITION and 3-SAT, we prove that in other settings, the problem becomes strongly NP-complete—that is, no polynomial algorithms exist for computing optimal compilation orders unless NP=P, even if the compilation and execution times of the functions are all bounded to a polynomial in the length of the call sequence. Prompted by the NP-completeness, we develop the Init-Append-Replace (IAR) algorithm, a polynomial-time approximation algorithm that is able to produce near optimal compilation sequences efficiently in general settings. These findings make the examination of the full potential of compilation orders possible. More fundamentally, they add the first deep understanding to the compilation ordering problem.

Based on the theoretical findings, the second part of this work (Section 6) empirically evaluates the quality of the compilation order given by existing JIT-based runtime systems, and measures the potential of better compilation orders in enhancing the system performance. Specifically, we use a modern Java Virtual Machine (JVM) named Jikes RVM [4] as the basic framework, and strive to include some of the most relevant factors into our measurement. These factors include concurrency in a JIT, machine parallelism, the cost-benefit models a JIT may use to determine the suitable compilation levels for a function, and so on. We evaluate the compilation scheduling scheme used in Jikes RVM, as well as the one used in V8, the runtime Engine behind Google Chrome. Experiments on call sequences collected on a set of Dacapo benchmarks [6] conclude that a good compilation order can lead to an average speedup of as much as a factor of two over existing runtime systems, clearly demonstrating the importance and potential of compilation ordering. Meanwhile, the results unveil some insights in the relations among compilation order, machine concurrency, and parallel JIT, and provide a discussion on enhancing compilation orders in practice.

Overall, this work makes the following major contributions.

- It provides the first systematic exploration to the impact of compilation ordering for the performance of JIT-based runtime systems in various settings.
- It reveals that the compilation scheduling schemes used by existing runtime systems are far from optimal, pointing out the importance of research efforts in enhancing compilation schedules—a direction largely overlooked by previous studies.
- It, for the first time, unveils the inherent complexity of the compilation scheduling problem, proving the NP-completeness of getting optimal compilation schedules in a general setting. It further shows that intelligent searching algorithms like A^*-search, although offering some alleviation, are not feasible in practice. The result is significant as it will help the community avoid wasting time in finding algorithms to compute the optimal orders and instead devote the efforts into the more promising direction of designing effective approximation algorithms.
- It proves the optimality of a simple algorithm for computing optimal compilation schedules for single-core settings, and contributes an efficient approximation algorithm, the IAR algorithm, for getting near optimal schedules in general settings. These findings are crucial for assessing the room left for improvement of practical runtime systems, and meanwhile, shed insights for designing better compilation scheduling modules in practice.

This work may benefit the JIT community in several ways. (1) It points out an important research direction for enhancing JIT-based runtime systems. (2) Its complexity findings will help avoid efforts in searching for algorithms to compute optimal JIT schedules. (3) It offers the first practical way to assess the JIT scheduling of runtime systems, from which, virtual machine developers can easily see the room left for improvement and allocate their efforts appropriately. (4) It provides an effective scheduling algorithm (IAR) that is efficient for possible online uses (although some other conditions are needed for its integration in an actual runtime system.)

The rest of the paper is organized as follows. We start with some background on JIT in Section 2, provide a definition of the optimal compilation scheduling problem in Section 3, then present the algorithms and complexity analysis in Sections 4 and 5, and finally report the experimental results, discuss related work, and conclude with a summary.

2. Background

This section takes Jikes RVM as an example to provide some brief background on how JIT compilation is currently being used in many runtime systems. Jikes RVM [4] is an open-source JVM originated from IBM. It uses method-level JIT compilation[2]. Its employment of JIT is typical. A function can be compiled at four levels by the JIT; the higher the level is, the deeper the optimizations are, and also the larger compilation overhead it incurs. In an execution of a program, Jikes RVM maintains a queue for compilation requests. When it encounters a Java method for the first time, it enqueues a request to compile the method at the lowest level. During the execution of the program, it observes the behaviors of the application through sampling, whereby, it determines the importance of each Java method in the application. When it realizes that a Java method is important enough to deserve some deeper optimizations, it enqueues a request to recompile the method at a higher level. The level is determined by a cost-benefit model prebuilt inside Jikes RVM, the information which draws on includes the hotness of the method and its code size.

Jikes RVM has a number of threads, including a compilation thread. The compilation thread dequeues the compilation tasks and conducts the corresponding compilation work. As a result, the order in which the compilation tasks are enqueued and the time when they are enqueued determines the compilation sequence of the program in that execution.

3. Problem Definition and Scope

This section provides a formal definition of the optimal compilation scheduling problem, and then discusses the assumptions and scope of the problem.

Definition 1. Optimal Compilation Scheduling Problem (OCSP)

Given: *A sequence of function invocations, where, each element (say m_i) represents an invocation of a function m_i. A function can appear once or multiple times in the invocation sequence. For each function, there are multiple possible compilation levels. Let $c_{i,j}$ represent the length of the time needed to compile function m_i at level j, $e_{i,j}$ represent the length of the corresponding execution time of the produced code. Assuming all $c_{i,j}$ and $e_{i,j}$ are known and independent of compilation orders, and $\forall\ i$ and $j_1 < j_2$, we have $c_{i,j_1} \leq c_{i,j_2}$, $e_{i,j_1} \geq e_{i,j_2}$. A function cannot run unless it has been compiled at least once (no matter at which level) in an execution of the invocation sequence. A function can be compiled once or multiple times in a run. The code produced by the latest compilation is used for execution.*

Goal: *To find a sequence of compilation events that minimizes the make-span of the program execution. Here, make-span is the time spanning from the start of the first compilation event to the end of the program execution.*

The definition contains several assumptions. We explain them as follows.

- *Assumption 1:* The compilation and execution time of a function is all known. In actual runtime systems, such as Jikes RVM [4], there are often some kind of cost-benefit models. They estimate the compilation and execution time from function size and hotness (to determine the suitable compilation levels for a function.) How to accurately estimate the time is not the focus of this work; we use measured time in our experiments. In addition, in the definition of OCSP, both $c_{i,j}$ and $e_{i,j}$ through a program execution are assumed to be constant for some given i and j. In reality, the compilation time of a function $c_{i,j}$ is largely stable, but the execution time $e_{i,j}$ may differ from one call of function m_i to another, thanks to the differences in calling parameters and contexts. The variations however do not affect the major conclusions our analysis produces, as Section 8 will discuss. In our experiments, we set $e_{i,j}$ with the average length of all calls to function m_i that is compiled at level j.
- *Assumption 2:* A function can be compiled at multiple levels. This assumption is consistent with most runtime systems (e.g., V8, Jikes RVM, J9, Hotspot.) A higher level usually means deeper optimizations, and hence a longer compilation time and shorter execution time of the function.
- *Assumption 3:* The sequence of function invocations is known. This assumption is necessary for revealing the full potential impact of compilation scheduling. In our experiment, the sequence is collected through profiling. How to estimate such a sequence in an actual deployment of a runtime system is a problem relevant to the construction of a practical scheduler, but orthogonal to the investigation of optimal compilation scheduling problem.

In addition, besides function compilation, function inlining may also affect the overall performance. We discuss its effects in Section 8.

4. Algorithms and Complexities for Finding the Optimal

In this section, we investigate the feasibility in finding solutions to the OCSP. The motivation for this investigation is multifold. First, an optimal solution, if it is feasible to obtain, exposes the limit of the scheduling benefit, which is essential for examining the full potential of compilation scheduling. Second, understanding the computational complexity of the problem is important for guiding the direction of efforts: If the problem is strongly NP-hard, efforts may be more worthwhile to be spent on finding effective heuristic algorithms

[2] In this paper, "function" and "method" are inter-changable terms.

than designing optimal algorithms. We describe our findings by classifying the scenarios into two categories.

4.1 Optimality in the Case of Single Core

When there is only one CPU core available, both compilation and execution have to happen on that core. In this case, determining the best compilation order is straightforward. For each function (say m_i), we find the *most cost-effective compilation level* for that function, denoted as l_i, such that $\forall k, n_i * e_{i,l_i} + c_{i,l_i} \leq n_i * e_{i,k} + c_{i,k}$, where, n_i is the number of invocations of function m_i in the execution sequence. We have the following theorem:

Theorem 1. *An arbitrary compilation sequence of the methods, as long as every method appears once in it and is compiled at its most cost-effective compilation level, yields the minimum make-span for the program.*

The correctness of the theorem is easy to see. As there is only one core, the machine is always running, doing either the compilation or execution jobs. The make-span is hence the sum of the compilation and execution times. The most cost-effective level ensures that the total time taken by each method is minimum. Hence the conclusion. The optimality is subject to the assumptions listed in the previous section, which will be discussed in Section 8.

The implication of this result is that on single-core machines, research efforts should be put on the prediction of the most cost-effective compilation levels. The reason is that if those levels are predicted accurately, using on-demand compilation—that is, compiling a function when it gets invoked for the first time, as what most existing runtime systems are using—can already give us an optimal compilation order. This insight has its practical values: Despite that most machines are equipped with multicores, lots of times, an application is using only one core when multiple applications are running. It is especially common on smartphone-like portable devides that contain only several cores.

4.2 Complexity in the Case of Multiple Cores

In the case of multiple cores, the compilation and execution can happen largely in parallel. The determination of a best compilation order becomes complicated.

An Example To illustrate the complexity, consider the example showing in Figure 1. The call sequence is simple, containing just four calls to three functions. For simplicity, we assume that compilation and execution happen on two different cores, and only one level of compilation is worthwhile for functions f_0 and f_2 and two levels for function f_1. Just based on the execution times, it may be tempting to think that the best way to compile the functions is to pick the highest compilation levels for all the functions—as they provide the shortest running times for the functions—and then schedule the compilations in the order of the first-time appearance of those functions. The middle bottom box in Figure 1 shows

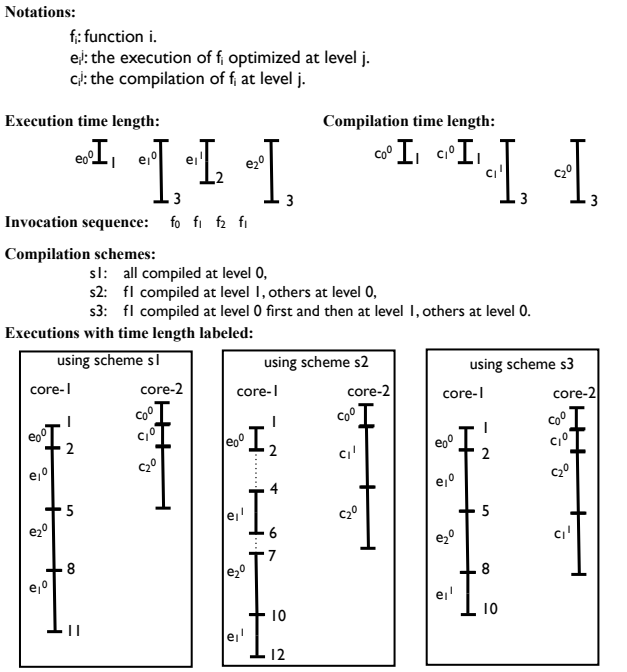

Figure 1. An example illustrating the complexity of compilation scheduling. Assuming execution happens on core-1 and compilation happens on core-2.

this schedule. It turns out to result in the longest make-span among all the three schedules shown in Figure 1. The reason is that the highest level of compilation of f_1 causes delays to the start of the second and third function calls—we call such delays *bubbles* from now on. The best schedule of the three is the third schedule, in which, function f_1 is compiled twice: The first time is at the lowest level to avoid causing delays, and the second time is at the highest level to speedup its second invocation. From this result, one may think that a schedule that compiles all functions at the lowest level first and then recompiles them would work well.

But a simple extension of the example can immediately prove the hypothesis incorrect. As Figure 2 shows, we add another call to function f_2. It also shows that the compilation of f_2 at level 1 takes 5 time units and execution at that level takes 1 time unit. The bottom three boxes in Figure 2 show how the three schedules in Figure 1 would perform in this case when we consider the possibility of appending a recompilation of f_2 at level 1 to the schedule. This appending turns the previously best schedule (schedule 3) to the worst. (The third box in Figure 2 does not have that recompilation appended because appending it is apparently not beneficial.) The first schedule with such an appending becomes the best of the three. This schedule has function f_2 but not others recompiled, despite that f_2 takes the longest time to recompile.

These two examples are simple, involving only three functions and five invocations. They reflect the high complexity OCSP can have on real program executions. Which

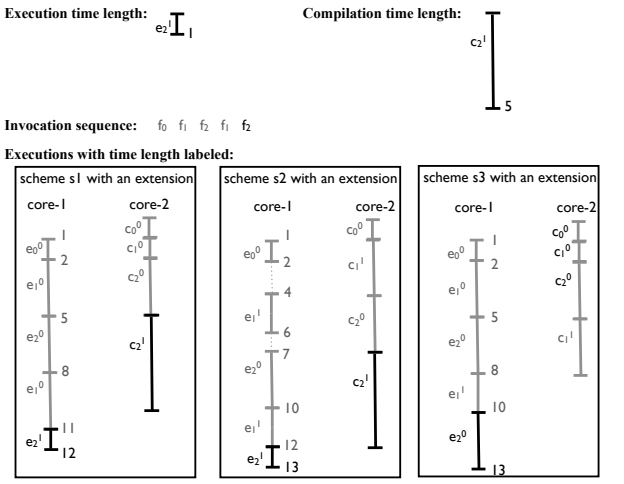

Figure 2. Continuing the example in Figure 1 with a call to f_2 appended to the invocation sequence.

functions need to be recompiled and when each (re)compilation should happen depend on how frequently the functions are each invoked, when they are invoked, how long the execution and compilation of each take, and so on.

Prompted by the observed complexities, we conduct a theoretical analysis of the computational complexity of the OCSP. Findings are shown next.

Strong NP-Completeness Our analysis produces the following conclusion:

Theorem 2. *When compilation and execution can happen in parallel, the OCSP problem is NP-complete.*

We prove the NP-completeness through a reduction from the classic NP-complete problem, PARTITION, to a simple case of OCSP, where there are only two machines, both compilation and execution are single threaded, and there are only two optimization levels.

Proof. Suppose we are given a set of non-negative integers $S = s_1, s_2, \cdots, s_n$. Let $t = (1/2) \times \sum_{i=1}^{n} s_i$. The goal of the PARTITION problem is to find a subset X whose sum is t. (By the definition of t, this ensures that $S - X$ also sums to t.)

For each s_i, we construct one function with $c_{i1} = 1$, $c_{i2} = s_i + 1$, $e_{i1} = s_i + 1$, $e_{i2} = 1$. Pick an arbitrary order to execute these functions. We then construct two additional functions, which have the same compilation time and execution time for both levels. One of these functions, which we require to execute first, has compilation time 1 and execution time $t+n$. The other function, which we require to execute last, has compilation time $t + n$ and execution time 1.

Suppose that we have a set X satisfying the PARTITION constraints. Then we can construct the corresponding compilation sequence by compiling all of the functions in the set X at level 1, and compiling all other functions at level 2. The long execution time of the initial function guarantees that we have lots of time to compile all of the functions in the middle before we have to execute them. So if we let one machine (named E) conduct only executions and the other (named C) compilations, both machines can work constantly (with the exception of the first time unit for Machine E and the last time unit for Machine C).

Because the numbers in X sum to t, the total execution time for functions in X is $|X| + t$, and the total compilation time for functions in X is $|X|$. Similarly, the total execution time for functions not in X is $n - |X|$, while the total compilation time for functions not in X is $n - |X| + t$. Hence, the total compilation time for all but the first and last functions is $t + n$. The execution time is also $t + n$. When combined with the time for the first and last functions, as well as the slack time, the total make-span is $2(1 + t + n)$.

It is easy to see that $2(1 + t + n)$ must be the minimum make-span. Machine E cannot work in the first timestep (before anything is compiled) while Machine C can't work in the last timestep (while the last function is executing). So with make-span less than $2 + 2t + 2n$, the total processing time that we get out of Machines C and E is less than $2 \times (2 + 2t + 2n - 1) = 2(2t + 2n + 1)$. However, the total amount of work to be done (execution and compilation combined) is at least $2(2n+2t+1)$ regardless of which level the functions are compiled at, so we cannot have make-span less than $2 + 2t + 2n$.

We can also show the converse: that a schedule with that make-span must imply the existence of a partition. The total run time of all functions (execution plus compilation) is fixed at $(2n+2t)+2(n+t)+2 = 2(2n+2t+1)$. To finish that work within time $2(1 + t + n)$, Machine C has to work constantly (aside from the last time-step, which is unavoidable), and Machine E also has to work constantly (aside from the first time-step, which is unavoidable). This means that the total compilation time must sum to exactly $t+n$, and so the set of functions compiled at level 2 must correspond to a set that sums to exactly t. □

We further prove that the 3-SAT problem can be also reduced to the OCSP problem in polynomial time and hence prove that the OCSP is not only NP-complete, but also strongly NP-complete. It means that even if the compilation and execution times of the functions are all bounded to a polynomial in the length of the call sequence (which could be true in practice), the problem remains NP-complete. The implication to practice is that the OCSP is unlikely to admit efficient solutions in practice. We put the proof into a technical report [12].

5. Challenging the NP-Completeness

The strong NP-completeness proved in the previous section has some significant implications. It enhances the fundamental understanding to the compilation scheduling problem of JIT, suggesting that polynomial-time solutions are unlikely

to exist for OCSP as soon as the number of cores goes beyond one. This result entails difficult challenges for studying the full potential of compilation scheduling for JIT as the optimal schedules are probably infeasible to obtain for a real program execution.

This section presents several options we have pursued to overcome the NP-completeness to reveal the potential of compilation scheduling. The first option explores heuristic algorithms for approximating the optimal; the algorithms include a single-level scheme and a several-round algorithm. The objective of this option is to obtain some schedules that are both valid and close to the optimal in performance so that they can provide a tight upper bound for the minimum make-span. The second option computes the lower bound of the make-span. Together, these two options lead to a range covering the minimum make-span. The rationale is that if that range is small, it would be sufficient to substitute the optimal in the examination of the potential of compilation scheduling. At the end of this section, we describe the third option, which tries heuristic search to directly find the optimal.

5.1 Approximating the Limit

The goal of this option is to obtain some legal schedule with a performance close to the optimal.

Single-Level Approximation Following Occam's razor, before designing sophisticated algorithms, we start with a simple way to do the approximation. It limits the compilations of all methods to only one level. In this case, as there are no recompilations, it is easy to prove that the best schedule is just to order the compilations of all the functions based on the order of their first-time appearance in the call sequence. We call this simple approach single-level approximation.

IAR Algorithm The single-level approximations are simple but ignores recompilation, an operation that can be beneficial as Figure 1 illustrates. To better approximate the optimal schedules, we design an algorithm that consists of an initialization step and several rounds of refinement with each round enhancing the schedule produced by earlier rounds. We call this algorithm the *IAR algorithm* in short for it centers around *initialization*, *appending*, and *replacement* operations.

Figure 3 outlines the algorithm. For simplicity of explanation, the description of the algorithm assumes that there is one execution thread and one compilation thread only and they run on two cores respectively. Additionally, it assumes that the JIT has only two levels of compilation. We will discuss the case with more than two levels at the end of this section and the case with more threads in Section 6.2.3.

The first step of the IAR algorithm creates an initial compilation schedule, which consists of the lowest-level compilations of all functions in the program, arranged in the order of the first-time appearances of the functions in the call sequence. Such a schedule tries to prevent bubbles (i.e. waiting time) from occurring in the execution that the long compilation time at the higher compilation level could cause.

It then classifies the functions into three categories: $A(ppend)$, $R(place)$, $O(ther)$. The categorization examines the benefits of doing a higher-level compilation for each function. The examination employs the collected compilation and execution time at all compilation levels. If a higher-level compilation appears to be not beneficial for a function (Formula 1 in Figure 3), it is put into O. Otherwise, it should be compiled at the higher level either at its first-time compilation (i.e., to replace its low-level compilation in the initial compilation sequence with a high-level compilation of it) or after the initial compilation stage is done (i.e., to append its high-level compilation to the end of the current compilation sequence.) These two cases correspond to adding the function to R and A respectively. The Formula 2 in Figure 3 shows the criterion for classifying these two cases. The rationale is that when the high-level compilation overhead is much larger than the attainable benefits at the beginning part of the run, putting the high-level compilation into the initial schedule may cause bubbles into the execution, and it is hence better to be appended to the end of the initial schedule. In our experiment, we tried different values of "K" in Formula 2 and found that as long as it falls into a range between 3 and 10, the results are quite similar (in our reported results, K=5.)

Based on the classification, the second step of the IAR algorithm simply replaces the low-level compilations of all functions in R with high-level compilations in the initial compilation sequence. To append functions in A to the sequence, however, we need to decide the order of appending. The heuristics we use is the compilation overhead. The rationale is that putting a costly recompilation early would cause large delays to the attainment of high-quality code for other functions.

The third step is a fine adjustment to the compilation sequence. It tries to find the slacks in the initial part of the schedule and exploit the slacks by replacing a low-level compilation with a high-level compilation as long as the replacement does not add bubbles into the execution. A slack is defined as the time between the finish of the first-time compilation of a function and its first invocation in the execution. The selective replacement accelerates the execution of the function without causing a negative effect. At a replacement, the following high level compilation of that function is deleted if there is one.

The final step is another fine adjustment. It tries to append more high-level compilations as long as there is time between the finish of all compilations and the finish of all executions. This filling does not cause any negative effect to the execution but could accelerate the execution of some functions that are not yet compiled at the high level. We tried various ways to prioritize these additional appending opera-

```
// Eseq: the call sequence;
// F: the set of functions;
// K: a constant (5 in our experiment);
// f.cl, f.ch: the compilation time of function f at low and high levels;
// f.el, f.eh: the execution time of function f at low and high levels;
// f.n: the number of calls to function f in Eseq;

step1 (init):
  Cseq = {};
  // get the sequence of first-time calls of all the functions
  Eseq1 = getSeq1stCalls (Eseq);
  Cseq = Eseq1;

  // categorize functions into sets A(ppend), R(eplace), O(thers).
  O={}; A={}; R={};
  foreach f in F
    if (f.ch+f.n*f.eh > f.cl+f.n*f.el)   // Formula 1
       O.append (f);
    else {
       f.n1 = numbers of calls to f during the compilation of current Cseq;
       if (f.ch - f.cl > K * f.n1 * (f.el - f.eh))   // Formula 2
          A.append (f);
       else
          R.append (f);
    }

step2 (append & replace):
  // ascending sort on the compilation time at the high level
  A = ascendingSortOnCh (A);
  Cseq.append (A); // append ch of members of A
  Cseq = replace (Cseq, R); // replace cl with ch for members of R

step3 (fill slack through replacement):
  Tc1 = computeTheEnd1stCompile (Cseq);
  Te1 = computeTheStart1stCall (Eseq, Cseq);
  slacks = findSlacks (Tc1, Te1);
  // replace some cl with ch to fill slacks
  Cseq = fillSlacksWithReplacement (Cseq, slacks);

step4 (append more to fill ending gap):
  L  = functions that are compiled at low level only in Cseq;
  M =  numbers of calls to functions in L in Eseq after Cseq finishes;
  L  = desendingSort (L, M);
  Tgap = time between end of compilations and end of executions;
  Append as many ch of functions in L as possible until filling up Tgap.
```

Figure 3. IAR algorithm.

tions by considering factors ranging from optimization overhead, to benefits, and positions of the calls in the sequence. But they do not outperform the simple heuristics Figure 3 shows. As Section 6 will show, the reason is that there is only a marginal room left for improvement by this fine adjustment.

This IAR algorithm offers a more sophisticated way to approximate the optimal schedules than the single-level approximation. It capitalizes on both the order of first-time appearances of the functions and their invocation frequencies, considers both the compilation overhead and overall benefits, and exploits each individual level of compilation and their combinations. Section 6 will show that the design enables a much better approximation than other methods do.

The time and space complexity of the IAR algorithm is $O(N+MlogM)$, where N is the length of the call sequence, and M is the number of unique functions in the sequence. Typically $N >> M$ and the complexity is linear to N in practice.

The description of the IAR algorithm has assumed that the JIT has only two candidate compilation levels. That assumption is only for explanation simplicity. The algorithm itself works as long as a function has two levels of optimizations to choose; the two levels can differ for different functions. For a JIT with more than two levels, our design is to take the most responsive level and the most cost-effective level of a function as the two candidate levels for that function when applying the IAR algorithm. The most responsive level is the one taking the least time to compile, usually corresponding to the lowest level. The most cost-effective level is the level at which the compilation time plus the total execution time of all invocations to the function is minimum. The cost-effective level can be determined by the default **cost-benefit model** in most virtual machines. The cost-benefit model takes the hotness of a function, its size, and other information as inputs and estimates the most cost effective compilation level for that function [4]. An alternative way is offline profiling that measures the performance at every compilation level.

Our experiments in Section 6 will confirm that IAR on the two levels is enough to get a schedule close to the optimal.

5.2 Bounding the Limit

The algorithms in the previous subsection produce schedules that are attainable, the results of which may be regarded as the upper bounds of the minimum make-span of the program execution. Getting a lower bound of the minimum make-span is relatively simple. It is clear that the make-span cannot be smaller than the sum of the shortest execution time of each function call in the call sequence. Therefore, the lower bound is computed as $\sum_{i=1}^{N} {e_{f_i}}^K$, where K is the highest compilation level, $e_{f_i}^K$ is the execution time of the ith function call in the invocation sequence at level K. The upper bound and lower bound reveal the limits of the value range of the minimum make-span. If the range is tight, it may serve as an alternative to minimum make-span for examining the potential of compilation scheduling.

5.3 Searching for the Limit

In addition to approximation, another way to solve the OCSP is to use some search algorithms to cleverly search through the design space for an optimal schedule. These search algorithms may substantially outperform naive search by avoiding some subspaces that are impossible to contain the optimal solution. However, they do not lower the computational complexity of the problem and may still consume too much time, space or both for large problems.

For completeness of the exploration, we examine the feasibility by applying A*-search, an algorithm that is optimally efficient for any given heuristic function—that is, for a given heuristic function, no other search-tree–based optimal algo-

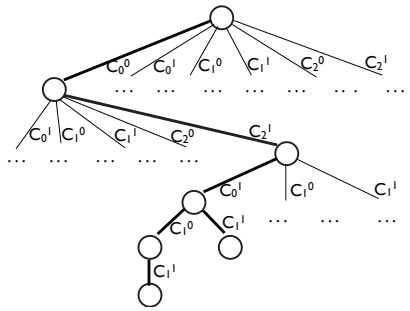

Figure 4. An example for modeling the compilation scheduling of three functions as a tree search problem. Notation: C_i^j for the compilation of method i at the jth level. The two highlighted paths denote two complete compilation sequences.

rithm is guaranteed to expand fewer nodes than A* search does [31].

A*-Search For a tree search, where the goal is to find a path from the root to an arbitrary leaf with the total cost minimized, A* search defines a function $f(v)$ to estimate the lowest cost of all the paths passing through the node v. A* search maintains a priority list, initially containing only the root node. Each time, A* search removes the top element—that is, the node with the highest priority—from the priority list, and expands that node. After the expansion, it computes the $f(v)$ values of all the newly generated nodes, and puts them into the priority list. The priority is proportional to $1/f(v)$. This expansion process continues until the top of the list is a leaf node, indicating that no other nodes in the list need to be expanded any more as their lowest cost exceeds the cost of the path that is already discovered.

The definition of function $f(v)$ is the key for the solution's optimality and the algorithm's efficiency. There are two properties related to $f(v)$:

- A* search is optimal if $f(v)$ never overestimates the cost.
- The closer $f(v)$ is from the real lowest cost, the more effective A*-search is in pruning the search space.

Problem Modeling A typical way to apply A*-search is to model the target problem as a tree-search problem. Figure 4 illustrates how we model the compilation scheduling problem. The tree contains all possible schedules for compiling three functions at two compilation levels. Every path from the root to a leaf represents one feasible compilation sequence. Each node on the path represents the compilation of a function at a level. The construction of the tree ensures that a lower-level compilation of a function does not appear after a higher-level compilation of the function in any path, as it is impossible to be an optimal schedule under the assumptions on execution times in Section 3. For instance, in Figure 4, C_2^0 never appears in a path below C_2^1.

To explain the search heuristic function $f(v)$, we use the following notations: $s(v)$ for the compilation schedule represented by the path segment from the root to node v, $t(v)$ for the time period from the start to the end of compilations in $s(v)$. In our exploration, the most effective definition of $f(v)$ we find is $f(v) = b(v) + e(v)$, where, $b(v)$ is the sum of the lengths of all the execution bubbles in $t(v)$ when $s(v)$ is used, and $e(v)$ is the sum of the extra execution time in $t(v)$ because the functions are not compiled at the highest level in $s(v)$.

6. Experiments

We design our experiments to answer the following questions:

- **Limit Estimation or Computation** Can the algorithms in Sections 5.1 and 5.2 provide a tight range for the minimum make-span? How effective and scalable is the A*-search algorithm in searching for optimal compilation schedules?
- **Potential of Existing Systems** Is there a substantial room left for improving the compilation scheduling used in modern runtime systems?
- **Influence of Other Factors** How do other factors, such as the concurrency and cost-benefit model in a JIT, influence answers to the questions above?

6.1 Methodology

Experimental Framework To validate whether the proposed algorithms can provide a good estimation of the optimal compilation schedules, one just needs to apply the algorithms to a set of data that include a call sequence of a number functions and the compilation and execution times of those functions at various optimization levels. For collecting some realistic data, we develop a data collection framework for collecting the required data from real Java programs.

The collection framework is based on Jikes RVM [4] v3.1.2. As an open-source JVM that has gone through many years of enhancement by the community, Jikes RVM now shows competitive performance with top commercial systems [1]. It contains a basic compiler and an optimizing compiler, both single-threaded. The optimizing compiler supports three compilation levels. So counting them together means that a function can be compiled at any of the four levels. Jikes RVM has a replay mode, in which, it takes some *advice files* that indicate how each Java method should be compiled, and runs the program with the JIT following those instructions. By manipulating the advice files, we stimulate the compilation of each function (i.e. Java method) at each of the four levels in 20 repetitive runs, during which we collect the execution times of every function at each level, the average of which is taken as the execution time of that function at that level. By adding a timer into the JIT, we also collect

the compilation time of each function at every level [3]. The runs for measuring compilation time are separate from the runs for measuring execution times to avoid possible perturbations by each other.

Also through the Jikes RVM, we collect the call sequence for every run of the programs. For a multithreaded application, we still get a single sequence; the calls by different threads are put into the sequence in order of the profiler's output. This order roughly corresponds to the invocation timing order by those threads. An alternative is to separate calls by different threads into different call sequences. But given that the threads typically share the same native code base generated by the same JIT, we find it more reasonable to put them together when considering compilation scheduling.

These collected data together form the data set we use for assessing the quality of the various schedules. Examination of the data shows that for most functions, the condition in Definition 1 holds—that is, $\forall\ i$ and $j_1 < j_2$, we have $c_{i,j_1} \leq c_{i,j_2}, e_{i,j_1} \geq e_{i,j_2}$.

The harness in the Dacapo suite, when running a benchmark, starts the JVM and runs the benchmark for a number of times (without restarting the JVM.) So in the late runs, the program enters a steady-state without many compilations happening. In the reported performance results, we concentrate on the first-time run (or called warmup run), for two reasons. First, compilation scheduling is mostly relevant to warmup runs as that is when most compilations happen. Second, despite that steady-state performance is critical for server applications, for most utility programs and smartphone applications, warmup run performance is often more crucial. These programs usually do not have a long execution time, but their response time is essential for user satisfaction.

In addition to data collection, the experimental framework includes a component that, for a given compilation schedule, computes the make-span of a call sequence based on the compilation and execution times of the involved functions, along with the number of cores used for compilation and execution.

Comparing the make-spans reported by such a framework provides a clear, direct evaluation of the scheduling algorithms. An alternative method is to deploy the compilation schedules in real executions of JVMs and measure the program finish time. That method, unfortunately, cannot provide a direct measurement of the quality of the scheduling algorithms, as we will discuss in Section 8.

Benchmarks and Machines Our experiment uses the Dacapo [6] 2006 benchmark suite (default input)[4]. Two programs are not used: *chart* fails to run on Jikes RVM (3.1.2), and *xalan* cannot run in the replay mode of Jikes RVM. Table 1 reports the basic information of the benchmarks. Two of the programs, *hsqldb* and *lusearch*, are multithreading. The lengths of the call sequences range from 467K calls to 43M calls, and the numbers of unique functions in sequences range from 543 to 2128. All experiments happen on a machine equipped with X5570 with totally 16 cores. It has Linux 3.1.10 installed.

Table 1. Benchmarks

Program	parallelism	#functions	call seq length	default time(s)
antlr	seq	1187	2403584	1.6
bloat	seq	1581	9423445	5.0
eclipse	seq	2194	467372	28.4
fop	seq	1927	1323119	1.5
hsqldb	parallel	1006	8022794	2.9
jython	seq	2128	23655473	6.7
luindex	seq	641	20582610	6.1
lusearch	parallel	543	43573214	3.2
pmd	seq	1876	12543579	3.5

6.2 Results

We report the experimental results in three parts. The first part examines the full potential of compilation scheduling compared to the default scheduling scheme in Jikes RVM. The second and third parts investigate how an oracle cost-benefit model and concurrent JIT influence the observed potential. Then we examine the potential when the scheduling scheme in V8 is applied to those Java programs, the feasibility of finding optimal schedules through A*-search, and the potential of the IAR algorithm for online usage.

6.2.1 Comparison with the Jikes RVM Scheme

In this part, we compare various schedules to reveal the potential in enhancing the scheduling algorithm in the default Jikes RVM. We first describe the default scheduling algorithm briefly. At the first invocation of a function, it always compiles the function at the lowest level. During the execution of a Java program, after every sampling period, the runtime checks whether any function could benefit from a recompilation. The cost of a compilation at level j is computed as $e_j * k + c_j$, where e_j and c_j are the execution and compilation times of the function at level j, and k is the number of times the sampler has seen the function on the call stack since the start of the program. Let l be the level of the last compilation of the function, and m be the level with the minimal cost among all levels higher than l. If $e_m * k + c_m < e_l * k$, the JIT recompiles the function at level m.

We implement the scheduling algorithm in our experimental framework. Figure 5 shows the normalized make-span (i.e., end-to-end running time) of the benchmarks under the default scheduling scheme and other compilation schedules. The IAR algorithm considers only two compilation levels for a function: the lowest level, and the most cost-effective level that is determined by the default cost-benefit

[3] To minimize measurement errors, we insert a loop into the JIT so that we can time 100 compilations of a function and get the average time.

[4] We didn't use the latest version of Dacapo as it cannot fully work on Jikes RVM [20].

model in the Jikes RVM. These two levels are respectively used for the two single-level compilation scheme. The baseline used for the normalization is the lower bound computed with the algorithm in Section 5.2. So the lower a bar is in the figure, the better the corresponding schedule is.

The results lead to several observations. First, there is a large gap between the make-span by the default schedule and the lower bound. More than half of the programs show a gap greater than 50%. The average gap is over 70%. However, just from that gap, it is hard to tell whether there is a large room for improvement as the minimum make-span may exist anywhere between the lower bound and the default. So besides the lower bound, we still need a decent estimation of the minimum make-span.

Following descriptions in Section 5.1, we experiment with the single-level approximation, with the base level ("base-level") and the suitable highest compilation level ("optimizing-level") used respectively. Both of them show even longer make-spans than the default on most programs. The base-level scheme saves compilation time but results in a longer execution time for less efficient code it generates. The optimizing-level scheme, on the other hand, saves execution time, but results in a longer compilation time and execution bubbles. A more sophisticated approximation is necessary.

The IAR algorithm meets the needs. Using the schedules from IAR, none of the programs shows a gap wider than 17% from the lower bound. On average, the gap is only 8.5% wide. It indicates that to examine the full potential, it is sufficient to compare to either the lower bound or the make-span from the IAR algorithm, as the minimum make-span is close to both of them. Compared to the IAR results, the default compilation scheduling scheme in Jikes RVM is largely behind in performance, with a gap from the lower bound as wide as eight times of the IAR gap, suggesting that significant potential exists in enhancing the performance of Jikes RVM by optimizing its compilation scheduling. It is worth mentioning that the time savings by the IAR schedule is not only due to the reduction of compilation-incurred bubbles on the critical path, but also to the reduction of execution time as some highly optimized code become available earlier than in the default run.

As a side note, the results also confirm that using only two levels of optimizations for each function is enough for the IAR algorithm to work sufficiently well, despite more than two levels are supported by the JIT.

6.2.2 With an Oracle Cost-Benefit Model

In the study so far, we rely on the default cost-benefit model in Jikes RVM to determine the suitable optimizing levels for each method. This subsection examines whether the large impact of the scheduling diminishes if the cost-benefit model is improved. A positive answer would mean that enhancing the cost-benefit model could be sufficient.

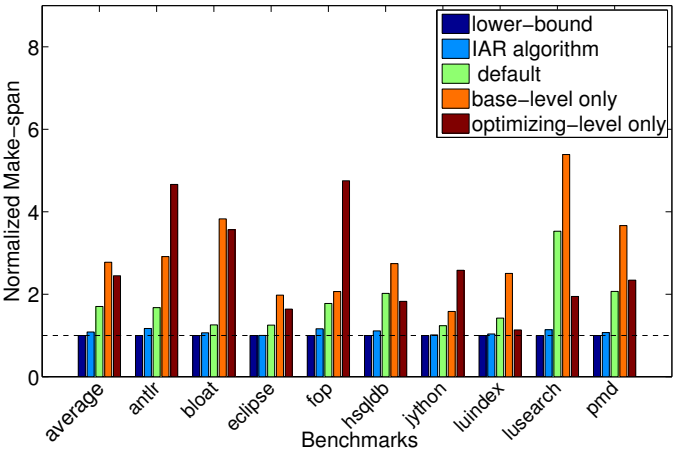

Figure 5. Normalized running time based on the JIT in the default Jikes RVM.

To answer the question, we repeat the experiments while replacing the cost-benefit model in Jikes RVM with an oracle model. The default cost-benefit model tries to estimate the compilation time and execution time of a function at various optimization levels, and pick the level that if it is used, the sum of the estimated compilation time of that function and the estimated execution time of all invocations to that function is minimized. In our oracle cost-benefit model, we simply replace the estimated time with the actual time. The model is not necessarily the optimal model, but it is the best the default cost-benefit model can do.

Figure 6 reports the result. Compared to the result in Figure 5, there are several differences. First, the gap between the IAR schedules and the lower-bound increases, but by no more than 6% on average. The range of the make-span defined by the IAR and lower-bound remains tight. Second, the average gap between the default and the lower-bound doubles in size. Using the optimizing level only results in a similar gap. The reason for the gap expansion is two-fold. The lower-bound becomes lower as the pure execution of the program gets faster with a better optimizing level determined, and meanwhile, the longer time in the higher-level compilation causes more bubbles into the execution in the default and optimizing-level only cases. Many of these bubbles however can be avoided by a better compilation schedule as the IAR results demonstrate.

Overall, the results suggest that the importance of compilation scheduling actually increases as the cost-benefit model gets enhanced, further reinforcing the importance of the compilation scheduling dimension in improving modern runtime systems.

6.2.3 With a Concurrent JIT

The JIT in the default Jikes RVM is single-threaded. There are some JVMs (e.g., Hotspot) that support concurrent compilation of multiple functions on a multicore system. The concurrency offers an alternative way to reduce compilation

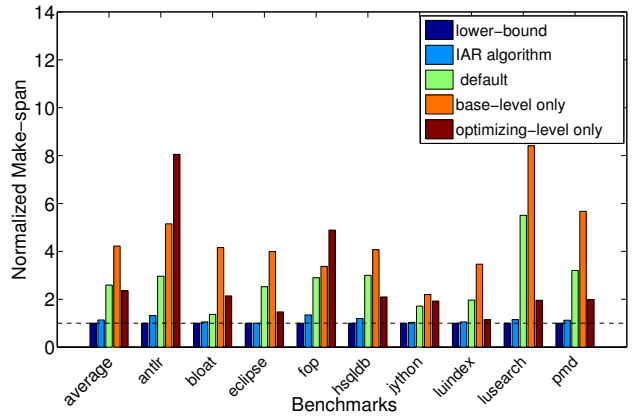

Figure 6. Normalized running time when an oracle cost-benefit model is used in the JIT.

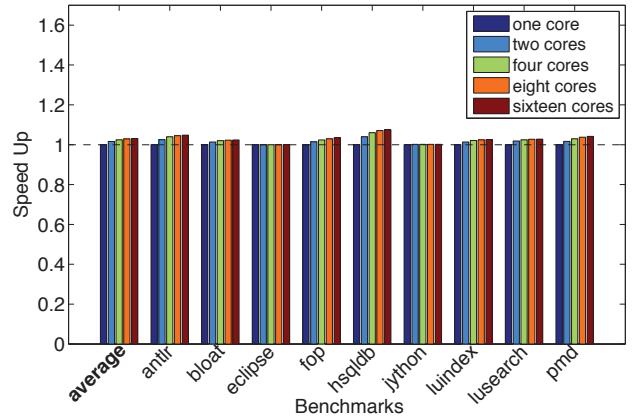

Figure 7. Speedups brought by concurrency JIT when the IAR schedule is used. The legend shows the number of cores used for compilation.

overhead and the delays to executions. We apply the idea to the IAR schedules by allowing multiple compilation tasks to be processed in parallel (the tasks are ordered in a queue based on the IAR schedules.) The graph in Figure 7 reports the speedups (in terms of make-span) brought by the concurrency, when the default cost-benefit model is used. As the number of cores increases, the speedup increases but slightly and always remains quite minor. The largest speedup is 13% when 16 cores and the oracle cost-benefit model are used. The average speedups are no greater than 7% in all the cases. Similar results are seen when the oracle cost-benefit model is used.

The reason for the speedups being minor is that most compilation time is already hided when the IAR schedules are used. In another word, when a good compilation schedule is used, there is a small room left for performance improvement by concurrent JIT. Given that concurrent JIT may compete for computing resources with worker threads, the results indicate that compilation scheduling is a more cost-effective way for enhancing modern runtime systems than concurrent JIT[5].

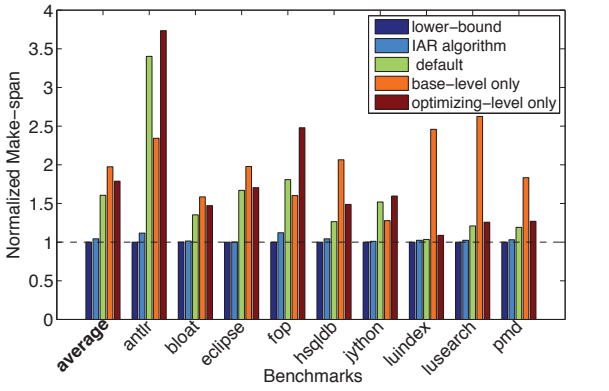

Figure 8. Normalized make-span, compared with the results by the compilation scheduling scheme in Google V8.

6.2.4 Comparison with the V8 Scheme

In addition to Jikes RVM, we also compare with the compilation scheduling algorithm used in V8, the primary Javascript runtime engine from Google, and also one of the most efficient Javascript runtime engines on the market. We apply its scheduling scheme to the Java programs we use. The purpose is to examine the inherent quality of the scheduling scheme, rather than the actual performance of the engine on Javascript applications.

In V8, there are only two optimization levels: *low* and *high*. By default, it compiles a function at the *low* level at the first encounter of it and then recompiles it at the *high* level at its second invocation.

We apply that algorithm to the call sequences we obtained from the Jikes RVM on the Java benchmarks, with the lowest two levels in Jikes RVM JIT as the *low* and *high* in this experiment. Figure 8 shows the make-spans. The gap between the IAR schedule and the lower bound remains small, only 4% on average. The schedule from the V8 scheme results in a smaller gap from the lower bound than the Jikes RVM results, 61% on average. The gaps between the two single-level compilation schedules and the lower bound also become smaller than in the Jikes RVM case. The main reason is not that the V8 scheme is better than the Jikes RVM one, but that as the *highest* optimizing level is lower than that in the Jikes RVM, the lower bound in this V8 experiment is higher than in the Jikes RVM experiment. But the high-level observation is consistent: The IAR algorithm still produces near optimal results while the default scheduling has a large room for improvement.

[5] Note that this point does not rule out the value of concurrent JIT. It could be quite valuable when an inferior compilation scheduling is used.

Table 2. IAR Algorithm Time

Program	IAR time (sec)	percentage over whole program time (%)
antlr	0.006	0.37
bloat	0.023	0.47
eclipse	0.001	0.004
fop	0.003	0.21
hsqldb	0.020	0.67
jython	0.059	0.87
luindex	0.051	0.84
lusearch	0.108	3.38
pmd	0.031	0.89

6.2.5 Effectiveness of Heuristic Search

For the nature of strongly NP-completeness, it is unlikely to find a search algorithm that can produce the optimal compilation schedule efficiently. Our experiments on the A*-search algorithm offers a confirmation. For tiny problems, the algorithm can find the optimal schedules by searching through a fraction of all possible paths. For instance, for a call sequence with six unique functions called for 50 times in total and two levels of compilations, the A*-search algorithm finds an optimal compilation schedule by searching through 96 out of 4 billion (12!) paths. However, due to the heuristics A*-search uses for space pruning, it stores all incompletely examined paths in memory. As the search tree grows exponentially with the problem size, the space requirement increases rapidly. In our experiments, when the number of unique methods is larger than 6, the A*-search program (written in Java) aborts for out of memory (2GB is used for the heap size.)

6.2.6 IAR Overhead

Table 2 reports the time overhead of the IAR algorithm. For most programs, the overhead is less than 1% of the whole program execution time. The low overhead makes it affordable for online usage. But to achieve a good scheduling result, many other factors have to be considered as Section 8 will discuss.

6.3 Summary of Results

Overall, the experiments show that the IAR algorithm is effective for approximating the optimal compilation schedules. It produces schedules efficiently with a near optimal performance on most programs; its linear time and space complexity grants it high efficiency and scalability. On the other hand, the experiments show that neither simple heuristic algorithms (e.g. the single-level approximation) nor heuristic search (e.g., A*-search) can provide scalable, satisfying results.

By comparing the scheduling used in Jikes RVM with the IAR algorithm and lower bounds, we see that a better compilation schedule can bring speedup by as much as 1.6X on average. And the potential increases to a factor of 2.3 when the perfect cost-benefit model is used. These results suggest that enhancing the compilation scheduling is an important direction to explore, and improvement in the cost-benefit model in a JIT compiler only highlights its importance further. Meanwhile, it shows that concurrent JIT adds minor improvement when a good compilation schedule is used.

7. Potential Impact

The primary goal of this work is to enhance understanding of how compilation ordering affects the performance of JIT-based runtime systems. This study produces some major findings towards that goal: It reveals the inherent computational complexity of the scheduling problem, offers methods for effectively approximating the optimal schedules, and shows the large room left for improvement in existing systems.

The potential impact of this work is multifold. It points out that compilation scheduling is an important direction for enhancing JIT-based runtime systems. Its complexity findings will help avoid efforts in searching for algorithms to compute optimal JIT schedules. It offers the first practical way to assess the JIT scheduling of runtime systems, from which, virtual machine developers can easily see the room left for improvement and allocate their efforts appropriately. In addition, the insights from this work may offer some immediate guidance for enhancing current virtual machines. For instance, it shows that the first-time compilation of a method should generally get a higher priority than recompilations of other methods.

8. Complexities from Actual Runtime Systems

The findings in this work shed some insights into enhancing compilation schedules in virtual machines, but producing a ready-to-use virtual machine with much enhanced compilation schedules is a goal beyond the scope of this paper. This section discusses some complexities for achieving that goal in an actual runtime system.

The first barrier is in getting or estimating the call sequence of a production run. It could be tackled through some recently developed techniques, such as cross-run learning and prediction. Some studies (e.g., [5, 33]) have shown the feasibility to learn various types of behavior patterns of a program (e.g., loop trip-counts, function call relations) by accumulating the data transparently sampled across *production runs* of the program. There have been a lot of studies in small-scope program behavior prediction [8, 18, 28, 35, 36]. And recent years have seen some development towards large-scope program behavior sequence prediction [34]. Extending these techniques into call sequence prediction could help remove this barrier.

The second barrier is in obtaining the accurate compilation and execution times of a function at various optimization levels. In some virtual machines, there are already some

kind of models for estimating those times. In Jikes RVM, for instance, the times are estimated through some simple linear functions of the size of the function. The parameters of those functions are obtained through some offline training during the installation of Jikes RVM on a machine. However, such static estimations are often quite rough. For example, different invocations of a function, even if they use the same executable, may have different running lengths due to differences in their calling parameters and contexts. Moreover, function inlining that happens in a run may substantially change the length and execution time of the caller function. Furthermore, the time when a compilation happens may also affect the execution time of a function. A later compilation in a run may produce better code as the online profiler collects more information about the program execution. All these factors suggest that existing static estimations of the execution times of a function are unlikely to provide an accurate estimation. Development of the aforementioned cross-run learning methods may help enhance the estimation quality. On the other hand, finding out the relations between estimation errors and the quality of an advanced scheduling algorithm will also help: If the scheduling can tolerate a good degree of estimation errors, building up an estimation model to meet the requirement may be still feasible.

It is worth mentioning that due to the variations in execution times as mentioned in the previous paragraph, desirable results often cannot get produced when one directly applies the results of the IAR algorithm—obtained on offline collected average times—to a real program execution. Some ways to extend the IAR algorithm to accommodate the variations in execution times will help its practical usage.

Unlike Jikes RVM, some runtime environments have interpreters as part of their code translators for quick code generation. In fact, the lowest level of compilation in Jikes RVM is designed in that spirit for the same purpose. It does not build an intermediate representation nor perform register allocation. In another word, that lowest-level compilation can be regarded as a method-level interpreter. Similar schemes are taken in other runtimes (e.g., V8). If we treat interpretation as the lowest level compilation in the optimal compilation schedule problem, the analysis and algorithms discussed in this paper can still be applied. Some extra care may be needed for the interpreters that operate at the level of a single statement.

We note that although these complexities complicate the enhancement of compilation scheduling in real systems, they do not alter the major conclusions from this study. For instance, recall that we have used the average value of $e_{i,j}$, for each given i and j, when computing the lower bounds of program execution times. Because the total time a function takes in a run equals the sum of the time it takes in all its invocations, using the average does not skew the computed lower bound of the execution time of the whole program run. For the same reason, the variation does not affect the optimality of the scheduling algorithm in the single-core case in Section 4.1.

9. Related Work

The ParcPlace Smalltalk VM [11] and the Self-93 VM [17] pioneer adaptive optimization techniques employed in current virtual machines. Recent years have seen increasing attentions payed to enhancing the performance of JIT-based runtime systems. These efforts can be largely classified into two categories, based on the main target of their optimizations.

The first category focuses on the internal design of the JIT in runtime systems. Examples include the work on using machine learning to help JIT better decide the optimizations suitable for a Java method [10, 24]. Several recent studies on Javascript JIT have explored static and dynamic inferences of dynamic types to enable better code specialization [2, 16, 19, 23]. In addition, a number of recent studies have tried to refine trace detection, management, and optimizations in trace JIT for modern scripting languages [9, 13]. As Section 6 shows, enhancing the design of JIT does not replace but reinforce the need for compilation scheduling by exposing an even higher potential benefit of scheduling.

The second category concentrates on JIT compilation policies, especially on the determination of the appropriate optimization level to use for compiling a function (or trace). Most adaptive virtual machines use selective compilation, which tries to find hot functions and optimize them more than other functions [3, 27]. They typically contain a cost-benefit model that decides the suitable optimization level for a function based on its hotness, size, and other information. Predicting the hotness of a function is critical for these systems. Most of them have relied on online profiling with the assumption that a hot method in the past will remain hot in the future [4, 21]. Some studies try to predict hotness based on loop trip-counts [26] or phases [14]. All these studies use information within a run to make the predictions. The scheduling of the compilation events in the those studies is mostly on-demand, just as the default scheme in Jikes RVM: When a function is loaded or is selected by the online profiling to be recompiled, the compilation event is inserted into a compilation queue and is processed in the order of their arrival. Their general guideline is to use a low level to compile a function when it is loaded to get a good startup time and later recompile it at a higher level when it appears to be hot. This current study shows that the intuition-based scheme leads to schedules that are far from being optimal in practice.

Several studies try to exploit cross-run observations [5, 32]. They make it possible to predict the suitable optimizing levels for a set of methods as program starts running. Such predictions allow more flexibility in scheduling (re)compilations beyond the on-demand scheduling. How-

ever, these studies did not exploit the flexibility and used similar scheduling schemes as the default system uses.

All these studies on compilation policies have aimed at answering the questions on what functions are hot and what levels to compile them, rather than when to (re)compile them or in what order to (re)compile them.

There have been explorations on using concurrent JIT to enhance the performance of virtual machines [7, 15, 22]. They have not systematically explored the influence of the order of compilation events. Concurrent JIT may give some performance boost to the default JVM, but introduce contentions for computing resources with the application threads. As Section 6 shows, when a good compilation order is used, the gain from concurrent JIT over sequential JIT becomes marginal. How to combine these two complementary approaches in practice is a problem for the future.

Another body of related work is ahead-of-time (AOT) compilation for Java and other managed code. The idea of AOT is to compile a program into native machine code before executions of the program. Another technique that shares a similar spirit of reusing native code cross runs is persistent code caching [29], which stores the native code produced in a run into some persistent storage for reuses by later runs. Both techniques have shown some benefits in performance enhancement. However, in practice, they have been used in just some limited settings. Most programs in managed languages are still JIT compiled at runtime, plausibly for the portability and seamless handling of dynamic types by JIT, and the maturity of modern runtime environments.

10. Conclusion

This paper describes the first principled study on unveiling the full potential of compilation scheduling for JIT-based runtime systems. Through some rigorous analysis, it reveals the inherent computational complexity of finding optimal compilation orders, proving the strong NP-completeness of the problem. It empirically demonstrates the difficulty in searching for an optimal schedule with the A*-search algorithm. Prompted by the challenges, it proposes a heuristic algorithm named the *IAR algorithm*, which yields near optimal schedules for a set of benchmarks. The results enable a systematic examination of the full potential of compilation scheduling in modern runtime systems. This paper reports experiments on two scheduling schemes, respectively corresponding to those used in Jikes RVM and V8, concluding that significant potential (1.6X on average) exists in adjusting compilation schedules. It further shows that the potential is even larger as the JIT internal design gets better. It also reveals that concurrent JIT gives only marginal gain in performance when the compilation order is near optimal, suggesting that enhancing compilation orders may be a more cost-effective option than concurrent JIT in enhancing the performance of JIT-based runtime systems. Overall, this study concludes that improving compilation scheduling in modern runtime systems is an important direction (although being largely overlooked so far) for future runtime system designs. The theoretical results, the IAR algorithms, and multi-fold insights produced in this work lay the foundation for this direction of research.

Acknowledgment

We thank the anonymous reviewers of the earlier versions of this paper for their helpful feedback. This material is based upon work supported by DOE Early Career Award, and the National Science Foundation under Grant No. 0811791, 1320796, and CAREER Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or DOE.

References

[1] Jikes rvm. `http://jikesrvm.org`.

[2] C. Anderson, S. Drossopoulou, and P. Giannini. Towards type inference for javascript. In *ECOOP*, 2005.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2), 2005.

[4] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 111–129, 2002.

[5] M. Arnold, A. Welc, and V.T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 297–311, 2005.

[6] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2006.

[7] I. Bohm, T. Koch, S. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2011.

[8] M. Burtscher and B. G. Zorn. Hybrid load-value predictors. *IEEE Transactions on Computers*, 2002.

[9] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages. In *OOPSLA*, 2012.

[10] J. Cavazos and M. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2006.

[11] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL*, 1984.

[12] Y. Ding, M. Zhou, Z. Zhao, S. Eisenstat, and X. Shen. The potential and complexity of compilation scheduling in jit-

based runtime systems. Technical Report WM-CS-2014-02, College of William and Mary, 2014.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, et al. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN Conference On Programming Language Design and Implementation*, 2009.

[14] D. Gu and C. Verbrugge. Phase-based adaptive recompilation in a JVM. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 24–34, 2008.

[15] J. Ha, M. Haghighat, S. Cong, and K. S. McKinley. A concurrent trace-based just-in-time compiler for single-threaded javascript. In *Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, 2009.

[16] B. Hacket and S. Guo. Fast and precise hybrid type inference for javascript. In *PLDI*, 2012.

[17] U. Holzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Language Systems*, 18(4), 1996.

[18] S. Hu, R. Bhargava, and L. K. John. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelism*, 2003.

[19] S. H. Jensen, A. Mller, and P. Thiemann. Type analysis for javascript. In *SAS*, 2009.

[20] Jikes rvm project and status. http://http://jikesrvm.org/Project+Status.

[21] T. Kotzmann, C. Wimmer, H. Mossenbock, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot client compiler for java 6. *Transactions on Architecture and Code Optimization*, 5(1), 2008.

[22] P.A. Kulkarni. Jit compilation policy for modern machines. In *Proceedings of the annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2011.

[23] F. Logozzo and H. Venter. Rata: Rapid atomic type analysis by abstract interpretation. In *CC*, 2010.

[24] R. Nabinger-Sanchez, J. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using machines to learn method-specific compilation strategies. In *CGO*, 2011.

[25] F. Nah. Study on tolerable waiting time: How long are web users willing to wait? *Behavior and Information Technology*, 23(3):153–163, 2004.

[26] M. A. Namjoshi and P. A. Kulkarni. Novel online profiling for virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, pages 133–144, 2010.

[27] M. Paleczny, C. Vick, and C. Click. The java hotspot server compiler. In *Symposium on JavaTM Virtual Machine Research and Technology*, 2001.

[28] C. J. Pickett, C. Verbrugge, and A. Kielstra. Adaptive software return value prediction. Technical Report 1, McGill University, 2009.

[29] V. Reddi, D. Connors, R. Cohn, and M. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *CGO*, 2007.

[30] A. Rushinek and S. Rushinek. What makes users happy? *Communications of the ACM*, 29(7):594–598, 1986.

[31] S. Russell and P. Norvig. *Artificial Intelligence*. Prentice Hall, 2002.

[32] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[33] K. Tian, E. Zhang, and X. Shen. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[34] B. Wu, Z. Zhao, X. Shen, Y. Jiang, Y. Gao, and R. Silvera. Exploiting inter-sequence correlations for program behavior prediction. In *Proceedings of the annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2012.

[35] T. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.

[36] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 31(6), 2009.