Reducing the Overhead of Dynamic Compilation

Chandra Krintz[†] David Grove[‡] Derek Lieber[‡] Vivek Sarkar[‡] Brad Calder[†]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]IBM T. J. Watson Research Center

UCSD Technical Report
CSE2000-0648, March 2000

Abstract

*The execution model for mobile, dynamically-linked, object–oriented programs has evolved from fast interpretation to a mix of interpreted and dynamically compiled execution. The primary motivation for dynamic compilation is that compiled code executes significantly faster than interpreted code. However, dynamic compilation, which is performed while the application is running, introduces execution delay. In this paper we present two dynamic compilation techniques that enable high performance execution while reducing the effect of this compilation overhead. These techniques can be classified as: 1) decreasing the amount of compilation performed (*Lazy Compilation*), and 2) overlapping compilation with execution (*Background Compilation*).*

We first evaluate the effectiveness of lazy compilation. In lazy compilation, individual methods are compiled on demand upon their first invocation. This is in contrast to Eager Compilation*, in which all methods in a class are compiled when a new class is loaded. Our experimental results (obtained by executing the SpecJVM Java programs on the Jalapeño JVM) show that, compared to eager compilation, lazy compilation results in* 57% *fewer methods being compiled and reductions in total time (compilation plus execution time) of* 14% *to* 26%*.*

Next, we present profile-driven, background compilation, a technique that augments lazy compilation by using idle cycles in multiprocessor systems to overlap compilation with application execution. Profile information is used to prioritize methods as candidates for background compilation. Our results show that background compilation can deliver significant reductions in total time (26% *to* 79%*), compared to eager compilation.*

1 Introduction

The execution model for mobile, dynamically-linked, object–oriented programs has evolved from fast interpretation to a mix of interpreted and dynamically compiled execution [21, 15, 26]. The primary motivation for dynamic compilation is significantly faster execution time of compiled code over interpreted code. Many implementations of object–oriented languages (such as Java [12], Smalltalk [11], and Self [27]) use dynamic compilation to improve interpreted execution time. Dynamic compilation also offers the potential for further performance improvements over static compilation since runtime information can be exploited for

optimization and specialization. Several dynamic, optimizing compiler systems have been built in industry and academia [2, 15, 24, 16, 17, 13, 6, 20].

Dynamic compilation is performed while the application is running and, therefore, introduces compilation overhead in the form of intermittent execution delay. The primary challenge in using dynamic compilation is to enable high performance execution with minimal compilation overhead. Unfortunately, a common practice thus far in the evaluation of dynamic compilers for Java, has been to omit measurements of compilation overhead and to report only execution time [26, 9, 24, 8]. Hence, it is difficult for users to evaluate the tradeoff between compilation overhead and execution speedup.

In this paper we present two dynamic compilation techniques that enable high performance execution while reducing the effect of compilation overhead. These techniques can be classified as (1) decreasing the amount of compilation performed (*Lazy Compilation*); and (2) overlapping compilation with useful work (*Background Compilation*).

We first examine the benefits of lazy compilation, in which individual methods are compiled on demand upon first invocation. This is in contrast to what we term *Eager Compilation*, in which all methods in a class are compiled when a new class file is loaded. Most Just-In-Time (JIT) compilers for Java[1] perform lazy compilation [24, 16, 17, 26]. We first evaluate the effectiveness of lazy compilation by performing a quantitative comparison with eager compilation. To the best of our knowledge, this is the first study to provide such an evaluation. We detail our experiences with lazy compilation and describe its implementation in the Jalapeño JVM, as well as its implications for compiler optimizations (some of which may have been overlooked in prior work). Our experimental results obtained for the SpecJVM Java programs show that, compared to eager compilation, lazy compilation results in 57% fewer methods being compiled and reductions in *total time* (compilation plus execution time) of 14% to 26%.

We then present profile-driven background compilation, a technique that augments lazy compilation by using idle cycles in multiprocessor systems to overlap compilation with application execution. Profile information is used to prioritize methods as candidates for background compilation. Our background compilation technique is designed for use in symmetric multiprocessor (SMP) systems in which one or more idle processors might be available to perform dynamic compilation concurrently with application threads. We believe that such systems will become more prevalent in the future, especially with the availability of systems built using single-chip SMPs. Our results using Jalapeño, show that background compilation can deliver significant reductions in total time (26% to 79%), compared to eager compilation.

The infrastructure used to perform our measurements of compilation overhead is Jalapeño, a new JVM (Java Virtual Machine) built at the IBM T. J. Watson Research Center. Jalapeño [2] is a multiple–compiler, compile–only JVM (no interpretor is used). Therefore, it is important to consider compilation overhead in the overall performance of the applications executed. Prior to the work reported in this paper, the default compilation mode in Jalapeño was eager compilation. After the results reported in this paper were obtained, the default compilation mode for Jalapeño was changed to lazy compilation.

The two main contributions of this paper can be summarized as follows:

1. An evaluation of the effectiveness of lazy compilation in reducing the overhead of dynamic compilation, compared to eager compilation, including a study of its impact on execution time.

2. Profile-driven background compilation, a new technique for reducing dynamic compilation overhead by overlapping compilation with application execution. Background compilation delivers the exe-

[1] The implementation results described in this paper are for Java, but the techniques are relevant to any dynamic compilation environment.

cution speed of optimized code while incurring very little overhead (similar to that of a fast, non-optimizing compiler) by exploiting idle cycles.

In the following section we provide an overview of the Jalapeño JVM, the infrastructure we use for this research. We also describe the general methodology we used to collect the experimental results in the paper. In Section 3, we present the implementation of lazy compilation in Jalapeño. The methodology specific to lazy compilation and the performance improvements we achieve using it are detailed in Sections **??** and 3.3. We then present an extension to the lazy compilation approach, background compilation, in Section 4.1, and describe the methodology specific to it as well as the performance results achieved using it in Sections **??** and 4.2. We then discuss related (Section 5) and future work (Section 6) and conclude in Section 7.

2 Methodology

The infrastructure in which we evaluate our compilation approaches is Jalapeño, a Java virtual machine (JVM) being developed at the IBM T. J. Watson Research Center [1, 2]. We first describe this infrastructure then detail our general experimental methodology.

2.1 Jalapeño

Jalapeño is written in Java and designed to address the special requirements of SMP servers: performance and scalability. Extensive runtime services such as concurrent garbage collectors, thread management, dynamic compilation, synchronization, and exception handling are provided by Jalapeño.

Jalapeño uses a compile–only execution strategy, i.e., there is no interpretation of Java programs. Currently there are two fully–functional compilers in Jalapeño, a fast *baseline* compiler and the *optimizing* compiler. The baseline compiler provides a near-direct translation of Java class files thereby compiling very quickly and producing code with execution speeds similar to that of interpreted code. The second compiler is highly optimizing and builds upon extensive compiler technology to perform various levels of optimization [8]. The compilation time using the optimizing compiler is much slower than the baseline (50 times on average in the programs studied), but produces code that executes 3–4 times faster. To warrant its use, compilation overhead must be recovered by the overall performance of the programs. All results were generated using a December, 1999 build of the Jalapeño infrastructure. We report results for both the baseline and optimizing compiler. The optimization levels we use in the latter include many simple transformations, inlining, scalar replacement, static single assignment optimizations, global value numbering, and null check elimination.

As shown in Figure 1, a Jalapeño compiler can be invoked in four ways. First, when an unresolved reference made by the executing code causes a new class to be loaded, the class loader invokes a compiler to compile the class initializer (if one exists). In eager compilation, the class loader also invokes a compiler to compile all of the class method's as part of the class loading process. In lazy compilation, instead of invoking a compiler, the class loader simply initializes all methods of the newly loaded class to a *lazy compilation stub*. When a stub is executed, a compiler is invoked to compile the method (denoted by the arrow from the executing code to the compiler). The implementation of lazy compilation in Jalapeño is a contribution of this paper, and is discussed in Section 3. A third compilation path involves a background compilation thread (denoted OCT in Figure 1) that uses off-line profile data to schedule pre-emptive optimizing compilations of performance-critical methods. The class loader notifies the OCT of class loading events, but the actions taken by the OCT are otherwise decoupled from the application's execution (they

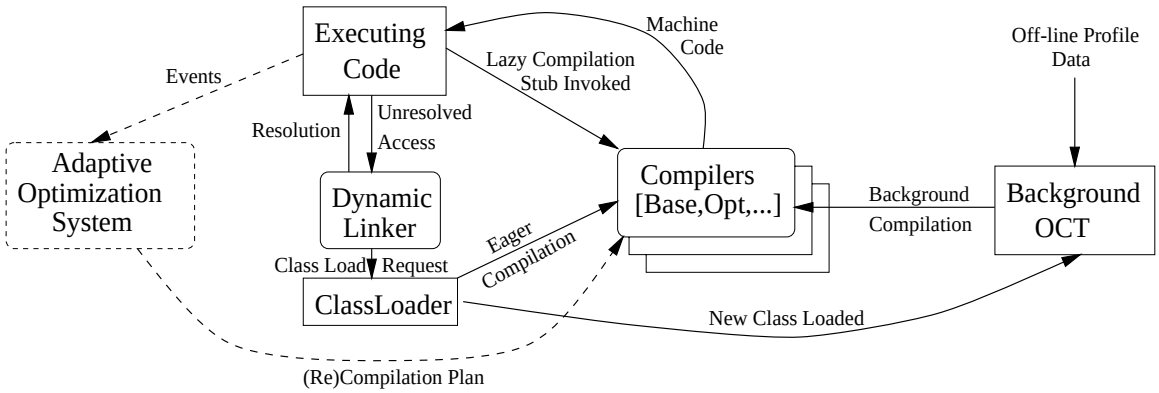

Figure 1: Compilation Scenarios in the Jalapeño JVM

occur in the background). Background compilation is a contribution of this paper, and is discussed in Section 4.1. Another potential compilation path is shown by the dashed lines in Figure 1). This represents an adaptive optimization system, which uses on-line profile data to selectively recompile performance critical methods using an optimizing compiler. The adaptive optimization system is currently being developed by Arnold et. al. [4]. The main contribution of that work is their development of low-impact on-line profiling techniques to guide which methods to optimize, and a controller that dynamically selects routines for optimization. In comparison, our approach in this paper uses off-line profiles to choose which methods to optimize for background compilation. In addition, we evaluate and compare background compilation to our lazy compilation implementation.

Jalapeño is invoked using a boot image [1]. A subset of the runtime and compiler classes are fully optimized prior to Jalapeño startup (and placed into the boot image); these class files are not dynamically loaded during execution. Including a class in the boot image, requires that the class file does not change between boot-image creation and Jalapeño startup. This is a reasonable assumption for Jalapeño core classes. This idea can be extended to also include application classes in the boot image to avoid runtime compilation of such classes, provided the class does not change prior to runtime. This topic is further described in [22]. Infrequently used, specialized, and supportive (library), Jalapeño class files are excluded from the boot image to reduce the size of the JVM memory footprint and to take advantage of dynamic class file loading. When a Jalapeño compiler encounters an unresolved reference, i.e., an access to a field or method from an unloaded class file, it emits code that when executed invokes Jalapeño runtime services to dynamically load the class file. This process consists of loading, resolution, compilation, and initialization of the class. If, during execution, Jalapeño requires additional Jalapeño system or compiler classes not found in the boot image, then they are dynamically loaded: there is no differentiation in this context between Jalapeño classes and application classes once execution begins. To ensure that our results are repeatable in other infrastructures, we isolate the impact of our approaches to just the benchmark applications by placing all of the Jalapeño class files required for execution into the boot image.

2.2 Experimental Methodology

We present results gathered by repeatedly executing applications on a dedicated, 4–processor (each 166Mhz) PowerPC-based machine running AIX v4.3. The applications we examine are the SpecJVM programs [23]. We report numbers using inputs of size 10 and size 100. According to Spec, input 100 is full execution of

Benchmark	Total Size in KB	Total Count Classes	Used Class Count small	Used Class Count large	Optimized Time (Secs) (Used Classes) small ET	Optimized Time (Secs) (Used Classes) small CT	Optimized Time (Secs) (Used Classes) large ET	Optimized Time (Secs) (Used Classes) large CT	Baseline–Compiled Time (Secs) (Used Classes) small ET	Baseline–Compiled Time (Secs) (Used Classes) small CT	Baseline–Compiled Time (Secs) (Used Classes) large ET	Baseline–Compiled Time (Secs) (Used Classes) large CT
Compress	17	12	11	12	7.4	8.2	84.0	8.1	47.0	0.1	525.1	0.1
DB	9	3	3	3	1.9	8.2	102.7	8.0	2.9	0.3	162.6	0.3
Jack	129	57	46	46	9.9	16.0	84.3	16.0	10.9	0.4	93.2	0.4
Javac	548	176	132	139	2.0	38.6	66.3	38.5	3.0	0.6	103.5	0.6
Jess	387	151	133	134	2.5	27.2	45.2	27.6	6.4	0.3	109.8	0.3
Mpeg	117	55	42	37	7.3	15.9	71.3	15.9	47.6	0.4	452.1	0.4
Average	201	78	61	62	5.2	19.0	75.6	19.0	19.6	0.4	241.1	0.4

Table 1: **Benchmark characteristics.** The first column of data provides benchmark sizes in kilobytes. The second column is the total number of classes in each application. The third and fourth columns show the dynamic class file counts for each input (classes used during execution). The middle four columns contain the execution (ET) and compile (CT) times when the Jalapeño optimizing compiler is used. The last four columns are the execution and compile times when the Jalapeño baseline compiler is used. Times for both input sizes are given. The small input is the SpecJVM 10% input and the large is the 100% input.

an example application and input 10 is execution that is 10% of the full execution. Spec also includes a size 1 (1% of full execution time) which we exclude here since size 10 and 100 better represent execution times of existing Java programs. Throughout this study we refer to size 10 as the small input and size 100 as the large.

Table 1, shows various characteristics of the benchmarks used in this study. Total size and static class count are given as well as dynamic counts (classes used) of class files for each input (small and large). In addition, compilation time (CT) and execution time (ET), in seconds, using the Jalapeño optimizing and the fast baseline compilers are shown for each input. The compilation time includes the time to compile only the class files that were used. Despite repeated execution, some noise occurs in the collected results. For example, the DB data shows that the compile time for the small input is 0.2 seconds slower than that for the large, even though both inputs compile the same classes. The variance is due to system side–effects, e.g., files system service interruptions.

3 Lazy Compilation

Dynamic class loading in Java loads class files as they are required by the execution (on demand). One approach to dynamic compilation is *Eager Compilation*: compile the entire class file as soon as it is loaded. Eager compilation enables analysis and optimization across methods within the same class. Interprocedural optimization can exploit spatial locality within the Java class file to improve performance. In addition, eager compilation reduces the overhead caused by switching between execution and compilation and it simplifies verification. The time required by class file loading, however, increases with eager compilation since the entire class files must be compiled before execution continues. This delay is experienced the first time each class is referenced. In some cases, it may take seconds to compile a class if high optimization levels are used, affecting a user's perception of the application performance. In addition, many methods may be compiled

and optimized but never invoked, leading to unnecessary compilation time and code bloat.

We first examine **Lazy Compilation**. Lazy (method–level) compilation, is commonly used in Dynamic Java compilers [19, 26, 15, 17]. We compare this compilation alternative with that of eager compilation and describe the implementation of lazy compilation in Jalapeño, which prior to this work had only used eager compilation.

3.1 Implementation

As part of loading a class file in Jalapeño, entries for each method declared by the class are created in the class' virtual function table and/or a static method table. These entries are the code addresses that should be jumped to when one of the class's methods is invoked. In eager compilation, these addresses are simply the first instruction of the machine code produced by compiling each method. To implement lazy compilation, we instead initialize all virtual function table and static method table entries for the class to refer to a single, globally shared stub. When invoked, the stub will identify the method the caller is actually trying to invoke, initiate compilation of the target method (if necessary[2]), update the table through which the stub was invoked to refer to the real compiled method, and finally, resume execution by invoking the target method. Our implementation of lazy compilation is somewhat similar to the backpatching done by the Jalapeño baseline compiler to implement dynamic linking [3] and shares some of the same low-level implementation mechanisms. After the stub method execution completes, all future invocations of the same class and method pair will jump directly to the actual, compiled method.

3.2 Results

To gather our results using this lazy approach, we time the compilation using internal Jalapeño performance timers. Whenever a compiler is invoked the timer is started; the timer is stopped once compilation completes. To measure the execution time of the program, we use the time reported by a wrapper program (SpecApplication.class) distributed with the Spec JVM98 programs [23]. Programs are executed repeatedly in succession, and timings of the execution are made separately.

To analyze the effectiveness of lazy compilation we first compare the total number of methods compiled with and without lazy compilation. Figure 2 depicts the percent reduction in the number of methods compiled using the large input. The numbers are very similar for the small input since the total number of methods used is similar in both inputs. Above each bar is the number of methods compiled lazily (left of slash) and eagerly (right of slash). On average, lazy compilation compiles 57% fewer methods than eager compilation.

To understand the impact of lazy compilation in terms of reduction in compilation overhead, we measured compilation time in Jalapeño with and without lazy compilation. Figure 3 shows the percent reduction in compilation time due to lazy compilation in relationship to eager compilation for both the optimizing (left graph) and baseline (right graph) compiler for the large input. The data shows that lazy compilation substantially reduces compilation time for either compiler. On average, for the optimizing compiler, 29% of the compilation overhead is eliminated. Using the baseline compiler, on average 50%, is eliminated.

Table 2 provides the raw execution and compilation times with and without lazy compilation using the optimizing compiler for both inputs. The data in this table includes compilation times used in Figure 3 as well as execution times. Data for the baseline compiler is not shown because compilation overhead is a very

[2] Because we lazily update virtual function tables on a per-class basis, it is possible that the target method has already been compiled but that some virtual function tables have not yet been updated to remove the stub method.

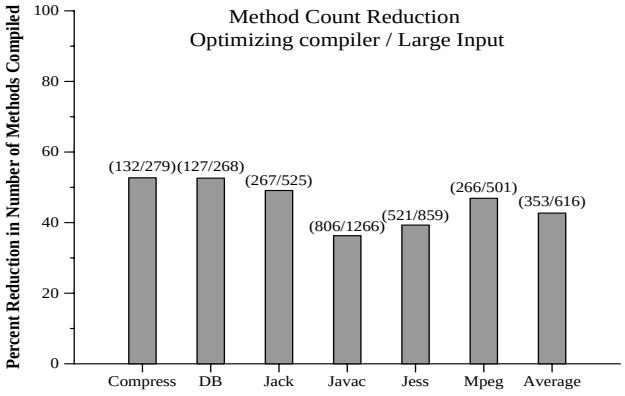

Figure 2: **Percent reduction in the number of methods required for eager compilation using lazy compilation.** We only include data for the large input since the number of used methods is similar across inputs for the Spec JVM98 benchmarks. In addition these numbers are typical regardless of which compiler (optimizing or baseline) is used.

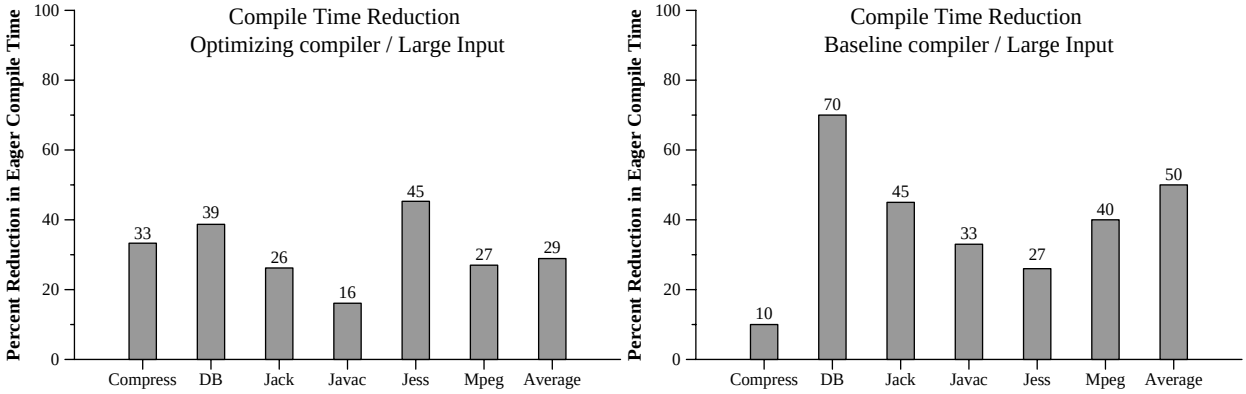

Figure 3: **Reduction in compilation time due to lazy compilation.** The left graph shows the reduction in compilation time over eager compilation for the optimizing compiler and the right graph shows the reduction for the baseline compiler. Since the results are similar for the small and large inputs, we only report data for the large input here.

	Small (in seconds)					Large (in seconds)				
	Eager		Lazy		Ideal	Eager		Lazy		Ideal
Benchmark	ET	CT	ET	CT	ET	ET	CT	ET	CT	ET
Compress	7.4	8.2	5.3	5.4	5.3	84.0	8.1	58.3	5.4	58.3
DB	1.9	8.2	1.9	5.0	1.7	102.7	8.0	98.8	4.9	98.8
Jack	9.9	16.0	9.4	11.6	9.1	84.3	16.0	80.1	11.8	77.6
Javac	2.0	38.6	2.0	31.2	1.9	66.3	38.5	68.1	32.3	62.6
Jess	2.5	27.2	1.8	14.7	1.8	45.2	27.6	38.4	15.1	37.9
Mpeg	7.3	15.9	6.7	11.7	5.4	71.3	15.9	61.7	11.6	51.3
Average	5.2	19.0	4.5	13.3	4.2	75.6	19.0	67.6	13.5	64.4

Table 2: **Raw data: Execution (ET) and compile (CT) times in seconds with and without lazy compilation using the optimizing compiler.** The sixth and eleventh columns contains the benchmark execution time when the application is batch compiled off-line. Batch compilation (Ideal) eliminates dynamic linking code from the compiled application and enables more effective inlining. Columns 2 through 6 are execution and compile times for the small input and columns 7 through 11 are for the large input. For each input, times for both the eager and lazy approaches are given.

small percentage of total execution time and thus, the 50% reduction in compilation time only results in a 1% reduction in total (execution plus compilation) time. Columns 2 through 6 are for the small input and 7 through 11 are for the large. The sixth and eleventh columns, labeled "Ideal" contain the execution time alone for batch–compiled applications. *Batch Compilation* is off–line compilation of applications in their entirety. We include this number as a reference to a lower–bound on the execution time of programs (given the current implementation of the Jalapeño optimizing compiler). Batch compilation is not restricted by the semantics of dynamic class file loading; information about the entire program can be exploited at compile time. In particular all methods are available for inlining and all offsets are known at compile time.

Columns 2 and 3, and 7 and 8, are the respective execution and compile times for eager compilation. Columns 4 and 5, and 9 and 10, show the same for the lazy approach. In addition to reducing compilation overhead, the data shows that lazy compilation also significantly reduces execution time when compared to eager compilation. This reduction in execution time was caused by the direct and indirect costs of dynamic linking. In the following section, we provide background on dynamic linking and explain the unexpected improvement in optimized execution time enabled by lazy compilation.

3.3 The Impact of Dynamic Linking

Generating the compiled code sequences for certain Java bytecodes, e.g. `invokevirtual` or `putfield`, requires that certain key constants, such as the offset of a method in the virtual function table or the offset of a field in an object, be available at compile time. However, due to dynamic class loading, these constants may be unknown at compile time: this occurs when the method being compiled refers to a method or field of a class that has not yet been loaded. When this happens, the compiler is forced to emit code that when executed, performs any necessary class loading (thus making the needed offsets available) and then performs the desired method invocation or field access. Furthermore, if a call site is dynamically linked because the callee method belongs to an unloaded class, optimizations such as inlining cannot be performed. In some cases, this indirect cost of missed optimization opportunities can be quite substantial.

Dynamic linking can also directly impact program performance. A well-known approach for dynamic linking [7, 10] is to introduce a level of indirection by using lookup tables to maintain offset information. This table-based approach is used by the Jalapeño optimizing compiler. When it compiles a dynamically linked site, the optimizing compiler emits a code sequence that, when executed, loads the missing offset from a table maintained by the Jalapeño class loader.[3] The loaded offset is checked for validity; if it is valid it can be used to index into the virtual function table or object to complete the desired operation. If the offset is invalid, then a runtime system routine is invoked to perform the required class loading (updating the offset table in the process) and execution resumes at the beginning of the dynamically linked site by re-loading the offset value from the table. The original compiled code is never modified. This scheme is very simple and, perhaps more importantly, avoids the need for self-modifying code that entails complex and expensive synchronization sequences on SMP's with relaxed memory models (such as the PowerPC machine used in our experiments). The tradeoff of simplicity is the cost of validity checking: subsequent executions of dynamically linked sites incur a four-instruction overhead.[4]

If dynamically linked sites are expected to be very frequently executed, then this per-execution overhead may be unacceptable. Therefore, an alternative approach based on backpatching (self-modifying code) can be used. In this scheme, the compiler emits a code sequence that when executed invokes a runtime system routine that performs any necessary class loading, overwrites the dynamically linked sites with the machine code the compiler would have originally emitted if the offsets had been available, and resumes execution with the first instruction of the backpatched (overwritten) code. Using backpatching there is an extremely high cost the first time each dynamically linked site is executed, but the second and all subsequent executions of the site incur no overhead.

The Jalapeño optimizing compiler used in this paper uses the dynamic linking approach instead of backpatching. This design decision was driven by the need to support type-accurate garbage collection (GC); Jalapeño relies on GC-safe points. At all GC-safe points, Jalapeño's compilers must produce mapping information detailing which registers and stackframe offsets contain pointers. By definition, all program points at which an allocation may occur (either directly or indirectly) must be GC safe-points, since the allocation may trigger a GC. Because allocation will occur during class loading, all dynamically linked sites must be GC-safe points. It proved to be challenging for the optimizing compiler to provide GC mapping information if backpatching was used, so dynamic linking was used instead.

When using lazy compilation, we delay compilation and thus increase the probability that an accessed class will be resolved at the time the referring method is compiled. Table 3 shows the number of times dynamically linked sites are executed with eager and lazy compilation. On average, code compiled lazily executes through dynamically linked sites 92% fewer times than eager compilation for the small input (99% fewer times for the large input). Although the reduction in direct dynamic linking overhead can be quite substantial (e.g. roughly 25 million executed instructions on compress large), the missed inlining opportunities are even more important. For example, virtually all (greater than 99%) executions of dynamically linked sites in the eager version of compress large are calls to very small methods that are inlined in the lazy version. Thus, the bulk of the 25 second reduction in large compress execution time (see Table 2) is due to the direct and indirect benefits of inlining, and not only to the elimination of the direct dynamic linking overhead. Similar inlining benefits also occur in `mpegaudio`.

The effect of lazy compilation on total time is summarized in Figure 4. The graph shows the relative effect by lazy compilation both on execution time as well as compilation time using the optimizing compiler. The left graph is for the small input and the right graph is for the large input. The top, dark-colored portion

[3] All entries in the table are initialized to 0, since in Jalapeño all valid offsets will be non-zero

[4] The four additional instructions executed are two dependent loads, a compare, and a branch.

	Small			Large		
	x 100,000		Percent	x 100,000		Percent
Benchmark	Eager	Lazy	Reduced	Eager	Lazy	Reduced
Compress	492	3	99	6202	3	100
DB	12	3	75	455	4	99
Jack	32	28	13	71	51	28
Javac	27	17	37	480	33	93
Jess	64	7	89	790	8	99
Mpeg	133	5	96	1547	6	100
Average	127	11	92	1591	18	99

Table 3: **Dynamic execution count of dynamically linked sites.** Columns 2–4 are for the small input and 5–7 are for the large. Columns 2 and 5 give the counts (in 100,000's) of executed sites that were dynamically linked using the optimizing compiler. Columns 3 and 6 are the counts when lazy compilation is used, (Columns 4 and 7 show the percent reduction).

of each bar represents compilation time, the bottom light-colored portion represents execution time. A pair of bi-colored bars is given for each benchmark. The first bar of the pair results from using the eager approach; the second bar from lazy compilation. Lazy compilation reduces both compilation and execution time significantly when compared to eager compilation. On average, lazy compilation reduces overall time (execution plus compilation) by 26% for the small input and 14% for the large. Execution time alone is reduced by 13% and 11% on average for each input, respectively, since lazy compilation greatly reduces both indirect and direct costs of dynamic linking.

4 Background Compilation

We will now describe how to further reduce compilation overhead by overlapping compilation with execution. Lazy compilation delays method compilation until first invocation, however, the cost of optimizing a method can still be expensive and the execution characteristics of the method may not warrant optimization. In addition, on–demand compilation in an interactive environment may lead to inefficiency. In environments characterized by user interaction, the CPU often remains idle waiting for user input. Furthermore, the future availability of systems built using single-chip SMPs, makes it even more likely that idle CPU cycles will intermittently be available. The goal of background compilation is to extend on–demand and lazy compilation to further mask compilation overhead by using idle cycles to perform optimization.

4.1 Implementation

Background compilation consists of two parts. The first occurs during application execution: when a method is first invoked, it is lazily compiled using a fast non–optimizing compiler (or the method is interpreted). This allows the method to begin executing as soon as possible. However, since this type of compilation can result in poor execution performance, methods which are vital to overall application performance should be optimized as soon as possible.

This is achieved with the second part of the background compilation by using an Optimizing Compiler Thread (OCT). At startup we initiate a system thread that is used solely for optimizing methods. The OCT is

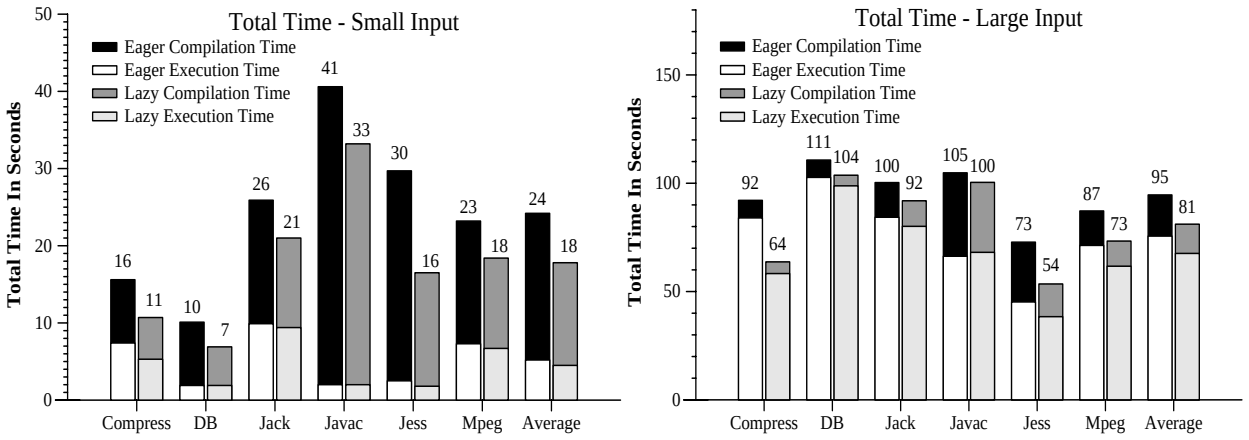

Figure 4: **Overall impact of lazy compilation application performance.** The left bar results from using eager compilation, the right bar lazy. The top, dark colored, portion of each bar is compilation time, the bottom (light-colored) execution. The number above each bar is the total time in seconds required for both execution and compilation time. Lazy compilation reduces both execution time as well as compilation time.

presented with a list of methods that are predicted to be the most important methods to optimize. The OCT processes one method at a time, checking whether or not the class in which it is defined has been loaded. If it has, then the method is optimized. Once compiled, the code returned from the optimizing compiler is used to replace the baseline compiled code (or the stub if the method has not yet been baseline compiled). Future invocations of this method will then use the optimized version.

In order to predict which methods should be optimized by the OCT, we use profiles of the execution time spent in a method. To generate these profiles, we execute the application off–line and compute these times. The profiles accumulate the total execution time spent in each method during execution of the small input. In many studies, this input is referred to as the training set. This profile is then used to generate results using the small input and the large input. The numbers we report for the large input then, show how well the small input predicts the time spent in methods when executing the large input.

Using the profile, each method is assigned a global priority ranking it by execution time with respect to all other methods executed by the application. Once global priorities are assigned to each method, we record it with the methods within each class. As each class is loaded, any methods of the class that have been prioritized are inserted into the OCT's priority queue for eventual optimization. If the priority queue becomes empty, the OCT sleeps until class loading causes new methods to be added.

For background compilation, we collect all profile data during prior executions and hence, do not adapt to execution paths that do not occur during profiling. Background compilation makes use of idle cycles and available processors. If a class is loaded, a performance-critical method can be optimized in the background and (the stub) replaced before first invocation of the method. Compilation overhead for such methods is completely eliminated; a benefit not offered by current adaptive compilation environments [5, 14, 21].

4.2 Results

In this section *total time* refers to the combination of compilation, execution, and all other overheads. The total time associated with background compilation includes:

- Baseline compilation time of executed methods (fast compiler)

- Execution time from methods with baseline-compiled code
- Execution time from methods invoked following code replacement by the optimizing background thread, and
- Thread management overhead

The examples in Figure 5 illustrate the components that must be measured as part of total time for a different scenarios involving a method, Method1. In the first scenario, Method1 is invoked, baseline compiled, and executed. Following its initial execution the OCT encounters Method1 in its list and optimizes it. By the time it is able to replace Method1's baseline compiled code, Method1 has executed a second time. For the third invocation, however, the OCT has replaced the baseline compiled code and Method1 executes using optimized code. Total time for this scenario includes baseline compilation time of Method1 and execution time for two Method1 invocations using baseline compiled code and one using optimized code.

In the second scenario, the OCT encounters, optimizes, and replaces Method1 before it is first invoked. This implies that the class containing Method1 has been loaded prior to OCT optimization of Method1. The OCT replaces a stub that is in place for Method1 with the optimized code. All executions of Method1 use the optimized code. Total time for this scenario includes only the execution time for three invocations of Method1 using optimized code. In the final scenario, the OCT replaces Method1 after its first execution so that the remaining two invocations use the optimized version. Total time for this scenario includes baseline compilation time of Method1 and execution time for one Method1 invocation using baseline compiled code and two using optimized code.

To measure the effectiveness of background compilation, we provide results for the total time required for execution and compilation using this approach. Figures 6 and 7 compare total time with background compilation to total time for the eager, lazy, and ideal configurations results from Table 2 (for the small and large input, respectively). Four bars (with absolute total time in seconds above each bar) represent the total time required for each approach for a given benchmark. The first bar shows results from eager and the second bar from the lazy approach. The third bar is the total time using background compilation and the fourth bar is "ideal" execution time alone. Ideal execution time results from a batch-compiled application (complete information about the application enables more effective optimization and removes all dynamic linking, and there is no compilation cost).

The summary figures show that background compilation eliminates the effect of almost all of the compilation overhead that remains when using the lazy approach. On average, background compilation provides an additional 71% average reduction in total time over lazy compilation for the small input (14% for the large). In comparison with eager compilation, background compilation reduces the total time (execution plus compilation) by 79% and 26% for the small and large input, respectively. The percentage of total time due to compilation is 79% and 20%; hence background compilation reduces total time by more than just the compilation overhead. This occurs since background compilation extends lazy compilation and thereby avoids the dynamic linking effects and enables additional optimization (as discussed in Section 3.3).

Most importantly, however, are the similarities between background and "ideal" execution time. Total time using the background approach is within 21% and 8% (on average for the small and large inputs, respectively) of the ideal execution time. Background compilation therefore, achieves the goal of masking almost all compilation overhead while enabling highly optimized execution times.

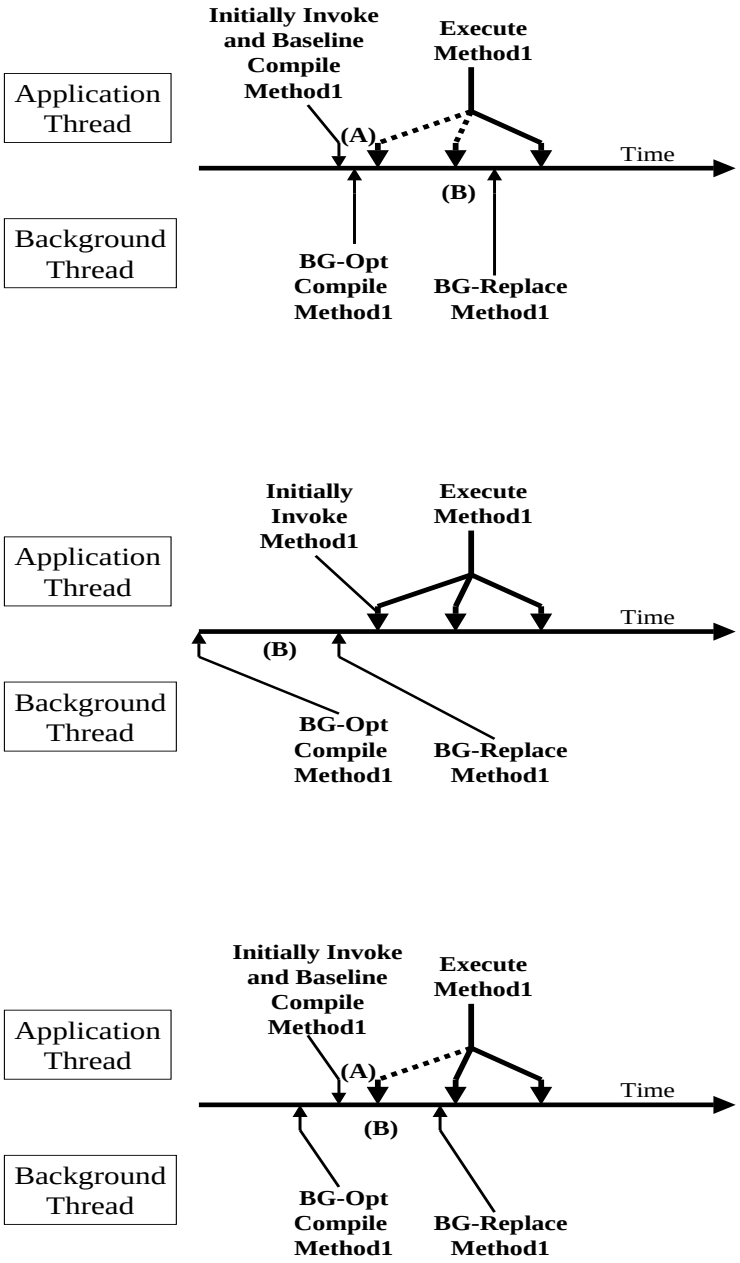

Figure 5: **Example scenarios of background compilation.** In the first scenario, upon initial invocation of Method1, execution suspends and Method1 is baseline-compiled ("(A)" in each figure represents the time required for baseline compilation). When Method1 is executed the code invoked is the baseline-compiled version (represented in all figures by the dotted arrow). Next, in the background (below each timeline), the optimizing compilation thread (OCT) then optimizes Method1. Due to the time required to optimize Method1 ("(B)" in each figure represents the time required for optimization), Method1 is invoked and executed a second time with the baseline-compiled code before the OCT replaces the baseline-compiled code with the optimized version. Once replaced, Method1 executes using the optimized version of the code (represented by a solid line). In the second scenario, the OCT is able to compile and replace Method1 before any invocations of Method1 occur; therefore, all executions use the optimized code. The third scenario shows the OCT replacing Method1 between its first and second invocations so that all executions but the first execute optimized code.

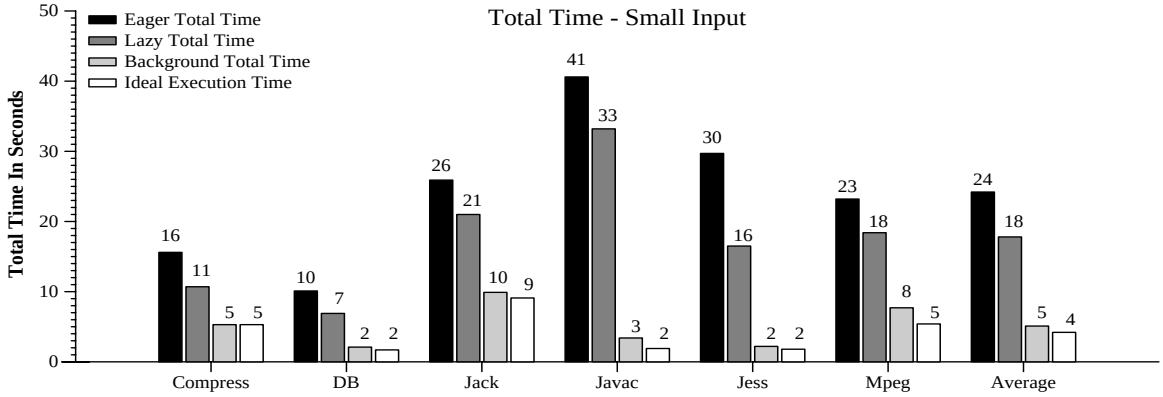

Figure 6: **Summary of total time (in seconds) for all approaches including background compilation for the small input.** Total time includes both compilation and execution time. Four bars are given for each input. The first three bars show total time using eager compilation, lazy compilation, and background compilation, respectively. The fourth bar shows "ideal" execution time alone (from execution of off-line compiled benchmarks). Absolute total time in seconds appears above each bar.

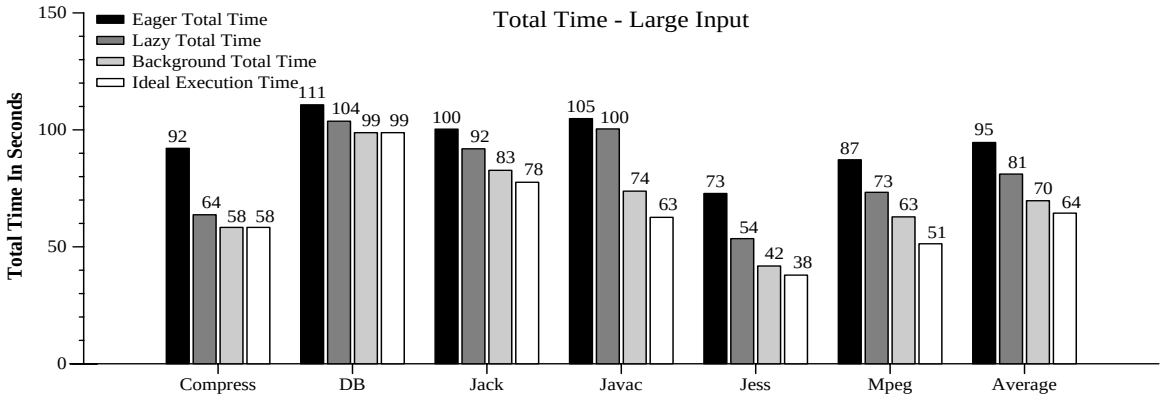

Figure 7: **Summary of total time (in seconds) for all approaches including background compilation for the large input.** Total time includes both compilation and execution time. Four bars are given for each input. The first three bars show total time using eager compilation, lazy compilation, and background compilation, respectively. The fourth bar shows "ideal" execution time alone (from execution of off-line compiled benchmarks). Absolute total time in seconds appears above each bar.

5 Related Work

Our work reduces the compilation overhead associated with dynamic compilation. Much research has gone into dynamic compilation systems for both object–oriented [8, 15, 27] and non–object–oriented [13, 6, 20] languages. Our approach is applicable to other dynamic compilation systems, and can be used to reduce their compilation overhead.

Lazy compilation, as mentioned previously, is used in most Just-In-Time compilers [24, 19, 26, 15, 17] to reduce the overhead of dynamic compilation. However, a quantitative comparison of the associated tradeoffs between lazy and eager compilation, to our knowledge, has not yet been presented. In addition, we provide a detailed description of the implementation and the interesting effects on optimization due to lazy compilation in the Jalapeño VM.

Most closely related to our background compilation work, is that by Hölzle and Ungar [14]. They describe an adaptive compilation system for the Self language that uses a fast, non–optimizing compiler and a slow, optimizing compiler like those used in Jalapeño. The fast compiler is used for all method invocations to improve program responsiveness. Program "hotspots" are then recompiled and optimized as discovered. Hotspots are methods invoked more times than an arbitrary threshold. When hotspots are discovered, execution is interrupted and the method (and possibly an entire call chain) is recompiled and replaced with optimized versions. In comparison, background compilation uses off–line profile information to determine which methods to optimize and never causes stalls in execution due to optimization. In addition, background compilation can potentially eliminate all compilation overhead for some methods since method stubs (for lazy compilation of loaded, but as yet unexecuted, methods) can be replaced with the optimized code prior to initial invocation of the methods. That is, for methods for which optimization is vital to overall application performance, no threshold of invocation count or execution time has to be reached for optimization to be initiated. This is an advantage over all dynamic, adaptive, compilation environments. Another advantage is that our techniques introduce no runtime overhead due to dynamic measurement. Lastly, we perform optimization on a separate thread of execution and exploit idle processors; the combination of which, to our knowledge, has not been examined and published prior.

In other work, Arnold, et. al. [5] uses profiles to guide static compilation. The goal of this project was to determine the performance potential of dynamic, adaptive compilation based on selective optimization in a feedback-based system. In comparison, we incorporate similar, off-line profiles but use them to drive on-line compilation using background compilation.

Another project that attempts to improve program responsiveness in the presence of dynamic loading and compilation is continuous compilation [21]. Continuous compilation overlaps interpretation with Just–In–Time (JIT) compilation. A method, when first invoked, is interpreted. At the same time, it is compiled on a separate thread so that it can be executed on future invocations. They extend this to Smart JIT compilation: on a *single* thread, interpret *or* JIT compile a method upon first invocation. The choice between the two is made using profile or dynamic information. Our background compilation approach uses a separate thread and processor to selectively, background compile only methods predicted as important for application performance. Only a single processor is used in this prior work (and only a single thread in Smart JIT compilation). Our infrastructure uses a compile–only approach, so interpretation in our project is replaced by fast compilation. Interpretation and JIT compilation overlap is also used in the Symantec Visual Cafe JIT compiler, a Win32 JIT production compiler delivered with some 1.1.x versions of Sun Microsystems Inc. Java Development Kits [25].

Another form of background compilation is described in the HotSpot compiler specification [15] from Sun Microsystems. A separate process is used for compilation which is moved to the background when

a threshold of time has been spent compiling a single method. A foreground process then interprets the method to reduce the effect of the compilation overhead for the method. This implementation depends upon the operating system for process management; in Jalapeño each thread of execution and compilation is managed by the Jalapeño thread scheduler. In addition, no profile information is used and the documentation does not provide measurement of the impact of background compilation as a separate process.

Lastly, we previously proposed prefetching class files on separate threads to overlap the overhead associated with network transfer in [18] with execution. Network latency increases the delay during class file loading much like compilation overhead does. In this prior work, we show that by premature access (and transfer) of a class file by a separate (background) thread during application execution, we are able to mask the transfer delay and reduce the time the application stalls for class loading of non-local class files. As in background compilation, we generate profiles off–line but use them to determine the order in which class files are first accessed. Background compilation differs in that we attempt to overlap compilation with execution; hence these techniques are complementary and can be used in coordination to reduce both transfer and compilation overhead.

6 Future Directions

As part of future work, we will extend our background compilation approach to further reduce the effect compilation overhead. We plan to annotate class files so that when non–local class files are loaded by Jalapeño, a profile list can be constructed, eliminating the need for the OCT to read in a list from the local file system. In addition, we plan to extend our single OCT approach to multiple OCTs. That is, we will include the option of using multiple available processors for background optimization. Currently, Jalapeño's optimizing compiler is not re-entrant; only one thread can use the optimizing compiler at a time. Once this changes, our background optimization will be used to exploit multiple idle processors.

One limitation imposed on background compilation by the current Jalapeño implementation is lack of a mechanism for setting thread priorities, i.e., on a single processor, we are unable to *only* compile when the primary thread(s) of execution is idle. With the current infrastructure, the application thread and the OCT must contend equivalently for resources restricting potential for improvement on a single processor. Once thread priorities are implemented as part of future work, we will use background compilation to mask compilation overhead on single processor machines as well. We believe that for interactive programs, execution on a single processor can benefit from background compilation since optimization can be performed when the application suspends waiting for user input.

7 Conclusion

In this work, we focus on reducing the effect of compilation overhead imposed by dynamic compilation. We first quantitatively compare the tradeoffs between eager (class–level) and lazy (method–level) compilation. Lazy compilation reduces the number of methods compiled, thereby reducing compilation overhead. We also introduce and evaluate background compilation using an SMP, an approach that optimizes important methods on a background thread to mask compilation overhead due to optimization.

The infrastructure we use to examine the impact of our compilation strategies is the Jalapeño Virtual Machine, a compile–only execution environment being developed at IBM T. J. Watson Research Center. Currently in Jalapeño, two compilers are used, the fast baseline compiler that produces code with execution speeds of interpreted versions, and the optimizing compiler, a slow but highly optimizing compiler that

produces code with execution speeds two to eight times faster than the code produced by the baseline compiler. Our goal was to design and implement optimizations that enable compilation times of the baseline compiler and execution speeds of optimized code.

We first empirically quantify the effect of lazy compilation on both compilation time and execution time. We show that lazy compilation requires 57% fewer methods (than eager) be compiled on average for each input of the benchmarks studied. In terms of compilation time, this equates to approximately 30% reduction on average for either input, since the number of methods used between inputs is relatively the same. In addition to reducing compilation overhead, lazy compilation also improves execution time by greatly reducing the number of dynamically linked sites, thus avoiding both the direct costs of dynamic linking and the indirect costs of missed optimization opportunities. Lazy compilation reduces optimized execution time 13% and 10% on average for the small and large input, respectively. In terms of total time, lazy compilation enables a 26% and 14% reduction over eager compilation using the optimizing compiler.

We also present a compilation approach that extends lazy compilation. Background compilation masks the overhead incurred by compilation by overlapping it with useful work. With this optimization, we use the Jalapeño optimizing compiler on a background thread to compile only those methods we predict as important for optimization. On the primary thread(s) of execution, the Jalapeño baseline compiler is used so that methods can begin executing much earlier than if they are optimized. The background thread then replaces the baseline compiled method with an optimized version so that future invocations of the method call the optimized version. No prior research, to our knowledge, has presented the effect (on compilation overhead) of utilizing an idle processor for optimization. Our results show that background compilation achieves execution times of optimized code with compilation overhead of baseline compilation. On average, background compilation effectively reduces total time (execution plus compilation) by 79% and 26% for the small and large input, respectively. When compared to lazy compilation, the background optimization reduces total time of 71% for the small input and 14% for the large. We also show that background compilation achieves the runtime performance of applications that are batch compiled (off–line optimization of the entire application at once).

The Java programming language provides an architecture–independent intermediate representation that is and will continue to be exploited by the distributed execution of Internet–computing applications. In order for the execution of these applications to be practical, execution speeds must be fast and overheads associated with execution, i.e., optimization, must not create performance bottlenecks. Dynamic compilation enables state-of-the-art optimizations to improve the execution speeds of Java programs, but also introduces compilation overhead due to optimization. Compilation approaches like the ones presented here are important since they enable optimization while reducing the effect of compilation overhead.

References

[1] B. Alpern, C. Attanasio, J. Barton, A. Cocchi, S. Hummel, D. Lieber, T. Ngo, M. Mergen, J. Shepherd, and S. Smith. Implementing jalapeño in java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1999.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] B. Alpern, M. Charney, J. Choi, A. Cocchi, and D. Lieber. Dynamic linking on a shared-memory multiprocessor. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[4] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the jalaepeño jvm. *Submitted to* ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 2000, 2000.

[5] M. Arnold, M. Hind, and B. Ryder. An empirical study of selective optimization. Technical Report RC 21703, IBM T. J. Watson Research Center, March 2000. Also available as Rutgers Technical Report dcs-tr-411.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. Technical Report HP Laboratories Tech Report HPL-1999-78, 1999. `http://www.hpl.hp.com/techreports/1999/HPL-1999-78.html`.

[7] J.L. Bash, E.G. Benjafield, and M.L. Gandy. The Multics operating system-an overview of Multics as it is being developed. Technical report, 1967. Project MAC, MIT, Cambridge, Mass.

[8] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The jalapeño dynamically optimizing compiler for java. In *ACM Java Grande Conference*, June 1999.

[9] M. Cierniak and W. Li. Optimizing java bytecodes. In *Concurrency: Practice and Experience*, volume 9 (6), pages 427–444, June 1997.

[10] R.C. Daley and J.B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, 1968.

[11] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983. ISBN 0-201-11371-6 `http://users.ipa.net/~dwighth/smalltalk/bluebook/bluebook_imp_toc.html%` .

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[13] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Dyc: An expressive annotation–directed dynamic compiler for c. Technical Report Tech Report UW-CSE-97-03-03, 1997. To appear in *Theoretical Computer Science*. `http://www.cs.washington.edu/research/projects/unisw/DynComp/www/`.

[14] U. Hölzle and D. Ungar. A third–generation self implementation: Reconciling responsiveness with performance. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1994.

[15] The java hotspot performance engine architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[16] Kaffe – an opensource java virtual machine. `http://www.kaffe.org/`.

[17] A. Krall and R. Grafl. Cacao - a 64 bit javavm just-in-time compiler. In *Concurrency: Practice and Experience*, volume 9 (11), pages 1017–1030, November 1997. `http://www.complang.tuwien.ac.at/java/cacao/index.html`.

[18] C. Krintz, B. Calder, and U. Hölzle. Reducing transfer delay using java class file splitting and prefetching. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1999.

[19] Latte: A fast and efficient java vm just-in-time compiler. `http://latte.snu.ac.kr/`.

[20] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[21] M. Plezbert and R. Cytron. Does just in time = better late than never? In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, January 1997.

[22] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. Quassi-static compilation in Java. *Submitted to* ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 2000, April 2000.

[23] Spec jvm98 benchmarks. http://www.spec.org/osg/jvm98/.

[24] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 39(1), 2000.

[25] Sun microsystems jit compiler. http://java.sun.com/products/jdk/1.1.6/download-jdk-windows-006.html.

[26] The symantec just–in–time compiler. http://www.symantec.com/domain/cafe/rguide_21/closerlook.html.

[27] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87*, pages 227–242, December 1987.