Using Annotations to Reduce Dynamic Optimization Time

Chandra Krintz   Brad Calder
University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA 92093-0114
{ckrintz,calder}@cs.ucsd.edu

UCSD Technical Report
CS2000-0663, November 2000

Abstract

Dynamic compilation and optimization are widely used in heterogenous computing environments, in which an intermediate form of the code is compiled to native code during execution. An important tradeoff exists between the amount of time spent dynamically optimizing the program and the running time of the program. The time to perform dynamic optimizations can cause significant delays during execution and also prohibit performance gains that result from more complex optimization.

In this research, we present an annotation framework that substantially reduces compilation overhead of Java programs. Annotations consist of analysis information collected off-line and are incorporated into Java programs. The annotations are then used by dynamic compilers to guide optimization. The annotations we present reduce compilation overhead incurred at all stages of compilation and optimization as well as enable complex optimizations to be performed dynamically. On average, our annotation optimizations reduce optimized compilation overhead by 78% and enable total time speedups of 7% on average for the programs examined.

1. INTRODUCTION

The execution model for mobile, Java programs, is one in which code and data are transfered to remote destinations for execution. Typically, architecture-independent bytecodes are shipped to the execution site and interpreted by a virtual machine (the JVM). However, to overcome the performance limitations interpretation usually imposes, the next generation of Java execution systems [23, 2, 18, 10] employ just-in-time compilation. These new JVMs dynamically compile the bytecode stream (on a method-by-method basis) into machine code before executing it. The resulting execution performance is higher than for interpreted bytecodes, but execution must pause each time a method is initially invoked so that it may be compiled.

Most systems attempt to reduce the delay introduced by the compilation during execution in one of two ways. Either they invoke an optimizing compiler and resort to interpretation if the compilation delay exceeds an arbitrary threshold [10, 19], or they use a combination of fast, non-optimized compilation and slow, optimized compilation [5, 7]. In this paper, we present a framework and a set of methodologies for minimizing the overhead introduced by dynamic compilation while at the same time achieving optimized execution speeds. The core of our approach relies on the introduction of annotations into the Java bytecode stream that can be used to guide optimization. Previous annotation-based techniques have focused on including information that aids optimization, but is too time-consuming to collect on-line [4, 12, 20, 9]. The goal of these efforts has been to make costly optimizations effective in dynamic compilation settings. We extend the use of annotations both to include new profile-based optimization techniques (which might prove too costly to implement on–line) *and* to control the use of simple, fast techniques as well. The resulting infrastructure is general. It allows the compilation system to consider the use of existing dynamic optimization techniques, profile-based techniques, and rudimentary optimizations, alone and in combination. Our work substantially reduces dynamic compilation overhead while enabling optimized execution time.

In addition, we find that most of the dynamic compilation in Java programs occurs at startup. In the programs studied, 77% of of the compilation overhead occurs in the first 4 seconds (initial 10%) of program execution on average. Applying the techniques we have developed reduces the startup delay of Java programs by more than 2 seconds in some cases. As such, this work compliments many of the efforts to reduce startup delay that have been the focus of current research [14, 15, 22].

Finally, we investigate the impact of annotations on program transfer time in mobile computing environments. Our annotation framework introduces a very small number of bits into the bytecode stream so as not to increase transfer delay. Using compression techniques, the annotations we implement increase the size of annotated applications by less than 0.05% on average.

The contributions of this paper are:

- We define a methodology for guiding just-in-time (JIT) compilation using annotations to reduce the cost of runtime interruptions during execution.
- We present an annotation framework that is optimized to keep the annotation image as small as possible so that the increase in transfer time due to annotations is unnoticeable.
- We effectively use our annotation framework to imple-

ment and analyze novel annotation optimizations that significantly reduce compilation overhead without effecting program execution performance.

- We analyze different optimization phases to determine how annotation-guided compilation can reduce both startup delay and compilation overhead incurred during program execution.

2. THE OPEN RESOURCE PLATFORM

The performance of mobile Java programs is greatly improved through the use of dynamic compilation over interpretation. However, the compilation process is more complex and imposes longer, intermittent delays during execution since execution must pause waiting for compilation to complete. To compensate for compilation overhead, many systems use a combination of a very fast interpreter and an optimizing compiler or two compilers (one simple and very fast and the other optimizing).

For this research, we use an open-source, dual-compiler system called the Open Runtime Platform (ORP) which was recently released by the Intel Corporation [18]. The first compiler (O1) provides very fast translation of Java programs [1] and incorporates a few very basic bytecode optimizations that improve execution performance. The second (O3) compiler performs a small number of commonly used optimizations on bytecode and an intermediate form to produce improved code quality and execution time. O3 optimization algorithms were implemented with compilation overhead in mind, hence only very efficient algorithms are used [7]. The compilation overhead, total time, and the number of methods compiled by the ORP compilers is shown in Table 2. Total time consists of both compile and execution time. For comparison, O3 execution time is 5% faster than O1 execution time on average and the compilation time of the O3 compiler is 89% slower than that for O1 on average in the programs studied.

The goal of our work is to use both ORP compilers to reduce O3 compilation overhead while enabling O3 execution time. To evaluate where ORP compilation time is spent, Figure 1 gives a breakdown of the O3 compilation phases over time. The y-axis is time in seconds; the average O3 compile time for the benchmarks is approximately 2.7s. The bar for each application is broken down into eight pieces. `Other` includes memory allocation of data structures and any other code transformation costing less that 100ms. `Const-prop` is constant and copy propagation. `Global-reg` is global register allocation, i.e., the time to allocate physical registers to the local variables of a method. `Build-ir` is the intermediate form translation time; the bytecode of each method is converted to a lower-level form for further optimization. `DCE` is dead code elimination. `Local-reg` is local register allocation, i.e., the time to allocate physical registers to temporary variables required by the translation. `Fg-create` is the time for flow-graph construction, and `loop-opts` is the time for loop optimization. The applications are commonly used Java applications including some of the SPECJvm benchmarks.

3. ANNOTATION-BASED OPTIMIZATIONS

Table 1: ORP Compilation and total time (execution plus compilation). O1 is the ORP fast compiler, O3 is the ORP optimizing compiler. Columns 2 and 3 is O1 and O3 compilation overhead, respectively. Columns 3 and 4 show total time. Columns 5 and 6 show the number of classes and methods compiled, respectively (over the total number of classes and methods).

	Cmpl Time(s)		Total Time(s)		Classes	Methods
Program	O1	O3	O1	O3 (SpdUp)	Cmpl'd	Cmpl'd
Jack	0.30	2.80	42.2	41.5 (1.7%)	46/56	295/315
JavaCup	0.30	3.20	48.9	47.7 (2.4%)	31/36	216/385
Jess	0.32	2.55	44.1	42.9 (2.7%)	134/151	445/690
JSrc	0.26	3.06	49.6	48.2 (2.8%)	48/194	436/2066
Mpeg	0.28	2.37	37.8	31.2 (17.5%)	42/55	223/322
Soot	0.33	1.74	7.0	6.9 (1.4%)	379/721	1119/3607
Avg	0.30	2.62	38.3	36.4 (4.8%)	113/202	456/1231

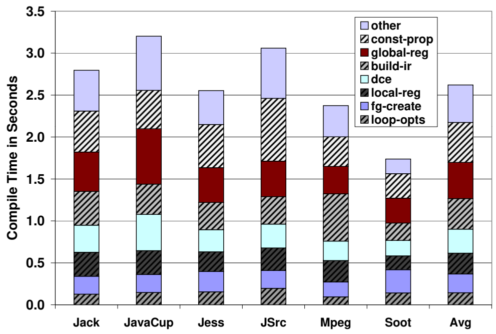


Figure 1: ORP O3 (Optimizing) Compilation Time Breakdown. The y-axis is time in seconds.

A compiler annotation is additional information attached program code and data that help to guide optimization. Annotations have been widely used on program source code in various languages to exploit parallelization and optimization opportunities in parallel and distributed codes. More recently, annotation-based techniques have focused on communicating information that aids optimization, but is too time-consuming to collect on-line [4, 12, 20, 9]. The goal of these efforts has been to make costly optimizations effective in dynamic compilation settings.

We extend the use of annotations both to include new profile-based optimization techniques (which might prove too costly to implement on-line) *and* to control the use of simple, fast techniques as well. It enables the compilation system to consider the use of existing dynamic optimization techniques, profile-based techniques, and rudimentary optimizations, alone and in combination. The goal of our work is to present a framework with which annotations can be implemented that substantially reduce dynamic compilation overhead while enabling optimized execution performance.

In addition, we intend for annotations to be used in a mobile computing environment. Therefore, we must minimize the number of bytes we add for annotations so that the transfer delay for an annotated program does not negate the performance improvements achieved by annotated execution. To ensure that our resulting class file size is kept small, we incorporate compression into our framework and ensure that all annotations implemented impose very little space overhead in the program bytecode stream.

3.1 Annotation Framework

Using our framework, annotations are inserted into a Java program using a bytecode rewriting tool called BIT [17]. Annotations are included in class files using *attributes* and are transfered as part of the application when remotely executed. An attribute is a named piece of information defined as part of the class file structure [8]. The name of the attribute is stored in the constant pool (a table of symbolic data about the class file) and is used to parse and make use of the attribute. Attributes are defined for class files as well as for the methods and fields contained in the class. For example, a `Code` attribute is defined in the Java language specification as a method-level attribute and contains the actual bytecode. The name of each attribute at any level (class, method,field), is included in the constant pool of the class file and is used to distinguish, parse, and make use of the attribute. When a virtual machine encounters an undefined attribute it is required to ignore it. This allows class files to contain user-defined attributes and yet remain compatible with all virtual machines that do not recognize the attribute.

In this work, we add a *single*, user-defined, method attribute for annotations. We combine multiple annotations into the same attribute and use a single character of Unicode [8] to represent the name of the attribute in the constant pool. These two design decisions minimize the size increase of a class file required for annotations. To further reduce annotation size, annotations within each attribute compressed using gzip compression. Gzip is a standard compression utility, commonly used on UNIX operating system platforms. These decisions also distinguish our framework from prior research in this area (see section 5.2).

In our design, annotations of variable length appear sequentially in the attribute. We therefore, must indicate the size of the annotations within each method attribute to the virtual machine executing the class file. As an example, consider an annotation that indicates that a method is very important to optimize. This information can be represented by a single bit: 1 indicating that the ORP O3 compiler should be used or 0 indicating O1, fast compilation. The compilation system must be able to identify that this bit indicates which compiler to use on this method. To enable this, we distinguish annotations using opcodes, or identifiers. This opcode tells the compiler how to parse and make use of the annotation. In addition, since there are possibly many annotations in a single attribute, the compiler must be able to determine where one starts and the next begins. This is done by including the size of the attribute after the opcode. The annotation itself is then concatenated onto the opcode and size (in this case, the annotation is a single bit of data). The general format of an annotation attribute then is a series of triples of the form: $< opcode, size, data >$. The elements of each triple are summarized as:

- *opcode*: (2 bytes) The identifier of this annotation that tells the compilation system how to parse and make use of the following annotation. The end of the attribute section, and thus all annotations, is indicated by a 0 opcode.
- *size*: (4 bytes) The number of bytes for the annotation data that follows.
- *data*: (variable number of bytes) The annotated information.

To incorporate the annotations, the compiler decompresses the annotations contained in the attribute (using the gzip compression library) and reads the opcode and size elements of the first triple (6 bytes total). It then looks up the opcode to determine the use of the annotation. The annotation is parsed and placed in the appropriate data structure in memory or processed directly. This procedure is repeated for the next annotation until the end of the attribute is reached. We use a 2-byte opcode of 0 to delimit the annotation attribute.

3.2 Annotation Implementation

A goal of this research is to reduce compilation overhead while maintaining optimized execution time. The annotations we describe next are meant to achieve this goal for the ORP compiler. As shown previously in Figure 1, the dynamic optimization time in ORP is spread across multiple operations. To reduce time spent overall we consider annotations that reduce the largest contributors in terms of compile time. We examine the effect of annotating results from both static and profile-based analysis to guide efficient optimization of methods. Static information is structural and syntactic information explicitly available in Java bytecode and class files. Profile-based information consists of runtime program characteristics and is collected by instrumenting and executing the programs off-line. We present four types of annotations and describe an implementation of each: those that provide static analysis information, those that enable optimization reuse, those that enable selective optimization, and those that enable optimization filtering. The name of each specific annotation is shown in parentheses prior to each description.

3.2.1 Provision of Static Analysis Information

All compilers collect static information about the code it is compiling to perform translation, transformation, and optimization. For example, information about local variables, control flow, exception handling, etc, may be collected for an optimization by scanning the code. For information for which use is independent from the collection procedure, the analysis and acquisition of it can be performed off-line. We present annotations that communicate analysis data to the compiler as needed in an efficient format. Since the analysis is performed off-line dynamic compilation overhead is reduced.

Global Register Allocation Annotation (global-reg). Since the Java bytecode format is stack-based, it is difficult to achieve acceptable performance of Java programs on register-based architectures without complex analysis and algorithms for register allocation. Many commonly used allocation routines prioritize variables in order to apply more advanced algorithms for the assignment of registers, e.g., prioritized graph-coloring. In ORP, priorities are determined by static counts of local variable uses in the bytecode. To reduce this search time, we use annotations to indicate the priorities. We are able to perform more advanced prioritization (via profiling or some other heuristic) to improve register allocation in ORP, but due to time and space constraints we leave this to future work. For this optimization, we make static counts of local variable usage just like those made in ORP and communicate this information via the annotation.

Flow-graph Generation Annotation (fg-create). A flow graph is a data structure commonly used by compilers to identify changes in program control flow for effective and correct optimization. Most Java compilers generate a flow graph for every method to find basic block boundaries and other pertinent control flow information [5, 7, 13]. This construction requires multiple passes of the Java bytecode. To reduce the time required for such passes we implemented a flow graph generation annotation using our framework. We characterize the control flow structure of each method and use an annotation to present it to the compiler for single pass flow graph construction.

3.2.2 Optimization Memory Reuse

The goal behind the implementation of this next annotation is to enable reuse of analysis information required for multiple optimizations during execution. Most dynamic compilers regenerate information about a method compilation instead of storing it for possible reuse to reduce the memory footprint of the execution. Since it is unclear to the compiler which information can be efficiently reused, much of the analysis traditionally has been repeated. Annotations can be used to indicate which data it is cost effective for the compiler to store for reuse.

Inlining (inlining). Inlining is a common optimization used by all compilers to reduce method or function call overhead. By inlining the code of a method that is called by another, no call is needed, since the code can be executed directly. Commonly in dynamic compilation systems [5, 7], when a method is inlined into another, its (control) flow graph is generated, processed, and possibly optimized prior to insertion into the method it is inlined into. If this method is later inlined into a different method, the process of flow graph creation and optimization is repeated. For methods that are inlined many times, much redundant work is performed by the compiler. It is not desirable to keep all flow graphs in memory in case of reuse, since it can dramatically increase the size of the memory footprint unnecessarily. We therefore, analyzed off-line profiles to determine which executed methods might be inlined multiple times. We indicate these methods to the dynamic compiler at runtime using annotation.

3.2.3 Selective Optimization

We next present selective optimization annotations that aid a multiple-compiler system with selection of the appropriate compiler. In ORP, the O1 fast compiler or the O3 optimizing compiler can be used. For this type of annotation we identify the most important methods to optimize and indicate to the compilation system that all other can be fast-compiled. This annotation has potential to reduce compilation overhead since methods that are fast-compiled are more prevalent. It also limits execution time degradation since methods that impact overall execution time to the greatest extent are optimized.

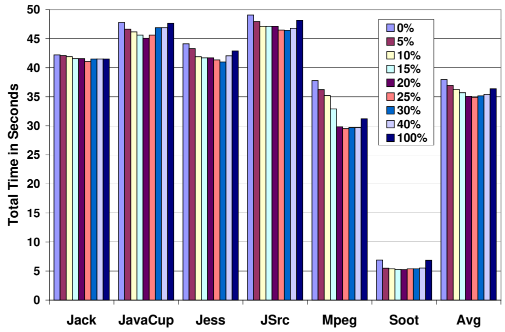


Figure 2: The histogram used to determine the percent of methods most important to optimize. Each bar shows the total time (compilation plus execution) for the program when different percentages of the most frequently executed methods are optimized. The remaining methods are compiled with the O1 compiler. The top 25% frequently executed methods should be optimized on average for the best performance.

Method Priority Annotation (top25%). For this annotation, we use off-line profile data to predict the methods that should be optimized. This is similar to the function of an adaptive compilation system in which methods are first compiled with a very fast, non-optimizing compiler, then optimized when deemed *hot*. Hot-ness is identified using analysis of on-line profiles enabled by method instrumentation. With this annotation, we indicate whether a method is hot using off-line profiles obviating the need for on-line instrumentation and profiling. The annotation indicates whether a method should be compiled using the fast O1 compiler or optimized. To determine the percentage of methods that are important to optimize we gathered execution times for the histogram shown in Figure 2. The data implies that the top 25% of frequently executed methods should be optimized for the best performance. We used profile data to determine which methods were contained in the top 25% in terms of invocation frequency. For each method in the top 25%, we include a one bit annotation to indicate that it should be compiled with the ORP O3 compiler.

3.2.4 Annotations for Optimization Filtering

The final type of annotation we describe is used for optimization filtering. Currently an optimizing compiler performs all available optimizations on a method. To reduce compilation overhead, it may be beneficial to only perform an optimization when static or profile-based analysis of the method indicates that it is profitable to do so.

For this annotation we construct a 1-byte bit mask as part of the annotation data that maps to a list of expensive optimizations. For example, the first bit might map to inlining,

Jack	Spec Benchmark: Java parser generator based on the Purdue Compiler Construction Toolset
JavaCup	LALR Parser Generator: A parser is created to parse simple mathematics expressions
Jess	Spec Expert System Shell Benchmark: Computes solutions to rule based puzzles
JSrc	Bytecode Processing Tool: converts Java class files to HTML pages in a javadoc-like format
Mpeg	Spec Audio File Decompression Benchmark: Conforms to the ISO MPEG Layer-3 audio specification
Soot	Bytecode Processing Tool: converts Java class files to an intermediate format: Jimple

Table 2: Description of Benchmarks Used.

Table 3: General characteristics of execution using two different inputs (Large and Small).

	Executed/Total Classes		Executed/Total Methods	
Program	Large	Small	Large	Small
Jack	46/56	46/56	295/315	265/315
JavaCup	31/36	28/36	216/385	213/385
Jess	134/151	133/151	445/690	412/690
JSrc	48/194	48/194	436/2066	436/2066
Mpeg	42/55	42/55	223/322	201/322
Soot	379/371	379/371	1119/3607	1106/3607
Avg	113/202	113/202	456/1231	439/1231

the second to register allocation, the third to constant propagation, and so on. Each bit that is set in the mask indicates to the compiler that the associated optimization should *not* be performed for that method. This annotation has the potential for reducing compilation overhead by filtering time-consuming optimizations that do not result in substantially improved execution performance on a per method level. We are able to determine which optimizations improve performance of methods using profiling.

Constant Propagation Filtering. In Figure 1, the ORP O3 compiler spends a substantial amount of time performing constant propagation. For some methods, the empirical data we have collected shows that this expensive optimization does not reduce runtime performance enough to warrant the optimization time. For those methods, we use the bit mask of this annotation to indicate that the copy propagation optimization should be excluded. Due to space constraints, we do not provide the full implementation and results of optimization filtering for copy propagation, but will do so in the final version of this paper, if accepted and space permits. Our initial results do indicate performance improvement potential.

4. RESULTS

The results we present were gathered by repeatedly executing applications on a dedicated 300Mhz x86 Intel Pentium II machine running Linux v2.2.15. The applications we examine, described in Table 2, are part of the SPEC Java benchmark suite and additional applications commonly used in Java studies [15, 7, 21]. Two of the optimizations we present (inlining and top25%) require profile information for annotation construction, as described in the previous section. Since compilation overhead (method use) is dependent upon program inputs, we present profiles gathered using two different inputs. The input we used to generate all of the results is the larger (in terms of execution time) of the two inputs, and its statistics are shown in table 2. The second input, which we call small, results in a shorter execution time (size 10 for the SpecJvm benchmarks) than the large input. Table 3 shows some differences in the execution characteristics between the two inputs. Above the slash is the number of classes (and methods) used during execution of the application using the associated input; below the slash we show the total class (and method) counts. We use both inputs to generate profiles for our results. For results gathered using the large input to generate both the profile and the results we append `Same` to the name and refer to them as same-input. For results that are gathered using the large input for the results and the small input for the profile, we append `Diff` to the name and refer to them as cross-input. The inclusion of both cases enables the reader to compare the impact of our techniques using perfect (Same) and imperfect (Diff) profile information.

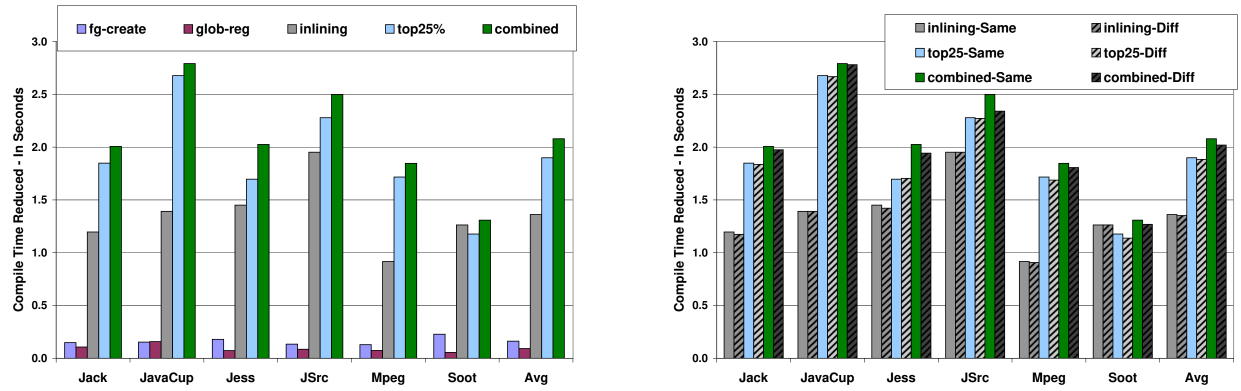
The overhead associated with annotations consists of class file size increase and ejection overhead needed to process and make use of the annotations. We detail the former in Section 4.2. Any ejection overhead imposed by annotations is included in the overall results we now present. We first present results in terms of the reduction in compilation time resulting from the use of our annotations. The number of seconds of compilation overhead reduced is shown in Figure 3(a). Each bar shows the reduction due to each of the individual optimizations. The right-most bar for each benchmark shows the combined reduction in compilation overhead for all of the annotations together.

The right graph, Figure 3(b), shows the inlining, top25%, and combined bars from (a) for both same-input and cross-input results. Figure (a) shows the same-input results alone. These additional results in (b) indicate that even with imperfect profiling information, our annotated optimizations substantially reduce compilation overhead. On average, our annotation optimizations reduce compilation overhead by 79.0% (78.1% for cross-input).

Figure 4 shows the time required for O3 (optimizing compiler), 01 (fast compiler), and the combined results (called *Annot*) from Figure 3 for same-input and cross-input configurations. On average, annotated compilation time is approximately 250ms slower than O1 compile time and 2 seconds faster than O3 compile time.

Next we present the effect of our annotation optimizations on total time: compilation and execution time combined. First though, notice that in Figure 3 both graphs indicate that the majority of compile time reduction is due to two optimizations, inlining and top25%. As such, we also have collected results for the combination of these two optimizations alone. This data is shown by the *Annot-IT* results (I=inlining, T=top25%) in the following graphs.

Figure 5 shows the speedup achieved through the use of annotation over O3 total time for both of our annotation



(a) (b)

Figure 3: Seconds of compilation overhead reduced. The left graph shows the number of seconds of compilation overhead reduced due to the annotation optimizations we present. They are broken down by optimization. The right-most bar of each group is the number of seconds reduced due to all of the annotations combined. The right graph (b) duplicates the inlining, top25%, and combined bars from (a) (now with the -Same extension) and adds cross-input results (Diff).

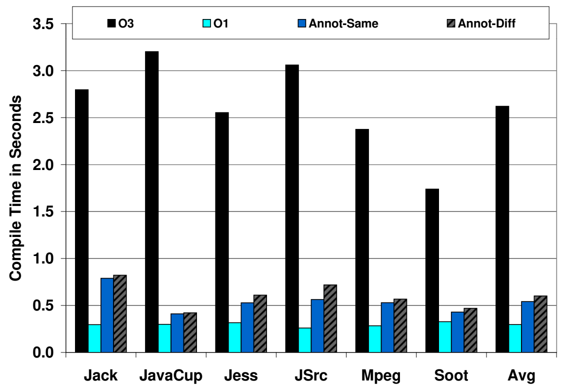


Figure 4: Total compilation overhead for O3 and O1 compilation, and combined annotated compilation.

schemes (Annot and Annot-IT). The first two bars show the speedup achieved through the use of all annotations using both input configurations (Annot); the second two bars show the speedup from the inlining and top25% annotations alone for both input configurations (Annot-IT). The overall effect on total time is more dramatic the shorter the execution time, as expected. Our annotation optimizations achieve 2% to 23% speedup over O3 total time for the programs studied.

These results indicate that our annotation optimizations reduce compilation overhead by 78% with little effect on execution performance. Moreover, we achieve total time speedups of 7% on average across all benchmarks. We now consider the effect of these optimizations when we only have access to the application class files, as opposed to all classes, for analysis prior to execution.

4.1 Local v/s Remote Execution

Mobile Java programs are commonly transfered over a network for remote execution either through the use of dynamic class file loading of individual class files or by archiving and compressing the application as a single file, e.g. Java archive (jar). In addition, these programs commonly use Java class libraries during execution that are not part of the application itself, but are shared by all such programs. These library classes are not transfered for remote execution but are located at the destination for use by remotely executed Java programs. To this point, we have assumed that we are able to annotate both the application as well as the libraries it uses. However, this is not always the case and as such, we present results on the effect of annotating only non-local, or application class files. To our knowledge, this has not been addressed in any prior or related work yet is vital to the empirical evaluation of annotation-based techniques.

We show in Figure 6 the effect of our annotations when they are incorporated into only non-local class files. Local files are optimized with the O3 compiler. These results are achieved by using only the two best-performing annotations inlining (I) and top25% (T). The two input configurations included (Same and Diff) indicate that imperfect information has little effect on the overall performance of these annotations. Our inlining and top25% annotation optimizations on remote class files alone achieve 1% to 21% speedup over O3 total time for the programs studied. The average speedup across benchmarks is 6.1% and 5.6% for the same-input and cross-input configurations, respectively. The percent reduction in compilation overhead is not shown by this figure but were generated in the same way as Figure 3; we've omitted this graph due to space constraints. However, these annotations reduce compilation overhead by 76.9% (74.6% for cross-input configurations) on average.

As part of future work we will consider the effect of providing general annotations for local (library) class files that can

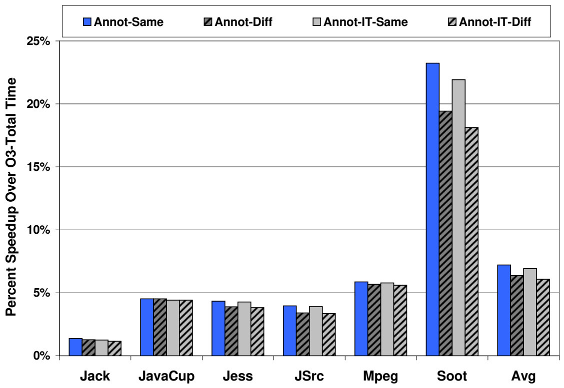


Figure 5: Speedup over optimized (ORP O3) total time due to annotated execution.

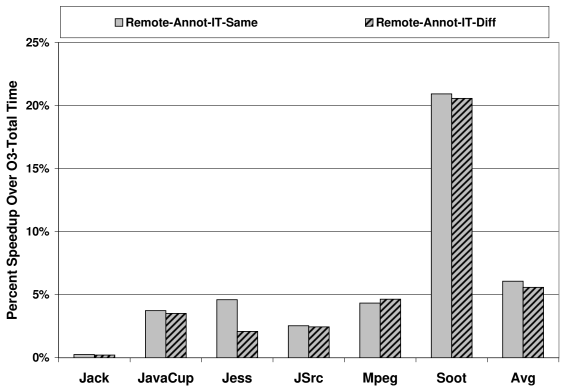


Figure 6: Speedup over optimized (ORP O3) total time due to annotated execution on remote classes only. For these results, only the inlining and top25% annotations are used.

be used to improve performance regardless of the invoking program. For example, it has been shown for C and Fortran that execution behavior of commonly used Unix libraries is similar across different programs and that this information can be used to guide optimization [6]. This implies that profile-based techniques for a subset of programs can potentially be used to optimize shared libraries. We are investigating such techniques as part of future work.

4.2 Annotation Size

Since annotations are added to class files that transfer for remote execution, we must ensure that our framework and annotation implementations increase class file size minimally. Since it is this size that dictates the transfer time on a particular network, we present overhead as the number of kilobytes added for annotation. Table 4.2 shows this data for the different annotations implemented across all non-local class files. In parentheses is shown the number of kilobytes added for annotations in both the non-local and local class files combined. On average, the annotations add less than 0.05% to 3.4% to applications alone on average. By only incorporating the two best annotations (inlining and top25%) we increase application size by less than 0.05%. This is compared to the 7% to 97% increase in size by existing annotation implementations for a single annotation optimization [11, 12, 20]

Table 4: The added size in kilobytes due to annotation.

	Size of Annotation in KBytes for Non-Local Classes - Total for All Classes in Parens.			
Program	Prepass	Regalloc	Inlining	Top25%
Jack	12.55 (16.41)	0.74 (1.25)	0.04 (0.06)	0.04 (0.06)
JavaCup	7.02 (13.61)	0.47 (1.37)	0.03 (0.07)	0.03 (0.07)
Jess	9.74 (14.62)	1.02 (1.61)	0.05 (0.08)	0.05 (0.08)
JSrc	12.29 (15.39)	1.22 (1.68)	0.05 (0.08)	0.05 (0.08)
Mpeg	5.15 (8.35)	0.71 (1.17)	0.03 (0.05)	0.03 (0.05)
Soot	6.44 (9.66)	0.60 (1.09)	0.04 (0.07)	0.04 (0.07)
Avg	8.87 (13.01)	0.79 (1.36)	0.04 (0.07)	0.04 (0.07)
Incr	3.6% (5.2%)	0.3% (0.5%)	0.0% (0.0%)	0.0% (0.0%)

4.3 The Effect on Startup Time

Lastly we consider a different performance metric with which we can empirically evaluate our annotation optimizations. It is our experience that most of the compilation in Java programs occurs at startup: In the programs studied, 77% of of the compilation overhead occurs in the first 4 seconds (initial 10%) of program execution on average. By reducing compilation overhead, annotated execution should substantially reduce this startup cost. Figure 7 confirms this with graphs of the cumulative distribution of compile time over program lifetime (in seconds on the y-axis). We show the actual cumulative time in seconds on the x-axis as opposed to the percentages commonly shown for a cumulative distribution function. We do this so that the duration of startup time is evident. That is, throughout program execution, the number of seconds of compilation overhead that have occurred since the start of the program is expressed by the data. The overhead for the O3 compiler is shown by the top (dark) line on each graph.

The bottom two lines indicate the effect of our two best performing annotation optimizations alone (inlining and top25%): startup overhead is substantially reduced for every program. For example, for Mpeg, all compilation completes in 3 seconds. Using annotated-execution this point is reached in approximately one second. Using annotated execution for all programs, the majority of startup compilation completes more than 2 seconds earlier with annotated execution than without.

Another interesting detail shown by these graphs is the compilation overhead that occurs at the end of execution for JavaCup and JSrc. The methods compiled during this period are those for I/O and clean up. We plan to use this characteristic to guide future annotation implementation. For example, optimization of such methods for an interactive program can be avoided since they are only used at the end of the program.

To summarize, our annotation optimizations reduce compilation overhead by 78% and speedup total time by almost

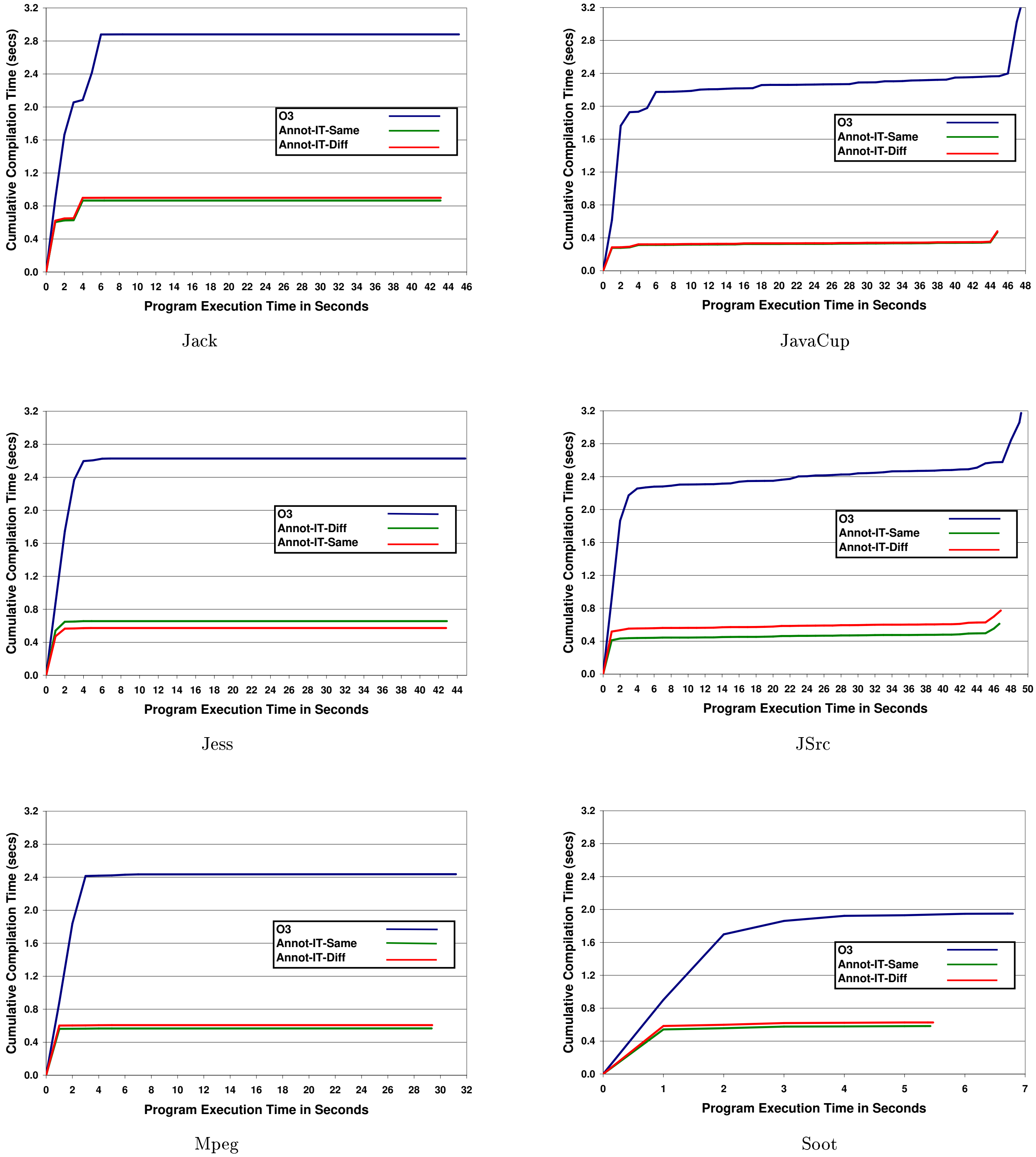


Figure 7: Cumulative compilation time in seconds over the lifetime of each program (y-axis in seconds). These graphs show, throughout program execution, the number of seconds of compilation overhead that have occurred (x-axis) since the start of the program. The overhead for the O3 compiler is shown by the top dark line on each graph; the bottom lines indicate the effect of annotation using our inlining and top25% optimizations. On average, 77% of the compilation overhead occurs in the first 4 seconds (10%) of program total time, e.g., for Jack, almost all of the 2.8 seconds of compilation overhead occur in the first 6 seconds. For all programs, a majority of the startup compilation completes more than 2 seconds earlier than with execution without annotations.

7% on average across all benchmarks. We have shown empirically that there is little effect in performance due to imperfect information. That is, when we use a different input to generate profile information that that used to generate the final results, the compilation overhead reduction and speedup percentage is similar to that resulting from the use of perfect information (same input for result and profile generation). We've also shown that by annotating only remote class files (only application files without libraries) we are able to achieve speedups of within 1% of performance due to annotating both local and remote files. Lastly, we've shown that annotated execution substantially reduces the effect of compilation overhead on startup time. All of which is vital to the user's perception of program performance as well as the raw performance of mobile Java programs.

5. RELATED WORK

In this section, we discuss related Java annotation research. We first describe related work on dual-compiler systems and then describe recent annotation research.

5.1 Fast v/s Optimizing Compiler

Existing Java systems use two compilers in an adaptive compilation scheme [3, 7, 10] in which methods are first compiled with a very fast compiler that converts bytecode into instrumented executable code. Instrumentation enables online measurements to be made to discover hot methods. Hot methods are those that are frequently executed [7, 10] or those in which a substantial amount of execution time is spent [3]. Hot methods are re-compiled by the optimizing compiler. The optimized code is used by future invocations of the method to improve execution time. This process amortizes the cost of optimized compilation since only frequently executed (or methods otherwise deemed important) are optimized.

In our work, we perform off-line instrumentation and analysis to determine important methods to optimize and pass this information to the compiler using our selective optimization annotation. This obviates the need to spend time for both the instrumentation, on-line measurement, and recompilation. Our methods may be complementary and prove to reduce both compilation delay and execution performance. As part of future work though, we will provide and empirical evaluation of the performance advantages to using off-line static and profile-based analysis communicated via annotations or adaptive compilation or both.

In [16], the authors use a technique similar to our selective optimization annotation to optimize important methods on a background processor. If a method is encountered before it has been optimized then it is compiled with a fast compiler and possibly replaced with an optimized version if deemed important. Off-line profiles are generated to determine which methods are important based on the time spent executing them. This infrastructure, however differs from our selective optimization annotation since they provide no mechanism for communicating the information to the compiler except as a file on disk. This prior work also does not differentiate between local and non-local files and their profiles include both local and non-local methods. In addition, our work extends both of these prior works by including many other novel annotation optimizations not yet considered for the reduction of compilation overhead.

5.2 Annotation-based Dynamic Optimization

Pominville et.al. [20] recently published a framework for annotation of Java bytecode that enables static and dynamic, off-line analysis to be incorporated for optimization. The design of their framework differs from ours in that it does not consider the size of their annotations or the transfer delay incurred through their use. They use separate class file attributes for each annotation increasing the class file constant pool size an arbitrary amount (for the name of the attribute) for each. The single annotation optimization they implement as a prototype of their framework is array bounds check elimination which increases application size 7% to 16% for the benchmarks they included in their study. They provide no implementation or description of how dynamic (profile) information is used to construct annotations in their framework. They only provide details on the array bounds check elimination, an optimization already performed in the ORP O3 compiler. Our empirical breakdown of the time spent in O3 compilation groups the time for this optimization in to `Other` indicating that it alone requires less than 100 ms for the entire application.

Hummel et.al. [11], show how static information about register allocation in the Kaffe JIT can be conveyed using annotations. They use a virtual register scheme in which they assume an infinite number of registers, make virtual register assignments off-line, and communicate this information to the JIT compiler for register allocation. They provide no other annotation implementation or mechanism for reducing the transfer delay imposed by their 33% to 97% increase in class file size. They also, as in the aforementioned work, use a separate attribute for each annotation. In our work, we use a single attribute for all annotations and a very short name to indicate that the attribute is an annotation. In a project similar to this prior work, Jones et.al. in [12] describe an annotation for register allocation. The annotation assigns 256 virtual register numbers to the bytecode of each method to improve execution performance using the Kaffe JIT compiler. They also provide no mechanism for annotation size reduction and increase class file size 31% to 38% for the programs presented. Lastly, they provide no general framework or format for the annotations.

In addition, these projects do not consider the use of annotation to reduce optimization overhead in general as we have done in this paper. As part of future work, we will implement and evaluate the effect of expensive optimizations that are now feasible in a dynamic compilation setting through the incorporation of annotation. Specifically, we plan to use our annotation framework to implement more complex register allocation and bounds check elimination algorithms in the ORP O3 compiler to evaluate compilation overhead and execution time benefits effectively. Such an annotation implementation will enable a more complete empirical performance comparison of our work with this related work.

6. CONCLUSION

We have presented an annotation framework for Java programs that substantially reduces compilation overhead in the ORP dynamic optimizing compiler. The annotation language is highly extensible, representing an instruction set with opcode, size of data, and annotation data, and only requires one annotation attribute. The framework enables incorporation of highly-compressed, static and profile-based information into Java class files using a user-defined attribute for each method. These *annotations* enable reduction in compilation overhead of 75% on average, while increasing class file size (and hence transfer delay) by less than 0.05%.

Compilation overhead in execution environments for mobile code is expensive since optimization, resulting in improved execution time, uses time-consuming analysis and processing even for very simple optimizations. However, the potential for execution speedup is large since runtime information can be used for program optimization and specialization. The annotations we present provide analysis information to the optimizing compiler to obviate the need for its collection at runtime. In addition, we pass dynamic information from off-line profiles via annotations to the compilation system so that methods, predicted to be most important, are selectively optimized. Our cross-input results show that our annotations are able to achieve performance within 2% of that achieved with perfect profile information. In addition, our annotation optimizations achieve an overall average speedup of 7% over using ORP O3 dynamic optimization. We also show that over 77% of the compilation occurs within the first 4 seconds of execution on average, and we are able to remove 2 seconds of this startup delay on average. This substantially reduces the impact of compilation on program startup time.

Our annotation framework, provides information from off-line, static and dynamic (profile-based) analysis to a dynamic, optimizing compilation system, to substantially reduce compilation overhead and enable more complex optimizations to be performed. We are currently designing and developing new annotations using this compact annotation representation and the ORP O3 compiler. Some examples include annotations that improve garbage collection and memory performance e.g., code and data placement, on-stack allocation of objects, lifetime estimates, and annotations that aid in branch and loop optimizations.

7. REFERENCES

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. Fast,Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, October 2000.

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

[4] A. Azevedo, A. Nicolau, and J. Hummel. Java Annotation-Aware Just-In-Time Compilation System. In *ACM Java Grande Conference*, June 1999.

[5] 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.

[6] B. Calder, D. Grunwald, and A. Srivastava. The Predictability of Branches in Libraries. In *28th International Symposium on Microarchitecture*, November 1995.

[7] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, October 2000.

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

[9] 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, University of Washington, 2000.

[10] The Java Hotspot performance engine architecture.

[11] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the Java Bytecodes in Support of Optimization. In *Journal Concurrency:Practice and Experience, Vol. 9(11)*, November 1997.

[12] J. Jones and S. Kamin. Annotating Java Bytecodes in Support of Optimization. In *To appear in the Journal of Concurrency: Practice and Experience*, 2000.

[13] Kaffe – An opensource Java virtual machine.

[14] 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.

[15] C. Krintz, B. Calder, H. Lee, and B. Zorn. Overlapping execution with transfer using non-strict execution for mobile programs. In *Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[16] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the Overhead of Dynamic Compilation. In *Software—Practice and Experience*, 2000. To appear.

[17] H. Lee and B. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS97)*, pages 73–82, Monterey, CA, December 1997. USENIX Association.

[18] Open Runtime Platform (orp) from Intel Corporation. `http://intel.com/research/mrl/orp`.

[19] 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.

[20] P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A Framework for Optimizing Java Using Attributes. In *Sable Technical Report No. 2000-2*, 2000.

[21] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. Toba: Java for applications a way ahead of time (wat) compiler. In *Proceedings of the Third Conference on Object–Oriented Technologies and Systems*, 1997.

[22] E. Sirer, A. Gregory, and B. Bershad. A practical approach for improving startup latency in Java applications. In *Workshop on Compiler Support for Systems Software*, May 1999.

[23] 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.