Open access • Proceedings Article • DOI:10.1145/378795.378831

Using annotations to reduce dynamic optimization time — Source link

Chandra Krintz, Brad Calder

Institutions: University of California, San Diego

Published on: 01 May 2001 - Programming Language Design and Implementation

Topics: Dynamic compilation, Just-in-time compilation, Single Compilation Unit, Optimizing compiler and Compiler

In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2001.

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

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 speedups of 7% on average for the programs examined.

1 Introduction

The execution model for mobile programs consists of code and data first being transferred to a remote destination and then executed. Typically, an architecture-independent program representation (e.g., bytecodes for the Java language) is shipped to the execution site and interpreted by a virtual machine. However, to overcome the performance limitations interpretation usually imposes, these systems now employ just-in-time compilation [25, 2, 19, 11]. These new virtual machines dynamically compile the bytecode stream (on a method-by-method basis) into machine code before executing it. The resulting execution time is lower than for interpreted bytecodes, but execution must pause each time a method is initially invoked so that it may be compiled.

Dynamically compiling and optimizing a program can cause significant delays during execution. Most systems attempt to reduce this delay in one of two ways. The first approach is to invoke an optimizing compiler and resort to interpretation if the compilation delay exceeds an arbitrary threshold [11, 20]. The other approach uses two dynamic compilers, a fast, non-optimizing compiler, and a second optimizing compiler [5, 7]. The program is compiled first with the fast compiler, and then frequently executed methods are compiled later with the optimizing compiler based on dynamic information gathered during execution. In [17], we examine the use of off-line profile information to decide which compiler to use initially, however, we provide no automatic mechanism for the communication of this information to compilation system. In this paper, we present such a mechanism that introduces annotation into the bytecode stream to communicate compilation analysis as well as off-line profile information. The goal of our research is to minimize the overhead introduced by dynamic compilation while achieving optimized execution speeds.

Existing annotation-based techniques annotate Java bytecode with analysis information that is time-consuming to collect to guide dynamic compilation [4, 13, 21, 10]. The goal of this prior work was to make costly optimizations feasible in dynamic compilation settings. In this paper, we extend annotation-based compilation and optimization (1) to provide a general annotation representation to guide dynamic compilation, (2) to examine the effects of using annotations to reduce the startup delay and intermittent interruption caused by dynamic optimization, (3) to examine new profile-based annotations to guide optimization, and (4) to generate annotations that do not increase the application transfer size. The latter is very important if annotated-execution is to be used in a mobile environment. If the size of the annotations are not very small, they can introduce significant transfer delay which can negate any benefit achieved through the use of the annotations. Since we intend for our annotation optimizations to be used in a mobile environment, we ensure that they not only improve performance at runtime but do not introduce transfer overhead. A primary contribution of this work is the implementation of annotations that increase the size of annotated applications by less than 0.05% on av-

erage.

Another contribution our work makes is the reduction in program startup time. We have found that most of the dynamic compilation for Java programs occurs at program startup. In the programs studied, 77% of of the compilation overhead occurs in the first 4 seconds (initial 10%) of program execution on average. The application of our techniques reduces startup delay by more than 2 seconds in many cases which enables significantly more progress to be made by the programs. Startup delay has been the focus of much past research since it substantially effects a user's productivity and perception of program performance [8, 24]. Using annotations extends and compliments these and many other efforts [15, 16, 23] to substantially reduce the startup time of mobile programs.

We use the Open Runtime Platform (ORP) [19] from Intel Corporation as our experimental implementation infrastructure. The compilation system and performance characteristics of ORP are similar to other existing platforms [2, 19, 11] for dynamic Java optimization. In the next section, we detail the ORP compilation environment and the overhead inherent in its optimization system. In Section 3, we describe our annotation framework and multiple annotation-based optimizations. We then empirically evaluate the effect of our optimizations in Section 4. We conclude the paper with related work and our conclusions in Sections 5 and 6.

2 The Open Runtime 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.

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 [19]. 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 1. The applications are commonly used Java applications including some of the SPECJvm benchmarks. The input used for collection of these statistics is denoted as Input1 in the results section. 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 for the programs studied.

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

To evaluate where ORP compilation time is spent, Figure 1 gives a breakdown of where time is spent during the different O3 compilation phases. We use these results along with the speedups resulting from the different optimizations and their combinations in order to determine which optimizations might benefit from using annotations. The y-axis is time in seconds; the average O3 compile time for the benchmarks is approximately 2.7 seconds. The bar for each application is broken down into eight pieces. *Other* denotes memory allocation of data structures and any other code transformation costing less that 100 milliseconds. *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.

3 Annotation-based Optimizations

A compiler annotation is additional information attached to program code and data to help 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, 13, 21, 10]. The goal of these efforts has been to make costly optimizations feasible in dynamic compilation settings.

We extend annotation-based compilation and optimiza-

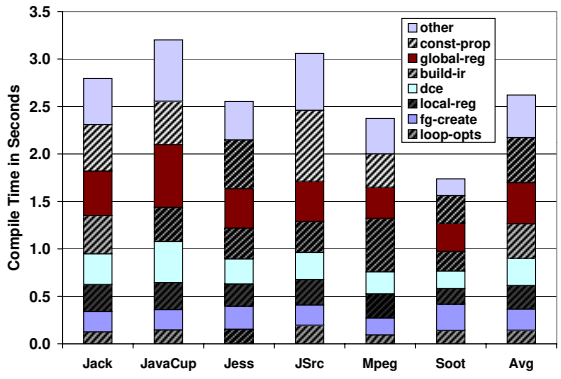


Figure 1: ORP O3 (Optimizing) Compilation Time Breakdown. The y-axis is time in seconds.

tion to provide a general annotation representation that minimizes the number of bytes used to represent the annotation. To this end, we incorporate compression into our framework and ensure that all annotations implemented impose very little space overhead in the program bytecode stream. In addition, we examine using new static and profile-based optimization techniques to guide dynamic compilation.

3.1 Annotation Framework

Our framework incorporates a bytecode rewriting tool called BIT [18] with which we insert annotations into a Java program. Annotations are included in class files and are transferred as part of the bytecode stream when remotely executed. Annotations are stored in a bytecode data structure called an *attribute* as defined in the Java language specification [9]. An attribute data structure is 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 makes attributes ideal for the storage of annotations since it allows annotated class files to remain compatible with all JVMs that are not annotation-aware.

In this work, we add a *single*, user-defined, class attribute for annotations. We combine multiple annotations into the same attribute and use a single character of Unicode [9] 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 are 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).

In our design, annotations of variable length appear sequentially in a given attribute. The encoding we chose for our annotation language is very similar to an instruction set architecture (ISA) format for a variable length ISA. The general format of an annotation attribute is a series of triples of the form: $< opcode, size, data >$. The 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 data then appears right after the opcode and size. The elements of each annotation 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.

- $size$: (2 bytes) The number of bytes for the annotation data that follows. The maximum size of an annotation is 64KB.

- $data$: (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. For annotations that are specific to a method within the class, the first two bytes of data contain a method identifier. Annotations can also be used across all methods in a class file; for these no method identifier is needed. This parsing procedure is repeated for the next annotation until the end of the attribute is reached. We use a 1-byte opcode of 0 to delimit the end of an annotation stream.

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 effect those phases that are the largest contributors to total compilation overhead.

We examine using annotations of both static and profile-based analysis to enable efficient, dynamic 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 we will provide results for is shown in parentheses at the start of each paragraph describing the annotation.

3.2.1 Provision of Static Analysis Information

All compilers collect static information about the code they are compiling to perform translation, transformations, and optimization. For example, information about local variables, control flow, exception handling, etc., may be collected for an optimization by scanning the code. If the collection of data analysis can be performed independent of its use, the analysis and acquisition of it can be performed off-line. We first present annotations that communicate such analysis information to the compilation system in an efficient format. Since the analysis is performed off-line, dynamic compilation overhead is reduced.

Global Register Allocation Annotation (global-reg). The Java bytecode format is base upon a stack architecture and hence, 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. Counting is performed by walking through each method. To avoid this bytecode scan, we use annotations to indicate the priorities. In addition, more advanced prioritization (via profiling or static heuristics) can be used to improve register allocation in ORP, but this is left for future work.

For this global register allocation annotation, we make static counts of local variable usage just like those made in ORP and communicate this information via the annotation. The data element of the annotation triple consists of two bytes to indicate the method and one byte for each local variable. The bytes are arranged in the same order as are the local variables in the local variable array [9]. In the programs studied, the maximum number of local variables used by a single program is 1719 (JSrc); the maximum for any single method is 31 (Mpeg). The average number of locals per method is 2.5. These numbers include methods in the system class libraries also. For non-local class files, the maximum number of local variables used by a program is 1247 (JSrc) with an average method use of 2.6 variables.

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, 14]. 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.

We construct this annotation for each method to enable automatic generation of the flow graph without the prepass operation. As with all other optimizations, fg-create is implemented with a single annotation per method (using a single opcode). The annotation is a list of the basic blocks in the method. The annotation begins with a two-byte method identifier (id) followed by the count of the total number of basic blocks that follow. Each basic block representation includes the block id, its (annotation) size, id numbers of the predecessor and successor blocks, start and ending bytecode indices, and other special information (loop header flag, exception handling block, etc), if any. The annotation enables construction of an ORP flow graph for a method with a single scan of the annotation. Across all benchmarks studied there are just under 14000 basic blocks and each method requires 4.2 blocks on average.

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. Since a compiler is unable to predict which analysis information will be reused by future phases of optimization, it must store all of the information or repeatedly regenerate it. Regeneration is performed since storage of the analysis information can substantially increase the size of the memory footprint of the execution. Some compiler stages may also modify analysis information requiring additional copies to be stored for reuse. Annotations can be used to indicate which data it is cost effective for the compiler to store for reuse. The annotation optimization we implement is for the reuse of inlining information in ORP.

Inlining (inlining) Annotation. Inlining is a common optimization used by all compilers to reduce method or function call overhead. By inlining a method call, the call and return are removed, the inlined method code becomes part of the method which contained the call, and that code is optimized along with the rest of the code in the method during optimization. Commonly in dynamic compilation systems [5, 7], when a method is inlined into another, its (con-

trol) 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.

For this optimization, we include one annotation per method, the annotation contains the method identifier and a single bit of information as the data element. When this bit is set, it indicates to the compiler that the optimized flow graph should be stored in memory for reuse. When unset, the bit indicates that the flow graph should not be stored but generated each time.

3.2.3 Selective Optimization

We next present selective optimization annotations that determines if a function should be optimized or not, or selects among the existing optimizing compilers available on a system. 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 can potentially reduce compilation overhead since methods that are fast-compiled are more prevalent. It also enables methods that are most frequently executed (and hence, important for the overall execution performance) to be optimized.

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 graph shows the total time (execution plus compilation). For each benchmark, each bar (within a set of nine) indicates the total time given optimization of some percentage of the most frequently executed methods. For example the 0% bar (leftmost in each set) shows the total time when 0% of the methods are O3-compiled (optimized) and 100% of the methods are O1-compiled (fast). The 100% bar (right-most of each set) shows the total time when 100% of the methods are

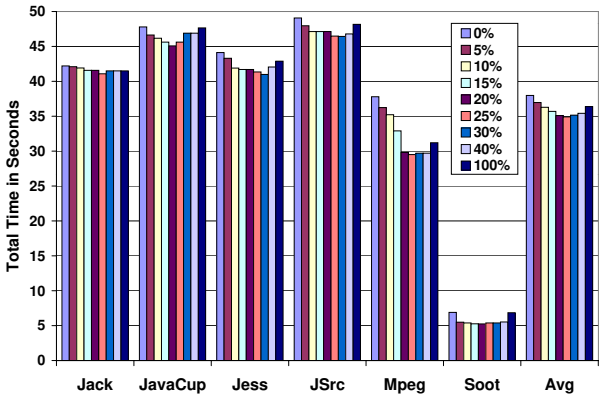


Figure 2: The histogram used to determine the percent of methods most important to optimize. Each bar shows the total time (compilation plus execution) for the program when different percentages of the most frequently executed methods are optimized. The remaining methods are compiled with the O1 compiler.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

O3-compiled (optimized) and 0% of the methods are O1-compiled (fast). The remaining bars represent the total time for various percentages between 0 and 100.

When 0% of the of the methods are O3-compiled (all of the methods are O1-compiled), the total time (compilation plus execution) is dominated by the execution time. O1-compilation time is very small (0.3 seconds for the entire application on average) but since no optimizations are performed, execution time is slow (38 seconds on average). At the far right of the spectrum (right-hand bars of each set) in which all methods are O3-compiled, compilation time is very high (2.6 seconds on average) and optimization enables improved execution time (33.8 seconds on average). We generated this histogram to discover the balance between these two extremes; the point at which both execution and compilation time are at their minimum.

The figure shows that the top 25% of frequently executed methods should be optimized to achieve the minimum compilation time as well as execution time. We used profile data to determine which methods were contained in the top 25% in terms of invocation frequency. The annotation for selective optimization is similar to that for the reuse optimization: it contains a one-byte method id and sets a single bit for each method in the top 25% most frequently executed methods. The bit indicates to the compiler that the method should be optimized.

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 perform a subset of the optimizations when static or profile-based analysis of the method indicates that it is profitable to do so.

We construct an annotation for each method that consists of a two-byte method identifier and a 1-byte bit mask as part of the annotation data for each method that maps to a list of expensive optimizations. For example, the first bit might map to inlining, 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 (const-prop). In Figure 1, the ORP O3 compiler spends a substantial amount of time performing constant propagation. For some methods, the reduction in runtime performance after applying this optimization is not enough to warrant the use of the optimization. For those methods, we use the bit mask of this annotation to indicate that the copy propagation optimization should be excluded.

We gathered total times (execution plus compilation) for the benchmarks for which constant propagation was used on various percentages of the most frequently executed methods. We created a histogram of these values like we did for the selective optimization annotation and discovered that 70% of the methods benefit from using constant propagation on average. For all other methods, it is not cost effective in terms of execution time to warrant using constant propagation. For each of these latter methods, we include an optimization filtering annotation that has the constant propagation bit set. This indicates to the compilation system that constant propagation should be bypassed during optimization of the method.

3.3 Annotation Security

An important issue that must be addressed by annotation-based systems for mobile-computing environments is security. Annotations must be verified or guaranteed that their use will not corrupt the JVM or machine on which they are used. Use of most existing bytecode annotation systems poses serious security risks since the annotations implemented using these systems affect program semantics and no verification mechanism is provided. If the bytecode stream is intercepted and modified by an untrusted party, illegal and possibly destructive program behavior can result.

The annotations we present here with the exception of the flow-graph generation optimization, if modified with harmful intent, can only effect program performance. As part of the empirical evaluation of our techniques, we measure the effect of the optimizations in a mobile environment for which only remote (non-library) classes are annotated. We do not include the flow-graph optimization in that part of the study (Section 4.2) to guarantee that untrusted execution using our annotations are safe. Benefits from our secure annotations are two-fold: their modification does not effect program semantics and they do not require the additional runtime overhead of verification. The latter is significant for our work, since our goal is to eliminate (not introduce) as much runtime overhead as possible.

4 Results

The results we present were gathered by executing the applications 100 times on a dedicated 300Mhz x86 Intel Pentium II machine running Linux v2.2.15 and taking the average. The applications we examine, described in Table 2, are part of the SPEC Java benchmark suite and additional applications commonly used in Java studies [16, 7, 22]. 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 gather execution profiles using two different inputs, called Input1 and Input2. Table 3 shows some differences in the execution characteristics between the two inputs. The first two columns show data for class file counts, the last two columns for method counts. Above the slash in each column of data is the number of classes (and methods) used during execution of the application using the associated input; below the slash we show the total static class (and method) counts. Input1 is used to generate all of the results in this section as well as the compilation time statistics shown in Table 1.

We generate profiles and guide our optimizations using both inputs. We denote results that use the profile generated using Input1 by "Same". Using the same data set for both profile and result generation provides ideal performance since we have perfect information about the execution characteristics of the programs. Results denoted by "Diff" are those that use the profile generated using the Input2 to guide optimization. Diff, or cross-input, results indicate realistic performance since the characteristics used to perform the optimization can differ across inputs and the input that will be used is not commonly known ahead of time. As mentioned above, results are executed using Input1 regardless of the input used for profile creation.

The overhead associated with annotations consists of class file size increase and the execution overhead needed to process and make use of the annotations. We detail the former in Section 4.3. Any execution overhead imposed by

Table 2: Description of Benchmarks Used.

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 3: General characteristics of execution using two different inputs.

	Executed/Total Classes		Executed/Total Methods	
Program	Input1	Input2	Input1	Input2
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

annotations is included in the overall results. 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 time reduced is shown in Figure 3. Each bar shows the reduction due to each of the individual optimizations. The two, right-most bars of each group is the number of seconds reduced using all of the annotations we examine in this paper. *Const_prop, Inlining, and top25* use profile information to guide the optimization. We therefore show two bars of each of these optimizations (as well as for the combined results). The first bar of each pair (Diff) shows the cross-input results (different inputs were used to generate the profile and the results). The second bar of each pair (Same) shows the effect of using the same input for profile and result generation. On average, our annotation optimizations reduce compilation overhead by 1.9 seconds using imperfect information and over 2 seconds using perfect information (this equates to a 78.1% reduction for cross-input and a 79% reduction for same-input results).

Figure 4 shows the total compilation time required for optimized compilation (O3), annotated compilation (Annot), and unoptimized compilation (O1) for same-input and cross-

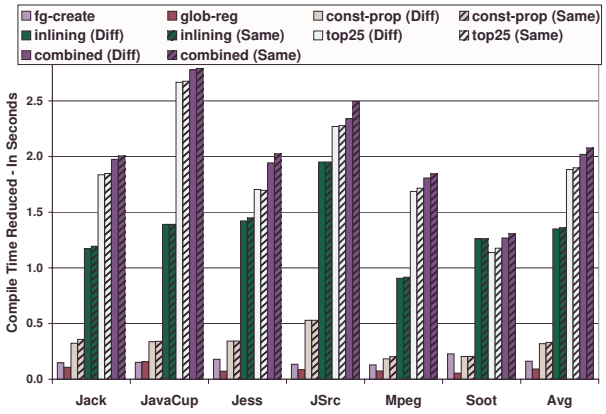


Figure 3: The graph shows the number of seconds of compilation overhead reduced due to the annotation optimizations we present. They are broken down by optimization. The two, right-most bars of each group is the number of seconds reduced using all of the annotations we examine in this paper. *Const_prop, Inlining, and top25* use profile information to guide the optimization. We therefore show two bars of each of these optimizations: the first bar of each pair shows the cross-input (Diff) results and the second shows the same-input results (Same).

input configurations. Annot results show the combined effect from all of the annotation optimizations we describe in this paper. These results are the same as shown in Figure 3, however, they are in terms of resulting compile time instead of the number of seconds reduced and include O1-compilation for comparison. On average, annotated compilation time is approximately 250 milliseconds slower than O1 compile time and 2 seconds faster than O3 compile time.

The results in Figure 3 show that the majority of compile time reduction is due to two optimizations, inlining and top25%. This indicates that using these two optimizations alone is enough to achieve substantial performance benefit. As such, we also have collected results for the combination of these two optimizations alone. This data is shown by the *Annot-Inlining_top25* results in the following graphs. We next present the effect of our annotation optimizations on total time: compilation and execution time combined.

Figure 5 shows the speedup achieved through the use of annotation over O3 total time for both of our annotation schemes, Annot and Annot-Inlining_top25. The former are the results for all of the annotations and the latter are when only the inlining and selective optimization annotations are used. Results are shown for both input configurations (Same and Diff). The cross-input (Diff) results are very similar to having perfect information (Same results). The overall effect on total time is more dramatic the shorter the execution time, as expected. Our annotation optimizations achieve

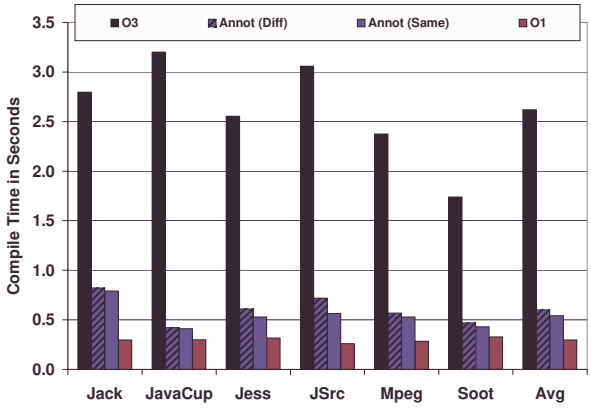


Figure 4: Total compilation overhead for O3 and O1 compilation, and combined annotated compilation. All of the annotation-guided optimizations presented in this paper are included in the latter denoted by *Annot*.

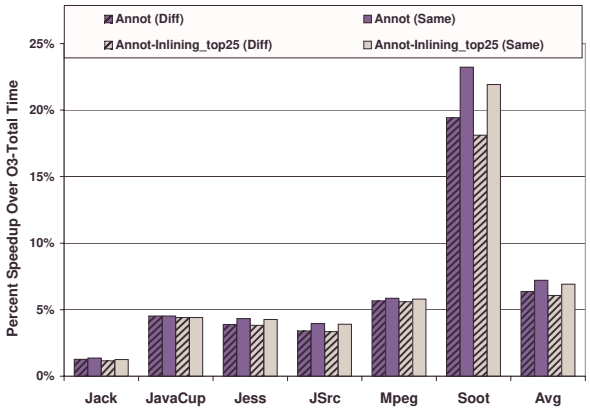


Figure 5: Speedup over optimized (ORP O3) total time due to annotated execution. Annot-Inlining_top25 denotes results that use the inlining and top 25% annotations alone to guide dynamic compilation.

2% to 23% speedup over O3 total time for the programs studied. However, a more important result from annotated-compilation is in its effect on startup time.

4.1 The Effect on Startup Time

A significant contribution of this work is the reduction in startup time that is achieved. Startup time is arguably more important to an end user over a few percent speedup in execution time. Our studies revealed that almost all compilation in Java programs occurs at startup: In the programs studied, 77% 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 6 confirms this with graphs of the cumulative distribution of compile time over program lifetime (in seconds on the y-axis). We show the cumulative time in seconds on the x-axis as opposed to the percentages commonly shown for a cumulative distribution function. The compilation overhead in seconds that has occurred since the start of the program's execution is expressed in this figure. The overhead for the O3 compiler is shown by the top (dark) line on each graph.

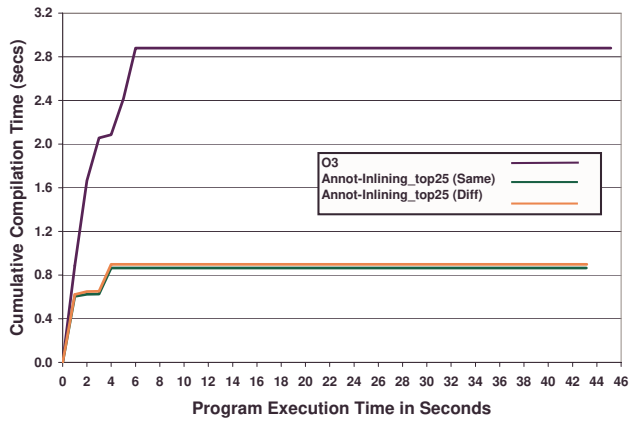
The bottom two lines in each graph indicate the effect of our two best-performing annotation optimizations (inlining and top25%). The results show that startup overhead is substantially reduced for every program. For example, for Mpeg, all compilation completes in 3 seconds. Using annotated-execution this point is reached in approximately one second. 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, startup compilation completes more than 2 seconds earlier using annotation.

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.

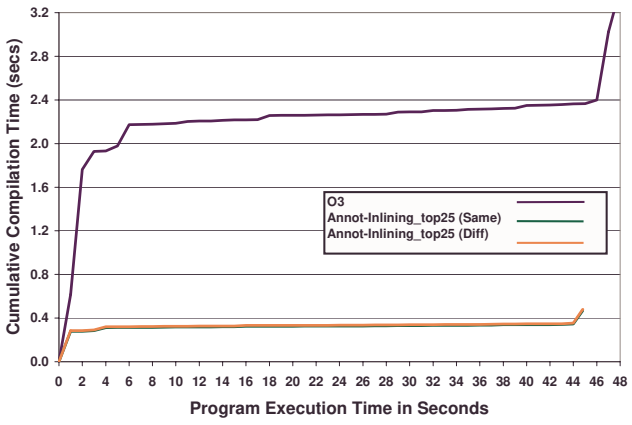
4.2 Local v/s Remote Execution

Mobile Java programs are commonly transferred 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 transferred 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. Most of the prior work [4, 13] on bytecode annotations does not address the effect of limiting optimization to remote class files yet it is vital to the empirical evaluation of annotation-based techniques.

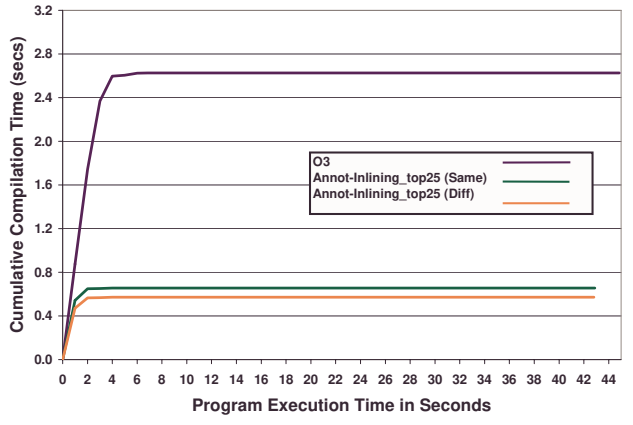
Figure 7 shows the reduction in compile time due to annotated-execution of only non-local class files. Class files that are non-local (or remote) include non-library files used during execution of each program. Local (library) files are compiled using the O1 compiler (no-optimization). Realisti-



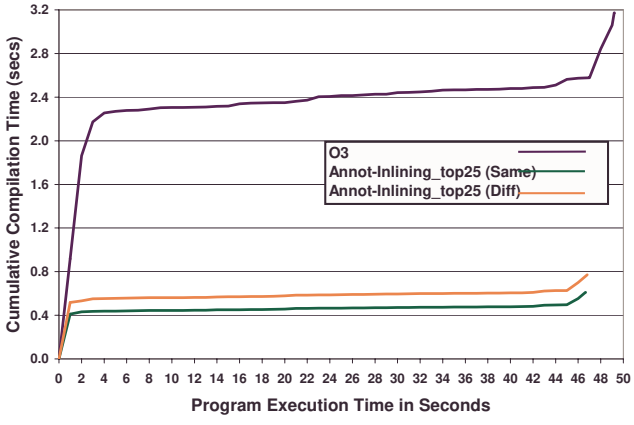
Jack



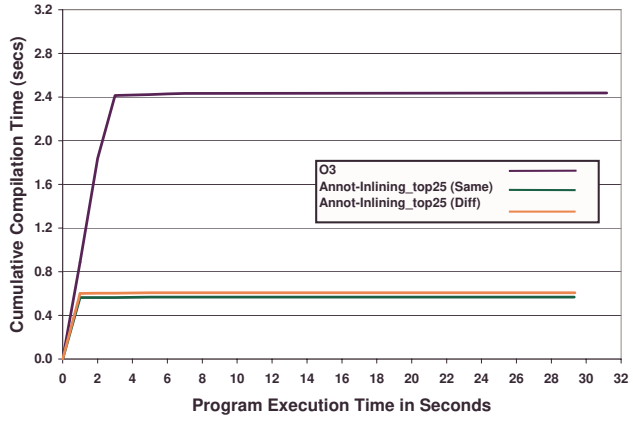
JavaCup



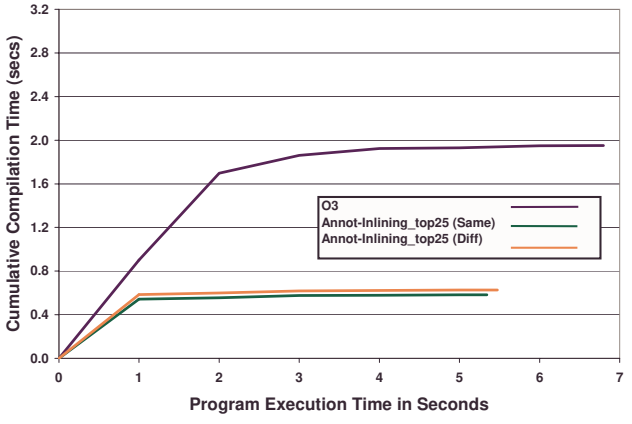
Jess



JSrc



Mpeg



Soot

Figure 6: 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.

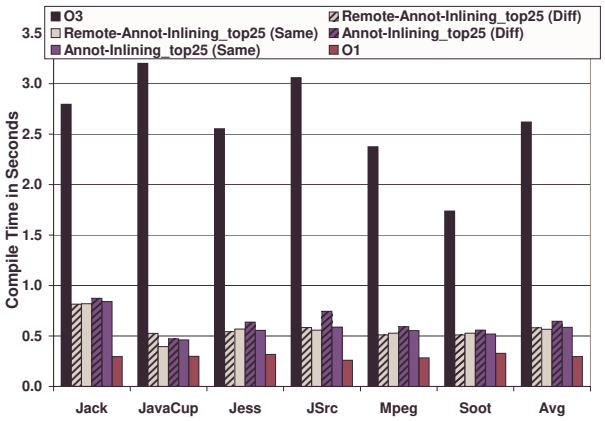


Figure 7: Total compilation overhead for O3, O1, and annotated compilation (both all class files and remote class files only). Annot-Inlining_top25 denotes results that use the inlining and top 25% annotations alone to guide dynamic compilation. "Remote" indicates that only non-local class files use annotation-guided optimization. For this configuration, all other class files are compiled using the O1 compiler.

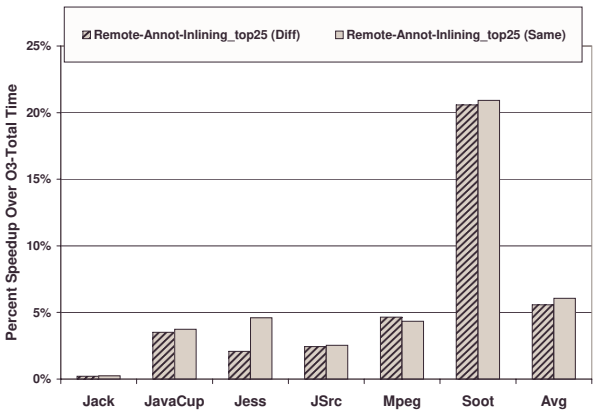


Figure 8: Speedup over optimized (ORP O3) total time due to annotated execution on remote classes only. For these results, only the inlining and top25% annotations are used.

cally, library files should be optimized but the overhead associated with their compilation should not degrade startup or cause intermittent interruption. Because of this we collected results for O1-compilation of these files. Some execution environments store optimized versions of library files on disk and dynamically load them [2]. As part of future work we will incorporate such functionality into ORP.

The graph in this figure is the same as the one presented earlier in Figure 4 with two additional bars: one for cross-input remote-only compile time (second bar) and one for same-input remote-only (third bar) results. These results are achieved by using only the two best-performing annotations inlining and selective optimization (top25%). The two input configurations included (Same and Diff) again indicate that imperfect information has little effect on the overall performance of these annotations. Over 80% of the compilation delay is reduced by using annotation-guided optimization for non-local class files and O1-compiling all others. Figure 8 shows the percent speedup over O3-compilation due to annotations on non-local class files. 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.

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

4.3 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 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. By only incorporating the two best performing 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 [12, 13, 21]

5 Related Work

In this section, we discuss work related to our Java annotation research. We first describe related work on dual-compiler systems, then describe recent annotation research, and conclude with relevant research for the reduction of startup delay.

Table 4: The added size in kilobytes due to each annotation optimizations. Columns 2 through 5 contain the added size from each the flow graph creation, register allocation, constant propagation, reuse optimization for inlining, and selective optimization using the top 25% of frequently executed methods, respectively. Remote class file annotation alone is shown first in each column. In parenthesis, we show the annotation size across all class files in an application. The results show the percent increase in application size due to the annotation (in that column) on average across all benchmarks.

	Size of Annotation in KBytes for Non-Local Classes - Total for All Classes in Parens.				
Program	FG-Create	Regalloc	Const-prop	Inlining	SelOpt
Jack	12.55 (16.41)	0.74 (1.25)	0.28 (0.50)	0.04 (0.06)	0.04 (0.06)
JavaCup	7.02 (13.61)	0.47 (1.37)	0.20 (0.54)	0.03 (0.07)	0.03 (0.07)
Jess	9.74 (14.62)	1.02 (1.61)	0.39 (0.64)	0.05 (0.08)	0.05 (0.08)
JSrc	12.29 (15.39)	1.22 (1.68)	0.41 (0.62)	0.05 (0.08)	0.05 (0.08)
Mpeg	5.15 (8.35)	0.71 (1.17)	0.22 (0.40)	0.03 (0.05)	0.03 (0.05)
Soot	6.44 (9.66)	0.60 (1.09)	0.33 (0.54)	0.04 (0.07)	0.04 (0.07)
Avg	8.87 (13.01)	0.79 (1.36)	0.28 (0.54)	0.04 (0.07)	0.04 (0.07)
Incr	3.6% (5.2%)	0.3% (0.5%)	0.1% (0.3%)	0.0% (0.0%)	0.0% (0.0%)

5.1 Fast v/s Optimizing Compiler

Existing Java systems use two compilers in an adaptive compilation scheme [3, 7, 11] in which methods are first compiled with a very fast compiler that converts bytecode into instrumented executable code. Instrumentation enables on-line measurements to be made to discover hot methods. Hot methods are those that are frequently executed [7, 11] 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 re-compilation.

In [17], we proposed using a technique similar to 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 it provided 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.

5.2 Annotation-based Dynamic Optimization

Pominville et al [21] 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. 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 include in their study. They do not provide an 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 milliseconds for the entire application.

Hummel et al [12], 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 do not provide an annotation implementation or mechanism for reducing the transfer delay imposed by their 33% to 97% increase in class file size. In a project similar to this prior work, Jones et al in [13] describes 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 do not provide a mechanism for annotation size reduction and increase class file size 31% to 38% for the programs presented. Lastly, no general framework or format for their annotations is provided.

These related projects do not consider the use of annotations to reduce optimization overhead as we have done in this paper. The goal of each of the prior work is to enable an expensive optimization to be performed, which is a side-effect of our framework. We plan to evaluate the effect of expensive optimizations that are now feasible in a dynamic compilation setting through annotation. In doing this, we will consider the transfer delay trade-off associated with such optimizations to ensure that total time (transfer, compilation, and execution) is not degraded. As mentioned above, each of these related works increase transfer delay substantially which can negate the impact of their execution time improvements. One reason for this is the use of multiple annotations per class file. This increases the size of the constant pool substantially. In our work, we use a single attribute for all annotations and a single Unicode character name to indicate that the attribute is an annotation to keep constant pool expansion to a minimum.

Another issue the must be addressed by any annotation implementation is security. For annotations to be trusted, they must be verified or implemented so as to guarantee safety of the JVM or machine on which annotated execution is performed. The annotation-guided optimizations presented in [21, 4] are unsafe since the annotations contain information that effect the semantics of the program and no verification is performed at the destination. For example, in [21], the authors present an annotation for array bounds check elimination. If this annotation in intercepted and modified by an untrusted party, a boundary check might be eliminated and cause illegal memory access. Likewise, in [4], the authors implement register allocation, and annotation manipulation can cause program behavior that can potentially harm the JVM in which it is executing as well as the underlying machine. For such annotations to be trusted, some mechanism for verification must be implemented. The annotations we present for remote execution (Section 4.2) are safe without requiring verification since their modification only affects program performance not semantic behavior.

5.3 Reducing Startup Delay

In Section 4.1, we showed that annotation-guided compilation substantially reduces startup time in mobile programs. Related work which reduces startup delay focuses on reducing the effect of transfer delay on such programs. Our annotation optimizations are complementary to these techniques and when combined with them can further reduce the startup time of mobile programs.

Existing transfer delay techniques that improve startup performance commonly restructure mobile programs so that only those methods that will be executed are transferred across the network. One such technique is non-strict execution [16], in which the transfer and execution model of Java is modified to allow execution to begin much earlier. The changes to the JVM required include method-level transfer and execution; when the code and data necessary for execution has arrived at the destination execution can begin. Methods are reordered across class files by profile-guided techniques to enable their arrival at the destination in the order they are predicted to execute.

Another form of program restructuring to improve startup delay is class file splitting as described in [15], and similarly in [23]. In these works, Java class files are repartitioned to enable more effective utilization of the available bandwidth during transfer without requiring modification to the JVM. Profile information is used to identify methods that are unused during instrumented execution. Unused methods are then split out into new class files. Using existing Java class file loading techniques, the class files containing the methods that are used during execution are transferred. If methods predicted as unused and split out are not used during actual execution, the methods are never transferred and the transfer delay is reduced. If such methods are used, then the class files containing them are transferred.

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 and represents an instruction set with opcode, size, and annotation data, and requires only one bytecode attribute for multiple annotations. The framework enables incorporation of highly-compressed, static and profile-based information into the Java bytecode stream for use in dynamic optimization. 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 annotation optimizations we present perform analysis off-line and communicate it 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.

A significant contribution of this work is the substantial reduction in startup time that our optimizations enable,

while at the same time reducing the overall execution time of the program. Startup delay negatively effects the user's perception of program performance as well as the raw performance of mobile Java programs. In the programs studied, 77% of of the compilation overhead occurs in the first 4 seconds (initial 10%) of program execution on average. Using our annotation-guided optimizations startup delay is reduced by more than two seconds on average, enabling substantial improvement in the initial progress made by program execution.

In summary, 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. 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.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF CAREER grant No. CCR-9733278, and a grant from Intel Corporation.

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.

[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] W. Doherty and R. Kelisky. Managing VM/CMS systems for user effectiveness. *IBM Systems Journal*, pages 143–163, 1979.

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

[10] 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.

[11] The Java Hotspot performance engine architecture.

[12] 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.

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

[14] Kaffe – An opensource Java virtual machine.

[15] 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.

[16] 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.

[17] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the Overhead of Dynamic Compilation. *Software—Practice and Experience*, 31(8):717–738, 2001.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] A. Srivastava. From communication on reducing startup delay in Microsoft Applications.

[25] 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.