USENIX Association

Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium (JVM '02)

San Francisco, California, USA
August 1-2, 2002

An Empirical Study of Method Inlining for a Java Just-In-Time Compiler

Toshio Suganuma Toshiaki Yasue Toshio Nakatani

IBM Tokyo Research Laboratory
1623-14 Shimoturuma, Yamato-shi, 242-8502 Japan
{suganuma, yasue, nakatani}@jp.ibm.com

ABSTRACT

Method inlining is one of the optimizations that have a significant impact on both system performance and total compilation overhead in a dynamic compilation system. This paper describes an empirical study of online-profile-directed method inlining for obtaining both performance benefits and compilation time reductions in our dynamic compilation system. We rely solely on the profile information in deciding which methods should be inlined along which call paths, without employing the existing static heuristics, except that methods with extremely small bodies are always inlined. The call site distribution and invocation frequency are collected using dynamically generated instrumentation code, and are given to the recompilation controller for analyzing and organizing the inlining requests. The experimental results show that the strategy of relying on the profile information in method inlining decision is quite effective for reducing compilation overhead and/or for improving the performance. It can also provide flexibility for inlining decisions based on the type of profile, such as spiky or flat profiles. The results show that the strategy can be the basis of automatic optimization selection for a future efficient dynamic compilation framework.

1. INTRODUCTION

Dynamic compilation systems need to reconcile the conflicting requirements between fast compilation speed and fast execution performance. We would like the system to generate highly efficient code for good performance, but the system needs to be lightweight enough to avoid any startup delays or intermittent execution pauses that may occur due to the runtime overhead of the dynamic compilation.

The high performance implementation of Java Virtual Machines (JVM) is moving toward exploitation of adaptive compilation optimizations on the basis of online runtime profile information [1][9][20][23]. Typically these systems have multiple execution modes. They begin the program execution using the interpreter or baseline compiler as the first execution mode. When the program's frequently executed methods or critical hot spots are detected, they use the optimizing compiler for those selected methods to run in the next execution mode, obtaining better performance. Some systems employ several different optimization levels to select from based on the invocation frequency or relative importance of the target methods.

In these systems, the set of optimizations provided for each optimization level is predetermined. Basically those optimizations considered to be lightweight, such as those with linear order to the size of the target code, are applied in the earlier optimization levels, and those with higher costs in compilation time or greater code size expansion are delayed to the later optimization levels. However, the classification of optimizations into several different levels has been either intuitive work or just based on the measurements of compilation cost and performance benefit on a typical execution scenario analyzed offline [2]. Consequently, the optimizations applied are not necessarily effective for the actual target methods compiled with the given optimization level. That is, some optimizations may not be able to contribute to any useful transformation for performance improvement, but can result in a waste of compilation resources.

The problem here is that we equally apply the same set of optimizations for those methods selected to compile at that optimization level, regardless of the type and characteristics of each method. Ideally it is desirable to dynamically assemble a set of suitable optimizations depending on the characteristics of the target methods so that we can apply only those optimizations known to be effective for the method. It would be better for the total cost and benefit management, if we could not only selectively apply optimizations on performance critical methods, but also selectively assemble the optimizations that are effective for those methods.

Method inlining, a well-known and very important technique in optimizing compilers, expands the target procedure body at the method invocation call sites, and it defines the scope of the compilation boundary. Toward the big picture of the efficient compilation framework, method inlining can naturally be the first target for a dynamic compilation strategy by exploiting the profile information collected at runtime, since it is one of the op-

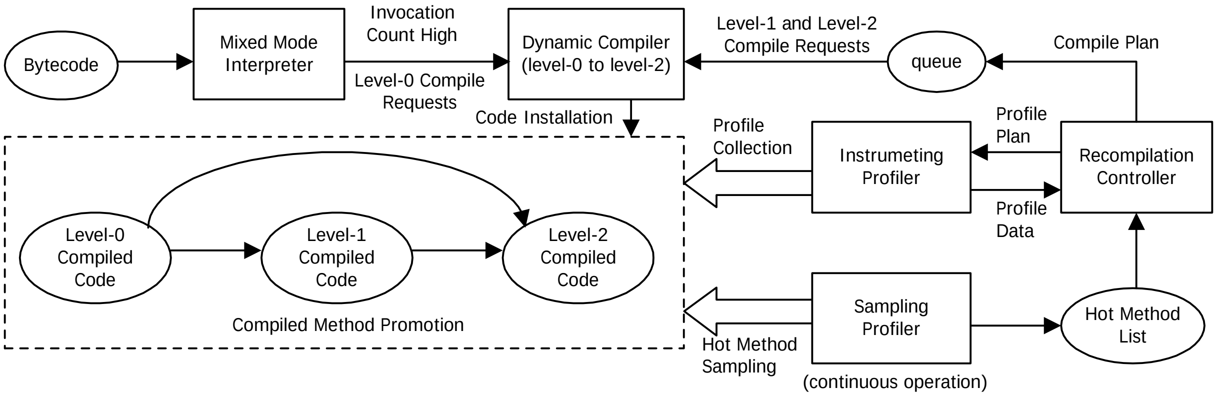

Figure 1. System architecture of our dynamic compilation system.

timizations that have a significant impact on both performance and compilation overhead.

In this paper, we describe an empirical study of an online-profile-directed method inlining strategy in our dynamic compilation system. We rely solely on the profile information for deciding which methods should be inlined along which call paths. Though many static heuristics exist, the only one we use is to always inline very small methods. The ultimate goal of our research is to explore the opportunities for dynamic automatic optimization selections, and the first step is to evaluate the potential benefits and limitations of this approach for method inlining.

The following are the contributions of this paper:

- **Online collection of call site information:** We present a mechanism for dynamic instrumentation, which collects the call sites distribution and invocation frequency for hot methods in order to help make decisions on method inlining during recompilation. This is quite different from the existing approach of constructing a dynamic call graph through sampling during the entire period of program execution.
- **Detailed evaluation of profile-directed method inlining policies:** We present detailed evaluation results of the benefits and limitations regarding both performance and compilation overhead for several inlining policies: two variations of profile-directed method inlining, existing static inlining heuristics, and a hybrid of both. We also evaluated inlining based on offline-collected profile information to know the upper bound of the potential of the profile-directed method inlining.

The rest of this paper is organized as follows. The next section is an overview of our dynamic compilation system, and describes the existing inlining policy and potential impacts on both performance and compilation time. Section 3 describes our design and implementation of online-profile-directed method inlining using the instrumentation-based information about call site. Section 4 presents the experimental results on both performance and compilation overhead by comparing several inlining policies. Section 5 discusses the results and possible future research towards automatic optimization selection. Section 6 summarizes the related work, and finally Section 7 presents our conclusions.

2. BACKGROUND

The overall system architecture of our dynamic optimization framework is described in detail in [23]. In this section, we briefly describe the system characteristics and features that are closely related to this study, and then we describe the existing static-heuristics-based inlining policy and its impact on both compilation time and performance.

2.1 System Overview

Figure 1 depicts the overall architecture of our system. This is a multi-level compilation system, with a mixed mode interpreter (MMI) and three compilation levels (level-0 to level-2). Initially all methods are interpreted by the MMI. A counter for accumulating both method invocation frequencies and loop iterations is provided for each method and initialized with a threshold value. The counter is decremented whenever the method is invoked or loops within the method are iterated. When the counter reaches zero, the method is considered as frequently invoked or computation intensive, and the first compilation is triggered.

The dynamic compiler has a variety of optimization capabilities. The level-0 compilation employs only a very limited set of optimizations for minimizing the compilation overhead. For example, it considers method inlining only for extremely small target methods as described in Section 2.2. It disables most of the dataflow optimizations except for very basic copy and constant propagation. The level-1 compilation enhances level-0 by employing additional optimizations, including more aggressive full-fledged method inlining, a wider set of dataflow optimizations, and an additional pass for code generation. The level-2 compilation is augmented with all of the remaining optimizations available in our system, such as escape analysis and stack object allocation, code scheduling, and DAG-based optimizations[1].

The level-0 compilation is invoked from the MMI and is executed as an application thread, while level-1 and level-2 compilations are performed by a separate compilation thread in the background. The upgrade recompilation from level-0 compiled code to higher-level optimized code is triggered on the basis of the hotness level of the compiled method as detected by a timer-based sampling profiler [24]. Depending on the relative hotness level, the method can be promoted from level-0 compiled code to either level-1 or directly to level-2 optimized code. This decision is made based on different thresholds on the hotness level for each level-1 and level-2 method promotion.

The sampling profiler periodically monitors the program counters of application threads, and keeps track of methods in threads that are using the most CPU time by incrementing a *hotness count* associated with each method. The profiler keeps the current hot methods in a linked list, sorted by the hotness count, and then groups them together and gives to the recompilation controller at every fixed interval for upgrade recompilation. The sampling profiler operates continuously during the entire period of program execution to adapt effectively to the changes of the program's behavior.

There is another profiler, an instrumenting profiler, used when detailed information needs to be collected from the target method. The instrumenting profiler, when invoked, dynamically generates code for collecting specified data from the target method, and installs it into the compiled code by rewriting the entry instruction of the target. After collecting a predetermined amount of data, the generated instrumentation code automatically uninstalls itself from the target code in order to minimize the performance impact. Thus the compiler can take advantage of the online profile information dynamically collected at runtime for the level-1 and level-2 compilations. This instrumentation mechanism is used for collecting call site distributions and execution frequencies to guide method inlining in this study.

This dynamic instrumentation mechanism allows the intensive monitoring of the runtime application behavior for a certain interval of the program's execution. By combining this technique with the sampling profiler, we can benefit from two nice properties of online profiling: low overhead and reliable profiling results. It is low overhead because we can limit the targets of the instrumentation to only those hot methods we are interested in for recompilation. It is reliable because we can obtain representative profiles for real applications by delaying the instrumentation installation until the program hot regions are discovered.

2.2 Static Heuristics Based Inlining

Our dynamic compiler is able to inline any methods regardless of the context of the call sites and the types of caller or callee methods. For example, there is no restriction in inlining synchronized methods or methods with try-catch blocks, nor against inlining methods into call sites within a synchronized method or a synchronized block. Thus the inlining decision can be made purely from the cost-benefit estimate for the method being compiled.

There is a class of methods in Java that simply result in a single instruction of code when compiled, such as the one just returning an object field value. Other methods may turn out to be very small number of instructions for their procedure body after compilation. We call these *tiny methods*. Our implementation identifies them based on the estimated compiled code size[2].

Tiny method**:** This is a method whose estimated compiled code is equal or less than the corresponding call site code sequence (argument setting, volatile registers saving, and the call itself). That is, the entire body of the method is expected to fit into the space required for the method invocation.

Since invocation and frame allocation costs outweigh the execution costs of the method bodies for these methods, and since inlining them is considered completely beneficial without causing any harmful effects in either compilation time or code size expansion, they are always inlined at all compilation levels. Otherwise we employ static heuristics to perform method inlining in an aggressive way while keeping the code expansion within a reasonable size.

The inliner first builds a possibly large tree of inlined scopes with allowable sizes and depths using optimistic assumptions, and then looks at the total cost by checking each individual decision to come up with a pruned final tree. When looking at the total cost, the inliner manages two separate budgets proportional to the original size of the method: one for tiny methods (and profile-directed methods, if any), and the other for any type of method. It tries to greedily incorporate as many methods as possible in the given tree using the static heuristics until the predetermined budget is used up. Currently the static heuristics consist of the following rules.

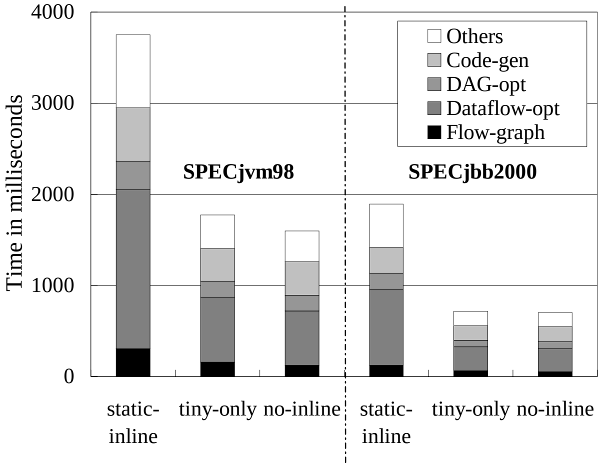

Figure 2. Compile time breakdown for SPECjvm98 and SPECjbb2000 benchmarks with three different inlining policies: static-inline, tiny-only, and no-inline.

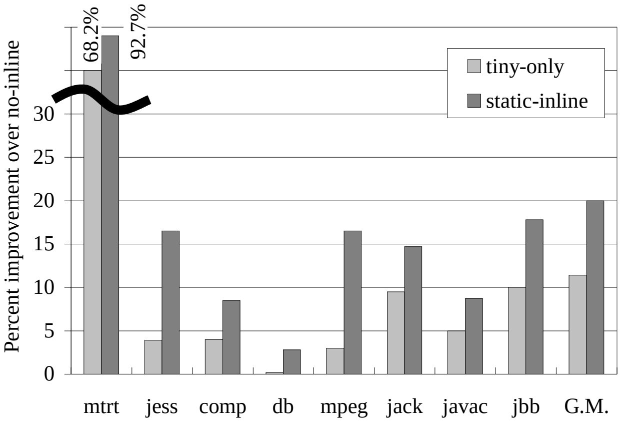

Figure 3. Performance difference for SPECjvm98 and SPECjbb2000 benchmarks with static-inline and tiny-only inlining policies over the no-inline policy.

- If the total number of local variables and stack variables for the method being compiled (both caller and callees) exceeds a threshold, reject the method for inlining.
- If the total estimated size of the compiled code for the methods being compiled (both caller and callees) exceeds a threshold, reject the method for inlining.
- If the estimated size of the compiled code for the target method being inlined (callee only) exceeds a threshold, reject the method for inlining. This is to prevent wasting the total inlining budget due to a single excessively large method.
- If the call site is within a basic block which has not yet been executed at the time of the compilation, then it is considered a cold block of the method and the inlining is not performed. On the other hand, if the call site is within a loop, it is considered a hot block, and the inlining is tried for deeper nest of call chains than for outside a loop.

The devirtualization of dynamically dispatched call sites is done at all compilation levels based on the class hierarchy analysis (CHA) and type flow analysis, and it produces either guarded code (via class tests or method tests) or unguarded code (via code patching on invalidation) [15]. Preexistence analysis is also performed to safely remove guard code and back-up paths without requiring on-stack replacement [11]. The resulting devirtualized call sites then may or may not be inlined according to the static rules described above.

Only tiny methods are inlined in level-0 compilation. In level-1 and level-2 compilation, the broader range of methods within the conditions of the above static heuristics are considered for inlining. There is no difference between level-1 and level-2 compilation in terms of the scope and aggressiveness of inlining.

As a result of method inlining, a data structure called an inlined method frame (IMF) is produced for each call site to provide information about the inlining context for the runtime system, which needs to know the exact call stack and call context prior to inlining. For example, the security manager requires the correct depth of the current call-chain on the stack, or a runtime exception is raised. The exception handler also requires the inlining context of call sites in order to keep track of the handlers for catching exceptions from the current context.

Figure 2 shows the compilation overhead for three different policies of method inlining when running two industry standard benchmarks. *Static-inline* indicates using the existing static heuristics for inlining in level-1 and level-2 compilation, *tiny-only* means inlining only tiny methods in level-1 and level-2 as well as level-0 compilation, and *no-inline* performs no method inlining at any level of compilation regardless of the size of the target methods. Note that devirtualization is still enabled for all three cases. The data was collected based on the system configuration described in Section 2.1 and using the methodology described in Section 4.1.

Each bar gives a breakdown of where in the optimization phase time is spent for compilation. *Flow-graph* indicates the time for basic block generation, inlining analysis to determine the scope of the compilation boundary, and flow graph construction for the expanded inlined code. *Dataflow-opt* is the time for a variety of dataflow-based optimizations, such as constant and copy propagation, redundant null-pointer and array-bound check elimination. *DAG-opt* is for DAG-based loop optimization and pre-pass code scheduling. *Code-gen* includes the time for register assignment, code generation, and code scheduling. *Others* denotes memory management cost and any other code transformations, including live analysis and peephole optimizations, each costing less than 5% of the total compilation time. Figure 3 shows the performance differences between the same set of three inlining policies.

As we can see in these figures, method inlining has significant impact on both compilation time and performance. With the static-inline policy, the time spent for inlining analysis and flow graph construction itself is not a big part of the total overhead, but optimizations in later phases, dataflow optimizations in particular, cause a major increase of the compilation time due to the largely expanded code that has to be traversed. The code size expansion and the work memory usage also have similar increases with this inlining policy, although the data is not shown here due to space limitations.

The tiny-only inlining policy, on the other hand, has very little impact on compilation time, as expected, while it produces significant performance improvements over the no-inline case. For some benchmarks, it provides a majority of the performance gain that can be obtained using the static-inline policy. On average, it contributes a little over half of the performance improvement compared to the static-inline policy. From these figures, our current policy of always inlining tiny methods is clearly justified.

3. PROFILE-DIRECTED INLINING

Method inlining can significantly impact both costs and benefits of compilation as shown in the previous section. We would like to apply this expensive but effective optimization selectively, rather than statically, based on dynamic program execution behavior, only for program points that are expected to provide the performance benefits. We basically remove any existing static heuristics for method inlining as described above, except for always inlining tiny and small sized methods, and totally depend on the online profile data. Section 3.1 describes how the profile data can be collected for extracting potential candidates for method inlining, and Section 3.2 provides the procedure for identifying exact call paths to make inlining requests before recompilation. Section 3.3 describes the problem of handling small methods, and Section 3.4 discusses how to make inlining decisions more appropriate for flat profile applications.

3.1 Collection of Call Site Information

As described in the previous section, we exploit the dynamic instrumentation mechanism for collecting information on the call site distribution and execution frequency at runtime. The instrumentation is basically applied for all hot methods that are detected by the sampling profiler and which are candidates for promoting to the next level of optimization. Since the instrumentation code in this case is to find the most beneficial candidates among the call sites where the current target method can be inlined, we need to ensure whether the current target is appropriate for inlining, or otherwise performing instrumentation is just overhead. Thus those methods that are apparently not inlinable into other methods (due to excessively large size, for example) are excluded as targets for instrumentation. Methods already compiled with the highest level of optimization are also excluded, since they are methods already considered for inlining but not inlined in the previous round of compilation.

As shown in Figure 4, the code generated by the instrumenting profiler is dynamically installed in the target code by replacing the entry code (after copying it into the instrumentation code region) with an unconditional jump instruction to direct control to the instrumentation code. At its first entry, the instrumentation code stores the current time stamp. It then examines the return address stored on the top of the current stack, and records it in a form of table with values and their corresponding frequencies. Because of the overhead, only the single-level history in the call stack is recorded. The maximum number of entries in the table is fixed. When the predetermined number of samples has been collected, the code again checks the current time and records how long it took to collect those samples by subtracting the start time. The code then uninstalls itself from the target code by restoring the original code at the entry point[3].

Thus two kinds of information can be collected for each method during an interval in the program's execution: the distribution of callers and the frequency of invocations. The time recorded for sample collection indicates call frequency for the method for that fixed number of instrumentation samples. Since the hotness-count provided by the sampling profiler can largely depend on the size of the target method (it is easier for larger methods to get more sampling ticks), small but very frequently called methods may not receive the attention they deserve. In that sense, the fine-grained frequency information can be useful in identifying hot call edges critical for inlining, and in compensating for the rather coarse-grained hotness count of the sampling profiler.

The instrumentation is successively applied to a group of recompilation target methods provided by the sampling profiler at every fixed sampling interval. After installation, the recompilation controller periodically monitors the completion of profile collections on all of these methods, and if completed, it proceeds to the inlin-

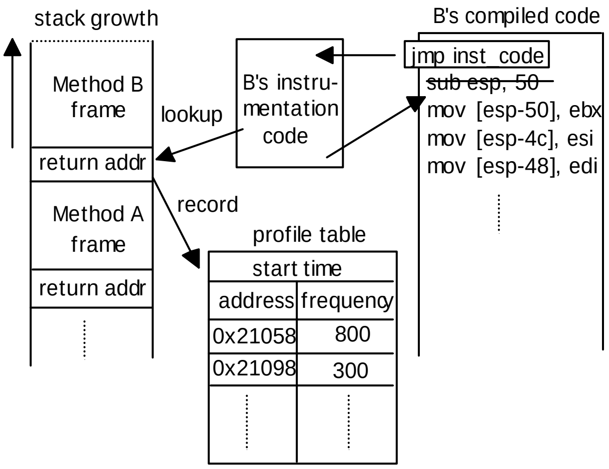

Figure 4. Dynamic instrumentation for collecting information on call site distribution and execution frequency (after code patching).

ing analysis step described in the next section. This is implemented by setting a maximum wait count for the recompilation controller to give up the thread execution. If necessary, it stops profiling for methods that are invoked very infrequently and which will never reach the predetermined number of samples by directly manipulating the current profile count for those methods.

3.2 Profile-Based Inlining Decision

The recompilation candidates grouped together and given to the recompilation controller are then examined to identify the hot call paths appropriate for inlining among them. Upon completion of collecting the call site information, the decision on requesting method inlining proceeds with the following steps.

1. **Examination of the call site distribution profile:** For each of the instrumented methods, the profiling result is examined in the first step. If there are just one or a few dominant call sites found in the profile information and the execution frequency is not too low, these are considered important candidates for inlining and thus a connection is created between the caller and the callee for each hot call edge. The edge is associated with a distribution ratio based on the count given to the corresponding call site address in the profile[4]. At the end of this step, a partial call graph is constructed, where nodes indicate recompilation candidate methods, each of which has a hotness count, and edges show dominant call connections from caller to callee, each assigned a distribution ratio (see Figure 5).
2. **Identifying exact call paths appropriate for inlining:** In the next step, we identify the exact call paths for inlining requests by traversing the connection between multiple methods. We begin with a method having no outgoing edge, and traverse the connections up to the top of the call chain. In the traversal of the connections, the following two operations need to be done:

   Call site address conversion: Since the raw data of the profile information indicates a set of compiled code addresses where the invocation was made to the current target method, it does not specify the exact call sites in the original form of the method bodies. The compiled code address is therefore converted to the offset of the corresponding bytecode instructions using the table generated at compile time.

   Hotness count adjustment: This is to correct the relative hotness level by assuming that inlining for the connected methods is performed. A portion of the hotness count of the callee (including values from its children) is distributed to each of its callers according to the ratio for each incoming edge. When dividing hotness counts from children into their multiple callers, the distribution ratio cannot be known precisely, because the profile information is not available for multiple-level histories of the call paths. Therefore, we use a *constant ratios assumption* [21] to approximate the actual ratio. That is, we assume the ratio is the same as the one recorded in the profile.
3. **Request for method inlining:** Finally, inlining requests are made for the hot call paths identified above. These requests are stored in a persistent database so that inlining decisions can be preserved for future recompilation to incorporate the previous inlining requests. This is important, as described in both the Self-93 [14] and Jalapeño [1] systems, to

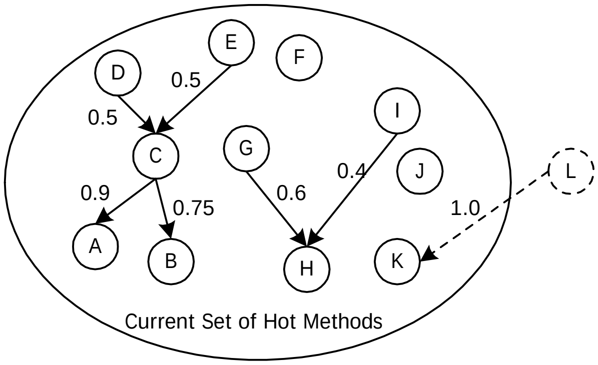

Figure 5. A simple graph is constructed for the profile-based inlining decisions. Nodes indicate methods to be recompiled, and edges show the dominant call edges from caller to callee. These dominant call edges are likely to be included within the hot call paths that should be considered for inlining requests.

avoid a performance perturbation resulting from oscillating between two compiled versions of a method, each with a different set of inlining decisions.

4. **Recompilation request:** After all the relationships between the hot methods in the given group are examined and appropriate inlining is requested, the list of hot methods is again traversed to issue the final recompilation requests. If a method in the list is requested to be inlined into another method, and there is no other call site for this method found in the profile information, then the method is removed from consideration for recompilation. Likewise, those methods whose hotness count is below a threshold value as a result of the hotness count adjustment are discarded. A method already compiled with the highest optimization level will be recompiled again only when a new inlining request has been made and its estimated impact is above a threshold value.

Since we consider the inlining possibilities among hot methods appearing in the group of hot methods, and since this group comes from sampling during an interval in the program execution, the resulting inlining requests are expected to contribute for a performance boost. In Step 2, however, there may be a call path where a caller is not included in the current group of hot methods, and it is possible to request inlining for such a case as well, as shown by the dashed line circle in Figure 5. This can be considered to be more aggressive inlining, and is evaluated in Section 4 as a variation of our profile-directed inlining. We do not trigger the recompilation for the caller in this case, since the method is not known to be worth upgrading to higher optimization at this time. We leave such a method for the future until it is be promoted by the sampling mechanism, at which time the inlining requests previously made for this method will all be incorporated in the recompilation.

3.3 Small Methods

Inlining with the above mechanism is only possible when the target method is identified as hot for dynamic instrumentation. Although tiny methods are always inlined, there are still some chances that other methods with very small bodies will be missed for selection as recompilation candidates due to the sampling nature of the profiler, and in that case it is unable to consider the inlining possibilities for those methods. To remedy this problem, small methods are also inlined, in addition to tiny methods, in level-1 and level-2 compilation under profile-directed inlining.

It is difficult to define small methods. In contrast with tiny methods, inlining them may cause some additional overhead for both compilation time and code size expansion. The timer interval of the sampling profile affects which small methods may be overlooked as inlining candidates. Our goal here is to spot as many of these overlooked candidates as possible, but also to minimize additional compilation overhead that can result from statically inlining these methods. In our current implementation, small methods are defined such that the estimated compiled code is less than the total invocation overhead occurred in both caller and callee (the method prologue and epilogue, in addition to the call site code sequence itself).

3.4 Flat Profile Applications

Benchmarks and applications can be roughly categorized into two different sets, each having distinct characteristics in their profiles: spiky or flat profiles. For the spiky profile applications, there are only a few very hot or scorchingly hot methods, and therefore fully optimizing those methods is sufficient to obtain the best possible performance. The remaining methods mostly have nothing to do with the total system performance. It is relatively easy for any type of profiling mechanism to detect those important methods, and thus this type of application is amenable in general for a dynamic optimization system.

On the other hand, the flat profile applications, such as jack and javac in SPECjvm98, are considered more difficult and challenging for the dynamic compilation system to improve performance. For example, method inlining is less effective for these applications involving a large number of methods that are almost equally important. Consequently, method inlining based on static heuristics, applying the same strategies for any type of method, does not work well for this type of application, as we can see from the relative performance difference between tiny-only and static-inline in Figure 3.

Profile-based inlining is expected to work more effectively than the static heuristics by dynamically changing the inlining decisions depending on the type of application. Since the cost/benefit trade-off line is considered to be higher for flat profile applications than for those with spiky profiles, due to the smaller impact on total performance from inlining individual call sites, method inlining needs to be performed even more selectively. In our implementation, we detect a flat profile for a given group of hot methods based on the average hotness count and the differences between maximum and minimum hotness counts among those hot methods not yet compiled with the highest optimization level. When making a inlining request, we require each edge of the call path to have either a single caller or a very high frequency with a few dominant call sites.

4. EXPERIMENTAL RESULTS

This section presents some experimental results showing the effectiveness and advantages of the profile-directed method inlining in our dynamic compilation system. We outline our experimental methodology first, describe the variations of inlining policies used for the evaluation, and then present and discuss our measurement results.

4.1 Benchmarking Methodology

All the performance results presented in this section were obtained on an IBM IntelliStation M Pro 6849 (Pentium4 2.0 GHz uni-processor with 512 MB memory), running Windows 2000 SP2, and using the JVM of the IBM Developer Kit for Windows, Java Technology Edition, version 1.3.1 prototype build. The benchmarks we chose for evaluating our system are SPECjvm98 and SPECjbb2000 [22]. SPECjvm98 was run in test mode with the default large input size, and in autorun mode with 10 executions for each test, with the initial and maximum heap size of 96 M. SPECjbb2000 was run in the fully compliant mode with 1 to 8 warehouses, with the initial and maximum heap size of 256 M.

The following parameters were used in these measurements. The threshold in the mixed mode interpreter to initiate level-0 compilation was set to 500. The timer interval for the sampling profiler for detecting hot methods was 5 milliseconds. The list of hot methods was examined every 200 sampling ticks. The number of samples to be collected for an instrumentation-based call site profile was set to 1024, and the maximum number of data variations recorded was 8. Exception-directed optimization (EDO) [19] was disabled, because EDO drives its own inlining requests based on the exception path profiles and the mixed inlining requests would make the evaluation for our approach more difficult.

Five inlining policies are compared as described below, one with the current static inlining heuristics, and the others using profile-directed inlining with different degrees of aggressiveness or combined with the static heuristics. The baseline of the comparison is with the policy inlining tiny methods only.

1. **Static:** This employs the existing static heuristics for method inlining in level-1 and level-2 compilation. No profile data on call site information is collected, and the recompilations are requested according to the relative hotness from the sampling profiler.
2. **Profile-1:** This disables all the static heuristics for inlining except for tiny methods, which are always inlined, and small methods, which are inlined in level-1 and level-2 compilations. Inlining other than for these small methods is based solely on the profile information. The inlining requests are conservative in that all the methods called through the hot call path must appear in the group of hot methods.
3. **Profile-2:** This is the same as Profile-1, but more aggressive inlining can be requested for callers that are not in the current group of hot methods. In such cases, the inlining is requested for the methods in the call path within the current group, as well as for the topmost method of the call chain outside of the group.
4. **Hybrid:** This is the same as Profile-1, but also using the current static heuristics if the inlining budget is still not used up after accepting all of the profile-based inlining requests.
5. **Offline:** This is the same as Profile-1, but inlining decisions are made using the offline collected profile information. Section 4.5 gives details regarding how the offline profile information is collected and then used for measurement.

When measuring the compilation overhead in Section 4.3, we measure only level-1 and level-2 compilation statistics, since level-0 is common among all four cases. The compiler is instrumented with several hooks for each part of the optimization process to record the value of the processor’s time stamp counter. The priority of the compile thread (for level-1 and level-2) is set higher than normal application threads, so the difference between each value from the time stamp counter is guaranteed to provide time spent for performing the corresponding optimization work, and also the difference in the value between the beginning and the end of the compilation should be the total compilation time.

4.2 Instrumentation and Inlining Statistics

Table 1 shows the compilation statistics when running the benchmarks with the profile-1 inlining policy. The second column of the table shows the total number of methods executed (without the effects of inlining). Native methods are not included. The next five columns show the number of compiled and instrumented methods for each of the optimization levels. No instrumentation is applied to level-2 compiled code, as mentioned in Section 3.1. The count is cumulative, so if a method is level-0 compiled and then promoted to level-1, then it is counted in both categories. The last five columns show the total number of inlining requests made through four steps described in Section 3.2. The number is per call site, so if two inline requests are issued for a method, inlining it into two different call sites within a method, then both are counted as inline requests.

The number of instrumented methods is mostly 10% or less of the compiled methods in level-0 for spiky profile benchmarks (top five in the table) and about 15% to

Benchmarks	# methods executed	level-0		level-1		level-2	inline requests				
		compiled	inst'd	compiled	inst'd	compiled	depth 1	depth 2	depth 3	depth>4	total
227_mtrt	333	182	18	16	4	7	13	2	1	--	16
202_jess	611	311	34	18	11	12	9	6	3	3	21
201_compress	204	118	5	1	0	3	3	--	--	--	3
209_db	199	99	6	2	1	3	6	--	--	--	6
222_mpegaudio	364	207	29	26	10	18	9	2	--	2	13
228_jack	440	363	52	43	15	15	18	4	1	1	24
213_javac	958	823	117	128	24	22	16	3	1	--	20
SPECjbb2000	2663	554	130	129	33	32	18	3	2	2	25

Table 1. Statistics of compiled and instrumented methods for each optimization level, and the number of inlining requests when running benchmarks with the profile-1 inlining policy. The depth columns represent the numbers of call chains for each call path that was requested to be inlined.

20% for flat profile benchmarks (bottom three). The number is about 30% or greater of the compiled methods in level-1. These figures are almost the same as the proportion of methods that are promoted from each level.

The inlining decision seems to work very selectively for each benchmark. The total number of inlining requests relative to the number of compiled methods at level-1 and level-2 is significantly smaller for flat profile benchmarks compared to spiky profile ones, reflecting the strict requirements for call paths to be requested for inlining in terms of the degree of bias in call site distributions.

4.3 Compilation Overhead

Figures 6 to 8 show various data for compilation overhead: the compilation time, compiled code size expansion ratio over the bytecode size, and compilation time peak work memory usage. Smaller bars mean better scores in these figures. The compiled code size used for profile-based inlining policies (profile-1, profile-2, and hybrid) includes additional code space that is dynamically allocated for instrumentation. The peak memory usage is the maximum amount of memory allocated for compiling methods. Since our memory management routine allocates and frees memory in 1 Mbyte blocks, this large granularity masks the minor differences in memory usage and this causes the result of Figure 8 to form clusters.

Overall the profile-based inlining (both profile-1 and profile-2) shows significant advantages over the existing static-heuristics-based inlining in most of the benchmarks. At worst, it does not exceed twice the baseline (the tiny-only inlining policy) in all three metrics of the compilation overhead. In the best case, it is almost equal to the level of tiny-only, apparently due to the minimum number of inlining requests made only for those call sites deemed beneficial for performance improvement. The code size expansion ratio is greatly reduced for some benchmarks, even taking the instrumentation code into account. On average, the compilation overhead (in all three metrics) is just about a 20% to 30% increase from the baseline, and almost a 40% reduction compared to static-heuristics-based inlining.

There are a few exceptions to these general observations. Compress shows degradation with profile-based inlining, and mtrt shows no significant improvement in all three indications of compilation overhead. As shown in Table 1, however, compress has a very few hot methods, resulting in relatively low level of compilation overhead regardless of the inlining policy. In the profile-based inlining case, two inlining requests were actually made for the same method, Compressor/compress, but with different timings because of the group of hot methods supplied by the sampling profiler. Thus the method was compiled twice with full optimization, first with one inlining request and then with an additional request identified later. Because of the low level of compilation overhead, this additional level-2 compilation was enough to cause a seemingly big difference in these ratios. Mtrt suffers for almost the same reason. An additional inlining request was made to one of the hottest methods OctNode/Intersect, after the callee for the call edge had become hot, triggering the second level-2 compilation, and this made the total compilation time increase significantly.

As expected, the hybrid inlining policy generally shows a similar amount of overhead as the static heuristics, but it sometimes causes significantly higher overhead. The overhead for compress is high for the same reason described above, but for the other benchmarks, the poor scores are due to the aggressiveness of the static heuris-

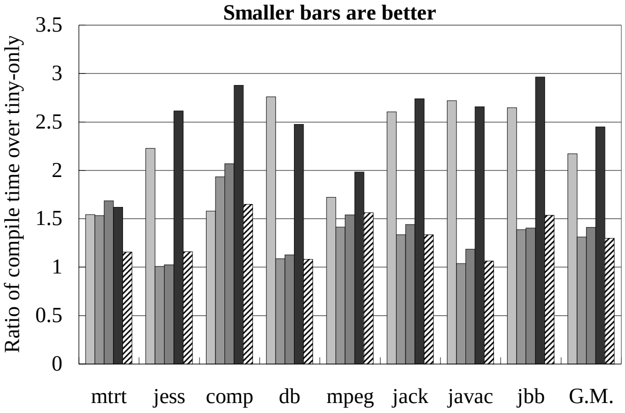

Figure 6. Ratio of compilation time for five inlining policies over the tiny-only case.

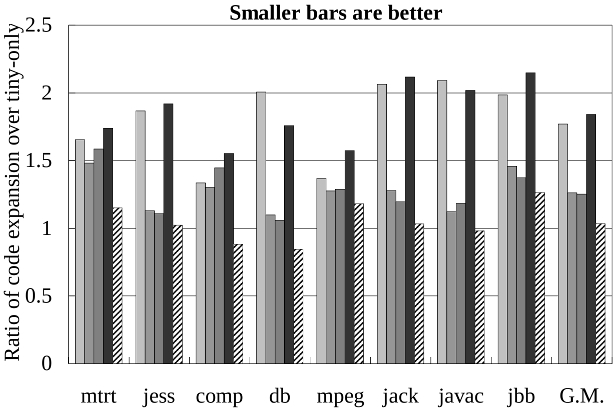

Figure 7. Ratio of compiled code size expansion for five inlining policies over the tiny-only case.

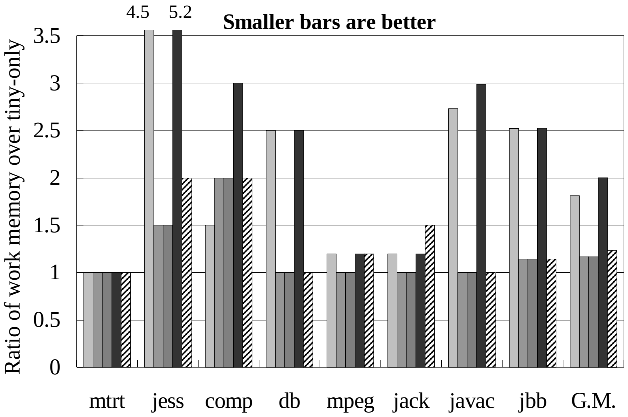

Figure 8. Ratio of compilation time peak memory usage for five inlining policies over the tiny-only case.

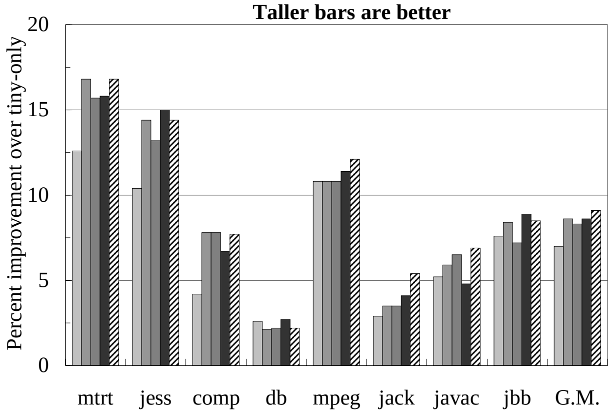

Figure 9. Performance comparison of five inlining policies over the tiny-only case.

□ Static □ Profile-1 □ Profile-2 ■ Hybrid ▨ Offline

tics applied after processing the profile-requested inlining. This inlining policy seems unattractive from the viewpoint of compilation overhead.

4.4 Performance Comparison

Figure 9 shows the performance improvement with the five inlining policies over the tiny-only inlining case. Taller bars show better performance. For each inlining policy, we took the best time from 10 executions of an autorun sequence for each test in SPECjvm98 and the best throughput from a series of successive executions from 1 to 8 warehouses for SPECjbb2000. These results show that the profile-based inlining, both profile-1 and profile-2, performs about as well as or better than the static-heuristics-based inlining for most of the benchmarks. The use of profiling seems particularly effective for jess and compress.

Based on the comparison of the actual inlining results between profile-based and static-heuristics-based inlining policies, it was found that for jess, many of the call paths that were identified as important by the profile-based policies were also inlined with the static heuristics, but there were several critical call paths that were not inlined with the static heuristics, and this caused the performance differences. One probable reason for this result is that the chance to inline performance critical call paths is decreased with the static heuristics because the limited inlining budget is wasted by greedily inlining methods based on static assumptions. For compress, the profile-based inlining policy made two critical inlining requests that the static heuristics declined to perform due to the excessive code size for inlining.

The performance is slightly degraded with the profile-based inlining for db. The fact that the hybrid inlining policy restores the performance shows that the current profile-based inlining is missing some opportunities for inlining performance-sensitive call sites in this benchmark. Since the degradation can be observed with the offline-based-profiling as well, it is probably not due to the short interval in online instrumentation to find dominant call edges. Rather it seems that target methods that should have been inlined were overlooked by the sampling profiler as recompilation candidates. The problem may be resolved by expanding the small method criteria, however, this would go against the goal of our approach of basically relying only on profile information. We need to come up with an effective way to deal with this situation.

The profile-2 inlining policy does not seem to produce any additional performance improvement over profile-1 in these benchmarks, regardless of the more aggressive inlining requests.

4.5 Accuracy of Inlining Decision

Because our profile-directed method inlining is on the basis of the call site profiling information collected for a fixed short time interval, there may be a concern as to whether the profile information is accurate enough for driving inlining when compared to a complete profile. In this section we evaluate the accuracy of our profile-based inlining decisions.

We define the accuracy of the inlining decisions relative to the best possible performance improvement and the smallest possible compilation overhead. To obtain complete profiling information during the entire period of program execution, we generated a hook to call a runtime routine at the entry point of the compiled code for each method, taking the caller's address and updating the profile table every time. The total invocation count was also recorded. The information was then written to disk at the end of the run. This provides the equivalent kinds of information to what is collected with our instrumenting profiler at runtime: call site distribution and execution frequency. In the next run, we used the complete profile information as input to make the inlining decisions, without using the instrumentation mechanism. This should provide the upper bound in terms of both performance and compilation overhead improvement for the system with profile-directed method inlining.

Figures 6 to 9 also show how the inlining based on offline collected profile information works relative to other inlining policies. Overall the offline-profile-based inlining finds more opportunities for inlining important call paths and hence is more aggressive compared to the online-profile-based inlining policies. Nevertheless, it shows lower compilation overhead than the profile-1 inlining policy in most of the cases because of the reasons described below.

Compress and mtrt no longer suffer from the problem of additional level-2 recompilation as we saw in Section 4.3. The code size expansion is consistently improved since no instrumentation overhead is imposed. Also some methods can be promoted from level-0 directly to level-2 compilation, because the important call paths for inlining and their impacts are known from the beginning, and this can reduce the total compilation time and code size expansion. All of these are, however, limitations of online-based profiling and its feedback system, and therefore the difference in compilation overhead is considered reasonable.

As for performance, the offline-profile based inlining shows slightly better numbers over profile-1 for some benchmarks, but the difference is not very significant on average. Despite the limited time interval for collecting profile information, our online profiling system seems to capture the majority of the performance sensitive call paths quite well, and hence the method inlining decisions based on this profile information are regarded as effective from the performance point of view.

5. DISCUSSION

We have described the design and implementation of online-profile-directed method inlining in our dynamic compilation framework. We present some observations and discussions based on the results in this section.

Overall the study shows the advantages of the pure profile-based inlining policy (without adding in static heuristics). It shows the potential for either significantly reducing the compilation overhead (measured in time, work memory, and code size) or for improving performance, or both, compared to other inlining policies. In the dynamic compilation environment, we have to be very careful about performing any optimizations that can have significant impact on compilation overhead. The online profiling can provide useful information to guide inlining only for those call sites and call paths that can be expected to produce performance improvements.

As described earlier, our ultimate goal is automatic optimization selections in the dynamic compilation system, and this study shows encouraging results as a first step, since inlining exerts the most influence on both performance and compilation overhead. When inlining decisions are profile-based and the compilation boundaries can be dynamically determined, then the next step will be to estimate the impact of each costly optimization based on the structure of a given method, using indications such as loop structure, characteristics of field and array accesses, and others, as suggested in [2].

The problem described in Section 4.3 is considered inherent to an online profiling and feedback system, since the information is partial, not complete, at any point in time, and we do not know when is the best time to drive recompilation with limited available information. When additional information that can contribute to the performance is later available, we then have to decide whether it is better to trigger additional recompilations. Currently we drive recompilation considering the estimated performance benefit of the additional inlining request based on the relative hotness counts of the caller and the callee, but also limit the maximum number of level-2 compilations for the same method in order to avoid code explosion. We need to explore better ways to manage this situation by, for example, introducing additional metrics of the impact of inlining on a particular call path.

In this paper, we consider only the relative strengths and distributions of the call edges when driving inlining. However, the significant impact of method inlining is not only through the direct effect of eliminating call overhead, but also due to indirect effects of specializing an inlined method body into the calling context, with better utilization of dataflow information at the call sites. Since our instrumentation mechanism can collect parameter values as well as return addresses, it should be possible to estimate whether inlining a method will be beneficial in terms of the effect of these optimizations as well, based on the result of impact analysis [23]. We will try to exploit this information in the future for pursuing better inlining strategies.

6. RELATED WORK

There are several previous studies that evaluate method inlining using profile information, mostly collected offline, in both static and dynamic compilation systems. Overall, it was concluded that the use of profile information for inlining is effective to improve program performance under careful management of the increase of the static code size.

Scheifler [21] shows that inlining optimization can be reduced to the well-known knapsack problem. He uses a greedy algorithm to minimize the estimated number of function calls subject to a size constraint. This relies on the runtime statistics of the program, collected with various sets of input data, to calculate the expected overhead of each invocation. The constant ratio assumption is used to avoid the cost of a multi-level history in gathering statistics. Kaser and Ramakrishnan [16] propose a probabilistic model to estimate the effect of using profile data, with a one-level history based on the constant ratio assumption. When evaluating inlinable calls remaining after optimization, they report good results with their technique compared to other compilers.

Chang et al. [7] describe profile-guided procedure inlining with the IMPACT optimizing C compiler. They first construct a weighted call graph using the offline-collected profile information on the number of invocations of each function and relative hotness counts of each call edge. A greedy algorithm is then applied bottom-up in the call graph to maximize the number of reductions of dynamic function calls while keeping the code size expansion within a fixed bound. Their result shows a significant performance improvement with a relatively low code expansion ratio. However they don't report on how much of the effectiveness is contributed by the use of profile information. Dynamically dispatched calls (through function pointers in C programs) are not inlined in this study.

Ayers et al. [5] describe the design and implementation of the inlining and cloning in the HP-UX optimizing compiler, and show the performance of the SPECint92 and 95 benchmarks can be substantially improved with their technique. They use profile information collected offline to prioritize the inlining or cloning candidates of the call site to be considered. The use of profile information is reported to be quite effective, however it is an intra-procedural profile of the basic block level execution frequency, not information on call edges for guiding inlining for a particular call path.

Arnold et al. [4] present a comparative study of static and profile-based heuristics for inlining with several limits on code expansion. In considering three inlining heuristics, based on a static call graph, a call graph with node weights, and a dynamic call graph with edge weights, they regard the selection of inlining candidates as a knapsack problem, and employ a greedy heuristic based on the benefit/cost ratio as a meta-algorithm for approximating the NP-hard problem. Their experiment, done with offline based profiling and an ahead-of-time compilation framework, shows that a substantial (sometimes more than 50%) performance improvement can be obtained with the heuristics based on the dynamic call graph with edge weights over the static call graph, even with modest limits on code size expansion. This work became the basis for the feedback-directed method inlining in adaptive optimization implemented in the Jalapeño JVM (now called the Jikes Research Virtual Machine) [1]. This system periodically takes a statistical sample of the method calls and maintains an approximation to the dynamic call graph during the course of the entire program execution. The online-profile-directed method inlining is shown to contribute to significant performance improvement for some benchmarks. Arnold [3] further continues to improve the effectiveness of the feedback-directed inlining by collecting execution frequencies of the control flow edges between basic blocks and letting this information be used for fine-tuning the inlining decisions.

Dean and Chambers [10] describe a system based on the SELF-91 compiler that uses the first compilation as a tentative experiment and records inlining decisions and the benefits of the resulting effect on optimizations in a database. The compiler can then take advantage of the recorded information for future inlining decisions by searching the database with the known information about the receiver and arguments. They do not exploit runtime profile information in their system, though the expected execution frequency of the call site is used for inlining decisions.

The SELF-93 system [14] is an adaptive recompilation system using online profile information. It collects call-site-specific profile information for receiver class distributions (type feedback) in un-optimized runs, and then when the method is recompiled, makes use of this information to optimize dynamically-dispatched calls by predicting likely receiver types and inline calls for these types. It was demonstrated that the performance of many programs written in SELF can be substantially improved with this technique. Grove et al. [12] complement this result by studying the various characteristics of the profiles of receiver class distributions collected offline, such as the degree of bias, effectiveness of deeper granularity, the stability across input and programs. They report that the compiler can effectively use deeper granularity of profile context to predict more precisely the target of dynamically dispatched procedure calls.

HotSpot [20] is a JVM product implementing an adaptive optimization system with an interpreter to allow a mixed execution environment. As in the SELF-93 system, it also collects profiles of receiver type distribution online in the interpreted execution mode. The profile information, together with class hierarchy analysis, is used when optimizing the code for virtual and interface calls.

There are several systems that use annotations to specify dynamic optimizations. Krintz and Calder [17] describe an annotation framework for reducing compilation overhead for Java programs. One of the proposed annotations is method inlining on the basis of analysis of profile information collected offline, which allows substantial reduction of startup overhead. Mock et al. [18] present Calpa, a system to automatically generate annotations for the DyC dynamic compiler. It evaluates the offline-collected information regarding basic block execution frequencies and a value profile based on its own cost/benefit model, and determines runtime constants for specialization and dynamic compilation strategies.

7. CONCLUSIONS

We have described an empirical study of online-profile-directed method inlining in our Java dynamic compilation system. We removed the existing static heuristics of method inlining except when dealing with tiny and small size methods, and rely solely on the online profile information to drive the method inlining in level-1 and level-2 optimizations. We presented our design and implementation showing how the information on call site distribution and execution frequency can be collected at runtime, and then showed how the hot call paths can be extracted to identify potential candidates for method inlining. We evaluated our approach using the industry standard benchmarks, and showed its potential to achieve better performance and/or smaller compilation overhead (measured in compilation time, peak memory usage, and code size expansion ratio) when compared to the static inlining heuristics.

In the future, we will further study the effectiveness of profile-directed method inlining by examining the accuracy and efficiency of our inlining approach when compared to other approaches such as dynamic call graph construction at sampling time [1].

8. ACKNOWLEDGMENTS

We would like to thank all the members of the Network Computing Platform group in IBM Tokyo Research Laboratory for helpful discussions and comments. The anonymous reviewers also provided many valuable suggestions and comments on the presentation of the paper.

REFERENCES and NOTES

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimizations in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 47-65, Oct. 2000.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive Optimizations in the Jalapeño JVM: The Controller’s Analytical Model. In *Proceedings of the ACM SIGPLAN Workshop on Feedback-Directed and Dynamic Optimization, FDDO-3*, Dec. 2000.

[3] M. Arnold. Online Instrumentation and Feedback-Directed Optimization of Java. Technical Report DCS-TR-469, Department of Computer Science, Rutgers University, Nov. 2001.

[4] M. Arnold, S. Fink, V. Sarkar, and P.F. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, Jan. 2000.

[5] A. Ayers, R. Gottlieb, and R. Schooler. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 134-145, Jun. 1997.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-12, Jun. 2000.

[7] P.P. Chang, S.A. Mahlke, W.Y. Chen, and W.W. Hwu. Profile-Guided Automatic Inline Expansion for C Programs. *Software Practice and Experience* 22(5), pp. 349-369, May 1992.

[8] W.Y. Chen, P.P. Chang, T.M. Conte, and W.W. Hwu. The Effect of Code Expanding Optimizations on Instruction Cache Design. *IEEE Transactions on Computer* 42(9), pp. 1045-1057, Sep. 1993.

[9] M. Cierniak, G.Y. Lueh, and J.M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13-26, Jun. 2000.

[10] J. Dean, and C. Chambers. Training Compilers for Better Inlining Decisions. Technical Report 93-05-05, Department of Computer Science and Engineering, University of Washington, May 1993.

[11] D. Detlefs, and O. Agesen. Inlining of Virtual Methods. In *The 13th European Conference on Object-Oriented Programming, ECOOP*, Lecture Note in Computer Science, 1628, pp. 258-277, 1999.

[12] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-Guided Receiver Class Prediction. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 108-123, Oct. 1995.

[13] R.E. Hank, W.W. Hwu, and B.R. Rau. Region-Based Compilation: An Introduction and Motivation. In *Proceedings of 28th International Conference on Microarchitecture*, pp. 158-168, Dec. 1995.

[14] U. Hölzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Ph.D. Thesis, Stanford University, CS-TR-94-1520, Aug. 1994.

[15] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 294-310, Oct. 2000.

[16] O. Kaser, and C.R. Ramakrishman. Evaluating Inlining Techniques. *Computer Languages*, 24(2) pp. 55-72, 1998.

[17] C. Krintz, and B. Calder. Using Annotations to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 156-167, Jun. 2001.

[18] M. Mock, C. Chambers, and S. Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In *Proceedings of 33rd International Conference on Microarchitecture*, pp. 1-12, Dec. 2000.

[19] T. Ogasawara, H. Komatsu, and T. Nakatani. A Study of Exceptions and its Dynamic Optimization for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 83-95, Oct. 2001.

[20] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pp. 1-12, Apr. 2001.

[21] R.W. Schiefler. An Analysis of Inline Substitution for a Structured Programming Language. *Communications of the ACM*, 20(9), pp. 647-654, Sep. 1977.

[22] Standard Performance Evaluation Corporation. SPECjvm98 and SPECjbb2000 benchmarks available at http://www.spec.org/osg.

[23] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, pp. 180-194, Oct. 2001.

[24] J. Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *Proceedings of the ACM SIGPLAN Java Grande Conference*, pp. 78-87, Jun. 2000.

[1] Note that each compilation level does not match the corresponding level of optimization described in [23], although the number of optimization levels is the same. The elements within each level have been rearranged. The purpose of level-0 compilation is to allow transition from MMI to compiled code with a lower threshold value without causing additional compilation overhead, and to provide the system with profile information for a broader range of methods.

[2] This estimate excludes the prologue and epilogue code. The compiled code size is estimated based on the sequence of bytecodes each of which is assigned an approximate number of instructions generated.

[3] The instrumentation code is just a small piece of code, less than 100 instruction bytes, and the corresponding table space is in practice less than another 100 bytes. This space is treated in the same way as the compiled code so that it can be reclaimed upon class unloading.

[4] Since tiny methods are always inlined in the compiled code and thus the call site found in the profile may actually be within an inlined tiny method, the runtime structure, IMF, is consulted to get the exact call path from the call edge.