A Comparative Study of Static and Dynamic Heuristics for Inlining

Matthew Arnold
Department of Computer Science
Rutgers, The State University of NJ
Email: marnold@cs.rutgers.edu

Stephen Fink Vivek Sarkar Peter F. Sweeney
IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
Email: {sjfink, vsarkar, pfs}@us.ibm.com

Abstract

In this paper, we present a comparative study of static and dynamic heuristics for inlining. We introduce *inlining plans* as a formal representation for nested inlining decisions made by an inlining heuristic. We use a well-known approximation algorithm for the KNAPSACK problem as a common "meta-algorithm" for the static and dynamic inlining heuristics studied in this paper. We present performance results for an implementation of these inlining heuristics in the Jalapeño dynamic optimizing compiler for Java. Our performance results show that the inlining heuristics studied in this paper can lead to significant speedups in execution time (up to 1.68×) even with modest limits on code size expansion (at most 10%).

# pages excluding title page & bibliography	=	15
# pages used by figures and tables	=	4
# pages of text	=	11

1 Introduction

Procedure *inlining* (*i.e.,* inline expansion of procedure calls) has been a well-known program transformation for almost three decades [1]. The inlining transformation replaces a call site by an "in-line" copy of the body of the procedure being called.

Procedure inlining can result in at least three kinds of benefits. First, the inlining transformation eliminates linkage overhead of the call. Second, the compiler can use data flow properties at the call site to generate more efficient code for the inlined copy of the procedure *i.e.,* the inlined routine is *specialized* to its calling context. Third, the compiler can uses data flow properties from the (specialized) inlined copy to generate more efficient code in the calling procedure. Unfortunately, the inlining transformation also has potential costs — excessive inlining can increase target code size, increase the number of I-cache misses, and increase the number of register spills with current register allocation algorithms. Additionally, inlining increases compile-time, a major factor in dynamic compilation.

Finding the best tradeoff among these benefits and costs presents a major challenge. The compiler will rely on an *inlining heuristic*, an algorithm for selecting call sites to inline. A *static inlining heuristic* uses only the static program text to guide its inlining decisions, without relying on any runtime information. A *dynamic inlining heuristic* additionally uses runtime profile information to guide its inlining decisions. A dynamic compiler might employ either static or dynamic inlining heuristics, or both.

In this paper, we present a comparative study of static and dynamic heuristics for inlining. We introduce *inlining plans* as a formal representation for nested inlining decisions made by an inlining heuristic. As in past work, we formalize the inlining optimization problem as a variant of the KNAPSACK problem [12]. We use a well-known approximation algorithm for the KNAPSACK problem as a common "meta-algorithm" for the static and dynamic inlining heuristics studied in this paper. The use of a common meta-algorithm makes it possible to perform a uniform "apples-to-apples" comparison among the inlining heuristics. We present performance results for an implementation of these inlining heuristics in the Jalapeño dynamic optimizing compiler for Java [6]. Since the Jalapeño JVM (Java Virtual Machine) is implemented in Java [3, 2], the scope for nested inlining through application, library, and JVM runtime layers exceeds the scope in traditional JVM's implemented in native code. Our performance results show that the inlining heuristics studied in this paper can lead to significant speedups in execution time (up to 1.68×) even with modest limits on code size expansion (at most 10%).

Our primary motivation for this study is to understand the effectiveness of various inlining heuristics, in order to choose the best candidate for use in the Jalapeño dynamic optimizing compiler. Since profiling overhead and compile-time overhead in dynamic compilation contribute directly to runtime performance, we must take care to ensure that the inlining algorithms do not use excessive time and space resources (as is typically the case with inlining algorithms for static compilation). Static heuristics enjoy the advantage that they do not require any run-time profiling overhead; to their disadvantage, they do not use runtime information and thus might make poor inlining decisions. Conversely, dynamic heuristics have the advantage of runtime information to make better inlining decisions, but collecting runtime information imposes extra overhead. We do not study the costs of profiling or of compilation overhead in this paper. Instead, this study focuses on the relative benefits of different inlining heuristics for a given limit on code size expansion. (As a rough estimate, one can consider the code size expansion to be indicative of the extra compilation overhead.)

The rest of the paper is organized as follows. Section 2 reviews the *static call graph* and *dynamic call graph* representations. Section 3 introduces our formalization of *inlining plans* and the optimization problem for cost-based inlining. Section 4 describes the three inlining heuristics considered in this study, which are based on the *static call graph* (SCG), *call graph with node weights* (CG-N), and *dynamic call graph with edge weights* (DCG-E), respectively. Section 5 outlines our algorithm for rewriting procedures according to a given set of inlining plans. Section 6 contains our experimental results. Section 7 discusses related work, and section 8 contains our conclusions.

2 Static and Dynamic Call Graph Representations

A call graph $G = (N, E)$ is a multigraph with N as its set of nodes and E as its set of edges. Each node in N represents a distinct procedure/method[1] in the program, and each edge represents a call site. Specifically, an edge from node n_i to node n_j represents a call instruction in n_i's method with n_j's method as the target. Since n_i's method may contain multiple such call sites, we also label the edge with the address of the call instruction. For Java bytecodes, it is convenient to use the bytecode index of the call/invoke instruction as the edge label. In the case of indirect calls such as virtual and interface calls, it is possible for multiple edges in a call graph to correspond to the same call site, since an indirect call can have multiple potential targets.

Call graphs can be static, or dynamic, or a hybrid of both. A *static call graph* (SCG) represents all possible calling sequences in the program *i.e.,* an edge (n_i, n_j) is included in the SCG for each *potential* call from n_i's method to n_j's method. A *dynamic call graph* (DCG) represents only those procedures and calls that were actually encountered in a given execution (or set of executions) of the program. Therefore, the DCG is a subgraph of the SCG.

Dynamic loading of classes/modules (as in Java) has two important consequences for call-graph-based optimizations. First, dynamic class loading via reflection makes it impossible to construct a complete SCG, since it is impossible to anticipate a priori the set of all procedures that might be executed. Second, it is infeasible to perform a common cleanup optimization after inlining *viz.,* removing a procedure from a program if it has been inlined into all its call sites (removal is not possible because additional call sites for the procedure might appear after dynamic loading).

Since the study in this paper is focused on Java programs (or, more generally, object-oriented programs with dynamic class loading), we address the above two issues as follows. First, for the scope of this paper, we modify the definition of an SCG to mean a call graph that has the *same set of classes* as the DCG, but whose nodes represent all the methods of these classes and whose edges represent all possible calling sequences among between these nodes. We believe this definition represents the world-view of a dynamic compiler in a JVM. Note that this definition may lead to smaller SCGs than pure static analysis since methods of classes that are referenced in the source code, but never loaded during the execution, do not appear as nodes in our SCGs. Second, we do not perform the optimization of removing a method from a program if it has been inlined at all its call sites.

Call graphs can be augmented with node and edge weights. A *call graph with node weights* (CG-N) includes a numerical weight for each node that represents the dynamic frequency of the node's method *i.e.,* the number of calls to the method. This graph is a hybrid between the static and a dynamic call graphs, as

[1] We will use the terms "procedure" and "method" interchangeably in this paper.

it has the same set of nodes as the dynamic call graph, but may also include edges with zero call frequency. A *dynamic call graph with edge weights* (DCG-E) augments a DCG with numerical weight for each edge that represents the dynamic frequency of the call corresponding to the edge.

There is an important difference in the granularity of information held by the three representations (SCG, CG-N, DCG-E). In both an SCG and a CG-N, there are no weights to distinguish among the different call sites (input edges) to a method[2]. Faced with the absence of this information, we will require the SCG and CG-N inlining heuristics described in this paper to pursue an "all or none" approach for inlining a method using an SCG or a CG-N *i.e.,* either *all call sites* for a target method will be inlined or none.

A DCG-E has edge weights that allow an inlining heuristic to distinguish the frequency of each call site to a given method. However, we require that a finer-grained "all or none" rule be imposed for inlining based on the DCG-E *viz.,* either *all instances of a given call site* should be inlined or none. Note that multiple instances of a given call site can be created by the code duplication that results from the inlining transformation.

An important consequence of the "all or none" requirement is that the inlining heuristics described in this paper will be unable to inline recursive calls. (Any attempt to inline all instances of a recursive call site would lead to unbounded code expansion.) In the future, we plan to study inlining heuristics based on calling context information that need not obey the all-or-none requirement, and thus will be capable of inlining recursive calls. It is important to note that the all-or-none restriction only applies to the SCG, CG-N, and DCG-E heuristics, and is not inherent to the problem formalization introduced in section 3 *e.g.,* the inlining plans in section 3 can be used to express inlining of recursive calls.

Figure 1 shows the SCG, CG-N, and DCG-E representations for an example program with ten procedures. The SCG contains all ten procedures. However, the CG-N and DCG-E do not include procedures `G`, `H`, and `I`; in our example program execution, the call sites in `MAIN` to `G`, `H`, and `I` were never called. The frequency of the other call sites appear in figure 1c. In section 4, we will use this example to illustrate the three different inlining heuristics studied in this paper.

3 Cost-based Inlining — Formalization

In this section, we introduce our formalization of *inlining plans* and the optimization problem for cost-based inlining. Given a static or dynamic call graph, an inlining heuristic selects an inlining plan for each method in the program. The inlining plan identifies which of the method's call sites should be expanded in-line. Section 5 discusses the algorithm for rewriting each method according to its inlining plan.

Given a method A, its inlining plan, IP_A, is a connected tree with a single root. Each non-root node in IP_A corresponds to a call site that will be inlined in the transformed code. The parent-child relationship in IP_A reflects the caller-callee relation for inlined method bodies. A tree with more than two levels specifies inlining of nested calls. An *inlining configuration* is a collection of inlining plans for all methods in the program.

There are several potential *benefits* due to inlining *e.g.,* fewer calls at runtime, larger regions of code for optimization, easy exchange of context information across the caller-callee boundary, specialization, etc. For the sake of simplicity in formulating the algorithms, and following practice in several previous

[2] In future work, we plan to extend a CG-N with edge weights that are estimated from the node weights, and compare its effectiveness with that of a DCG-E.

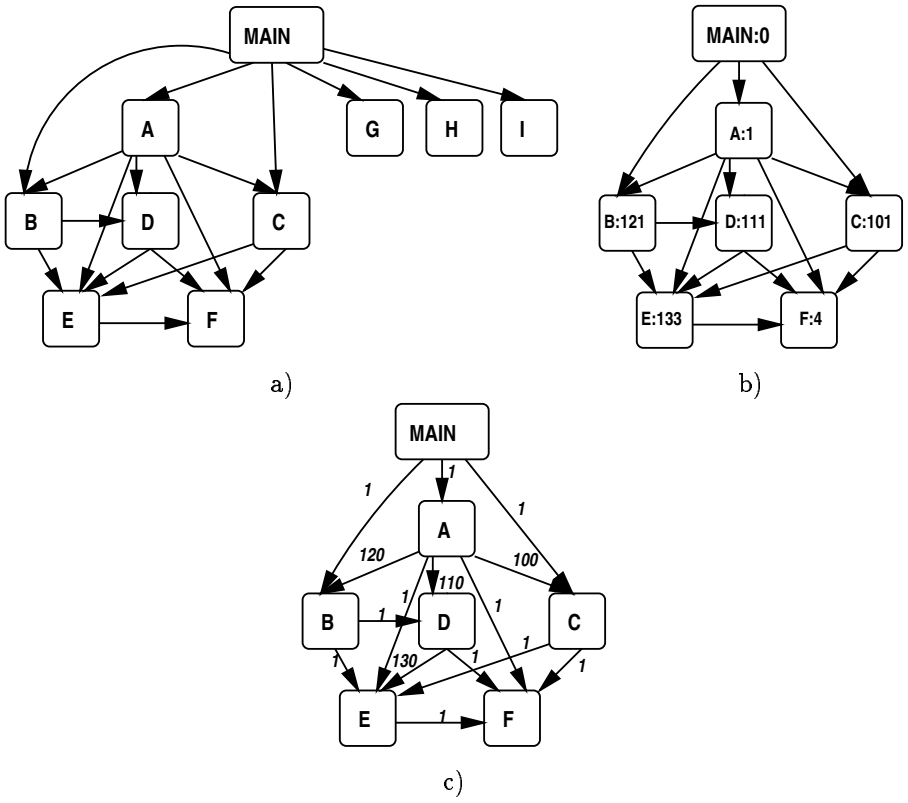

Figure 1: Call graph representations for an example program. a) SCG, b) CG-N, and c)DCG-E.

studies [16, 7, 15], we model the benefit of inlining as the number of runtime method calls eliminated. (The "bottom-line" benefit will of course be execution-time, which we will report in section 6.)

There also many potential sources of *overhead* due to inlining *e.g.,* increase in code size, decrease in code locality, increase in register pressure, etc. Again, for the sake of simplicity, we model the increase in code size as the sole cost of inlining.

Now, the optimization problem for selecting inlining plans can be formally stated as follows:

Given a limit on code size increase, LIMIT, select inlining plans for all methods such that:

1. $totalCost \leq LIMIT$, *where* $totalCost = \sum_{method\ A} cost(IP_A)$ *is the total code size increase across all inlining plans, and*
2. *totalBenefit is maximized, where* $totalBenefit = \sum_{method\ A} benefit(IP_A)$ *is the total benefit delivered across all inlining plans.*

In practice, *LIMIT* is set proportional to the total code size of all methods, using an expansion factor such as 10% or 50%. We observe that the above optimization problem is at least as hard as the well-known KNAPSACK problem [12], which is known to be NP-hard. However, there are some approximation algorithms to the KNAPSACK problem that are known to have tight performance bounds and to also be very effective in practice. One such notable approximation algorithm is to select knapsack items (in our case, call sites) in decreasing order of *benefit/cost* ratio. In the next section, we use this greedy heuristic as a common meta-algorithm for defining the three inlining heuristics studied in this paper.

```
totalCost := 0 ;
Initialize set S of candidate inlining decisions ;
while (totalCost < LIMIT) do
   Choose an inlining decision D from S with the largest benefit/cost ratio
   if (totalCost + cost(D) ≤ LIMIT) then
      Add inlining decision D to inlining plans ;
      totalCost := totalCost + cost(D) ;
      Update any candidate inlining decisions whose ratio may change due to D ;
   end if
   Remove inlining decision D from S ;
end while
```

Figure 2: Meta-algorithm for different inlining heuristics

4 Algorithms for Selection of Inlining Plans

In this section, we describe the three inlining heuristics considered in this study, and demonstrate how the meta-algorithm, presented as an outline in figure 2, can be parameterized by each of the inlining heuristics to form three separate algorithms. Recall that the underlying meta-algorithm makes an inlining choice that yields the largest *benefit/cost* ratio in each iteration of a greedy algorithm.

As we will see, the key differences among the three algorithms lie in the granularity of inlining choices (*e.g.,* inlining of all calls to a procedure in contrast to inlining of selected calls to a procedure) and in improved accuracy when estimating benefits due to inlining (*e.g.,* edge weights in a DCG-E lead to more accurate estimates of inlining benefits than node weights in a CG-N).

To simplify the following discussion, we assume that all methods in the running example, presented in figure 1, have the same *code size*. We also assume a code expansion limit of $LIMIT = 4$, where each method has $size = 1$.

4.1 Algorithm 1: using the Static Call Graph (SCG)

The first version of our meta-algorithm, denoted the SCG algorithm, uses an inlining heuristic that is based on the static call graph. Since the SCG includes no node weights or edge weights, no distinction can be made among inlining choices based on benefit. Hence, the heuristic of making an inlining choice with the largest *benefit/cost* ratio reduces to making a choice with the smallest *cost*. Since any SCG heuristic must take an all-or-nothing approach with respect to inlining call sites for a method, this heuristic can also be stated as choosing the method with the smallest value of (*# of instances of call sites*) × (*code size*) in each iteration.

A trace of the SCG algorithm will look as follows for the call graph in figure 2a:

1. $totalCost := 0$
2. Inline A in $MAIN$ (*i.e.,* inline all calls to procedure A) ; $totalCost := 1$
3. Inline G in $MAIN$ (*i.e.,* inline all calls to procedure G) ; $totalCost := 2$
4. Inline H in $MAIN$ (*i.e.,* inline all calls to procedure H) ; $totalCost := 3$

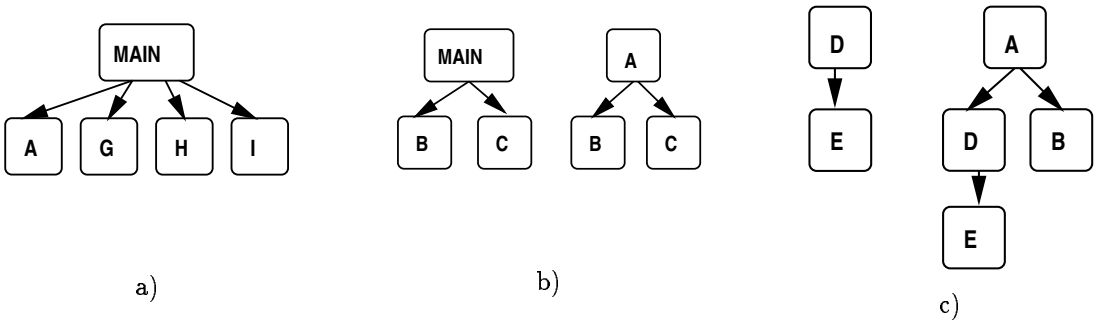

Figure 3: Inlining configuration output for example program by a) SCG, b) CG-N, and c)DCG-E.

5. Inline I in *MAIN* (*i.e.,* inline all calls to procedure I) ; *totalCost* := 4

6. Stop because any further inlining decision will cause *totalCost* to exceed *LIMIT*

Hence, the resulting configuration obtained by the SCG algorithm for $LIMIT = 4$ will have $totalCost = 4$ and $totalBenefit = 1$. (Inlining decisions for any method other than A, G, H, I, would have incurred a cost > 1, and hence were not considered above.) Figure 4.1a illustrates the inlining configuration (a single inlining plan for *MAIN*) obtained by the SCG algorithms.

Though more sophisticated heuristics for static inlining have been considered in past work (*e.g.,* giving priority to inlining leaf nodes in the call graph, or to calls contained within loops), we use the above heuristic in this study so as to enable a uniform comparison with the CG-N and DCG-E heuristics. In future work, we will consider adding some of these heuristics as extensions to all three inlining algorithms studied in the paper.

4.2 Algorithm 2: using the Call Graph with Node weights (CG-N)

The second version of our meta-algorithm, denoted the CG-N algorithm, uses an inlining heuristic that is based on a call graph with node weights. The advantage of the CG-N algorithm over the SCG algorithm is that node weights can now be used to compute benefits due to inlining choices. Since a node weight can not discriminate between the benefit of incoming edges, the CG-N algorithm takes an all-or-nothing approach with respect to inlining call sites for a method by inlining the method at every call site corresponding to an incoming edge. Therefore, the total *benefit* term for a method, to be inlined by the CG-N algorithm, simply equals its node weight. As in the SCG algorithm, the *cost* term equals (*# of instances of call sites*) $\times$ (*code size*).

A trace of the CG-N algorithm will look as follows for the example in figure 2:

1. *totalCost* := 0

2. Inline B in *MAIN* and A (*i.e.,* inline all calls to procedure B, which has largest *benefit/cost* ratio = $121/2 = 60.5$) ; *totalCost* := 2

3. Inline C in *MAIN* and A (*i.e.,* inline all calls to procedure C, which has the next largest[3] *benefit/cost* ratio = $101/2 = 50.5$) ; *totalCost* := 4

[3] We do not consider D as a candidate for inlining, because inlining all calls to D will cause *totalCost* to increase to 6, and thus exceed $LIMIT = 4$. (D would have to be inlined in A and B, and also in the two copies of B that were previously inlined in *MAIN* and A.)

4. Stop because any further inlining decision will cause *totalCost* to exceed *LIMIT*

Hence, the resulting configuration obtained by the CG-N algorithm for $LIMIT = 4$ will have $totalCost = 4$ and $totalBenefit = 222$. Figure 4.1b illustrates the inlining configuration (with inlining plans for *MAIN* and *A*) obtained by the CG-N algorithm. Compared to the SCG algorithm, the CG-N algorithm yielded a superior benefit because of its use of profile information (node weights).

4.3 Algorithm 3: using the Dynamic Call Graph with Edge Weights (DCG-E)

The third version of our meta-algorithm, denoted the DCG-E algorithm, uses an inlining heuristic that is based on a dynamic call graph with edge weights. The advantage of the DCG-E algorithm over the CG-N algorithm is that individual call site frequencies can be used to selectively perform inlining on call sites that deliver the greatest benefits. The all-or-nothing rule for call sites dictates that the total *benefit* for a call site to be inlined simply equals its edge weight, and its *cost* equals (*# of instances of the call site*) × (*code size*). The DCG-E heuristic is then simply a greedy algorithm that iteratively selects a call site to be inlined that has the largest *benefit/cost* ratio.

A trace of the DCG-E algorithm will look as follows for the example in figure 2:

1. $totalCost := 0$
2. Inline E in D (inline the call site that has the greatest frequency, 130) ; $totalCost := 1$
3. Inline B in A (inline the call site that has the next greatest frequency, 120) ; $totalCost := 2$
4. Inline D in A (inline the call site that has the next greatest frequency, 110) ; $totalCost := 4$
   /* *totalCost* increased by 2 because an extra copy of E was made for the inlined copy of D in A. */
5. Stop because any further inlining decision will cause *totalCost* to exceed *LIMIT*

Hence, the resulting configuration obtained by the DCG algorithm for $LIMIT = 4$ will have $totalCost = 4$ and $totalBenefit = 360$. Figure 4.1c illustrates the inlining configuration (with inlining plans for A and D) obtained by the DCG-E algorithm. Compared to the CG-N algorithm, the DCG-E algorithm yielded a superior benefit because of its ability to focus inlining choices on the most frequently executed call sites. (The CG-N algorithm is unable to distinguish between "hot" and "cold" edges in its call graph.)

Additional details on all three algorithms have been suppressed due to space limitations.

5 Code Transformation due to Inlining Plans

In this section, we outline our algorithm for rewriting procedures according to a given set of inlining plans. To correctly perform inlining of nested calls, the compiler maintains an *inline context* that represents the source of each instruction. An inline context is a sequence of $(method, address)$ pairs. For example, suppose that while compiling method `root`, the compiler inlines the call to `foo` at address 10, and then while compiling the inlined copy of `foo`, it inlines the call to `bar` at address 50. Then, the compiler will mark each instruction in the inlined copy of `bar` as having the *inline context*, $\langle(\texttt{root}, -1), (\texttt{foo}, 10), (\texttt{bar}, 50)\rangle$.

In general, whenever the compiler encounters a CALL instruction, it must decide whether or not the prevailing inlining plan requires the call to be inlined. Let $\langle(m_1, \alpha_1), (m_2, \alpha_2), \ldots, (m_n, \alpha_n)\rangle$ be the inline

Inputs:

- CALL instruction with inline context $\langle (m_1, \alpha_1), \ldots, (m_n, \alpha_n) \rangle$ and address α
- Inlining configuration C, with inlining plans for all methods

Output:

- Set of inlining targets for CALL instruction

Method:

```
Find inlining plan IP_{m_1} in C for method m_1 ;
If none found, return ∅ ;
caller := root node of IP_{m_1} ; i := 1 ;
while i < n do
   child := child node of caller with child-method m_{i+1} and address α_{i+1} ;
   if no such child found return ∅ ;
   caller := child ; i := i + 1 ;
end while
targets := children of caller in IP_{m_1} with address = α ;
return targets ;
```

Figure 4: Procedure for identifying inlining targets (if any) for a given call instruction

context for the call instruction, and α the address of the call instruction. The algorithm in figure 4 outlines how the compiler uses the inlining plan to decide if the call instruction should be inlined, and if so, what targets should be inlined at the call site. The inlining plan can be viewed as an "oracle" consulted by the compiler, and the inline context can be viewed as a path in the method's inlining plan from the root to the current call site.

If an inlining plan asks to inline a dynamically dispatched method, our compiler *guards* the inlined body with a conditional test for the correct target. These guards ensure correct program execution in the presence of other targets and dynamic class loading. Guards are not necessary if the compiler can establish that the call has only one target. In Java, all `static`, `final`, `private`, and class initializer methods do not require guards. Additionally, our compiler in some situations devirtualizes calls, eliminating the need for guards.

Other alternatives to ensure safety in the presence of dynamic class loading and dynamic dispatch are possible. For example, a compiler might insert a trap instruction that is triggered when the actual target of the call differs from the predicted target. Trap instructions incur significantly greater misprediction penalties than guards but have slightly better performance than guards in the correctly predicted case. We plan to study the relative performance of guards and traps in future work.

Additional details on the transformation algorithm have been suppressed due to space limitations.

6 Experimental Results

6.1 Methodology

We present experimental results using the Jalapeño Java Virtual Machine [3, 2, 6]. Jalapeño uses a *compile-only* methodology for running Java code; the system translates all Java bytecodes to native machine instructions, with no interpreter. A key feature of Jalapeño is that the entire virtual machine, including the thread scheduler, memory management systems, class loader, and compilers, is implemented in Java. With this design, the compiler can inline directly from user application code through standard Java libraries into the virtual machine itself. We will use the inlining algorithms to perform all these levels of inlining, in contrast to a Java compiler that can only inline application code and libraries, but cannot automatically inline into virtual machine routines.

For purposes of this study, we only inline call sites reachable from *main*, and exclude a few call sites into non-deterministic virtual machine systems. For example, we do not inline calls from user code into the thread scheduler subsystem, since a thread switch occurs non-deterministically when triggered by a timer interrupt. Nevertheless, inlining within the thread scheduling subsystem can occur.

To evaluate the inlining algorithms, we employ an experimental methodology with three steps:

1. Run the program with no inlining, instrumented to trace each method call, and write the resulting profile data (node weights and edge weights) to disk.

2. Feed the input from step 1 into one of the various inlining algorithms, and write the resulting output (the *inlining plans* defined in section 3) to disk.

3. Read the output from step 2, use it to compile all methods ahead of time with the appropriate inlining decisions, and run the program.

The programs generated from step 3 are then run to collect timing information.

The performance results in this section were obtained on an IBM F50 Model 7025 with four 166MHz PPC604e processors running AIX v4.3. The system has 1GB of main memory. Each processor has split 32KB first-level instruction and data caches and a 256KB second-level cache.

The Jalapeño system is continually under development; the results in this section use the Jalapeño system as of September 17, 1999. For these experiments, the Jalapeño optimizing compiler performed a basic set of standard optimizations including copy propagation, type propagation, null check elimination, constant folding, devirtualization, local common subexpression elimination, load/store elimination, scalar replacement of aggregates, dead code elimination, and linear scan register allocation. Previous work [6] has demonstrated that Jalapeño performance with these optimizations is roughly equivalent to that of the industry-leading IBM product JVM and JIT. All classes were compiled with a static linkage mode; no dynamic linking occurs during the run. The runs use Jalapeño's non-generational copying garbage collector and 300MB of heap.

We present results from five of the seven SPECjvm98 [9] benchmarks. The profile information was collected when each program was run with the standard medium-size (`-s10`) input [4]. The run times reported are the best wall-clock time from three executions of each benchmark for both medium-size (`-s10`) and large-size (`-s100`) inputs. All classes were pre-compiled; the reported times do not include compilation time, nor

[4] We did not collect results for the two other codes due to resource limitations with our current profiling implementation. We plan to present more complete results in the final version of this paper.

Benchmarks	Run Time w/no Inlining(sec)	call sites	virtual %	calls	virtual %
_201_compress	7.28	2953	61.9%	18,163,854	86.7%
_202_jess	2.33	4771	65.6%	6,188,662	94.0%
_209_db	1.74	3261	62.9%	3,202,215	72.8%
_222_mpegaudio	6.87	3612	64.4%	2,780,496	93.7%
_228_jack	9.91	5117	63.2%	5,643,344	65.0%

Table 1: For each benchmark, the running time with no inlining, the number of call sites, the percentage of call sites that are virtual, the number of calls during program execution, and the percentage of those calls that are virtual, where virtual includes both `invokevirtual` and `invokeinterface` calls. These runs use the medium `-s10` input size.

profiling time. Our goal is to compare the relative benefits of different inlining heuristics for a given limit on code size expansion. Note that these results do *not* follow the official SPEC reporting rules, and therefore should not be treated as official SPEC results.

Our results for CG-N may under estimate the benefit of this algorithm as we did not remove nodes with zero weight. Therefore, in this study, the CG-N algorithm uses a static call graph and may inline a method along an edge from a node with zero weight. In the final submission, our results for CG-N will correspond to what is presented in section 2.

6.2 Performance Measurements

Table 1 summarizes the calling characteristics for the five Java benchmarks studied. These numbers include not just application code, but also calls in Java libraries and Jalapeño Virtual Machine code. It is interesting to note that over 60% of the call sites in all programs correspond to virtual calls [5], and that this fraction is significantly larger for dynamic calls.

Table 2 shows the performance results obtained for our three inlining algorithms as a function of the code expansion limit. This limit represents the number of extra bytes compiled due to inlining divided by the number of bytes compiled with no inlining. For the results presented in this table, the profiling information is "perfect"; that is, we use the same input during the timing runs as used to generate profiling information. Figure 5 graphs the speedup results from Table 2 for 1% and 10% inlining limits, as well as results using a different input set, discussed shortly.

Overall, the results show encouraging speedups due to inlining, and emphasize the difference in quality between the three heuristics. Most strikingly, the DCG-E algorithm, even with a modest code expansion limit of 1%, improves the running time of the five codes by 46%, 57%, 22%, 14%, and 9%. In all cases except for `compress`, the performance of DCG-E with a 1% limit exceeds *any* result using the two other algorithms.

Looking more closely at `compress`, we determined that only a tiny percentage of the static call graph edges account for over 99% of the dynamic calls. For `compress`, any inlining algorithm that inlines these hot edges will obtain good speedup. The three algorithms differ in the code size expansion required to find theses hot edges. The SCG algorithm requires the code size be tripled (200% limit), while the CG-N and DCG-E algorithm require code size expansion of 5% and 1%, respectively.

[5] Virtual calls include both *invokevirtual* and *invokeinterface*.

Program	Limit (%)	SCG			CG-N			DCG-E		
		Time (s)	Speed up	% calls inlined	Time (s)	Speed up	% calls inlined	Time (s)	Speed up	% calls inlined
compress	1	7.14	1.02	11.30	5.74	1.27	86.72	4.98	1.46	99.97
	5	6.58	1.10	35.70	4.98	1.46	99.98	5.06	1.44	99.99
	10	6.08	1.20	67.89	5.12	1.42	99.99	5.19	1.40	99.99
	25	5.51	1.32	86.73	4.91	1.48	99.99	5.07	1.44	99.99
	50	5.24	1.39	93.36	4.98	1.46	99.99	5.09	1.43	99.99
	100	5.30	1.37	93.36	5.04	1.45	99.99	5.00	1.45	99.99
	200	5.04	1.45	99.99	5.05	1.44	99.99	5.16	1.41	99.99
jess	1	2.32	1.00	4.22	2.06	1.13	64.39	1.48	1.57	90.69
	5	2.24	1.04	17.37	1.77	1.32	71.84	1.43	1.63	96.53
	10	2.12	1.10	29.19	1.72	1.36	73.60	1.39	1.68	98.01
	25	2.09	1.11	34.58	1.71	1.36	75.66	1.71	1.36	99.38
	50	2.09	1.11	36.60	1.69	1.38	76.54	1.74	1.34	99.59
	100	1.93	1.21	62.50	1.66	1.40	77.11	1.77	1.32	99.68
	200	1.93	1.21	63.42	1.64	1.42	77.21	1.77	1.32	99.71
db	1	1.80	0.97	10.44	1.49	1.17	76.9	1.43	1.22	88.20
	5	1.70	1.02	28.26	1.47	1.18	89.57	1.44	1.21	97.55
	10	1.69	1.03	31.71	1.47	1.18	92.13	1.42	1.23	99.37
	25	1.67	1.04	41.41	1.47	1.18	93.90	1.43	1.22	99.90
	50	1.48	1.18	91.68	1.47	1.18	94.07	1.40	1.24	99.93
	100	1.48	1.18	94.20	1.47	1.18	94.16	1.43	1.22	99.94
	200	1.52	1.14	95.07	1.47	1.18	94.17	1.43	1.22	99.95
mpegaudio	1	6.92	0.99	0.71	6.86	1.00	49.97	6.01	1.14	78.97
	5	6.90	1.00	2.53	6.81	1.01	62.16	6.04	1.14	93.54
	10	6.93	0.99	36.53	6.84	1.00	66.36	6.03	1.14	96.58
	25	6.90	1.00	37.95	6.86	1.00	67.75	5.98	1.15	99.14
	50	6.61	1.04	40.64	6.92	0.99	67.99	5.98	1.15	99.75
	100	7.26	0.95	49.59	6.91	0.99	68.15	6.02	1.14	99.89
	200	7.28	0.94	52.65	6.94	0.99	68.17	5.98	1.15	99.95
jack	1	9.89	1.00	17.60	9.52	1.04	62.22	9.11	1.09	73.45
	5	9.80	1.01	28.37	9.45	1.05	80.20	8.78	1.13	91.00
	10	9.76	1.02	39.59	9.36	1.06	85.00	8.78	1.13	94.81
	25	9.57	1.04	63.03	9.37	1.06	89.37	9.07	1.09	98.33
	50	9.62	1.03	76.45	9.36	1.06	90.42	9.00	1.10	98.95
	100	9.81	1.01	76.64	9.56	1.04	90.96	9.28	1.07	99.28
	200	10.13	0.98	78.99	9.93	1.00	91.27	9.45	1.05	99.41

Table 2: For each benchmark and a code expansion limit, the execution time, the speedup over no inlining, and the percentage of dynamic calls inlined for each inlining heuristic. Profile data and timing measurements were both collected with medium-size inputs.

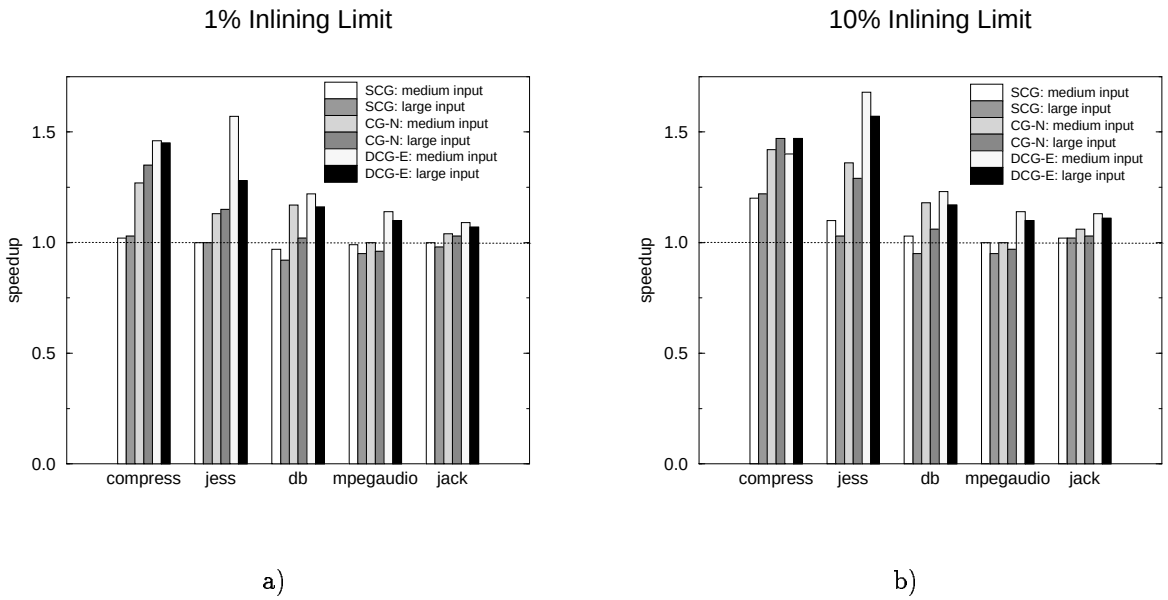

Figure 5: Speedups due to inlining with an a) 1% inlining size limit, and b) 10% limit. Profile data were collected for medium-size inputs. Time measurements were collected for medium-size and large-size inputs, as indicated.

Surprisingly, across all five codes, the DCG-E algorithm speedup does not generally improve as we increase the inlining limit. In effect, the 1% limit catches almost all useful inline sites. In fact, the 1% limit DCG-E covers at least 73% of the dynamic calls on all five benchmarks. Furthermore, for all codes, the 10% limit DCG-E covers at least 94% of dynamic calls. This result encourages the use of the DCG-E algorithm for a dynamic compiler, where the inlining limit should be reasonable predictor of compilation time. Intuitively, one should expect to realize over 90% of the potential benefit from inlining without increasing compile-time by more than 10%.

In three of the five benchmarks, CG-N significantly outperforms SCG for inlining limits up to 25%. Of the other two benchmarks, CG-N outperforms SCG slightly for `jack`, and `mpegaudio` realizes little or no speedup for both CG-N and SCG, even though the algorithms inline up to 68% of the dynamic calls. This latter fact demonstrates that the number of dynamic calls inlined does not accurately represent the benefit of inlining. Nevertheless, we are encouraged to find significant speedups even with such a crude benefit estimate for the algorithms.

For all the benchmarks, more detailed profiling information allows significant increases in the number of inlined calls with smaller expansion limits. Although this trend occurs in all the benchmarks, it is most pronounced in `jess` where a 1% code expansion for CG-N has a greater percentage of inlined calls, 64.39%, than 200% code expansion for SCG with 63.4% of the calls inlined; and a 1% code expansion for DCG-E has a greater percentage of inlined calls, 90.69%, than 200% code expansion for CG-N with 77.2% of the calls inlined.

In several cases, we see a degradation due to "over-inlining". For example, the DCG-E algorithm on `jess` suffers with inlining limits of 25% and greater. As noted earlier, register pressure or cache behavior might account for this effect. In addition, in the presence of dynamic dispatch, we must guard virtual calls to preserve safety. Other unexpected compiler interactions may contribute (see, *e.g.,* [8]);

Somewhat surprisingly, a dramatic increase in code size, tripling code size with a 200% expansion limit,

did not have a dramatic impact on performance. In the worst case, `jess` had a slow down of 21.4% when increasing code expansion from 25% to 200% for DCG-E. We conclude that for these programs, a static compiler (where compile-time is not an issue) using post-mortem profile data might reasonably employ a large expansion limit.

Table 3 shows the speedups obtained by running the programs with a different input than used to obtain profiling information. We profiled the codes with the medium input size (`-s10`), and Table 3 shows speedups running with the large input size (`-s100`). Even with "imperfect" profile data, the DCG-E algorithm still shows significant speedups: performance on the the five programs improved by 47%, 57%, 18%, 12%, and 11%, respectively.

In summary, we see that the use of edge weights provides significant benefit over node weights and static heuristics for the set of benchmarks studied. In particular, edge weights can effectively inline high percentages of calls with low percentage of code expansion. Although our simple estimate of benefit led to effective algorithms, it does not accurately predict run-time performance. In future work, we will endeavor to find more realistic estimates of inlining benefit, in the hope of improving on these results. See the next section for a brief discussion of related work along these lines.

Performance results for additional benchmarks will be included in the final version of the paper.

7 Related Work

Several previous works have explored profile-directed inlining, formulating inlining as an optimization based on post-mortem dynamic call graph data. Scheifler [16] introduced the formulation, and noted that the optimization is equivalent to the KNAPSACK problem. Scheifler presented a greedy algorithm which relied on a *constant ratios* assumption to infer expected frequencies of inlined call sites.

Chang et al. [7] report results with profile-directed inlining with the IMPACT optimizing C compiler. Similar to our study, the IMPACT work used only post-mortem dynamic call graph data, and used a greedy algorithm to maximize the number of call sites inlined subject to a fixed space bound. Their algorithm uses profile information equivalent to the dynamic call graph, but unlike ours, used a bottom-up heuristic to choose inline sites. Chang et al. did not inline dynamically dispatched calls.

Ayers et al. [5] present an evaluation of inlining and cloning in the Hewlett-Packard optimizing compilers. When solving the inlining optimization problem using a greedy algorithm, they chose to use a cost function quadratic in the size of the method, to account for quadratic-time algorithms in the back end. This work uses *intraprocedural* profile information at the basic block level, in contrast to the other works which rely on interprocedural information. Ayers et al. did not inline indirect call sites.

Kaser and Ramakrishnan [15] extended Schiefler's work by examining probabilistic variations to the greedy algorithm based on the constant ratio assumption. They report good results compared to the IMPACT algorithm. We plan to test this approach for Java in future work.

While these works estimate the benefit of inlining based on the number of dynamic calls inlined, a few works have attempted to estimate inlining's impact on optimizations. Dean and Chambers [10] present "inlining trials" for a SELF compiler. In this work, the compiler tentatively inlines a call sites, and monitors the resultant effect on optimizations. The compiler caches this result and uses the information to make future inlining decisions. Their work also factors in characteristics of the static call site when making inlining decisions. Dean and Chambers relied on static estimates of method frequencies, but note that their

Program	Limit (%)	SCG		CG-N		DCG-E	
		Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
compress	1	82.8	1.03	63.1	1.35	58.6	1.45
	5	75.9	1.12	57.5	1.48	58.7	1.45
	10	69.8	1.22	57.8	1.47	57.9	1.47
	25	62.8	1.35	58.4	1.45	57.7	1.47
	50	61.0	1.39	59.8	1.42	58.1	1.46
	100	61.8	1.37	59.2	1.43	58.1	1.46
	200	60.0	1.42	60.5	1.40	58.5	1.45
jess	1	49.1	1.00	42.6	1.15	38.1	1.28
	5	48.6	1.01	39.9	1.23	34.4	1.42
	10	47.7	1.03	38.0	1.29	31.1	1.57
	25	46.5	1.05	37.7	1.30	31.6	1.55
	50	46.0	1.06	37.8	1.29	32.0	1.53
	100	43.9	1.11	34.3	1.43	33.7	1.45
	200	45.8	1.07	36.8	1.33	35.9	1.36
db	1	138.9	0.92	125.0	1.02	110.2	1.16
	5	134.0	0.95	120.0	1.06	108.4	1.18
	10	134.6	0.95	121.0	1.06	109.3	1.17
	25	131.6	0.97	119.6	1.07	109.7	1.16
	50	121.8	1.05	120.3	1.06	108.5	1.18
	100	120.3	1.06	120.9	1.06	108.3	1.18
	200	121.4	1.05	120.2	1.06	110.7	1.15
mpegaudio	1	67.3	0.95	66.4	0.96	58.0	1.10
	5	67.0	0.95	65.3	0.97	57.8	1.10
	10	67.0	0.95	65.7	0.97	57.7	1.10
	25	66.4	0.96	66.4	0.96	57.4	1.11
	50	63.7	1.00	66.8	0.95	57.1	1.12
	100	70.2	0.91	66.7	0.95	57.3	1.11
	200	70.3	0.91	67.5	0.94	57.3	1.11
jack	1	84.9	0.98	81.3	1.03	78.4	1.07
	5	84.9	0.98	80.7	1.04	75.4	1.11
	10	83.6	1.00	81.0	1.03	75.6	1.11
	25	82.9	1.01	81.0	1.03	77.3	1.08
	50	83.2	1.00	80.9	1.03	77.4	1.08
	100	85.7	0.98	83.3	1.00	77.1	1.08
	200	89.6	0.93	87.7	0.95	76.7	1.09

Table 3: For each benchmark and a code expansion limit, the execution time and the speedup over no inlining, for each inlining heuristic. Profile data and timing measurements were collected with medium-size and large-size inputs respectively.

approach can easily incorporate profile data. Waddell and Dybvig [17] use a similar approach in the context of an on-line Scheme compiler. Jagannathan and Wright [14] use control-flow analysis to guide top-down static inlining heuristics for Scheme.

While many of these previous works justify using profile data for inlining, few works have tried to evaluate different types of profile data in this context. Grove et al. [13] address this question, evaluating granularity of profile information to predict the targets of dynamically dispatched method calls. This work did not approach inlining as an optimization problem. Grove et al. compare profile information with different levels of context information (we considered only two cases, node and edge counts; they evaluate deeper context). They report that for a set of C++ and Cecil programs, the compiler effectively uses deeper levels of profile context to predict dynamic dispatch targets. Dean et al. [11] report that static class hierarchy analysis improves the predictions, over and above using profile data alone.

8 Conclusions and Future Work

In this paper, we presented a comparative study of static and dynamic heuristics for inlining. We introduced *inlining plans* as a formal representation for nested inlining decisions made by an inlining heuristic. As in past work, we formalized the inlining optimization problem as a variant of the KNAPSACK problem [12]. We used a well-known approximation algorithm for the KNAPSACK problem as a common "meta-algorithm" for the static and dynamic inlining heuristics studied in this paper. We present performance results for an implementation of these inlining heuristics in the Jalapeño dynamic optimizing compiler for Java [6]. Our performance results show that the inlining heuristics studied in this paper can lead to significant speedups in execution time (up to 1.68×) even with modest limits on code size expansion (at most 10%).

A major direction for future work is to evaluate the use of *calling context* information [4] in inlining, and to see if it can lead to better inlining decisions than the DCG-E algorithm. For example, consider a program execution for the example in figure 2 in which the total frequency of 130 for the $D \rightarrow E$ call site gets split into 128 when D is called from A and only 2 when D is called from B. In that case, a smarter inlining decision would be to only make one copy of E in the inlined copy of D in A, and to not make a second copy of E in D itself. Doing so will enable C to be inlined in A as well, thus increasing *totalBenefit* to 458, while maintaining $totalCost = 4$.

Additional possibilities of future work were mentioned earlier in the paper *e.g.,* use of static structural heuristics (inlining leaf nodes, etc.), extension of CG-N with edge weights that are estimated from node weights, use of traps instead of guards for inlining dynamically dispatched calls, more realistic estimates of inlining benefits and costs, and the use of the constant ratio assumption from [15].

Acknowledgments

We thank Brian Cooper for the use of his profiling code that he developed as a 1998 summer student at IBM Research, and Dave Grove and Mike Hind for helpful comments on a draft of this paper. Thanks also to the entire Jalapeño team for helping to build the infrastructure which enabled this study.

References

[1] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[2] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark Mergen, Janice Shepherd, and Stephen Smith. Implementation of Jalepeño in java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[3] Bowen Alpern, Anthony Cocchi, Derek Lieber, Mark Mergen, and Vivek Sarkar. Jalapeño — a Compiler-Supported Java Virtual Machine for Servers. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)*, May 1999. Also available as INRIA report No. 0228, March 1999.

[4] Glen Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997.

[5] Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. In *SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 134–145, June 1997.

[6] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

[7] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for c programs. *Software – Practice and Experience*, 22(5):349–369, May 1992.

[8] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected side effects of inline substitution: A case study. *ACM LOPLAS*, 1(1):22–32, March 1992.

[9] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998.

[10] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, 1994.

[11] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *the 9th European Conference on Object-Oriented Programming*, 1995.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[13] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, October 1995.

[14] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 193–205, May 1996.

[15] Owen Kaser and C.R. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 24:55–72, 1998.

[16] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *CACM*, 20(9), September 1977.

[17] Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. In *4th International Symposium on Static Analysis*, September 1997.