Some New Approaches to Partial Inlining

Bowen Alpern
Lehman College, CUNY
Bowen.Alpern@lehman.cuny.edu

Anthony Cocchi
Lehman College, CUNY
Anthony.Cocchi@lehman.cuny.edu

David Grove
IBM Research
groved@us.ibm.com

Abstract

This paper proposes two novel techniques for partial inlining. *Context-driven partial inlining* uses information available to the compiler at a call site to prune the callee prior to assessing whether the (pruned) body of the callee should be inlined. *Guarded partial inlining* seeks to inline the frequently taken fast path through the callee along with a test and a call to the original method to handle those instances where the fast path is *not* taken.

A fragile implementation of guarded partial inlining is described. An example, very loosely based on a simplified web server, is fabricated. Experimental evidence establishes the superiority of partial inlining over no inlining and over complete inlining *on this contrived example*. We show that these approaches to partial inlining are applicable in situations where previous approaches are not.

Potential effects of the widespread availability of partial inlining on software development are considered.

Categories and Subject Descriptors Software and its engineering [*Compilers; Procedures,functions and subroutines; Parallel programming languages*]

General Terms compiler optimization, inlining

Keywords partial inlining, X10

1. Introduction

Compilers and virtual machines can do more to help programmers achieve high performance without distorting the natural structure of their programs.

Inlining is a compiler optimization that replaces a call to a method with the (suitably mangled) body of the method. It is well known that inlining frequently called methods can significantly improve the performance of a program. The *direct* benefit of inlining a call is the elimination of a method call overhead. Additional *indirect* benefits are often realized in the form of opportunities to further optimize the combined code of the caller and inlined call. (The genesis of this work lies in a desire to exploit the existence of such opportunities to help drive the choice of which calls to inline.)

The cost of inlining is *code bloat*: usually inlining a call increases the size of the caller method.[1] During compilation, bloat is felt in different ways. Typically, a compiler is given a space-budget that limits the final size of a method being compiled. If inlining a method increases the size of the caller, it limits the compiler's ability to inline other methods. It also makes the caller itself a less attractive candidate for inlining into its callers. During execution, code bloat can degrade performance by increasing instruction cache misses or otherwise overflowing fixed-size hardware resources such as branch prediction tables.

Partial inlining is a technique for realizing (much of) the benefit of complete inlining with less bloat. Here, only part of the called method is inlined. Accessing the uninlined portion still requires a method call, but the call is not made if the execution stays within the inlined portion. Previous work on partial inlining (see section 5) has been done in a context (such as a JIT compiler) where the compiler has profile information that allows it to conclude that certain paths through a method will be *hot*, often taken, while others will be *cold*, seldom taken.

The long-term goal of this research is to enable partial inlining in cases where such information is either unavailable or given only in terms of hints (annotations) supplied by the programmer. In particular, our ultimate aim is to exploit information at the call site to prune the body of the callee with a view to evaluating the pruned body as a candidate for inlining.

The *lazy initialization* idiom, which appears in a wide variety of guises in programs, provides a prime example illustrating the *context-driven partial inlining* approach. Consider the Java method in Figure 1 for adding an edge to a directed graph.

If the compiler could establish at the call site that the `neighborhood` map contains a non-null entry for `v` then the `addEdge` method could be inlined *without the if-block*. For

[1] If the inlined size of a method body is no bigger than the call to the method, there is no downside to inlining the call. Such methods should always be inlined. This paper will not consider them further.

```
final void addEdge(Vertex v, Vertex w) {
  Set<Vertex> neighbors = neighborhoods.get(v);
  if (null == neighbors)  {
    neighbors = new HashSet<Vertex>();
    neighborhoods.put(v, neighbors);
  }
  neighbors.add(w);
}
```

Figure 1. A motivating example of the Lazy Initialization Pattern

```
final void addEdge$$(Vertex v, Vertex w) {
  Set<Vertex> neighbors = neighborhoods.get(v) {
  if (null == neighbors)  {
    addEdge(v, w);
  } else {
    neighbors.add(w);
  }
}
```

Figure 2. Synthetic addEdge method created for use by Guarded Partial Inlining

instance, suppose the caller was an input routine reading a graph for a file in a format where each vertex was followed by a list of its neighbors. The compiler might emit code with a call to `addEdge` for a vertex's first neighbor followed by a loop of partially inlined calls for the remaining neighbors.

In other circumstances, the compiler might be unable to determine that the `if-statement` is always unnecessary but be able infer (perhaps with the help of a hint from the programmer) that it is likely to be infrequently taken. In this case, a related technique, *guarded partial inlining*, would cause the synthetic `addEdge$$` method, shown in Figure 2, to be inlined in place of the call to `addEdge`. This synthetic method calls the original `addEdge` if its slow path is required. Otherwise, it performs the fast path in-line.

Guarded and context-driven partial inlining are closely related but different techniques. Guarded partial inlining inlines not only the fast path but the test and a call to the original code as well. If the compiler can deduce that the test will always hold at a call site (this is precisely the precondition for context-driven partial inlining), then other optimizations (e.g., constant propagation) will eliminate the test and the call, leaving code identical to that produced by context-driven partial inlining. However, the compiler can perform guarded partial inlining where context-driven partial inlining is *not* warranted. This is both a blessing and a curse. The compiler might be tempted to perform guarded partial inlining under circumstances where the fast path is very rarely taken. Thus, reliance on programmer hints seems to be required for guarded partial inlining, but not for context-driven partial inlining. For the purposes of this paper, the pertinent difference between the two techniques is that guarded partial inlining admits an easy prototypical implementation in the X10 compiler while supporting context-driven partial inlining would require implementing infrastructure in that compiler for keeping track of state information that it does not currently maintain (see sections 2 and 7). Consequently, a prototype of guarded partial inlining will be used in this preliminary exploration of the efficacy of both techniques.

The next section describes our prototypical implementation of guarded partial inlining. Section 3 explains an example loosely based on a simple web server. Section 4 establishes the efficacy of guarded partial inlining on this example. Section 5 relates the two approaches presented here to other approaches to partial inlining. Section 6 outlines paths to robust implementations of both guarded and context-driven partial inlining. Section 7 speculates that general availability of partial inlining would have a positive impact on software development. Section 8 concludes.

2. A Prototypical Implementation

To implement our prototype, we decided to build upon the existing inlining infrastructure in the X10 compiler. This choice may seem somewhat quixotic because of the relative immaturity of the X10 optimizer (the primary focus of the X10 language [6, 12] being on parallel and distributed computation) and the lack of extensive suites of X10 benchmarks. However, the compiler is open source and its internals are well-known to two of the authors. Furthermore, it is based on the Polyglot [11] extensible compiler framework, which makes it relatively easily to add new optimization passes. It also has mechanisms in place for processing program annotations and for annotation-directed inlining, both of which we will exploit.

The X10 compiler is a source-to-source compiler: it translates X10 programs to either Java or C++. The resulting programs are then "post-compiled" to Java bytecodes or binary code by invoking a Java or C++ compiler respectively. The option to compile X10 to C++ enabled us to simplify our experimental evaluation by avoiding the possibility of the JVM's just-in-time compiler doing additional profile-directed inlining, thus obscuring the impact of the inlining being done by the X10 compiler.

This prototype is intended as a mechanism for establishing the efficacy of guarded partial inlining. It does not purport to be robust. It relies heavily on the programmer to know what she or he is doing and to tell the compiler what to inline. Annotations are required both on the definition of a method to be partially inlined and on the calls to these methods. (This allows us to control exactly which calls are partially inlined.)

Two passes have been added to the X10 compiler. The first of these walks the abstract syntax tree representations of X10 classes looking for **method definitions** with the appropriate annotation. When such a method is found, a new method (with a suitably mangled name) is created and added to the class. The new method is an exact copy of the old one except that the else branch of the first conditional statement

```
1  def caller {
2      prelude;
3      @PartialInline callee();
4      postlude;
5  }
6  private @PartialInline def callee() {
7      prologue;
8      if (test) {
9         fastpath;
10     } else {
11        slowpath;
12     }
13     epilogue;
14 }
```

(a) The caller and callee before inlining.

```
1  def caller () {
2      prelude;
3      prologue;
4      if (test) {
5          fastpath;
6      } else {
7          callee();
8      }
9      epilogue;
10     postlude;
11 }
```

(b) The caller after inlining.

Figure 3. Guarded partial inlining pseudocode.

in the method body is replaced by a call to the original method.[2] The new method is marked with an annotation that will force the method to be inlined in a later phase of compilation.

The second new pass looks for appropriately annotated **method calls**. Such calls are rewritten to call the corresponding partial inline method with the mangled method name that was created in the former pass and are annotated to be inlined unconditionally. The standard inlining pass of the X10 optimizer does the actual inlining.

Given the skeletal X10-like pseudo-code[3] in figure 3(a), the X10 compiler will emit something like the pseudocode in figure 3(b) for the caller. This transformation is *invalid* unless `prologue`, `test`, and `epilogue` are idempotent. It is *inexpedient* unless all three and `fastpath` are small (or could be optimized away).

3. A Contrived Example

The purpose of this example is to present a situation in which guarded partial inlining is in some way better than both no inlining and complete inlining. To demonstrate that guarded partial inlining is better than complete inlining it suffices to show that the former produces less code bloat. Ideally, performance measurements would establish the superiority of the partially inlined (over uninlined) code. However, a myriad of factors contribute to the performance of parallel programs. It is difficult to tease out the component attributable to partial inlining.

The X10 compiler currently has two back-ends. It can either produce Java or C++ source code as its target. The Java code will run on a JVM with its own JIT compiler. As previously stated, it is difficult to determine whether to attribute any difference in performance to the transformation being measured, or to the JIT compiler. For this reason, our measurements are restricted to the C++ back-end.

The example is an abstraction of a simplified client-server architecture. Various *clients* generate work and put it on an *order queue*. A single *manager* takes work from the order queue and a *worker* from a *worker queue* and dispatches the worker on the work.

Figure 4 shows the initial version of the manager's X10 code. The `async` statement at line 6 creates a new X10 *activity*, a very lightweight thread for worker `w` to process order `p`. The dequeue method starting at line 9 shows the code for dequeuing work from the order queue (dequeuing from the worker queue is similar). The dequeue code exploits the knowledge that, while many activities may add items to the order queue, only one activity removes them. Thus, no synchronization is required unless there is only one item in the queue.

In the expected case, when there is more than one order on the order queue, removing an item entails a test, two assignments, and a return. The *unexpected* case (the `else` clause) is much worse than it looks because `when` and `atomic` are somewhat expensive synchronization operations in X10 generating many lines of C++ (or Java) code. Guarded partial inlining inlines the test and the two assignments. The return is naturalized into an already existing assignment of the result of the method to a local variable.[4]

[2] This is a minimal mechanism sufficient to allow us to explore the consequences of partial inlining. It is *unsafe*, an unwitting (or unscrupulous) programmer could use it to annotate a program in such a way as to change the program's semantics. To be absolutely clear, the authors do not advocate the inclusion of unsafe annotations in programming languages. We do feel that such annotations may be legitimately used in a research prototype.

[3] This pseudocode ignores method receivers and parameter passing to and returning values from methods. Also ignored is the problem of alpha renaming of local variables and formal parameters in the inlined method body. Such issues are important to get right in any implementation of inlining, but present no special problems for partial inlining. This paper does not consider them further.

[4] A call to the original method is also inlined as the else branch of the test. In the current implementation, this results in a redundant test (once at the

```
1  public def manage(count : int) {
2    for (n in 1..count) {
3      val p = @PartialInline orderQ.dequeue();
4      val w = @PartialInline workerQ.dequeue();
5      async w.work(p);
6    }
7  }
8
9  @PartialInline public def dequeue () : T {
10   if (null!=head && head!=tail) {
11     val item = head.item;
12     head = head.next;
13     return item;
14   } else {
15     if (null==head)
16       when (null!=head);
17     atomic {
18       val item = head.item;
19       head = head.next;
20       if (null == head)
21         tail = null;
22       return item;
23     }
24   }
25 }
```

Figure 4. Initial X10 code for the manage and dequeue methods from the example program.

```
1  public def manage(val numOrders : int) {
2    for (n in 0..(numOrders/buf.size-1)) {
3      for (i in 0..(buf.size-1)) {
4        buf(i) = @PartialInline orderQ.dequeue();
5      }
6      val w = workerQ.dequeue();
7      async w.work(buf);
8    }
9  }
10
11 @PartialInline public def dequeue () : T {
12   val item:T;
13   if (null!=head && differ(head, tail)) {
14     item = head.item;
15     head = head.next;
16     return item;
17   } else {
18     if (null==head)
19       when (null!=head);
20     atomic {
21       item = head.item;
22       head = head.next;
23       if (null == head)
24         tail = null;
25     }
26     return item;
27   }
28 }
```

Figure 5. Optimized X10 code for the manage and dequeue methods from the example program.

Thus, if the expected case predominates, guarded partial inlining will eliminate two method call overheads from each iteration of the manager's inner loop.

Unfortunately, inlined or not, this code does not perform as well as it should.

One problem has to do with returning a value from inside an `atomic` block. In the current X10 C++ back-end, this entails creating a very large `Finalization` object, which is expensive to create and which could force a very expensive garbage collection.[5] This return is on the slow path through `dequeue` so it shouldn't cause any problems, but it is easy to eliminate by moving the return outside the block.

Another problem is that `head!=tail`, the pointer inequality test, is more expensive than it appears (or should be). It entails a virtual function call.[6] For the purposes of our experiments we eliminated this call by using X10's `@Native` annotation to replace the inequality test with a call to an inlined C++ function that just compares the pointers directly.

A final problem concerns the `async` statement in the manager inner loop. *Activity* (thread) creation is supposed to be extremely lightweight in X10. It is significantly less expensive than a normal thread creation and is implemented entirely at user level. However, it does entail several method calls encompassing hundreds of instructions. In short, its costs dwarf the savings we are hoping to measure. The remedy here is to batch up orders processed by a worker amortizing `async` (and worker) overhead across multiple order queue dequeues.[7] Figure 5 shows the example code of Figure 4 after it has been adjusted to avoid these problems.

4. Experimental Results

The example in the previous section is a concurrent application, and the benefits of guarded partial inlining should be realized during parallel execution. However, parallel execution makes it exceedingly difficult to accurately *measure* such benefits. Moreover, the benefit itself in this case, the call overhead for a non-virtual method, should not be expected to be very big, merely a handful of machine instructions.

In order to try to obtain accurate measurements, we tried to eliminate as much interference with the execution of the manager inner loop as we could. A single client was constrained to run to completion before the manager started. Limiting the number of X10 threads to one should guarantee that the manager will complete before any of the workers begin. (In addition, we constrain the workers to ignore the

call site and once in the call). Section 6 discusses how the redundant test could be eliminated.

[5] This is a performance bug in the X10 2.2.3 compiler and runtime that was identified as a result of this work and has been fixed for the upcoming X10 2.3.0 release.

[6] The inequality of boxed structs in X10 requires checking the fields of the structs. This should not matter here since `head` and `tail` are known to be instances of a class (and thus not structs), but the current X10 compiler misses this optimization. As a result a virtual call is made whose body is cheap, but the function call overhead remains.

[7] A similar strategy is used by the clients to enqueue multiple orders in order to amortize enqueue synchronization costs.

work they are given and return immediately.) To minimize the possibility of garbage collection interfering with timing of the manager (and thus, to assure more accurate and repeatable results), a garbage collection is forced just before this timing starts.

We measure the size and performance of three different compilations of the application: **vanilla** has no inlining of calls in the manager's manage() method to the `dequeue` method on the order queue; **complete** inlines all such calls completely; and **partial** uses guarded partial inlining on them. Otherwise, the compilations are identical and the X10 compiler was invoked with the recommended [16] -O and -NO_CHECKS flags.[8]

Minimum, average, and maximum accumulated inner-loop times of the manage() method for fifteen trials of each flavor (after two discarded "warm up" trials) on $20,000,000$ orders are reported in Table 1. The final column of the table shows the size of the x86 object code for these methods. The experiments were run on a machine with two Intel Xeon E7530 Nehalem processors. Each processor has six 2-way SMT cores running at 1.87 GHz sharing a 12 MB L3 cache. The machine is configured with 16 GB of memory. The machine was running Red Hat Enterprise Linux 6.3.

	min	avg	max	size
vanilla	0.294	0.298	0.299	351
partial	0.284	0.287	0.289	399
complete	0.287	0.288	0.290	1007

Table 1. Accumulated time (in seconds) of the manage() method of manager's inner-loop with $20,000,000$ orders and the size (in bytes) of the method.

Both partial and complete inlining show a modest but measurable (about 3%) improvement in performance. Partial inlining achieves this benefit at considerably less cost in code bloat.

5. Related Work

Inlining is a fundamental optimization technique applied in almost all modern optimizing compilers, therefore there is an extensive body of previous research and practical experience. We refer the interested reader to surveys such as Arnold et al. [1] for broader coverage; here we will restrict our attention to previous work in partial inlining and in exploiting static information in the caller to obtain better cost-benefit estimates for the inlined callee.

Partial Inlining Previous work in partial inlining has primarily relied on profile information or static heuristics to distinguish between the frequently executed portions of the callee method, which should be inlined, and the infrequently executed portions, which should not be inlined. In the Self-91 system, the compiler heuristically applied *deferred compilation* to avoid generating code for uncommon branches [5]. In the presence of inlining, deferred compilation has a similar effect as partial inlining: the inlined code is smaller since code is not generated for the unexpected control flow paths. Whaley [15] applied the ideas of deferred compilation in a JVM using dynamic profile data to identify rarely executed basic blocks that should not be inlined. Zhao and Amaral [17] approached partial inlining by implementing a procedure outlining transformation[9] that was run before the normal inlining pass in the ORC compiler. Recently Lee et al. [10] proposed a new algorithm for identifying the portion of a callee method to be partially inlined by using control flow edges leading to frequently executed procedure returns as seeds to the sub-graph construction algorithm.

Partial inlining can also be viewed as a mechanism for allowing traditional procedure-based compilation systems to achieve many of the benefits of more flexible trace-based compilation. The Dynamo system [2] pioneered many of the key concepts in dynamic, trace-based compilation. Inspired by Dynamo, Bruening and Duesterwald [4] did an early study on the applicability of trace-based compilation to Java and confirmed that confining optimization decisions to procedure boundaries does result in sub-optimal selection of compilation units. A number of subsequent systems explored region-based and trace-based just-in-time compilation [8, 9, 14].

Inlined Size Estimation A critical step in the cost-benefit decision of inlining is constructing an estimate of the cost: how big will the callee method be when inlined at a particular call site? Many compilers do this estimation based purely on characteristics of the callee method: they do not take into account how any information they may have available about the actual parameters at a specific call site could influence the optimization of the callee method when it is inlined in a specific context.

One notable exception is the inlining trials system of Dean and Chambers [7]. The key innovation of their system was to enable the compiler to learn how static information available at a call site would impact a callee method, thus yielding a cost-benefit decision tailored to the call site and allowing much larger callee methods to be considered as potential inlining candidates. In practice, their system would select larger than normally allowed callee methods to be inlined at exactly those callsites where the static information available about the actual parameters would enable the elimination of significant portions of the inlined callee method. This is the same effect we want to achieve in context-driven partial inlining, but we propose to approach it in the opposite direction by analyzing callee methods to determine what static information would be needed in the caller to profitably enable partial inlining of the callee.

[8] -O enables the X10 optimizer and also causes the g++ compiler to be invoked with -O2 -finline-functions. -NO_CHECKS disables runtime checking for null pointers, divide by zero, and out-of-bounds array accesses

[9] *Outlining* is the dual of inlining: it replaces a block of code with a procedure call to a new method that contains the original code.

```
def caller () {
    prelude;
    prologue;
    if (test) {
        fastpath;
        epilogue;
        postlude;
    } else {
        newCallee();
    }
}
def newCallee () {
    slowpath;
    epilogue;
    postlude;
}
```

Figure 6. Potential results of aggressive guarded partial inlining of the pseudocode from Figure 3(a).

The Jikes RVM optimizing compiler also heuristically adjusts its estimate of the inlined size of the callee method based on static information known about the actual parameters of the call. Sewe et al. [13] describe the original heuristics used by Jikes RVM and how they can be improved by using the dynamic call graph.

6. Future Work

Our prototype implementation of guarded partial inlining is both unsafe and unnecessarily limited.

The requirement that the call to a guarded partial inline method be annotated could be made optional. The X10 compiler can determine whether the declaration of the called method has the requisite annotation.

The prototype assumes that the evaluation of the test (and any prologue or epilogue code) in the method to be inlined will be idempotent. At a bare minimum, this assumption should be checked.

It would be better to avoid the redundant test altogether by crafting a new method to be called instead of the original in the event that the test is false. This new method would omit the test along with any prologue and/or epilogue code that would be inlined at the call site.

Guarded partial inlining achieves the direct benefit of inlining (in the expected case) but only gets some of the indirect benefits because the inlined call is wrapped in a context that handles the possibility that the test might fail. Although it is able to optimize the callee for the specific context it is inlined in the caller, it is unable to optimize subsquent code in the caller based on information discovered from the inlined callee because the non-inlined call introduces a merge in the dataflow information. To achieve these downstream indirect benefits, it would be necessary to move the code following the call site into both arms of the `if` statement evaluating the test. This approach would be particularly attractive if the code along the else branch could be encapsulated in a call to a synthesized out-of-line method.

Figure 6 shows what might be achieved by such aggressive guarded partial inlining. The indirect benefits of partial inlining are realized when the compiler optimizes

```
fastpath; epilogue; postlude;
```

under the assumption that test must hold. If such benefits are significant enough, the caller could be come a candidate for guarded partial inlining.

The prototype also assumes there is a unique fast path through the method to be inlined, that first (top-level) `if` statement distinguishes that path from the slow path(s), and that the fast path is the `then` branch of that `if` statement. These assumptions are unnecessarily restrictive. The `@PartialInline` annotation of a method declaration could take a path as an argument. The path could be encoded as a sequence of boolean values indicating the values to be assumed for successive boolean expressions encountered by the compiler on the method.[10]

The brute force approach to achieving context-driven partial inlining is surely unworkable. In principle, one could at each call site re-compile the callee in the light of information available at the call site. A scheme that cached the results of such re-compilations and reused them for call sites where the available information was "similar" would cut down on the cost, but probably not enough to make this approach practical.

A different approach would preprocess the callee, identifying information that could substantially simplify it. Such information could be structured as a predicate on its formal parameters (including its `this` parameter) along with any statically available information. Once obtained, such predicates could be used for either context-driven or guarded partial inlining. How to derive such predicates is the major open research question of our work.[11]

To achieve context-driven partial inlining, when the compiler deduces that one of the predicates associated with the callee at a specific call site must be true of the arguments to the call, a version of the callee tailored under the assumption of this predicate would be inlined. A lazy cache could be used to map predicates to the version of the method tailored under the assumption that the predicate holds.

Heretofore, this discussion has blithely assumed that the compiler is able to determine whether a predicate must hold at a particular call site. This is, of course, an undecidable problem. A compiler can only approximate its solution. How good an approximation an optimizer can provide depends in part on the infrastructure it provides for maintaining information about the program under compilation.

[10] Note that multiple annotations could associate multiple partial inline paths with a method. If this were the case, some annotation mechanism at the call site would be needed to determine which path(s) to partially inline.

[11] As an interim measure, interesting predicates could be supplied as boolean expression parameters to annotations. Multiple annotations could associate different partial inline predicates with the same method.

The X10 compiler has potential for providing a powerful infrastructure to explore these ideas, but there are significant missing pieces that must be completed to fulfill this potential. On one hand, the X10 type system maintains useful compile-time information about variables (and constants), such as the rank of an array and whether a reference is known to be non null. On the other, there is currently no mechanism in the X10 compiler to keep track of state information that must hold at a particular point in a program. For instance, at the first position in the `then` branch of an `if` statement, the condition of the `if` must hold, but the X10 compiler currently does not keep track of this information. As a practical matter, implementing context-driven partial inlining in X10 would entail developing a subsystem to keep track of such information.

The primary downside to doing compiler optimization research in X10 is the lack of competitive benchmarks. A clear picture of the potential benefits of context-driven or guarded partial inlining will not emerge until these techniques have been implemented in either a C/C++ or Java compiler for which more extensive benchmark suites, such as SPECInt or DaCapo [3], are available. While working on inlining in Jikes RVM, we observed that a few of the DaCapo benchmarks do have frequently called methods that are too large to be selected as inlining candidates, but that would be profitable to partially inline. It would be interesting to investigate whether or not our approach to partial inlining would be able to successfully optimize these call sites and what impact that would have on overall performance.

7. Discussion

Why should anyone care about partial inlining? After all, the performance improvement reported in section 4 was not very impressive. Furthermore, a programmer interested in obtaining the benefits of inlining at the reduced costs of partial inlining can easily achieve the same result by manually refactoring the source code.

In the first place, we only measured the direct benefits of guarded partial inlining. Indirect inlining benefits can be at least as substantial [7]. As indicated in the previous section, achieving such benefits from guarded partial inlining should be difficult but not impossible. However, achieving indirect benefits from context-driven partial inlining should be straightforward.

The more important point concerns refactoring source code to obtain better performance.

Within living memory, there were programmers who prided themselves on their ability to write more efficient assembly code than could be produced by an optimizing compiler. Such programmers have largely gone the way of the punch card and the rotary dial telephone. Modern performance programmers rely on their ability to trick a compiler into producing the machine code they desire. Refactoring source code to obtain better performance is their stock in trade.

We contend that the concern for performance continues to exert a deleterious influence on the clarity and maintainability of commercial software. We further contend that it ought to be a goal of compiler writers (and language designers) to reduce that influence *without compromising the performance programmer's ability to write high-performance source code*.

Performance programmers recognize the costs in clarity and maintainability in refactoring code to obtain better performance. When the expected benefit is great enough, they will put up with these costs. However, if the expected benefit is less great programmers will be tempted to forgo it. Easy access to partial inlining, using annotations say, would allow programmers to realize the benefit without incurring the cost.

For example, arrays in X10 are implemented in the standard library. Since arrays are considered to be performance critical, the code implementing them has been carefully crafted to match the capabilities and limitations of the X10 optimizer. In X10, arrays are more general than in C or Java. The underlying index space for X10 arrays can be multidimensional. Along each dimension, it can start (and end) at arbitrary integer values. It may also have gaps in it. In the general case, a method call verifies that an array index is valid on every array access. A `Rail` is an X10 array whose index space is one-dimensional, zero-based, and contiguous. An array index operation in X10 gets translated to an inlined call to a method that checks to see if the array is a `Rail`. If it is, the necessary bound checks are performed in-line. Otherwise, a general index verification method is called. The array index method (which bears an `@Inline` annotation) *is* the guarded partial inlining of the general index verification method. The code implementing the array class would be a little clearer if the latter method bore an `@PartialInline` annotation and the former did not exist.

Ubiquitously available partial inlining might, or might not, significantly improve the performance of existing code. It would certainly allow performance-obsessed programmers to obtain high-performance programs with less distortion to the natural shape of their source code.

8. Conclusion

This paper presents two new approaches to partial inlining. These approaches differ from previous ones in that they do not assume that the compiler has a priori runtime (or profile) information as to which paths through the method to be partially inlined are *hot*. *Guarded partial inlining* relies on programmer hints to identify candidate methods (and their hot paths). It causes a test and the purported hot path to be inlined along with a call to the original method in the case that the test fails. *Context-driven partial inlining* uses information available at a call site to customize a version of

the callee to be inlined. Guarded partial inlining becomes equivalent to context-driven partial inlining precisely when the compiler can deduce that the guarded partial inlining test must hold.

An admittedly fragile prototype of guarded partial inlining is described along with a somewhat contrived example. Completely and partially inlined versions of this example are observed to run slightly faster than an uninlined version. The partially inlined version is slightly bigger than the uninlined version but dramatically smaller than the completely inlined version. Plans for a more robust implementation of guarded partial inlining along with context-driven partial inlining are presented.

Finally, it is argued that pervasive availability of partial inlining would have a positive impact on how real world programs are written.

References

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349303. URL `http://doi.acm.org/10.1145/349299.349303`.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167488. URL `http://doi.acm.org/10.1145/1167473.1167488`.

[4] D. Bruening and E. Duesterwald. Exploring optimal compilation until shapes for an embedded just-in-time compiler. In *Proceedings of the 2000 ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, Dec. 2000.

[5] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 1–15, New York, NY, USA, 1991. ACM. ISBN 0-201-55417-8. doi: 10.1145/117954.117955. URL `http://doi.acm.org/10.1145/117954.117955`.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094852.

[7] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, LFP '94, pages 273–282, New York, NY, USA, 1994. ACM. ISBN 0-89791-643-3. doi: 10.1145/182409.182489. URL `http://doi.acm.org/10.1145/182409.182489`.

[8] A. Gal, C. W. Probst, and M. Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8. doi: 10.1145/1134760.1134780. URL `http://doi.acm.org/10.1145/1134760.1134780`.

[9] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 246–256, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL `http://dl.acm.org/citation.cfm?id=2190025.2190071`.

[10] J.-P. Lee, J.-J. Kim, S.-M. Moon, and S. Kim. Aggressive function splitting for partial inlining. In *Proceedings of the 2011 15th Workshop on Interaction between Compilers and Computer Architectures*, INTERACT '11, pages 80–86, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4441-0. doi: 10.1109/INTERACT.2011.14. URL `http://dx.doi.org/10.1109/INTERACT.2011.14`.

[11] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for java. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00904-3. URL `http://dl.acm.org/citation.cfm?id=1765931.1765947`.

[12] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification. `http://x10.sourceforge.net/documentation/languagespec/x10-223.pdf`, 2012.

[13] A. Sewe, J. Jochem, and M. Mezini. Next in line, please!: exploiting the indirect benefits of inlining by accurately predicting further inlining. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 317–328, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1183-0. doi: 10.1145/2095050.2095102. URL `http://doi.acm.org/10.1145/2095050.2095102`.

[14] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 312–323, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781166. URL `http://doi.acm.org/10.1145/781131.781166`.

[15] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, lan-*

guages, and applications, OOPSLA '01, pages 166–179, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9. doi: 10.1145/504282.504295. URL http://doi.acm.org/10.1145/504282.504295.

[16] X10 Perf. X10 performance tuning. http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html, 2012.

[17] P. Zhao and J. N. Amaral. Function outlining and partial inlining. In *Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, SBAC-PAD '05, pages 101–108, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2446-X. doi: 10.1109/CAHPC.2005.26. URL http://dx.doi.org/10.1109/CAHPC.2005.26.