Partial Method Compilation using Dynamic Profile Information

John Whaley
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
jwhaley@alum.mit.edu

ABSTRACT

The traditional tradeoff when performing dynamic compilation is that of fast compilation time versus fast code performance. Most dynamic compilation systems for Java perform selective compilation and/or optimization at a method granularity. This is the not the optimal granularity level. However, compiling at a sub-method granularity is thought to be too complicated to be practical.

This paper describes a straightforward technique for performing compilation and optimizations at a finer, sub-method granularity. We utilize dynamic profile data to determine intra-method code regions that are rarely or never executed, and compile and optimize the code without those regions. If a branch that was predicted to be rare is actually taken at run time, we fall back to the interpreter or dynamically compile another version of the code. By avoiding compiling and optimizing code that is rarely executed, we are able to decrease compile time significantly, with little to no degradation in performance.

Furthermore, ignoring rarely-executed code can open up more optimization opportunities on the common paths. We present two optimizations — partial dead code elimination and rare-path-sensitive pointer and escape analysis — that take advantage of rare path information. Using these optimizations, our technique is able to improve performance beyond the compile time improvements.

1. INTRODUCTION

Dynamic compilation systems explore an interesting tradeoff. On one hand, we would like to have code performance that is comparable to static compilation techniques. However, we would also like to avoid long startup delays, long latencies, and slow responsiveness, which implies that the dynamic compiler should be fast.

Many dynamic compilation systems attack this problem by using an interpreter and an optimizing compiler. They begin by interpreting the code, and when the execution count for the method reaches a certain threshold or by some other heuristic, they use the optimizing compiler to dynamically compile the code for the method [42, 38]. Some systems use a fast code generator (baseline compiler) rather than an interpreter [15, 9].

The problem with these systems is that the execution speed of the interpreted or baseline compiled code is significantly worse than that of fully optimized code — typically 30% to ten times slower for baseline compiled code [15, 9] and ten to a hundred times slower for interpreted code [42, 38].

Therefore, we would like to transfer into the optimized version as quickly as possible. However, the optimizing compiler can take a long time to compile. Waiting for the optimizing compiler to finish hurts program startup and response times. Some systems use a multi-level compilation approach, whereby they progress through a number of different compilation "levels", and thereby slowly "accelerate" into optimized execution [3, 38, 43]. However, this simply exacerbates the problem of having a long delay until the program runs at full speed.

Unlike interpretation, compilation takes time that is proportional to the amount of code that is being compiled. Many analyses and optimizations are superlinear in the size of the code (basic blocks, instructions, registers, etc.) This can cause the compilation time to increase significantly as the amount of code being compiled gets large. Compilation of large amounts of code is the cause of undesirably long compilation times.

However, when compiling a method at a time, we do not really have much choice in the matter. Some methods are large to begin with, and others grow large after performing inlining. Even when being frugal and inlining only when it will make a noticeable difference in performance, methods can still grow large, and excessively restricting inlining can significantly hurt performance [42, 4].

The root of the problem is that method boundaries do not correspond to the code that would most benefit from opti-

```
void read_db(String fn) {
  int n = 0, act = 0;
  byte buffer[] = null;
  try {
    FileInputStream sif = new FileInputStream(fn);
    n = sif.getContentLength();
    buffer = new byte[n];
    int b;
    while ((b = sif.read(buffer, act, n-act))>0){
      act = act + b;
    }
    sif.close();
    if (act != n) {
      /* lots of error handling code, rare */
    }
  } catch (IOException ioe) {
    /* lots of error handling code, rare */
  }
}

int read(byte b[], int off, int len) {
  try {
    /* ... */
  } catch (IOException ioe) {
    /* lots of error handling code, rare */
  }
}
```

Figure 1: From `spec db`. Method boundaries do not correspond well to where the time is actually spent.

mizing compilation. Even "hot" methods typically contain some code that is rarely or never executed, but often contain frequently-executed call sites to methods (which in turn, contain their own rarely-executed code.) Figure 1 contains a paraphrased example from the `spec db` benchmark [44]. In this example, the `read_db` method is hot due to the `while` loop that it contains. However, the error handling code guarded by the `if` and the exception handler are rarely executed. Likewise, the call to `read()` is in the loop and therefore a good candidate for inlining. However, `read()` itself contains rarely-executed error handling code. The region that is important to compile — the `while` loop and the hot path in `read()` — have nothing to do with the method boundaries. Using a method granularity causes the compiler to waste time compiling and optimizing large pieces of code that do not matter.

This paper describes a technique to selectively compile and optimize partial methods. This gives us much better control over what we spend time compiling and optimizing. This technique uses dynamic profile data to make a prediction of what code will actually be executed, and selectively compiles and optimizes only that code. If the program actually attempts to branch to code that was not compiled (so-called "rare code"), the system falls back to interpretation or another dynamically compiled version.

We also describe some optimizations that take special advantage of rare path information to improve their effectiveness. We describe two optimizations — partial dead code elimination and rare-path-sensitive pointer and escape analysis — that make special use of rare path information to optimize for the common paths.

The idea of optimizing with respect to "hot" code regions is not new. The Multiflow compiler uses trace scheduling [33] to optimize code with respect to the most frequently executed path. The IMPACT compiler [13] uses superblock scheduling, dividing the program along the most frequently executed paths into single-entry multiple-exit regions called *superblocks*, and performs scheduling on those regions. Dynamo [5] uses dynamic profile information to perform instruction scheduling optimizations on instruction trace fragments. There are two main differences between our technique and prior techniques. The first difference is that our technique operates on arbitrary control-flow graph regions, whereas prior techniques only operate on a single trace. The second difference is that our technique does not generate code for rare traces until they are actually executed at run time.

Our technique allows all analyses and optimizations to successfully ignore the rare code, improving compilation time. We use three techniques to guarantee the safety of optimizing code while ignoring rare code. First, like superblock scheduling [13], we do not allow control flow merges from the rare code back into the optimized path. Second, during analysis we conservatively summarize the effect of the rare code in order to prevent potentially invalid transformations. The third technique is to generate compensation code on the rare paths to undo the effects of some transformations, à la trace scheduling. Because there are no control flow merges from rare code back into non-rare code, our technique avoids one of the major problems with trace scheduling — the complexity of generating the compensation code. Because code is generated for rare paths in an on-demand basis, we can also avoid the explosive code growth associated with tail-splitting in superblock scheduling.

Our technique is able to significantly reduce compilation time while simultaneously improving code quality. Ignoring rare code improves the data flow information on the common paths and thereby opens up more optimization opportunities. Furthermore, ignoring rare code improves cache locality, and the decrease in compile time potentially allows room for the compiler to use more aggressive, time-consuming analyses and optimizations.

1.1 Contributions

This paper makes the following contributions:

- **Partial method compilation technique:** It presents the general technique of partial method compilation, including the necessary modifications to the compiler and the method of falling back to interpretation or to another compiled version.
- **Partial dead code elimination exploiting rare code information:** It gives specific details of a new partial dead code elimination optimization. This algo-

rithm differs from other partial dead code elimination algorithms in that it is an optimistic approach. It efficiently identifies operations that are only necessary in rare blocks and determines when it is safe to move them into the rare blocks.

- **Pointer and escape analysis exploiting rare code information:** It gives details of a pointer and escape analysis algorithm that exploits rare code information. This algorithm is able to effectively ignore the existence of the rare path and fully optimize the common case.

- **Frequency of rare code in programs:** It gives experimental results showing that a significant amount of code in typical programs is never executed or only executed during startup. This differs from previous work in that it ignores program initialization, which has a very different profile than the rest of the execution.

- **Effectiveness of this technique:** It presents experimental results from a preliminary implementation that shows that this technique is very effective in reducing compilation time, and can also improve optimization opportunities on the common paths.

The remainder of the paper is organized as follows. Section 2 describes how we use profile data to distinguish between commonly-executed and rarely-executed code. It also presents experimental results that show the frequency of rare code in programs and the effectiveness of using dynamic profile data to predict rarely-executed code. Section 3 describes the general technique of partial method compilation. It details how to safely perform data flow analysis in the presence of rare blocks, and it describes two optimizations that use compensation code to reverse transformations — partial dead code elimination and pointer and escape analysis. It also describes the method of falling back to the interpreter. Section 4 presents experimental results from our preliminary implementation of this technique. Section 5 compares our work to related work, and we conclude in Section 6.

2. RARE CODE

We begin by providing an overview of our dynamic compilation system. In Section 2.2, we describe the technique that we use to predict rare blocks. Then, in Section 2.3, we provide experimental data showing the frequency of rare blocks in actual programs. In Section 2.4, we evaluate the effectiveness of our prediction technique.

2.1 Overview of dynamic compilation system

Our system is a multilevel compilation system, with an interpreter and two compilation levels. See Figure 2. Transition between the three execution types is triggered by per-method dynamic execution counts. A dynamic execution count is the sum of the number of times that the method was invoked and the number of backward branches taken within the method. Execution begins with the interpreter. Once a method execution count reaches a certain threshold (`t1`), we upgrade it to a compiled version. This version

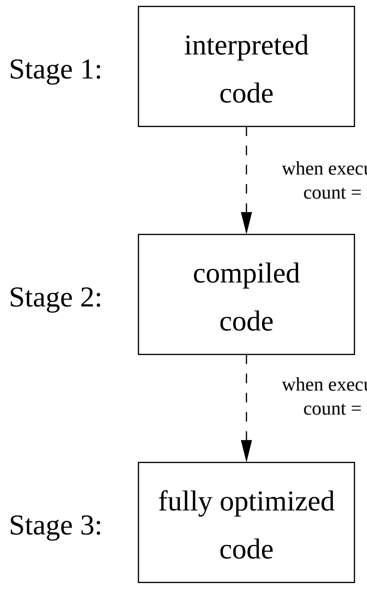

Figure 2: Overview of the multilevel compilation system. Execution transfers from interpreter to compiled code to fully optimized code.

includes some simple optimizations and limited method inlining. Later, after another threshold (`t2`) is reached, we upgrade to a fully-optimized compiled version, which uses more aggressive optimizations and inlining. Note that our system can transition to a newly compiled version of a method at any backward branch. Therefore, even long-running methods can be upgraded.

2.2 Identifying rare code

Our technique for identifying rare code is very simple: we say that any basic block executed during the second stage, the first compiled version, is not rare, and any basic block that is not executed in that version is rare. Because of our use of threshold values to trigger recompilation, this essentially means that we use the set of executed basic blocks in execution numbers `t1` through `t2` of a method as the set of non-rare blocks.

We gather the code coverage information by adding instrumentation code to basic blocks. By using the same threshold values for profile data collection and compilation transition, we can avoid having to make modifications to the interpreter and/or the fully-optimizing compiler, and we do not need to worry about enabling or disabling profiling. These are handled implicitly by transitioning to a new compiled version.

Figure 3 contains code paraphrased from the `addElement` method of `java.util.Vector`. This code causes an element to be added to a `Vector`; when the array backing the `Vector` overflows, the code reallocates a new array twice the size. In method `foo`, we create a `Vector` with an initial capacity of 10 and then begin adding elements to it with `addElement`. Assume that the threshold `t1` is 3000 and the threshold `t2`

	Linpack	JavaCUP	JavaLEX	SwingSet	check	compress	db	javac	mpegaud	mtrt	jack
1	64.20%	54.24%	66.94%	46.24%	50.79%	64.49%	61.05%	56.66%	56.89%	58.19%	65.04%
10	54.94%	45.19%	52.19%	30.07%	15.07%	60.00%	46.82%	55.65%	50.13%	51.88%	64.81%
100	54.94%	43.54%	47.00%	14.10%	4.25%	45.31%	39.33%	53.33%	49.71%	46.13%	61.19%
500	40.12%	38.61%	35.43%	5.32%	2.07%	45.31%	26.97%	49.79%	47.87%	43.03%	55.70%
1000	40.12%	35.76%	33.76%	2.84%	2.07%	43.27%	17.98%	47.45%	46.20%	42.92%	49.41%
2000	40.12%	29.30%	26.96%	1.56%	2.07%	43.27%	17.98%	44.86%	44.70%	42.92%	46.74%
5000	40.12%	26.01%	23.44%	0.59%	1.22%	41.63%	15.36%	39.56%	44.28%	42.37%	38.85%

Figure 4: Table that shows how the percentage of touched basic blocks of executed methods varies by changing the threshold at which we begin measuring. Rows correspond to threshold values; columns correspond to benchmarks.

	Linpack	JavaCUP	JavaLEX	SwingSet	check	compress	db	javac	mpegaud	mtrt	jack
1	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
10	89.51%	82.91%	83.18%	59.06%	26.00%	88.98%	73.78%	97.45%	81.79%	84.85%	99.54%
100	89.51%	82.72%	72.29%	25.19%	7.05%	59.18%	58.80%	92.09%	81.04%	75.88%	91.77%
500	64.20%	75.00%	57.55%	8.86%	3.28%	59.18%	40.82%	84.01%	78.53%	71.35%	81.59%
1000	64.20%	67.22%	55.47%	4.62%	3.28%	55.10%	28.46%	79.08%	76.11%	71.35%	71.37%
2000	64.20%	58.99%	41.59%	2.50%	3.28%	55.10%	28.46%	74.35%	73.02%	71.35%	66.79%
5000	64.20%	53.42%	35.43%	0.93%	1.94%	52.65%	24.72%	65.78%	71.76%	69.69%	55.62%

Figure 5: Table that shows how the percentage of basic blocks in methods exceeding the execution threshold varies as we change the execution threshold. 100% corresponds to all basic blocks in all executed methods.

```
void addElement(Object obj) {
  int oldCapacity = elementData.length;
  int minCapacity = elementCount+1;
  if (minCapacity > oldCapacity) {
    Object oldData[] = elementData;
    int newCapacity = oldCapacity*2;
    if (newCapacity < minCapacity) {
      newCapacity = minCapacity;
    }
    elementData = new Object[newCapacity];
    System.arraycopy(oldData, 0,
            elementData, 0, elementCount);
  }
  elementData[elementCount++] = obj;
}

void foo(...) {
  Vector v = new Vector(10);
  for (i=0; i<5000; ++i) {
    v.addElement(o);
  }
}
```

Figure 3: Code paraphrased from `java.util.Vector`.

is 5000.[1] After the 3000th element is added, execution will change from the interpreter to the instrumented compiled version and it will begin collecting profile data. After the 5000th element is added, execution will transfer to the fully optimized version. However, even though the code to grow the backing array has executed many times before, it will still be marked as rare, because it was not executed between the 3000th and 5000th call (it executed on the 10th, 20th, ..., 2560th call).

[1]This example is only used to illustrate how rare code is determined; it is not intended to be realistic.

There are many techniques to minimize the placement and execution of instrumentation code [6]. Furthermore, because we only care about code coverage, each instrumentation point only needs to be executed once; we can rewrite code or branch targets such that each instrumentation point is disabled after it is executed the first time. Our implementation is incredibly simple — we instrument the targets of conditional forward branches, and do not bother disabling instrumentation points. However, even with this naïve technique, we saw no substantial performance degradation. This is probably due to the fact that a majority of the benchmark execution time was spent executing fully optimized code, which does not contain any instrumentation code.

2.3 Frequency of rare code

Next, we give experimental data that shows that when compiling at a method granularity, a large amount of code is never executed, or only executed during initialization. We use a various set of GUI and non-GUI programs, including the industry-standard SPECjvm98 benchmark suite [44]. The numbers reported here only include the applications themselves — class libraries are not included. See the table in Figure 4 and the corresponding graph in Figure 6. Note that the graphs use a semi-log scale for the x-axis. In the tables, the numbers along the left correspond to the number of initial method invocations or iterations that were ignored (profile start threshold). The benchmarks are listed across the top. Each entry in the table is the percentage of touched basic blocks of executed methods after the given number of invocations or iterations are performed.

The first thing to notice when looking at this data is that even when the profile start threshold is one, less than 67% of the basic blocks of executed methods are ever executed. Furthermore, increasing the threshold causes the percent-

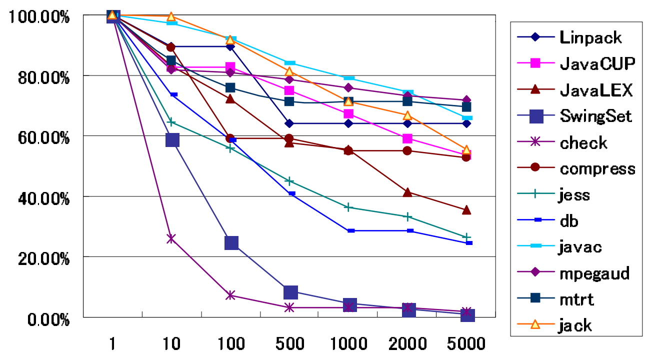

Figure 6: Graph of the data in Figure 4.

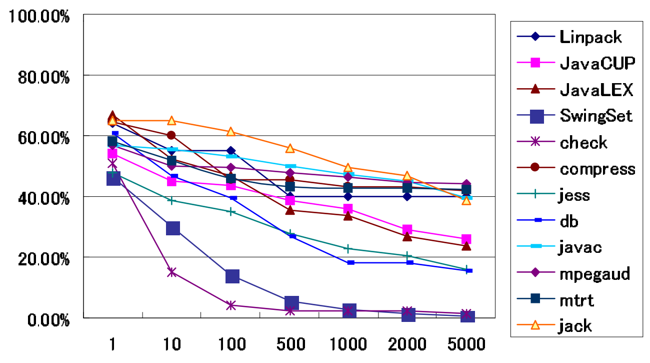

Figure 7: Graph of the data in Figure 5.

age to drop rapidly. As the threshold gets large, the percentage eventually stabilizes. This corresponds to the fact that a large amount of code is only executed during program startup, and the program eventually enters a "steady-state" where the same set of code is being executed.

Now let us compare this to the percentage of basic blocks that would be compiled using a method-at-a-time compilation strategy. The table in Figure 5 and the corresponding graph in Figure 7 show the ratio of the number of basic blocks in methods exceeding the execution threshold to basic blocks in all executed methods. This corresponds to the number of basic blocks that a method-at-a-time compilation strategy would compile. When we compare these numbers to those in Figure 4, we see that the method-at-a-time strategy typically compiles 50% to 100% more basic blocks, even when the threshold is high. Therefore, there is little correlation between method boundaries and executed basic blocks, even for frequently-executed methods.

This data shows that a significant amount of code is never executed after program startup, and that method boundaries are not a good determination of what code actually executes. Therefore, a partial method compilation strategy has the potential for significant savings in compilation time and code size.

2.4 Accuracy of rare code prediction technique

Next, we evaluate the effectiveness of our prediction technique. The table in Figure 8 shows the accuracy of the prediction at various `t2` (end profiling) threshold levels. We used a value of `t1`=2000 for these experiments. The first and second columns give the program name and the `t2` (end profiling) threshold. The third column shows the percentage of blocks that were found to be rare by our profiling in Section 2.2. The fourth column shows the percentage of non-rare blocks that were executed after the `t2` threshold was reached. The fifth column shows the percentage of rare blocks that were executed after the `t2` threshold was reached.

The table in Figure 9 shows the accuracy of the prediction in terms of dynamic execution counts. In this table, the third column shows the percentage of times that a non-rare block was executed, and the fourth column shows the percentage of times that a rare block was executed.

As we can see from Figure 8, the misprediction rate for rare blocks decreases as we increase the amount of profile data that we collect, sometimes decreasing all the way down to zero. The misprediction rate varies a lot between benchmarks, but is in general very low. Figure 9 shows us that rare blocks have extremely low dynamic execution counts compared to non-rare blocks. This means that the execution time of rare blocks should have almost no effect on the total execution time.

3. TECHNIQUE

This section describes the technique of partial method compilation and optimization. Section 3.1 describes the general technique. Section 3.2 argues the correctness of our technique. Sections 3.3 and 3.4 describe two optimizations, escape analysis and partial dead code elimination, that attempt to optimize for the common paths. Section 3.5 describes the process of reconstructing the interpreter state.

3.1 Overview

The general idea of the technique is to replace all entries into rare blocks with stubs that transfer control to the interpreter. The rare blocks are completely removed from the compiler's intermediate representation. Only very minimal changes to the compiler are necessary; optimizations can optionally use rare block information to attempt to better optimize the common paths. At the end of compilation, we store a map corresponding to each interpreter transfer point, which specifies how to reconstruct the interpreter state at that point.

We now describe each step of the process in detail.

1. **Based on profile data, determine the set of rare blocks.**
   The technique of determining rare blocks is described in Section 2.2. The entry points of the rare basic blocks are mapped to abstract program locations, which are

Benchmark	Threshold	Percentage rare blocks	Non-rare blocks executed	Rare blocks executed
Linpack	5000	50.96%	100.00%	26.42%
	10000	56.25%	100.00%	44.44%
	25000	30.77%	92.59%	0.00%
JavaCUP	5000	54.27%	99.74%	5.68%
	10000	41.90%	97.38%	6.82%
	25000	41.71%	97.25%	6.41%
JavaLEX	5000	45.69%	99.40%	26.69%
	10000	42.05%	86.60%	16.22%
	25000	33.25%	85.02%	18.80%
SwingSet	5000	40.09%	99.26%	9.89%
	10000	33.64%	100.00%	2.78%
	25000	19.23%	100.00%	20.00%
compress	5000	41.86%	100.00%	50.00%
	10000	42.28%	100.00%	51.92%
	25000	42.28%	100.00%	51.92%
db	5000	53.03%	100.00%	28.57%
	10000	53.03%	100.00%	28.57%
	25000	41.67%	100.00%	16.00%
javac	5000	45.37%	99.26%	15.61%
	10000	43.11%	99.85%	12.55%
	25000	41.15%	100.00%	3.40%
mpegaud	5000	39.81%	100.00%	3.80%
	10000	35.20%	100.00%	3.27%
	25000	32.91%	100.00%	3.40%
mtrt	5000	49.68%	99.37%	21.09%
	10000	48.56%	98.13%	19.80%
	25000	43.79%	99.37%	6.48%
jack	5000	38.73%	100.00%	22.12%
	10000	31.14%	100.00%	1.60%
	25000	31.76%	100.00%	1.48%

Figure 8: Accuracy of rare code prediction technique at different thresholds. `t1`=2000 and `t2` varies as the value in the second column.

Benchmark	Threshold	Non-rare block execution frequency	Rare block execution frequency
Linpack	5000	94.9564%	5.0436%
	10000	94.6827%	5.3173%
	25000	100.0000%	0.0000%
JavaCUP	5000	99.4227%	0.5773%
	10000	99.8612%	0.1388%
	25000	99.9863%	0.0137%
JavaLEX	5000	97.4265%	2.5735%
	10000	99.0399%	0.9601%
	25000	99.9690%	0.0310%
SwingSet	5000	99.9769%	0.0231%
	10000	99.9999%	0.0001%
	25000	99.9998%	0.0002%
compress	5000	99.9260%	0.0740%
	10000	99.9260%	0.0740%
	25000	99.9259%	0.0741%
db	5000	99.9967%	0.0033%
	10000	99.9967%	0.0033%
	25000	99.9998%	0.0002%
javac	5000	99.6938%	0.3062%
	10000	99.8573%	0.1427%
	25000	99.8805%	0.1195%
mpegaud	5000	99.9997%	0.0003%
	10000	99.9997%	0.0003%
	25000	99.9997%	0.0003%
mtrt	5000	98.5938%	1.4062%
	10000	98.6618%	1.3382%
	25000	99.8954%	0.1046%
jack	5000	94.8186%	5.1814%
	10000	99.9981%	0.0019%
	25000	99.9981%	0.0019%

Figure 9: Accuracy of rare code prediction technique in terms of dynamic execution counts. `t1`=2000 and `t2` varies as the value in the second column.

then used to mark basic blocks as rare in the compiler's intermediate representation.[2]

2. **Perform live variable analysis.**
   Before any transformations are performed, we perform live variable analysis to determine the set of live variables at rare block entry points.

3. **Redirect the control flow edges that targeted rare blocks, and remove the rare blocks.**
   For each control flow edge from a non-rare block to a rare block, we generate a new basic block containing a single instruction that transfers control to the interpreter. This instruction uses all local variables and Java stack locations from the Java bytecode that are live at that point.[3] We redirect the control flow edge to point to this new block, and add an edge from the new block to the exit node. See Figure 10 for an example. After this process, rare blocks can be removed as unreachable code.

4. **Perform compilation normally.**
   All analysis, optimization, and code generation proceeds normally. Analyses treat the interpreter transfer point as an unanalyzable method call. Code generation treats the transfer point as a call to a glue routine, described in Section 3.5.

5. **Record a map for each interpreter transfer point.**
   When generating the code to call the glue routine, we also generate a map that specifies the location, in registers or memory, of each of the local variables and Java stack locations used in the original Java bytecode. This map is used by the glue routine to reconstruct the interpreter state. The map is stored immediately after the call in the instruction stream. Each map is typically under 100 bytes long.

Recently, server-oriented virtual machines like Jalapeño [9] and the MRL VM [15] have abandoned interpretation and gone instead with a compile-only approach. Although our system utilizes an interpreter, this technique can easily be used with compilation-only systems. Instead of recovering interpreter state, the glue routine can use the map to recover compiled code state at the rare path entry point. A mechanism to enter compiled code at basic block entry points is necessary in any case to support Java run time features such as exceptions and debugging [35].

3.2 Correctness

This technique relies on a few simple observations. The first observation is that performing control-flow tail duplication does not change program semantics. Control flow tail duplication is a technique by which control flow merges are eliminated by duplicating the code after the merge point. It

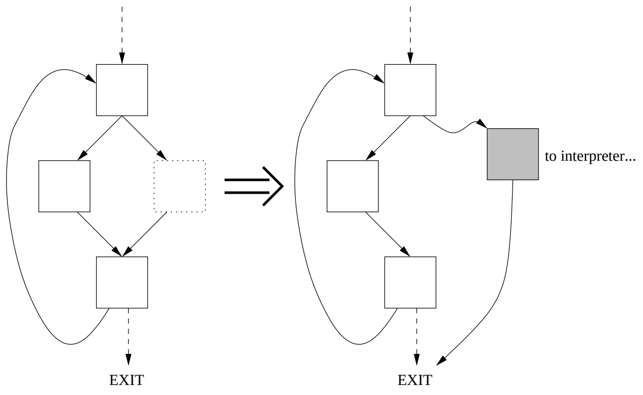

Figure 10: An example of redirecting the rare path. On the left, the dotted block is rare, so we redirect it to a block that calls the interpreter.

is typically used to allow more precise data flow information after the merge point and thereby increase optimization opportunities [13, 40, 1]. By removing edges from rare blocks to non-rare blocks, we are simply splitting the rare path from the common path, which obviously does not change the program semantics on the common path.

The second observation is the code that we replace the rare path with — the instruction to transfer to the interpreter — is a conservative summary of all of the code on the rare path. Analyses treat this instruction as a method call to an unanalyzable method, which takes all of the live variables as arguments. The body of the conceptual unanalyzable method can execute arbitrary code, which is by definition more general than the code on the rare path and is therefore a conservative summary.

The third observation is that reconstructing the interpreter state and continuing execution in the interpreter is computationally equivalent to continuing execution on the rare path. Assuming that the implementations of the compiler and interpreter are correct, and the reconstructed interpreter state is equivalent to the state if the interpreter was executing and entered the rare path, our technique is correct. The interpreter state can always be equivalently reconstructed because the live local variables and Java stack locations at that program point are marked as used and therefore their correct values are available; locations that are not marked as live contain values that will not be used and are therefore unimportant.

3.3 Partial dead code elimination

We modified our dead code elimination algorithm to treat rare blocks specially. This allows us to move computation that is only live on a rare path into the rare block, saving computation in the common case.

Our dead code elimination uses an optimistic approach similar to the one described by Muchnick [36], originally due to Kennedy [29]. That analysis begins by marking all instructions that compute essential values, and then recursively

[2]If the method entry point is marked as rare, we mark all rare blocks reachable from the root via other rare blocks as non-rare.

[3]This may include variables from multiple Java methods, if the transfer point is part of inlined code.

marking all instructions that contribute to the computation of essential values. Any non-essential instructions are then eliminated.

Our analysis operates on SSA form. It first computes the essential instructions in all non-rare blocks, completely ignoring all rare blocks. An essential instruction computes a value that is used in a predicate, returned or output by the method, or has a potential side-effect.[4] It then visits each rare block to discover instructions that are essential for that rare block, but not essential for non-rare blocks. If these instructions are recomputable at the point of the rare block, they can be safely copied there.

For each instruction in the rare block, it recursively visits all instructions that contribute to the computation of values for that instruction. If an instruction is marked as essential, it is skipped. If it is a ϕ function, it depends on an earlier predicate, and is therefore (recursively) marked as essential. Otherwise, the instruction is added to a set of instructions associated with the rare block.

After computing sets for all rare blocks, it adds each of the non-essential instructions in a set to its corresponding rare block. Then, all instructions in non-rare blocks that are not marked as essential are eliminated.

Now we make an argument of correctness. Any instruction that was eliminated on the main path either computed a value that was not essential anywhere, in which case it is obviously correct to eliminate it, or it was only essential in some number of rare blocks, in which case it would have been copied into those rare blocks. Copying the instruction into a rare block is legal because, as the instruction is not a ϕ function, the instruction dominates and is in the same loop as the rare block and therefore would have executed exactly once. Also, any instruction with a potential side effect or that read from or wrote to memory would have been marked as essential on the main path and therefore executed in its original location. Therefore, moving the instruction to a rare block does not violate exception or memory semantics.

3.4 Pointer and escape analysis

Treating an entrance to the rare path as a method call can be an overly conservative assumption. This conservativeness is normally not an issue because there are no control flow merges from the rare path back into the common path, and therefore it has no effect on forward data flow. However, in the case of escape analysis [46], the conservativeness prevents objects from being stack-allocated because they may escape into the "method call" on the rare path. Therefore, we modified our pointer and escape analysis to take advantage of rare block information. Objects can be stack-allocated as long as they do not escape in the common blocks; if a branch to a rare block is taken, stack-allocated objects are copied to the heap and pointers to them are updated.

[4]Reads from the heap are treated as having a side-effect because they can trigger null pointer exceptions. However, many of the reads from the heap are eliminated by a commoning optimization pass [42, 27].

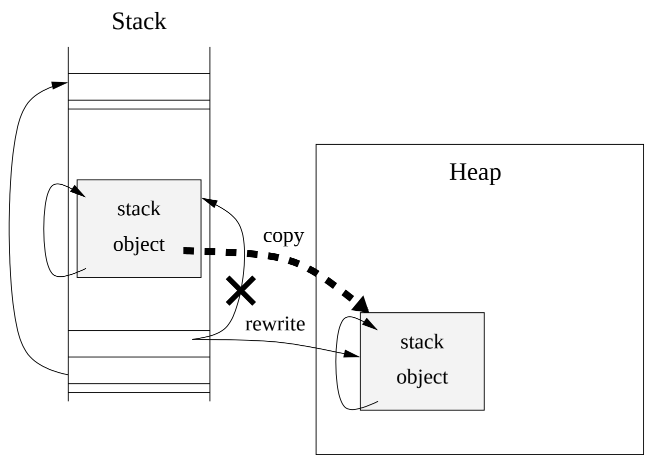

Figure 11: An example of copying an object from the stack to the heap. It copies the object and updates the pointers to point to the new location.

We present a modified version of the algorithm by Whaley and Rinard [46] that takes advantage of rare path information. This algorithm is a flow-sensitive, context-sensitive, compositional pointer and escape analysis. It discovers objects that can be accessed by only one thread and eliminates the synchronizations on them. It also computes the lifetimes of objects, and when the lifetime of an object is bounded by the activation of a method, it allocates the object in the stack frame of the method, avoiding heap allocation and garbage collection costs.

This algorithm works by building points-to escape graphs. A points-to escape graph characterizes how objects can refer to each other at run time. It also encodes escape information, which is information about which objects can be accessed by code outside of the scope of the analysis and how they can be accessed.

Our implementation performs more optimizations than the algorithm originally described by Whaley and Rinard [46]. First, it replaces the fields of non-escaping objects and the elements of non-escaping arrays by local variables whenever possible; this optimization is called scalar replacement of aggregates [31, 36, 10]. This transformation is possible for a given location when all loads and stores that can possibly access the location can access no other locations. In this case, the location is changed into a local variable; loads and stores are transformed into simple register move operations.

Second, it uses the points-to information generated by the algorithm to eliminate redundant type checks and resolve the target of dynamic calls. In the analysis, objects that are created at a specific creation site are grouped into a single node. Because an object's creation site precisely defines its type, we can use this type information along with the points-to information to eliminate redundant type checks and resolve the targets of dynamic calls.

The algorithm proceeds as follows. During the analysis phase, the algorithm treats the interpreter transfer instruction as a simple use, rather than a method call. Then, after performing a stack allocation transformation, we visit each of the interpreter transfer instructions in each of the rare blocks. If the object to be stack-allocated is potentially pointed to by one of the variables used by the interpreter transfer instruction, we insert a call before the interpreter transfer instruction to a runtime routine that copies the object from its location on the stack to a newly allocated area on the heap. We then insert conditional update instructions for every location that can potentially point to the stack-allocated object. Because our garbage collector cannot deal with references on the heap to stack objects, objects on the heap cannot refer to objects on the stack. Therefore, the only locations that can possibly refer to a stack object are local variables and fields of other stack objects. This list of locations is obtained from the points-to data at the point of the interpreter transfer instruction. See Figure 11 for a graphic example.

Scalar replacement transformations proceed similarly, but rather than inserting an instruction to copy the object from the stack, we insert instructions to allocate space on the heap and manually initialize the object using the local variables associated with the fields of the object.

When performing synchronization elimination on thread-local objects, we add synchronization enter operations on the object to every rare block in the synchronization region.

A small discussion on the legality of postponing synchronization enter operations until the rare block is executed is in order. According to our interpretation of the Java memory model [32], a memory barrier is required when successive synchronizations occur by different threads *on the same object*. Because the object is captured at the entry point to the rare block, it cannot be accessed by another thread, and therefore the synchronizations (and their associated memory barrier operations) can legally be postponed until the rare block is executed.

No change is necessary for the eliminated type checks and resolved method invocations; these transformations are valid even after an object escapes.

3.5 Reconstructing interpreter state

We use a runtime "glue" routine to construct a set of stack frames that represent the interpreter state, and then resume execution in the interpreter. Our system uses a mixed mode interpretation strategy where the interpreter uses the same stack as the compiled code. The glue routine first generates a sequence of interpreter stack frames corresponding to the inlining state at the rare block entry point. These stack frames are initialized with their corresponding method and bytecode pointers. Then, it iterates through each location pair in the map, and copies the value at the location to its corresponding position in the interpreter stack frames. Finally, it branches into the interpreter, and execution continues. Runtime support for the glue code (to support stack walking, exception handling, etc.) was already in place because of our support for multilevel compilation.

It is worth noting that execution does not stay in interpreted mode forever. The interpreter continues to keep track of the number of invocations and backwards branches. When this threshold hits `t1` again, the method is recompiled once again, and we recollect the profile information. Therefore, the system is able to dynamically adjust when the program profile changes.

4. RESULTS

We have implemented the partial method compilation technique in the joeq virtual machine [45], an open-source multi-platform virtual machine. However, the joeq Java compiler does not perform significant optimizations and therefore the effectiveness of our technique is limited. In order to evaluate the effectiveness of our partial-method compilation technique on a fully-optimizing compiler, we also implemented the technique as an offline step, using instrumented class files to collect the rare block information described in Section 2.2 and using that information to refactor the affected classes as described in Section 3.1. Interpreter transfer points at rare block entry points are implemented as method calls to synthetic methods that contain the code reachable from the transfer point. We then rely on the virtual machine's default method-at-a-time compilation policy to avoid the compilation of the code in the synthetic method. We also perform the partial dead code elimination and pointer and escape analysis described in Sections 3.3 and 3.4 on the refactored classes in a seperate offline step. Implementing our technique as a offline step also makes our technique virtual machine independent, and thereby it can be used as an effective standalone optimizer for improving application startup time on existing virtual machines.[5]

We used the joeq virtual machine [45] to collect the statistics in Section 2 and the profile data to perform the code transformations. We only transformed the application classes; further gains can be realized by applying the same technique to the class library and virtual machine internal classes. We used threshold values of `t1`=2000 and `t2`=25000. We used the Bytecode Engineering Library [16] to output the refactored classes. The escape analysis and partial dead code elimination passes are based on analyses in the FLEX compiler infrastructure [2], changed to work with joeq. To test the effectiveness of our technique, we used various programs, including the industry-standard SPECjvm98 benchmark suite [44]. SPECjvm98 benchmarks were executed in command line mode with the large input size.[6] Our performance information was collected using the IBM Developer Kit, Java(TM) Technology Edition [25] build cx130-20010626 on a Pentium 3 600 mhz machine with 512MB RAM running RedHat Linux 7.1.

4.1 Performance

Figure 12 and the associated table in Figure 13 show the improvement in performance from using our technique. Each

[5]The system will be available at press time as a standalone module at http://joeq.sourceforge.net.

[6]The SPECjvm98 benchmarks were not executed using the test harness and are therefore not official SPEC results.

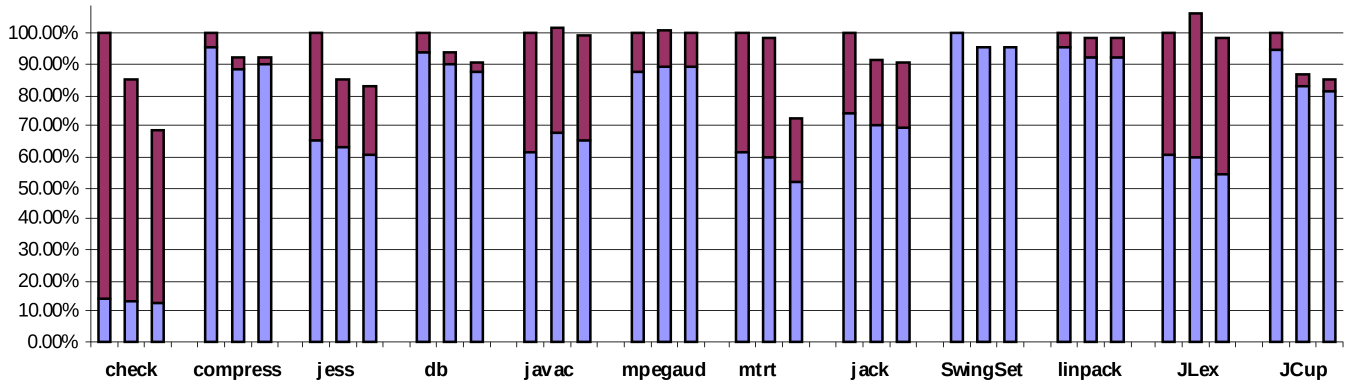

Figure 12: Normalized run time performance for each of our benchmarks. The first bar for each benchmark is the execution time using a standard whole-method compilation technique. The second bar is the execution time using our partial method compilation technique. The third bar adds partial dead code elimination and escape analysis. The length of each bar is the time of the first run; the bottom, light-colored part of each bar is the time of the best run. The difference can mostly be attributed to compile time.

	check	compress	jess	db	javac	mpegaud	mtrt	jack	SwingSet	Linpack	JLex	JCup
	Whole method											
First run	0.146	20.398	15.937	26.719	25.972	13.544	12.047	16.547	18.84	2.750	3.690	4.986
Best run	0.021	19.422	10.394	25.088	16.084	11.852	7.378	12.247	N/A	2.620	2.237	4.718
Difference	0.125	0.976	5.543	1.631	9.888	1.692	4.669	4.300	N/A	0.130	1.453	0.268
	Partial method											
First run	0.125	18.804	13.656	25.213	26.433	13.691	11.868	15.160	18.05	2.710	3.933	4.314
Best run	0.020	18.100	10.086	24.137	17.725	12.097	7.242	11.592	N/A	2.550	2.209	4.122
Difference	0.105	0.704	3.570	1.076	8.708	1.594	4.626	3.568	N/A	0.160	1.724	0.192
	Partial method plus optimizations											
First run	0.100	18.834	13.254	24.363	25.804	13.638	8.759	15.010	18.01	2.710	3.634	4.250
Best run	0.019	18.322	9.674	23.458	16.994	12.060	6.251	11.501	N/A	2.550	2.016	4.051
Difference	0.081	0.512	3.580	0.905	8.810	1.578	2.508	3.509	N/A	0.160	1.618	0.199

Figure 13: A comparison of whole method compilation versus partial method compilation. All times are in seconds.

benchmark has three bars — the first bar is the total execution time using a standard whole-method compilation strategy. Its run time is normalized to 100%. The second bar is the total execution time using our partial-method compilation technique. The third bar adds the partial dead code elimination and escape analysis optimizations from Sections 3.3 and 3.4, respectively. The height of each bar corresponds to the time of the first run of each benchmark. The bottom, light-colored portion of each bar corresponds to the time of the best run. The difference between these two times can mostly be attributed to compile time. SwingSet was measured by using wall-clock time, so we do not have best times for it. Although the times here do not include the processing in the offline step, our prior experiments indicate that the time spent for these is negligible, and that enabling partial-method sensitivity in the optimizations in Section 3 does not significantly add to their run time.

As we see from the figure, the effectiveness of our technique varies between the benchmarks. In most cases, it is effective in reducing compile time; however, in a few cases, compile time does increase due to a rare path being taken enough times to trigger compilation. This effect is very pronounced with JLex because the run time of the benchmark is small, so the recompilations have a larger effect. It is also worth noting that the optimizations improve compile time further, mostly due to the fact that the pointer analysis is able to resolve many more call targets, reducing code size by alleviating the need for generating extra code for virtual calls. This is the cause of the significant improvement in the compile time of mtrt when enabling the optimizations. Also, in some cases it is able to reduce best run time considerably, as is the case for compress and JCup. This improvement is due to improved cache locality and improved data flow information on the common paths. In mtrt and JLex, escape analysis was able to scalar replace an object in one of the core routines, which improved their best run time.

Overall, the partial method compilation technique improved total run time by an average of 5%, and up to 15%. Enabling the optimizations improved performance by, on average, another 5%, and up to 17%.

4.2 Code size

In Figure 14, we show the reduction in the number of bytes of bytecode compiled using our technique. The first row of the table contains the code size of all executed methods. The second row is the code size using a method-at-a-time strat-

	check	compress	jess	db	javac	mpegaud	mtrt	jack	SwingSet	Linpack	JLex	JCup
All executed methods	12441	4623	37983	4757	114608	56504	17570	53772	29894	3048	37838	33951
Method-at-a-time	360	2576	15114	1429	88798	20224	13178	36537	1814	1682	23349	20837
Partial method	175	2209	8467	912	50844	15222	10713	14669	1172	1744	14358	9042
Number of recompiles	0	2	2	0	5	0	4	0	0	1	4	0
Size of recompiles	0	606	21	0	1187	0	1195	0	0	92	1618	0

Figure 14: Number of bytes of bytecode compiled. The first row is the code size of all executed methods. The second row is the code size if we compile a method-at-a-time. The third row is the code size using the partial method compilation technique. The fourth and fifth rows show recompilation statistics from entering the rare path. We used threshold values of `t1`=2000 and `t2`=25000.

egy. The third row is the code size using the partial method compilation technique, including any recompilations. The fourth row shows the number of recompilations triggered by entering the rare path, and the fifth row shows the code size of those recompilations. We used threshold values of `t1`=2000 and `t2`=25000.

Our technique is able to significantly reduce the number of bytes of bytecode compiled. This translates into a direct compilation time speedup and code size reduction.

5. RELATED WORK

Our technique of partial method compilation has similarities to prior work. In this section, we compare and contrast our technique to related areas.

Dynamo [5] is a run-time optimizer that tries to improve performance by identifying and optimizing frequently taken paths (traces) through a program. Dynamo initially interprets the program, keeping track of execution frequencies. When an execution frequency hits a certain threshold, Dynamo records a dynamic trace of instructions, called a *fragment*, and compiles it as a single-entry, multi-exit contiguous sequence. This allows them to achieve sub-method and arbitrary cross-method granularity. Like our system, Dynamo uses dynamic execution counts to trigger upgrade compilation. However, Dynamo uses only the most recently executed trace to determine what to compile, and it can only operate on a single trace at a time. Our technique uses code coverage information from a number of executions, and operates on arbitrary code regions.

The Self dynamic optimizing compiler used a technique of deferring compilation of uncommon cases to help improve compilation time by over an order of magnitude [11]. Their technique uses type information to defer compilation for messages sent to receiver classes that are presumed to be rare. When a rare path is executed, the compiler reuses the current stack frame. Our technique is similar, but handles rare paths through arbitrary control flow. Like us, they also found that deferring compilation of the rare path improved optimization opportunities on the common path. In their case, they were able to use a simpler, faster register allocator because most of the calls were on the rare path.

The HotSpot server VM is able to avoid code generation for uncommon virtual method targets by using an "uncommon trap" mechanism [38]. Their mechanism of falling back to the interpreter is similar to our technique. Ishizaki *et al.* use a code patching mechanism to fall back when an assumption about devirtualization is invalidated [26]. Ogasawara *et al.* use a profiling technique to optimize exception handling [37].

Trace scheduling is a technique that predicts the outcome of conditional branches and then optimizes the code assuming the prediction is correct [18, 33]. The compiler generates compensation code to recover the state after an unpredicted branch is taken. Trace scheduling suffers from the complexity involved in the generation of compensation code. Superblock scheduling is a technique utilized by the IMPACT compiler [24]. Superblock scheduling simplifies the complexity of compensation code generation by using tail duplication to create *superblocks* — single-entry, multiple-exit regions. Like superblock scheduling, our technique does not allow control to flow from the rare path back onto the common path and thereby simplifies the generation of compensation code. However, unlike trace scheduling or superblock scheduling, our technique operates on arbitrary control-flow graph regions, not just a single trace. Furthermore, our technique does not generate code for rare traces until they are actually executed at run time. As long as our predictions are correct and the rare paths are not taken, we can avoid the code growth associated with tail duplication.

Poletto used tail duplication to increase the amount of data flow information that is available about a piece of code [40]. Like us, he found improved code quality due to better data flow information. Chang *et al.* describe a method of using edge profile information to assist classical optimizations [12]. They use edge profile information to discover frequently-executed paths and form superblocks from those paths, using tail duplication to avoid joins. Our technique achieves a similar effect. Ammons and Larus describe a technique of identifying and duplicating hot paths in order to improve the precision of data flow analysis along those paths [1]. The hot paths are constructed using an acyclic path profile [7]. Their technique uses qualified data flow analysis [22], which couples a conventional data flow problem with a deterministic finite automaton, to recognize hot paths. To avoid unnecessary code growth, the duplications that ended up to be unprofitable are removed. Our technique uses node profiling rather than path profiling, and therefore does not distinguish between different paths leading to the same point; extending our technique to use path information would be straightforward.

There are some techniques that use hardware extensions to allow an optimizer to ignore exceptional control flow [13, 41]. Delayed exceptions [17] and sentinel scheduling [34] are hardware extensions that allow speculative execution of excepting instructions. They allow an instruction scheduler to schedule as if exceptions did not exist. They add check and fix up code to deal with the case if the trap actually occurs. We achieve this in software by limiting optimization by making more conservative assumptions about exceptional control flow.

Gupta *et al.* describe a technique for performing optimizations in Java while ignoring dependence constraints among potentially excepting instructions [19]. They first identify the subset of program state that needs to be preserved in the case that an exception is thrown. They then remove dependencies between exceptions; when exceptions are reordered, compensation code is inserted to catch the exception and throw the correct one. This technique is complementary to ours; modeling rare code as a method call makes it a potentially excepting instruction, and so we could use this technique to remove dependencies between exceptions and the rare code entry point.

Bruening *et al.* explored the issue of finding optimal compilation unit shapes in an embedded Java JIT compiler [8]. They also pointed out that method boundaries are a poor choice for compilation. They suggest compiling only the hot traces or loops within a method. Because they did not have a working JIT compiler, they provided estimates of the code size benefits by using trace-based and loop-based compilation units. Like us, they found that a majority of the code in methods is rarely or never executed, and that they could drastically reduce code size with only a negligible change in performance. Our work improves on their work by providing the details of an actual implementation — how to determine rare code, how to compile partial methods, how to fall back to interpretation, performance results, etc. Hank *et al.* also claim that functions are not good compilation units [21] and they propose partitioning the program into disjoint regions that are analyzed and optimized seperately.

Our partial dead code elimination optimization is similar to other profile-sensitive code motion optimizations. Horspool and Ho present a version of partial redundancy elimination based on a cost-benefit analysis from edge profile information [23]. There has also been work on performing optimizations while making use of path profile information. Gupta *et al.* describe methods of using path profile information in a cost-benefit data flow analysis to perform partial redundancy elimination and partial dead code elimination and guiding speculation and predication decisions [30, 20]. The main difference between our algorithm and earlier algorithms is that because our algorithm is a sparse analysis and only makes a distinction between rare and non-rare blocks, it can avoid performing cost-benefit analysis, and therefore it has nearly the same run time as normal dead code elimination.

There has been some research devoted to using profile data to guide code placement strategies to improve instruction cache locality [39]. Our technique can be thought of as a very extreme profile-directed code placement strategy — rare code is not only placed away from common code, it is not even generated at all. Chen and Leupen describe a system of just-in-time code layout that dynamically loads and lays out procedures as they are called [14]; this is similar to our method, but we support this at the basic block level, and we also support deferring compilation.

6. CONCLUSION

In this paper, we presented a new technique for performing partial method compilation. This technique allows most optimizations to completely ignore the rare paths. This allows them to fully optimize for the common case. Furthermore, this technique allows us to avoid generating code for the rare path until it is actually executed. Because most rare paths are never executed, this decreases total code size and total compile time dramatically. It also improves the effectiveness of analyses, especially pointer and escape analysis.

Our experimental results are encouraging. With our benchmark suite, compile times were reduced drastically, and overall first run times improved by an average of 10%, and up to 32%.

The trend in compiler research has often been towards more and more powerful analyses, without regard for compilation time. This paper attempted to explore an oft-neglected area of compiler research — how to improve compilation time without giving up too much performance. With the advent of Java and corresponding surge in popularity of Just-In-Time compilers, we expect the interest in this area to grow.

Acknowledgements

The idea for ignoring the rare path during escape analysis and then copying the object to the heap if the rare path was taken is originally due to Akira Koseki of IBM Tokyo Research Laboratory. Part of this work was performed while I was a Student Fellow at IBM Tokyo Research Laboratory under the guidance of Toshio Nakatani, who was indispensable in supporting and promoting my research and providing a stimulating research environment. He, Toshio Suganuma, and Tamiya Onodera of IBM TRL also provided valuable input and comments on this paper. The research was also funded in part by an NSF Graduate Fellowship. I would also like to acknowledge all of the people who have worked on the IBM Java virtual machine and Just-In-Time compiler for providing the initial infrastructure in which this research was performed, and the developers of BCEL, FLEX, and the other tools that were used to collect the data contained in this paper. Finally, I would like to thank Monica Lam for her comments and assistance and the anonymous reviewers for their exceptionally helpful comments.

7. REFERENCES

[1] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 72–84, Montreal, Canada, June 17–19, 1998.

[2] C. S. Ananian. FLEX compiler infrastructure. http://www.flex-compiler.lcs.mit.edu, 2001.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, Oct. 2000.

[4] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and dynamic heuristics for inlining. In *Dynamo '00 workshop, Held in conjunction with POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, May 2000.

[6] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*, pages 59–70, New York, NY, USA, 1992. ACM Press.

[7] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, Paris, France, Dec. 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[8] D. Bruening and E. Duesterwald. Exploring optimal compilation unit 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.

[9] M. G. Burke, J. D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM SIGPLAN '99 Java Grande Conference*, June 12–14, 1999.

[10] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software: Practice And Experience*, 24(1):51–78, Jan. 1994.

[11] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, pages 1–15, New York, NY, 1991. ACM Press.

[12] P. P. Chang, D. M. Lavery, and W. M. Hwu. The effect of code expanding optimizations of instruction cache design. Technical Report CRHC-91-18, Coordinated Science Lab, University of Illinois, Jan. 1992.

[13] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An architectural framework for multiple instruction issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, volume 19, pages 266–275, New York, NY, 1991. ACM Press.

[14] J. B. Chen and B. D. D. Leupen. Improving instruction locality with just-in-time code layout. In USENIX, editor, *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 25–32, Berkeley, CA, USA, Aug. 1997. USENIX.

[15] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–26, 2000.

[16] M. Dahm. Byte code engineering library. http://bcel.sourceforge.net, 2000.

[17] M. A. Ertl and A. Krall. Delayed exceptions — speculative execution of trapping instructions. *Lecture Notes in Computer Science*, 786:158–171, 1994.

[18] J. A. Fisher. Trace scheduling : A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, 1981.

[19] M. Gupta, J. D. Choi, and M. Hind. Optimizing java programs in the presence of exceptions. In *14th European Conference on Object-Oriented Programming (ECOOP 2000)*, June 12–16, 2000.

[20] R. Gupta, D. A. Berson, and J. Z. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 358–368, Los Alamitos, Dec. 1–3 1997. IEEE Computer Society.

[21] R. Hank, W. Hwu, and B. Rau. Region-based compilation: An introduction and motivation. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 158–168, Ann Arbor, MI, Nov. 1995.

[22] L. H. Holley and B. K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, 7(1):60–78, Jan. 1981.

[23] R. Horspool and H. Ho. Partial redundancy elimination driven by a cost benefit analysis. In *8th Israeli Conference on Computer Systems and Software Engineering*, pages 111–118, June 1997.

[24] W. M. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, May 1993.

[25] International Business Machines. IBM Developer Kit, Java(tm) Technology Edition, 2001. http://www.ibm.com/java/jdk/index.html.

[26] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'00)*, Oct. 2000.

[27] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM SIGPLAN '99 Java Grande Conference*, pages 119–128, June 12–14, 1999.

[28] M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware traps. In *Proceedings of the 9th International Conference on Architectural Support on Programming Languages and Operating Systems (ASPLOS-IX)*, Nov. 2000.

[29] K. Kennedy. A Survey of data flow analysis techniques. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, 1981.

[30] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 147–158, New York, NY, USA, June 1994. ACM Press.

[31] D. A. Kranz, R. Kelsey, J. A. Rees, P. Hudak, J. Philbin, and N. I. Adams. Orbit: An optimising compiler for scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM, June 1986.

[32] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[33] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[34] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.-M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4):376–408, 1993.

[35] S. Microsystems. Java virtual machine debug interface reference. http://java.sun.com/~products/jdk/1.3/docs/guide/jpda/jvmdi-spec.html.

[36] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[37] T. Ogasawara, H. Komatsu, and T. Nakatani. A study of exception handling and its dynamic optimization in java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01)*, Oct. 2001.

[38] M. Paleczny, C. Vick, and C. Click. The Java HotSpot(TM) server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, 2001.

[39] K. Pettis and R. C. Hansen. Profile guided code positioning. *SIGPLAN Notices*, 25(6):16–27, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[40] M. Poletto. Path splitting: a technique for improving data flow analysis. Master's thesis, Massachusetts Institute of Technology, May 1995.

[41] M. D. Smith, M. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 344–354, 1990.

[42] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-time compiler. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[43] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for java just-in-time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01)*, Oct. 2001.

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

[45] J. Whaley. joeq virtual machine. http://joeq.sourceforge.net, 2001.

[46] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA'99)*, Denver, CO, 2–5 Nov. 1999.