Complex Program Transformations Via Simple Online Dynamic Analyses

Ajeet Ganesh Shankar

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-64
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-64.html

May 16, 2009

Complex Program Transformations Via Simple Online Dynamic Analyses

by

Ajeet Ganesh Shankar

A.B. (Harvard University) 2001

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Rastislav Bodik, Chair
Professor Eric Brewer
Professor Leo Harrington

Spring 2009

The dissertation of Ajeet Ganesh Shankar is approved:

Chair Date

Date

Date

University of California at Berkeley

Spring 2009

Complex Program Transformations Via Simple Online Dynamic Analyses

by

Ajeet Ganesh Shankar

Abstract

Complex Program Transformations Via Simple Online Dynamic Analyses

by

Ajeet Ganesh Shankar

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Rastislav Bodik, Chair

Online dynamic program analyses execute while a program is running in its production state. In this environment, they have access to a wealth of concrete information, such as heap values and call stacks, that can drive recompilation and program transformation based on behavior that is execution-specific or simply hard to analyze statically.

At the same time, because these analyses execute while a program is running, they are computationally constrained, since any overhead imposed by an analysis is reflected in the program's execution time. Thus, online dynamic analyses necessarily tend to measure simple events. Such constraints present a challenge: is it possible to construct sophisticated program transformations from simple dynamic analyses?

In this dissertation, we show that sophisticated program transformations are possible without proving complex program properties. Instead, dynamic analysis of simple properties can sufficiently approximate complex ones when coupled with runtime support

such as speculation. We describe two complex automatic program optimizations based on simple dynamic analyses that yield order-of-magnitude speedups.

The first is a specializer. While a program is running, it automatically determines which heap locations are constant, and selects portions of the program to recompile, using these heap constants to seed constant propagation, loop unrolling, and other classic optimizations. The specializer achieves speedups of up to 500% on programs that depend heavily on constant heap values.

The second transformation is an incrementalizer, known as DITTO. DITTO monitors data structure invariant checks as they execute, dynamically constructing models of their computation. When a check is invoked again, DITTO automatically reuses applicable portions of previous invocations and only computes anew computations based on new input. It speeds up common invariant checks by an asymptotic factor.

These two optimizations indicate that complex transformations are possible despite the limited complexity afforded to the online dynamic analyses that drive them. We find that these results can be achieved when (1) the transformation is tailored to a domain and (2) dynamic analyses are carefully constructed to exploit the common-case behavior of programs in that domain.

Professor Rastislav Bodik
Dissertation Committee Chair

Contents

List of Figures

Acknowledgements

I owe a huge debt of gratitude to my advisor, Ras Bodik. Over the course of my tortuous (but not tortured) graduate career, Ras displayed heroic patience as my interests shifted, supported my desire to teach, and showed genuine interest in my entrepreneurial endeavors with Modista. Most importantly, he refused to settle for lazy generalizations and taught me how to think and write clearly.

Matt Arnold was a fantastic friend and mentor during (and after) my IBM internship, and Professor Alistair Sinclair, for whom I was a Graduate Student Instructor, showed me the power of great teaching. Brian Fields, Percy Liang, the Turing Machines soccer team, and many other friends in the CS department made grad school fun. Manu Sridharan and Naveen Sastry were especially good friends (particularly during one veggie-and-gym filled summer in Seattle).

Bill McCloskey and Dave Mandelin were my closest companions in Soda Hall and I'm not sure I would have made it through school without them. They were always up for informed and hilarious debates, daily NYTimes crossword puzzles, and fun coding projects, and kept me sane during my moments of greatest doubt. My many technical conversations with Bill, Dave, and Manu were invaluable to my research and to my understanding of the field of programming languages as a whole.

Outside of Soda Hall, my close high school and college friends did an excellent job of getting married with pleasing regularity so that I'd get to see them all a few times a year. The Cal Men's Ultimate team provided me with four years of fantastic memories, culminating in this year's Nationals run. I'm grateful that they took a chance on an old

fogey like me. Arlo Faria has been a great friend and incredible business partner. I thank him for his patience and understanding as I finish grad school even as our startup company picks up steam.

My parents, brother Umesh, and sisters Meera and Maya have provided me with constant support, love, and guidance. In particular, I want to thank Umesh for inspiring me with his early love of computers. I followed him from high school to college to grad school; by the end I dropped all pretenses of being different and switched my major over to computer science. Good choices, Umesh.

Most of all, I would like to thank my fianceé, Bekah Sexton. She has been supportive of all my endeavors, understanding of all my concerns, insightful in all her comments, and endlessly affectionate. I only hope that I can provide the same support for her, and am thrilled to be heading out into the real world with such an amazing companion.

Chapter 1

Introduction

Program analysis is the task of automatically analyzing program behavior. Analyses can aid optimization, bug-finding, monitoring, correctness arguments, and a host of other applications.

Many analyses are *static*: they attempt to determine runtime behavior from a program's source code without any specific input, usually in an offline setting, such as in a compiler. The primary strength of this approach is that algorithms have no hard time constraints, and thus can be quite complex.

Another approach is *dynamic program analysis*, which seeks to analyze the program as it is running. The primary advantage of this approach is that since it deals with concrete behavior (an execution) rather than source code, it is more precise than static analysis. For instance, it has access to specific data – say, the layout of the heap – that static analysis can only infer via an abstract interpretation of the program code. However, if the analysis is *online* – that is, it is run while the program is executing in a production setting – it must

adhere to fairly tight overhead constraints. Thus, most online dynamic analyses are simple, yielding small amounts of information.

In this dissertation, I hope to demonstrate that it is possible to construct complex dynamic program transformations that depend on simple online dynamic analyses. I will do this by describing two nontrivial online dynamic optimizations, a specializer and an incrementalizer.

1.1 Runtime Specialization

A specializer optimizes code by partial evaluation: it selects a program state in which the values of certain variables are known to be constant, and uses these constant values to optimize the code that consumes them by constant propagation, inlining, branch evaluation, and loop unrolling.

Constant values that guide partial evaluation can originate from local variables or from heap data. Heap values are infrequently exploited because they are difficult to identify and verify as constant: their lifetimes can span an entire program execution and they can be mutated at any time.

We present a runtime specializer that exploits constant values on the heap. Programs amenable to specialization can see a 500% speedup as a result of this transformation.

- It is automatic. Specializeable functions and invariant heap values are identified via a lightweight runtime analysis. There is no barrier to use for programmers who do not want to learn an annotation system, delve into specific memory properties of programs they are developing, or find specializable regions.

- It is sound. Manual annotations may be incorrect (for instance, if a mutating heap locations is marked as invariant), whereas our approach uses runtime checks to ensure soundness.

- It is more complete than traditional static specializers, whose annotations are made offline, on static code, and cannot identify heap values that are invariant for just a particular program exeuction, or particular concrete elements of a data structure that are invariant even though the rest of the structure is not. Our specializer operates at runtime and is able to optimize any concrete heap locations.

1.1.1 Example: Interpreter

We illustrate how specialization works by examining the specialization of an interpreter. Interpreters are excellent candidates for specialization because a critical set of heap locations – those holding the program to be interpreted – is not modified. The locations in this set yield the constant values that drive the partial evaluation.

In a standard interpreter implementation, a main loop iterates over the intepreted program's opcodes, decoding and dispatching on each one. The opcodes are stored in an array on the heap and fetched one by one, depending on the value of the interpreted program counter. A specializer optimizes the interpreter by substituting the actual values of the opcodes in the main interpretation function, unrolling the loop and removing the dispatch overhead. In essence, it turns the interpreter into a simple JIT compiler. Thus, the specializer removes the interpretation overhead, the biggest overhead in an interpreter.

For example, consider the interpretation of an insertion sort routine.

```
void insertion_sort(int[] a)
  for (i = 0 ; i < a.length - 1 ; i++)
    min = i
    for (j = i+1 ; j < a.length ; j++)
      if (a[j] < a[min])
        min = j
    (a[i], a[min]) = (a[min], a[i])
```

A simple interpreter of this language might resemble the following.

```
int interpret(OpCodes[] o)
  pc = 0
  while 1
    switch o[pc].op
      case ASSIGN:
        mem[o.mem_a] = mem[o.mem_b]
        pc++
      case COMPARE_LT:
        if (mem[o.mem_a] < mem[o.mem_b])
          pc++
        else
          pc = o.destination
      ...
```

A specializer can partially evaluate this `interpret` function at runtime, using the concrete values of the opcodes (in this case, the opcodes from the insertion sort function) to bypass the `switch` statement and fill in the actual operation values, such as `mem_a`, `mem_b`, and `destination`, for each instruction. The resulting block of straight-line code is known as a specialization trace. Both our specializer and traditional specializers follow this same blueprint for optimization.

However, the principal way in which ours differs is in how it determines that (a) a particular program point in `interpret` is a good candidate for the start of a specialization trace and (b) `o` points to an array of invariant cells on the heap.

A traditional specializer, such as DyC [35], requires the programmer to annotate the interpreter with both of these pieces of information.

```
// indicates that this function should be specialized
specialize interpret(o, pc) int
interpret(OpCodes[] o)
  pc = 0
  // indicates that the values in the o array are constant
  // and should be used as a basis for partial evaluation
  make_static(o, pc:p_cache_one_unchecked, eager)
  while 1
    ...
```

As described above, there are several limitations to this approach: it is potentially error-prone (not always sound), incomplete, and also requires programmers to spend time on annotations. In constrast, our specializer determines both key pieces of information automatically using several dynamic analyses.

Because its predictions about what heap locations are invariant may become incorrect as the program execution progresses, our specializer must check that its assumptions about invariance hold before executing a specialization trace; if they no longer hold (i.e. a heap location's value has been modified), it must *invalidate* the trace. To ensure soundness in this way, our specializer automatically tracks its invariance predictions via write barriers as the program executes to determine when they have gone wrong. While write barriers can be an expensive modification, our specializer uses the language's type system to eliminate most writes as invalidators; it thus tends to insert only a modest number of write barriers, keeping the overhead low.

Its dynamic analyses perform:

- Fast identification of good specialization points via a new influence metric.

- Optimistic and accurate fine-grained detection of heap invariants by a store profile.
- Detection and invalidation of specialized regions when assumed heap invariants are modified.

All of these analyses are lightweight enough to be run in a production execution environment. Furthermore, since they are executed at runtime, they can provide better information to the specializer, by identifying behavior that is either (a) impossible to notate using offline annotations or (b) execution-specific.

To illustrate this benefit, we show how memory locations in an interpreted program's "memory space" are identified by our runtime specializer and not by an offline one. Consider the inner loop of the insertion sort function above, slightly rewritten:

```
j = i+1
while (j < a.length)
  if (a[j] < a[i])
    min = j
  j++
```

In Figure 1.1, we show a trace of an interpreter executing the inner loop bytecodes, along with the optimized versions of the trace that are produced by DyC, a traditional offline specializer, and our specializer. (Note that for simplicity, the trace does not include the instructions for dispatch and opcode decoding, since both specializers optimize them away; this optimization and loop unrolling contribute significantly to the speedup.)

Both specializers are privy to the fact that the bytecodes of the interpreted program are invariant, and thus eliminate loads that access fields of the bytecode objects. However, because DyC is an offline specializer that relies on code annotations, its optimizations must

apply to all executions of the interpreter. Thus, it cannot take advantage of any execution-specific behavior – namely, invariant memory locations present in the state of the *interpreted* program.

Our specializer, which operates directly on the heap as the program executes, is able to determine the invariance of several memory locations specific to the insertion sort, such as the address of the array to be sorted and its length. This address is fixed over the intepretation of the program, and appears to our specializer as constant as any other heap location. However, the address is nowhere in the code of the interpreter; it is only referenced in the bytecodes of the interpreted program. Thus, an offline specializer that only has access to the source code of the interpreter would have no way of identifying this memory location as constant. As a result, our specializer removes nearly 10% more loads than an offline specializer in this example.

In Chapter 2 of this document, we describe our specializer and the analyses it performs in greater detail, and provide an evaluation of it implemented in the Jikes RVM Java virtual machine.

Bytecodes	**Intepreter Trace**	**DyC**	**DynSpec**
// j = i + 1	case LD_IMM:		
LD_IMM r1, #1	r[op.dest] = op.constant	r[r1] = 1	r[1] = 1
	case ADD:		
ADD r2, r1, r3	r[op.dest] = r[op.src1] + r[op.src2]	r[2] = r[1] + r[3]	r[2] = r[1] + r[3]
// while (j < a.length)	case LD:		
LD r4, [a.addr], -1	r[op.dest] = m[op.addr+ op.offset]	r[4] = m[addr['a'] - 1]	r[4] = **20**
	case BGE:		
BGE r2, r4, [L1]	if (r[op.src1] >= r[op.src2])	if (r[2] > r[4])	if (r[2] > r[4])
	pc = op.target	pc = 12	pc = 12
// if a[j] < a[i]	case LD:		
LD r6, [a.addr], r2	r[op.dest] = m[op.addr + op.offset]	r[6] = m[addr['a'] + r[2]]	r[6] = m[**36** + r[2]]
	case LD:		
LD r7, [a.addr], r3	r[op.dest] = m[op.addr + op.offset]	r[7] = m[addr['a'] + r[3]]	r[7] = m[**36** + r[3]]
	case BGE:		
BGE r6, r7, [L1]	if (r[op.src1] >= r[op.src2])	if (r[6] >= r[7])	if (r[6] >= r[7])
	pc = op.target	pc = 12	pc = 12
//min = j	case ADD:		
ADD r5, r2, r0	r[op.dest] = r[op.src1] + r[op.src2]	r[5] = r[2]	r[5] = r[2]
L1: // j++	case ADD:		
ADD r2, r2, r1	r[op.dest] = r[op.src1] + r[op.src2]	r[2] = r[2] + r[1]	r[2] = r[2] + r[1]

Figure 1.1: Dynamic specialization of an interpreter executing insertion sort as performed by DyC and our specializer. Both specializers remove the bulk of the overhead caused by the interpretation loop; ours additionally optimizes constants present in the interpreted insertion sort program. Memory loads eliminated by both specializers are underlined; loads eliminated only by our specializer are double-underlined.

1.2 Automatic Incrementalization

Incrementalization is an optimization technique that exploits successive, iterative computations that vary slightly from one another. The results of repeated subcomputations are cached and reused; only computations based on changed or new input data are executed.

For example, to compute the successive sums of each m-element window in an n-element array (with m $<$ n), the following code might be used:

```
for (int i = 0 ; i < n-m ; i++)
  for (int j = i ; j < m ; j++)
    sum[i] += ary[j];
```

The nested loops yield an O(n^2) algorithm. However, we can incrementalize the code in the following way:

```
for (int j = 0 ; j < m ; j++)
  sum[0] += ary[j];
for (int i = 1 ; i < n-m ; i++)
  sum[i] = sum[i-1] - ary[i-1] + ary[i+m-1];
```

This incrementalized code takes advatange of the fact that for each window after the first, the previous window has already been computed. This previous window differs from the current one in only two values. Reusing the previous result and accounting for the two differing values is sufficient to compute the new window: sum[i] = sum[i-1] - ary[i-1] + ary[i+m-1]. The new algorithm runs in O(n) time.

Because the transformation requires algorithmic insight, incrementalization has traditionally been performed by hand. However, the manual approach is fraught with complications:

- It can be difficult to perform, especially if the code to be optimized relies on heap values that may be modified throughout the program. Program-wide changes are often necessary: everywhere the heap values are modified, incremental code must be inserted.
- Due to this complexity, hand-incrementalizing code is a time-consuming and challenging process and may induce errors or contain omissions that lead to unsoundness. For instance, we were unable to incrementalize the invariant checks for a red-black tree by hand.

Thus, a natural goal is to automate the incrementalization process.

To that end, we present DITTO, an automatic incrementalizer. Unlike existing incrementalizers, it operates on an imperative language (Java) and without any user annotations. It incrementalizes functions via a simple process.

1. The first time an invariant check is executed, DITTO builds a computation graph: a record of all function calls, arguments, heap accesses, and return values that occur during the check.

2. When control is returned to the rest of the program, DITTO monitors the heap to detect any changes to heap values accessed by the invariant check. Multiple changes can occur between check invocations.

3. When the check is invoked again, DITTO examines the new set of arguments. Using information about what heap values have been modified, it reuses results cached in the computation graph where possible. It only evaluates function calls with new arguments

or heap accesses to new or mutated data, and inserts these calls and their return values back into the comptuation graph, ensuring that the graph is up to date with the latest invocation of the invariant check.

4. By preserving the invariant that the computation graph is consistent with the most recent execution of the check, DITTO can continue to reuse previous results across subsequent checks, yielding an asymptotic speedup for each check.

The full process is described in Chapter 3.

1.2.1 Example: Red-Black Tree

To illustrate DITTO's benefit, consider verifying the correctness of a red-black tree implementation you have just created in Java. There are a number of properties to verify; for instance, each red node must only have black children. Here we focus on the most difficult property to verify, that every path from the root to a leaf contains the same number of black nodes.

```
// verify that each path has the same number of black nodes
Integer checkBlackDepth(Node n) {
  if (n == nil)
    return 1;
  int left = checkBlackDepth(n.left);
  int right = checkBlackDepth(n.right);
  if (left != right || left == -1)
    // error
    return -1;
  return left + (n.color == BLACK ? 1 : 0);
}
```

This property must be maintained across arbitrary insertions and deletions of elements. The simplest way to check the property would be to execute this `checkBlackDepth` function after

every insertion or deletion. However, the function traverses the entire tree for every check, making it prohibitively expensive if the check is to be executed frequently during the course of a program execution.

One option would be to manually incrementalize the function: insert local checks at every insertion and deletion point that verify the property for only those nodes in the tree that have changed. However, this is a very daunting task: a standard red-black tree implementation has five insertion and six deletion cases, each of which modifies the tree in a different way.

Furthermore, the operations often rotate the tree, modifying nodes that are not local to the insertion or deletion point. Even if a set of modified memory locations is provided, it is difficult to determine which nodes need to be verified without traversing the entire tree, as an unmodified node may need to be re-verified if it happens to precede a modified node on a path from the root to a leaf. Manual incrementalization in the face of these complex behaviors is challenging, and the chance for error is large.

DITTO automates this task. Presented with the `checkBlackDepth` function, it executes the function in full the first time it is invoked, and builds a computation graph of the recursive function calls it made as it traverses the nodes of the tree, counting how many black descendents each node has.

Assume that the next operation is an insertion of a new node, C, which causes a rerooting from node A to node B (Figure 1.2). Because DITTO has constructed a graph of the `checkBlackDepth` computation, it knows exactly what heap values the computation accessed, namely the nodes of the tree and accompanying metadata. It uses write barriers

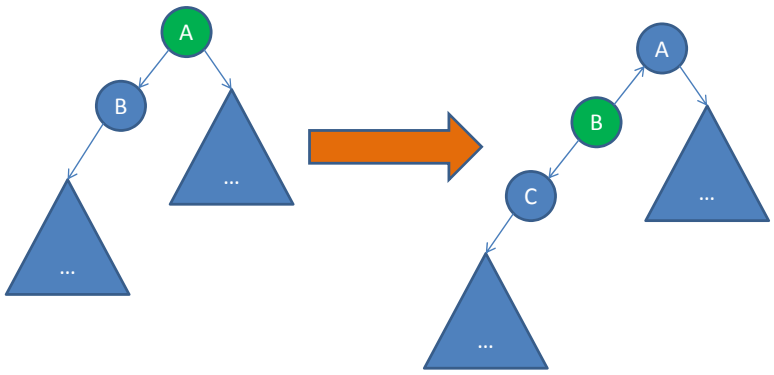


Figure 1.2: A tree that is rerooted after the insertion of a node.

to detect which of these values is modified by the insertion operation (nodes A, B, and C in Figure 1.2).

When `checkBlackDepth` is subsequently invoked, DITTO only re-executes those portions of the computation graph that consume modified data. Cached results are used instead of recomputing function calls that have no changed inputs. For example, the right-hand children of the previous root node have not changed at all, so DITTO reuses their previous counts of black descendents.

In this case, DITTO only has to recompute O(log n) nodes in order to verify the property for the entire tree structure. Furthermore, multiple updates (insertions and deletions) can occur between invariant checks. DITTO will track all changes and resynchronize the computation graph to the current state of the data structure.

Chapter 2

Dynamic Specialization

2.1 Introduction

Many virtual machines employ *dynamic optimization*, a technique that exploits the particular execution and memory behavior of each program run to produce optimizations tailored to that run. *Specialization*, or *partial evaluation*, is a related, more aggressive strategy by which hot portions of code are heavily optimized by "hard-coding" frequently occurring values, and other values that depend on them, directly into the instruction stream; when these values turn up again, the optimized code is invoked [4, 10, 21, 22, 29, 35, 36, 41, 44, 48, 54, 55, 68, 69, 71]. For certain classes of programs, such as interpreters, raytracers, and database query executors, in which a few popular values consistently dictate execution behavior, employing this technique can result in marked speedups, up to 5x in some cases [37].

Powerful specialization techniques have eluded inclusion in transparent dynamic

optimization systems, such as Java VMs, since existing specializers are *staged*: while they generate specialized code at runtime, they require an offline component of programmer annotation [22, 35, 48], or heavyweight program analysis [54]. This offline step forces staged specializers to abstract away from the concrete state of a particular program run, and employ only those specialization optimizations that apply to all executions of a program.

This chapter presents a program specialization technique that is able to exploit the unique opportunities offered by dynamic optimizers, in particular access to the concrete memory state and execution behavior of a program. This specialization technique has the following novel combination of properties:

- *It rapidly and automatically identifies specialization regions.* The specializer uses profile information with a novel linear-time algorithm based on a new notion of instruction *influence* to identify good specialization opportunities. The use of concrete execution behavior has the additional advantage of enabling specialization of the same function in different ways based on the execution pattern of a given program run.
- *It employs optimistic and precise automatic heap analysis.* The analysis exploits specialization opportunities that may not be easily detectable or annotatable in the source code, for instance data that is only invariant for certain program executions or in certain execution states, or constants as small as individual array elements.
- *It automatically monitors optimistic assumptions, and invalidates specialized regions if any assumptions are violated.* The specializer employs a low-overhead invalidation system that (a) detects when assumed constants have been updated and then (b) safely invalidates the corresponding specialized regions, even if they are currently on

the execution stack.

These three features enable the specializer to operate fully transparently at runtime: it requires no additional user input or other information. This transparency increases its ease of use: an end-user with a dynamic specialization-enabled runtime environment like a JVM can instantly reap its benefits on all of the specializable programs she runs, instead of hoping that developers will annotate each program individually with specialization directions. Even systems such as Calpa [54], a staged specializer that automatically infers annotations, require an off-line phase that a user may be unwilling or unable to perform. Additionally, the use of annotations means that such specializers cannot take advantage of per-execution runtime constants and behavior.

To the best of our knowledge, the system presented in this chapter is the first fully transparent specializer to use heap data. Suganuma et al. [71] have constructed a fully dynamic specializer, but unlike existing staged specializers, it does not utilize any heap constants (perhaps the most valuable pieces of runtime information) and so its speedups do not exceed 1.06x. In the rest of the chapter, when we refer to other specializers, we mean heap-aware specializers.

A strong motivation for this technology is that the massive popularity of scripting languages makes interpreter optimization a compelling goal. Application languages like Visual Basic and Tcl, web languages like JavaScript and VBScript, and general-purpose languages like Perl, Python, and Ruby, all have very large user bases. In addition, there are countless special-purpose languages that have significant followings in their niche areas. Given the success of these languages, new languages are constantly being developed, and

they need frameworks in which to run.

Interpreters have a number of advantages over compilers in providing such a framework: (1) writing an interpreter is much simpler than writing a compiler (and in many cases, such as in languages with "eval" functionality, a true compiler is infeasible); (2) it is generally much easier to verify an interpreter as correct; (3) it is easier to distribute an interpreter because there are fewer portability issues. As a result, most scripting languages are initially interpreted, and later, if there is sufficient demand for improved performance, a compiler may be painstakingly created. Dynamic specialization can provide an immediate level of optimization to interpreted code, reducing development time and making the use of new languages more appealing. By using concrete heap information, a dynamic specializer can also take advantage of constants induced by the *interpreted* program, rather than just those present in the interpreter, a level of optimization unavailable to existing staged specializers: it can not only specialize the interpreter, but also the program the interpreter is running.

Our primary contributions are:

- Fast identification of good specialization points via a new *influence* metric.
- Optimistic and accurate fine-grained detection of heap invariants by a store profile, and its use in specialization.
- An automatic mechanism for invalidating specialized regions when assumed heap constants are modified.
- An implementation of this system that runs transparently and with low overhead on the Jikes RVM and produces speedups of 1.2x to 6.4x.

This chapter is organized as follows. Section 2.2 is an overview of the system, and

presents three main challenges of dynamic specialization: region selection, heap invariance detection, and invalidation. Sections 2.3, 2.4, and 2.5 discuss our solutions to the three principle challenges, and Section 2.6 describes some details about region creation. Finally, we present an experimental evaluation of our work in Section 2.7. Related work is discussed in Section 2.8.

2.2 Overview

In this section, we describe the specialization procedure, discuss three critical problems that must be solved to implement it in a dynamic framework, and illustrate the process with an example.

2.2.1 Specialization Model

Classical specialization is applied in scenarios in which a program P is re-executed with a part of its input unchanged [42]. Technically, the input to P is divided into a static input s and a dynamic input d, where the former remains fixed across executions while the latter is unconstrained. For example, when specializing an interpeter P, the static input s is the program being interpreted and the dynamic input d is the input to the interpreted program. Since the static input s is fixed, computations that depend only on s produce identical outcomes in each of the executions. Specialization removes this redundant computation by specializing P with respect s. The specialized program P_s is obtained by evaluating (some) instructions that depend on s but not on d and residualizing remaining instructions. Executing the specialized program P_s on the dynamic input then yields the

desired output, formally $P_s(d) = P(s, d)$.

The computation that is "specialized away" may represent a significant portion of the original computation. For example, when specializing an interpreter P with respect to the program s being interpreted, the specialized interpreter may omit the entire interpretive overhead, producing a compiled version of s.

In practice, deployment of specialization differs from the scenario described above, either because programs are rarely executed with a part of their input fixed or because the static input is tedious to identify. In order to apply specialization to programs that are not reexecuted, pratical specializers identify fragments of the program that are reexecuted with same (static) inputs in the course of the execution. (In Tempo [21] and DyC [35], these fragments are syntactic code blocks, e.g., procedures and loops.) A given fragment can be specialized for multiple values of its static input, in which case a run-time dispatch selects the appropriate specialized version of the fragment (or its original, unspecialized version) each time the program is about to execute the fragment. In this "fragment-based" version of specialization, it helps to distinguish two kinds of static inputs: arguments, which are passed to the fragment by value; and heap inputs, which are obtained from the heap. In many specializers, including ours, the dispatch mechanism handles the two kinds of inputs differently. It turns out that it is heap input that delivers powerful specialization capable of eliminating safety checks. For example, an array-bounds check can be specialized away because the array length is a static input obtained from the heap.

Our specializer differs from this standard model in two ways. First, we simplify the specialization process by relying on fragments that are (dynamic) execution traces rather

than (static) syntactic code blocks. Traces are simpler to work with because they are free of control flow merges. Second, in order to exploit static input from the heap while keeping the dispatch simple, we optimistically assume that the heap locations used as static inputs are not modified after specialization. As a result, the dispatch needs to examine only the arguments, rather than also checking the heap inputs or relying on external guarantees that they are static. The optimistic assumption is inexpensively verified on the fly, by monitoring stores into the heap locations.

The combination of these two features yields a very simple yet suprisingly powerful specializer. The simplicity is the result of online specialization (i.e., specialization without binding-time analysis) [63] that is further simplified by specializing traces created in the spirit of the Dynamo specializer [11]. At a high level, our process is to interrupt an execution at a suitable program point and form a trace by following a hot execution path. While forming the trace, we evaluate all statements depending only on the static arguments of the trace and on static heap locations. Instructions that cannot be evaluated are emitted to the specialized trace. The power is gained by the access to run-time values in the heap, and by optimistically exploiting heap locations that may eventually be overwritten, which enables specialization that would be illegal or very difficult to verify if one conservatively required invariance of these locations.

We decompose this process into solving three key problems:

- What is a suitable program point to start a beneficially specializable trace, and what are suitable static arguments to this trace?

- Which heap locations should be assumed to be constant?

- How to detect if the heap locations used as static inputs have been modified, and if so, how to invalidate any specialized traces that depend on them?

We outline our solutions below. The rest of this section elaborates.

Identifying profitable specializable traces. To make the problem manageable, we establish a (mild) restriction that the specialized trace has only one static argument input (it can have an arbitrary number of static heap inputs). Under this restriction, the problem boils down to identifying an instruction whose result value would be a suitable static argument input; this instruction will form the start of the trace. Given the start point, the trace is formed by following the execution; this process terminates when a benefit function decides that specializing further appears no longer profitable, due to the ratio of instructions that are currently being specialized away (see Section 2.6).

To identify suitable trace start points, we have developed a metric called influence that estimates the benefit of specialization when a given instruction would serve as the static input. Influence over-approximates the size of the forward dynamic slice of the candidate instruction, which itself overapproximates the benefit (see Section 2.3). The metric is used to identify a few candidate instructions, which are then tentatively specialized to a limited degree, and the most promising candidate is selected as the trace start point. Our measurements show that the influence metric is fast enough for a runtime environment.

Identifying constant heap values. We identify locations unlikely to be overwritten by means of a form of value profile called a *store profile*. The store profile predicts whether a memory address is a likely constant by remembering (a sample of) addresses written by the program (see Section 2.4). We have found this profile to be sufficiently accurate.

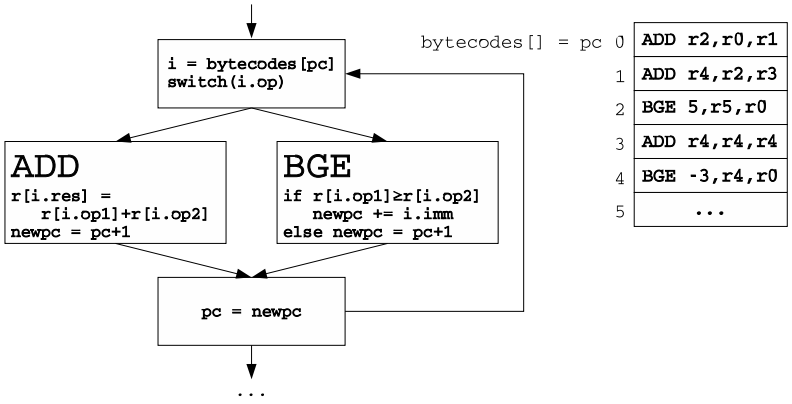


Figure 2.1: A program to specialize: a simple interpreter (left) and the set of bytecodes it is to interpret (right).

Since the store profile monitors individual concrete locations, its invariance detection is generally more precise than a static analysis working with a heap abstraction. Section 2.4.1 further discusses properties of the store profile. Thanks to recent advances in *sampling-based profiling*, the store profile can be collected with high accuracy, yet with overheads sufficiently low for dynamic optimizers [8, 17, 24, 39, 52, 66].

Invalidating optimistic specializations. Since the specializer cannot be sure that the memory locations that it assumes are constant will not change, it must ensure that if these locations are updated, specialized traces that rely on them are invalidated. We detect the invalidation of assumed invariants with write barriers, greatly optimized by relying on Java's type safety; overhead is generally well under 10%. Invalidation can occur even while the specialized code is being executed. We describe this system in Section 2.5.

2.2.2 Example

In this section, we illustrate the specialization procedure with a specific example. Figures 2.1 and 2.2 show a simplified interpreter before and after specialization. The interpreter is given an array of bytecodes, which it executes in turn, using a program counter *pc* to keep track of the current bytecode. We show only two bytecode types, `ADD`, which adds two values and increments *pc*, and `BGE`, which compares two values. During execution, a light-weight method profiler identifies the interpreter function as a hot method, and the specializer is invoked to assess the method for specialization potential. It runs the influence algorithm, described in Section 2.3, on the function and its associated dynamic execution information. The algorithm estimates the number of dynamic instructions that will follow each instruction and selects those with the most following instructions: the assignment to *i*, the `switch`, and the assignment to *pc*. These candidates are then tentatively specialized to a limited degree to gauge their effectiveness.

The candidate instruction that results in the best optimization opportunties turns out to be the assignment to *pc*. (If no candidate instructions revealed good optimization opportunities, the specializer would abort at this point.) A hot value profile, described in Section 2.7.1, indicates that the hottest values of this instruction are 0 and 2.

Based on these hot values of *pc*, the specializer (i) creates two traces, one for each of these values, and (ii) a dispatcher that jumps to the corresponding trace or falls back to the original code as appropriate; see Figure 2.2.

We will now walk through the creation of these traces, starting with hot value 0. Given the starting program point p that assigns 0 to *pc*, the specializer performs standard

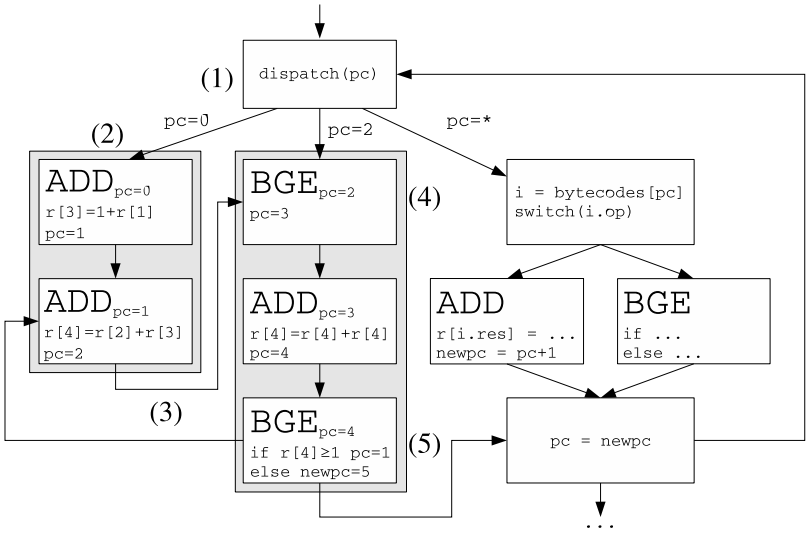


Figure 2.2: The interpreter in its specialized incarnation. The two shaded columns are specialized traces, while the diamond on the right is the original code.

constant propagation starting from p, with a few modifications: (i) it assumes pc is a constant equal to 0; (ii) it unrolls loops where appropriate; (iii) it evaluates load instructions on the concrete memory state of the program at the time of specialization, queries the store profile (detailed in Section 2.4) to see if the loads are of constant values, and eliminates them if so.

In our example, the constant propagator proceeds to the next instruction, `i = bytecodes[pc]`. The value of pc is now assumed to be 0, and the store profile indicates that `bytecodes[0]` is constant, so this load is eliminated (along with its associated null check and bounds check), and the contents of the fetched bytecode (stored in i) are themselves propagated further as constants.

The next instruction, the `switch`, is a branch. The specializer handles branches in two ways. If the predicate can be fully evaluated (i.e. all of its terms are constants known by the propagator at the time of specialization) then the branch is eliminated and the correct

path is followed. If not, the specializer uses an edge profile to determine the most likely branch outcome, inserting a failsafe check to the other branch.

In our example, since i is known to be `ADD r2,r0,r1`, the branch is eliminated and the specializer proceeds to the `ADD` basic block (2). In this block, several loads and null checks are eliminated since i is known. The pointer field `r` is a heap constant (it always points to the same array of "registers"), so its array size is inlined and several bounds checks are also eliminated. Furthermore, the store profile indicates that `r[0]` (corresponding to the interpreted program's register $r0$) is a heap constant with value 1, and so its load is eliminated as well. In total, the specialized interpreter requires only five instructions and one load to process this bytecode, whereas the unspecialized interpreter took 20 instructions, including seven explicit loads.

However, the specializer has made optimistic assumptions via the store profile about the invariance of several heap locations: the interpreter's `r` field, the instruction at `bytecode[0]`, and the value at `r[0]`. The elimination of their corresponding load instructions is safe only until these values are modified; thus the specializer must now monitor them for updates, as described below.

The following `ADD` block is automatically appended to the trace in a similar fashion, since the value of pc is known and propagated. Unrolling could continue further, but since there is already a trace being developed for the same constant values, the traces are *linked* together (3), eliminating the need for further code generation or an additional dispatch on subsequent executions.

The second trace, specialized for $pc = 2$, begins with a `BGE` (4). It normally

would require a jump (as in Figure 2.1) conditioned on the values of two heap locations, but the profiler identifies these locations as constants. Based on the two constant values, the conditional evaluates to the false (fall-through) block, all unnecessary instructions are eliminated, and unrolling continues.

After specializing an additional `ADD` instruction, the specializer reaches the `BGE` at bytecode index 4. It is unable to fully evaluate the predicate, since `r[4]` is not a constant. Thus, it consults an edge profile and determines that the true branch (the one that performs the jump) is most likely to be taken. It turns out that this branch sets $pc = 1$, and there is already a specialized block with the same set of constant values, so rather than generating new code, the specializer issues a jump linking the two specialized blocks together. Since the specializer cannot be sure that this branch will actually be taken every time, a failsafe jump to the corresponding point in the unspecialized code is inserted as well (5).

At this point, both specialized traces have terminated via trace linking. Traces can also end when too few instructions are being optimized away, or too many successive speculative branch decisions are made to the point that the probability that the code being generated actually gets executed is too low.

The specializer must ensure that the assumptions it made about particular heap locations being invariant hold. Thus, it inserts write barriers at memory store instructions indicated by the type system and address information. For instance, to track an update to $r[0]$, a write barrier need not be inserted at the store to $r[3]$ in the first specialized `ADD` block, since it is guaranteed to write to a different memory location. This procedure is described in detail in Section 2.5.

Finally, the specialized traces are compiled down to machine code and inserted into the execution stream.

Implementation. The entire specialization process runs completely independently from program execution to execution. Our dynamic specializer is implemented in the Jikes RVM Java virtual machine [30]. It leverages Jikes's solutions to many common issues involved in the general dynamic optimization problem, such as an efficient sampling-based profiling infrastructure, fast and efficient code generation, identification of hot methods, and on-stack replacement of recompiled methods [6, 7, 31], so we do not address these issues in this chapter.

2.3 Determining Specialized Regions: Influence

Our dynamic specializer creates a specialized trace by stopping the execution of a function at a dispatch instruction i, and then adding subsequent instructions to the trace until an end condition is met. The length and benefit of a trace is largely dependent on the dispatch instruction, and in this section we discuss our method for selected a good one. The end conditions are described in Section 2.6.

A simple algorithm for finding a good dispatch instruction is to simulate the specialization procedure on each instruction in the function without generating any actual code, and choose the one that results in the greatest optimization opportunity. In a runtime context, this approach is prohibitively costly for larger functions, since the specializer may have to be run on hundreds of instructions before selecting just one for which to generate code.

Thus, our specializer considers a small number of candidate instructions (in our

implementation, five), tentatively specializes each of them to a limited degree, and selects the most beneficial one. If no beneficial dispatch points are found, specialization is aborted.

The challenge arises in designing an efficient metric that consistently selects good candidates for tentative specialization without actually having to specialize them itself. Below, we describe several simple heuristics we tried that failed to work. We then present a better technique, *influence*, that does work.

Execution frequency. Instructions are ordered by a combination of execution frequency and hot value consistency, the cumulative frequency of their ten most frequent values. The idea is that the specialized traces of this metric's highest-ranked instructions will be executed very frequently. This approach fails because cold instructions (for instance, outside of a loop) can start beneficial traces that span multiple loop iterations, whereas traces from hot instructions inside loops often are limited to a single loop iteration.

First-n. Another heuristic that has limited success is based on the observation that function arguments or early values computed from them often make good dispatch points. The First-n heuristc simply suggests the first n instructions in a breadth-first traversal of the CFG. However, this heuristic also overlooks suitable instructions that precede backedges further down in the CFG. For instance, the `pc = newpc` instruction in Figure 2.1 would not be ranked highly by this heuristic, even though it greatly affects the execution of the program.

Control-flow domination. Instructions are ordered by the number of instructions in the control-flow graph they dominate; the idea is that these instructions at least have the potential to affect many others. This approach is also inaccurate because the

optimal dispatch point may dominate very few static instructions. For instance, loop variable updates (e.g. `i++`) can be good specialization candidates, enabling loop unrolling, but generally are not dominators.

All of these heuristics are "brittle" in the sense that they only find good dispatch points if they are provided with functions with a particular static control flow structure, while a dispatch point's true benefit seems to be more robustly tied to its function's dynamic execution behavior.

Our solution: Influence. This brittleness led us to recast the problem in terms of approximating a *forward dynamic slice*. Given an instruction i and a value v, the forward dynamic slice is the set of dynamic instructions affected when the value computed by i is v; see Tip [74] for a good summary.

The motivation behind using slices is that instructions that contribute to v's specialized trace's benefit are necessarily from i's forward dynamic slice. Thus, computing a forward dynamic slice for each of an instruction's hot values yields an over-approximation of the potential benefit of specializing on that instruction.

However, computing a dynamic slice is very costly; even the most efficient algorithms require extensive preprocessing [77], a luxury we cannot afford in a purely runtime environment. Thus we make a key simplifying assumption: we approximate the slice in a data-independent fashion by including all of the instructions in the CFG that dynamically follow instruction i. Thus, we simply need to compute the number of dynamic instructions that follow i.

Formally, we utilize a function f's control-flow graph and dynamic basic block and

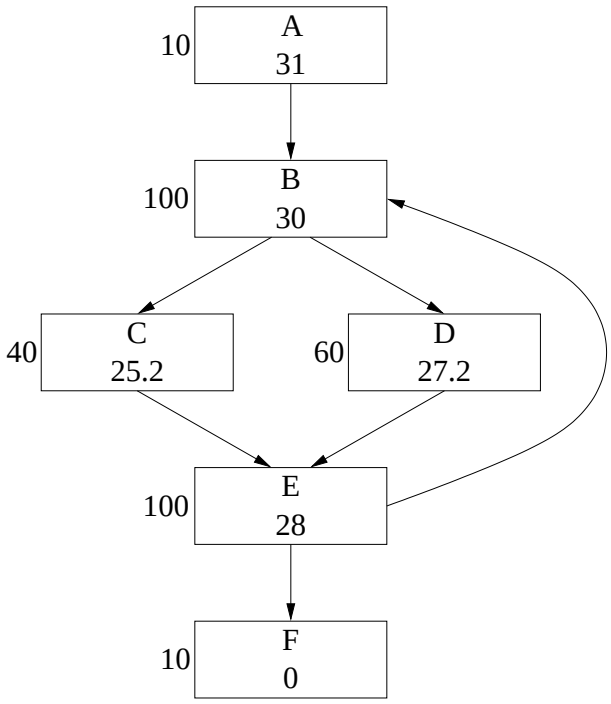


Figure 2.3: Influence numbers for a sample control flow graph. The number to the left of each block is its dynamic execution count, and the number inside is the influence.

edge execution counts. These two items induce a (possibly infinite) set of execution traces of the function, where a trace is a sequence of edges $e_1 e_2 ... e_n$ from the start of the function to the end. Let $count(x)$ be the dynamic execution count of graph component x. Assuming independence of branch outcomes, each trace t can be assigned a probability of occurrence, by taking the product of the probabilities of its branch choices:

$$occurrence(t) \equiv t \text{ is followed on an execution of } f \tag{2.1}$$

$$\mathbf{Pr}[occurrence(t)] = \prod_{\text{edge } e=(m,n) \in t} \frac{count(e)}{count(m)} \tag{2.2}$$

Together, the traces and their probabilities constitute f's *expected execution set*, or *EES*, dictated by the profile.

We define the *influence* of an instruction i with respect to a particular trace t as

$$\mathbf{Pr}[reach(m,s)] = \sum_{\text{edge } e=(m,n)} \frac{count(e)}{count(m)} \cdot \begin{cases} 1 & \text{if } n = s \\ \mathbf{Pr}[reach(n,s)] & \text{otherwise} \end{cases} \tag{2.5}$$

$$\mathbf{E}[len(m)] = 1 + \sum_{\text{edge } e=(m,n)} \frac{count(e)}{count(m)} \mathbf{E}[len(n)] \tag{2.6}$$

$$influence(i) = \mathbf{Pr}[reach(start,i)] \cdot \mathbf{E}[len(i)] \tag{2.7}$$

Figure 2.4: Equations for computing the influence of an instruction i as the solution of two systems of linear equations. *start* is the function's entry instruction.

the length of the subtrace from the first occurrence of i along t until the end of the trace:

$$influence_t(i) \equiv length(e_k ... e_n) \tag{2.3}$$

where e_k is the edge from the first occurence of i.

The overall influence of i is the expected length of this path over all traces, computed as the influence per trace, weighted by each trace's probability of occurring:

$$influence(i) \equiv \sum_{t \in EES} \mathbf{Pr}[occurrence(t)] \cdot influence_t(i) \tag{2.4}$$

See Figure 2.3 for a sample influence computation. The influence of 30 for B means that an average of 30 instructions follow the first execution of B on a given function invocation. Note that the influence of instruction C is nearly as great as that of B, even though it is executed on only 40% of loop iterations. This is because it affects *all* instructions after the *first* time it is executed — a property that we were unable to capture with other heuristics.

Unfortunately, the existence of loops in control flow graphs makes a computation

$$\mathbf{E}[num(m)] = \frac{count(m)}{count(start)} \tag{2.8}$$

$$\mathbf{E}[slen(m,s)] = 1 + \sum_{\text{edge } e=(m,n)} \frac{count(e)}{count(m)} \cdot \begin{cases} 0 & \text{if } n = s \\ \mathbf{E}[slen(n,s)] & \text{otherwise} \end{cases} \tag{2.9}$$

$$influence(i) = \mathbf{E}[num(i)] \cdot \mathbf{E}[slen(i,i)] \tag{2.10}$$

Figure 2.5: Alternative equations for computing the influence of an instruction i as the solution to a single system of linear equations. *start* is the function's entry instruction.

of influence from Equation 2.4 infeasible because the EES can be infinite in size.

Instead, we can recast the influence of i as the combination of two problems (Equation 2.7, Figure 2.4): the probability of ever reaching i during an invocation of f (*reach*, defined in Equation 2.5), and the expected length from i to the end of the function once i is reached (*len*, defined in Equation 2.6). Both are recursive definitions that produce systems of linear equations.

We use dynamic execution count data to simplify the problem further. We solve just one system of linear equations, by directly computing the expected number of times i is executed *per invocation* of f (*num*, defined in Equation 2.8). We can then view each trace as a series of shorter paths from an instance of i to the immediately next instance, or (in the case of the last short path) to the end of the function, and define a system of equations to compute the expected length of such a path (*slen*, defined in Equation 2.9). The influence of i is the product of these two expectations (Equation 2.10).

The systems of equations in Figures 2.4 and 2.5 can be solved via Ramalingam's data flow frequency analysis framework [61]. Ramalingam cites a number of algorithms for solving such systems on a reducible control flow graph in almost-linear time. For instance, a simplified version of Tarjan's algorithm [73] runs in $O(e \log v)$ time, and the Allen-Cocke interval analysis algorithm [3] runs in linear time on graphs with a bounded loop nesting depth. We also present a truly linear time algorithm for solving influence on a reducible control flow graph in Section 2.3.1.

Our implementation of influence operates on Java bytecodes. While Java programs must result in bytecodes that represent reducible control flow graphs, it is possible to construct irreducible graphs with arbitrary Java bytecodes (perhaps with a compiler for another language that outputs Java bytecodes), since there is a `goto` bytecode. Our implementation supports reducible control flow graphs only.

Figure 2.7 compares influence against the other heuristics mentioned above on the actual Java programs described in Figure 2.6. Each of the other metrics successfully ranked the optimal specialization point highly for several programs, but failed on the rest of the programs. Influence appeared to succeed at identifying good dispatch instructions both accurately and consistently.

Benchmark	Description	Input(s)
`convolve`	Transforms an image with a matrix; from the `ImageJ` toolkit	various images, fixed matrix (`conv-VI`)
		fixed image, various matrices (`conv-FI`)
`dotproduct`	Converted from C version used in DyC [35]	sparse constant vector: 75% zeros
`interpreter`	Interprets simple bytecodes	bubblesort bytecodes (`i-sort`)
		binary search bytecodes (`i-search`)
`jscheme`	Interprets Scheme code	partial evaluator
`query`	Performs a database query; converted from DyC	semi-invariant query
`sim8085`	Intel 8085 Microprocessor simulator	included sample program
`em3d`	Electromagnetic wave propagation (intentionally unspecializable)	-n 10000 -d 100

Figure 2.6: Description of benchmarks and their inputs.

Benchmark	Total	ExecFr	Dom	First-n	Inf
conv-VI	39	34	1	1	1
conv-FI	39	33	2	2	2
dotproduct	8	5	1	1	1
i-sort	52	3	46	52	4
i-search	52	3	46	52	4
jscheme	29	26	1	1	1
query	20	12	1	1	1
sim8085	86	3	31	21	5

Figure 2.7: Rank (1 is best) of the most beneficial instruction in the principal function of various benchmarks according to several ordering heuristics. Total is the total number of candidate instructions for each principal function.

2.3.1 Linear-time Influence Algorithm

In this section, we present an algorithm for computing the influence of an instruction n that is linear in the number of instructions blocks in the graph. We exploit the fact that the graph is reducible to precompute closed-form summaries of the needed expectation properties for loops.

Recall that the influence is the expected path length from the first occurrence of n to the end of the function. If the control flow graph is acyclic, it is easy to compute the influence of n via depth-first search, as there are a finite number of paths.

If we allow loops, but require that n is not in a loop itself, we can compute influence in the following manner. Assume that we have a way to compute the expected path length, l, from the beginning of a loop until it is exited (we describe such a way below). Then, since n is not in a loop, we can replace each loop in the graph by a summary node of "length" l, and compute influence the acyclic way.

The difficulty arises if n is in a loop. Consider a standalone loop, as in Figure 2.8. The loop can have arbitrary branches and so on inside it, as long as it does not have any

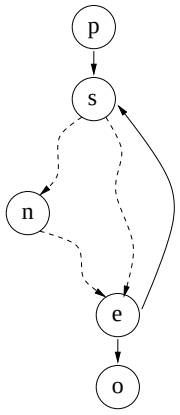


Figure 2.8: A loop. The dotted edges represent arbitrary acyclic control flow.

inner loops. (We will discuss nested loops further below.)

If n is in a loop, we can reformulate the expected path length from n to the end of the loop as the probability that the loop is reached in a given function invocation, times the probability of ever getting to n from the start of the loop, $F(n)$, times the expected path length from n to the end of the loop, $L(n)$. (The expected path length through the rest of the function is computed as normal.)

$$influence(n) = \frac{count(p)}{count(start)} F(n)L(n) \tag{2.11}$$

where *start* is the first instruction of the function.

To compute $F(n)$ and $L(n)$, we first need to describe some properties of the loop. See Figure 2.8 for a sample loop. s is the loop's entry node, and e is the loop's exit node.

b. Probability of taking the backedge. Simply $\frac{count(backedge)}{count(e)}$.

l. Expected length of a loop iteration, from s to s. Can be computed by DFS since all paths within the loop are acyclic.

f(n). Probability of reaching n from s on *one* arbitrary loop iteration (i.e. without

reaching s again). Computed like l above.

r(n). Expected length of an acyclic path from n to o. Computed like l above.

Computing $F(n)$

How can execution go from s to n? It can go directly on the first iteration of the loop, or miss n and hit it on the second iteration, or miss it again and hit it on the third iteration, and so on. Formally,

$$F(n) = \sum_{i=0}^{\infty}((1 - f(n))b)^i \cdot f(n) \tag{2.12}$$

Each $(1-f(n))b$ represents a loop iteration that missed n, and the final $f(n)$ is there because eventually a successful path to n must be taken. The closed form of $\sum_{i=0}^{\infty} r^i$ when $r < 1$ (as b must be) is $\frac{1}{1-r}$, so we have

$$F(n) = \frac{f(n)}{1 - (1 - f(n))b} \tag{2.13}$$

Computing $L(n)$

The expected length of a (cyclic) path from n to o is the expected length of the acyclic path from n to o, $r(n)$, plus the probability of doing one additional loop before exiting times the expected length of the loop, plus the probability of doing two additional loops before exiting times twice the expected length of the loop, etc.

$$L(n) = r(n) + (1 - b)\sum_{i=0}^{\infty} ilb^i \tag{2.14}$$

The $(1 - b)$ factor is needed because eventually the exit edge from e to o must be taken.

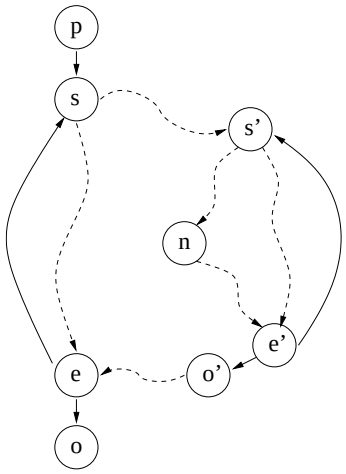


Figure 2.9: Nested loops. The inner loop variables have been primed.

The closed form of $x = \sum_{i=0}^{\infty} ir^i$ when $r < 1$ is $\frac{r}{(1-r)^2}$. Thus we have

$$L(n) = r(n) + \frac{lb}{1-b} \tag{2.15}$$

With these closed forms, we can compute the influence of any node in a loop relative to the rest of the loop in linear time.

Handling Nested Loops

Consider a properly formed nested loop, as in Figure 2.9. The variables in the inner loop have been primed, and we prime associated loop properties as well (e.g. the probability of taking the inner backedge is b'). We assume that these properties have already been computed.

Computing the influence of nodes that are in the outer loop but not in the inner one is straightforward: we can just replace the inner loop by a summary node with length of the expected path length through the loop. This number is identical to computing $L(n)$

from the top of the loop — one pass through the loop plus the chance of doing another iteration times its length, etc.:

$$l' + (1 - b') \sum_{i=0}^{\infty} i l' b'^{i} \tag{2.16}$$

$$= \quad l' + \frac{l'b'}{1 - b'} \tag{2.17}$$

$$= \quad \frac{l'}{1 - b'} \tag{2.18}$$

Thus this value is exactly equal to $F(s')L(s')$,

$$F(s')L(s') \quad = \quad \frac{f(s')}{1 - (1 - f(s'))b'}(r(s') + \frac{l'b'}{1 - b'}) \tag{2.19}$$

$$= \quad \frac{1}{1}(l' + \frac{l'b'}{1 - b'}) \tag{2.20}$$

$$= \quad \frac{l'}{1 - b'} \tag{2.21}$$

which is to be expected, since $F(s')L(s')$ computes the same value: the expected path length of one complete execution of the inner loop.

To expand the influence computation of nodes in the inner loop to the outer loop, we use the loop properties of the inner loop that we have already computed.

Specifically, to compute the influence of the node n in the inner loop, we determine the needed variables:

b. The backedge weight from e to s, as normal.

l. Also computed normally for the outer loop, using the summary node for the inner loop.

f(n). Probability of reaching n from p. This is just $f(s')F'(n)$: the probability of getting to s' from p times the probability of ever reaching n from s'.

r(n). Expected length of an acyclic path from n to o. This is $L'(n) + r(o')$: the expected length of the path from n to o' plus the expected length of the path from o' to o.

Note again that, given our old cached values from the inner loop, these new values are computed in linear time using only the nodes in the outer loop. We then apply these values to the closed-form influence equation above for the outer loop. We keep expanding outward in this fashion, and since the same node is never visited twice, the algorithm is linear in the size of the graph.

2.4 Identifying Heap Constants: The Store Profile

Loads from memory can be safely removed if the specializer knows that the values at those memory locations will not change throughout the rest of the program execution. Alternatively, the specializer can employ the more aggressive strategy of assuming that certain locations will remain constant. This strategy, which our specializer employs, uncovers more constants, those for which the guarantee of invariance is impossible or very difficult to acquire. However, if such an optimistic assumption turns out to be false – an assumed constant memory location is actually updated later in the execution – any optimizations that depend on it must be invalidated. Thus, since the cost of an incorrect assumption is high, a technique for making *accurate* optimistic guesses, guesses that are most often right, is needed.

We have designed the store profile to make such guesses. Ideally, it would determine

$const(a)$: will address a be updated in the remainder of execution?

While a number of static analyses [5, 46] can conservatively approximate this predicate, our

dynamic specializer requires as efficient a technique as possible, and also we would like it to exploit as many constants as possible, even those that may not reveal themselves to a static analysis. Thus, we use the past behavior of the program, from the start of execution until specialization time, as a guide to future execution. We simply assume that if a location has been constant, it will remain constant. This approximation of $const(a)$ is efficient but not conservative:

$var(a)$: has a been written to more than once?

If $var(a)$ holds, then a is assumed to remain variant for the rest of the execution; if not, then a has only been initialized and we optimistically consider it constant.

Monitoring every store is too expensive; hence, our specializer employs sampling-based profiling, described in greater detail in Section 2.7.1, in which only one out of every 1000 or so stores is monitored. The profile thus evaluates the following predicate, which approximates $var(a)$.

$w(a)$: does a random sample of observed stores contain a store to address a?

If so, then a has almost certainly not been constant, since it must have been updated enough to be detected by the profiler. Furthermore, rarely written locations are likely to be identified as constants, which can allow for beneficial specializations until the next time they are updated.

Implementation. The store profile records the addresses of all sampled memory updates in a hash table. The sampling interval is 1000. Since the profiler tracks the exact addresses of store instructions, it is able to identify constants as small as individual object fields or array elements.

Overhead. The time and space overheads of the store profile are generally under 5%; more numbers and a discussion are presented in Section 2.7.1.

Evaluation of accuracy. The store profile's accuracy can be assessed by measuring the fraction of memory locations it claims are constant that actually remain constant for the duration of the program execution.

Divide a program's execution into two phases: Phase 1, before the specializer is invoked, and Phase 2, afterwards. Let L be the set of *all* memory locations read by the program in Phase 1. L is an upper bound on the number of possible constants the specializer could infer, since it is a superset of all the concrete addresses the specializer could have identified.

Let S_w be the set of locations detected as written to by the store profile in Phase 1. $C_w := L - S_w$, then, is the set of memory locations the store profile reported as constant.

Let W be the set of *all* locations written to in Phase 2. $C := L - W$ is then the set of locations known in Phase 1 that actually were constant from specialization onward.

We formally define the accuracy of the store profile as $|C \cap C_w|/|C_w|$: the fraction of claimed constants that really were never modified by the end of the program.[1]

We instrumented 12 Java programs to compute this value. The mean accuracy was 95.6%, indicating that (1) locations that start out constant overwhelmingly tend to remain constant, and (2) variable locations tend to be updated frequently enough to be observed by the profiler. These observations provide a reasonable basis to employ the store profile, as few invalidations should occur as a result of its predictions.

[1]Note that some of the locations that were updated may have remained constants, if the updates did not change the actual value at those locations; this is known as the silent store phenomenon [49].

2.4.1 Benefits of Heap Profiling

This dynamic store profile, in addition to being very simple to implement, also enables more powerful specializations than existing hybrid approaches to heap constant detection. It does not require programmer understanding of a program's heap data structures, it is not susceptible to unsound programmer mistakes, and it can detect constants in library, dynamically-loaded, or otherwise unannotatable code. Furthermore, it exposes a new class of constants to specialization. Consider the three classes of constants below.

1. *Compile-time constants*: their values can be determined statically.
2. *Run-time constants*: they are known at compile-time to be constant, but their values can only be determined at run-time.
3. *Transient constants*: they are only constant for particular program inputs or intervals of execution.

Compile-time constants can be optimized by normal static compilers. Run-time constants include, for instance, the bytecodes fed to an interpreter: data that we know will not change, but whose values we can only access at run-time. This class of constants provides the most common optimization opportunities for current specializers: during their off-line phase, they annotate these constants and specialize with respect to them at run-time.

Transient constants include nodes in a semi-invariant data structure, or memory locations that are only constant depending on the program's input – those that static analysis or annotation would not identify as constant. A typical example is the memory region associated with an interpreted program. It may contain constants, but whether they exist

and where they are is naturally dependent on the particular program being interpreted. This class of constants is especially difficult to pinpoint via static annotations of the interpreter, whether produced by a programmer or a tool, since neither generally has access to each particular interpreted program. Identifying these constants enables the interpreted program to be specialized, not just the interpreter.

Another example is a database query processor. In the figure below, it iterates over the boolean predicates in a query, applying them in turn to each item in a dataset.

```
public boolean Satisfies(Predicate p) {
  switch(p.conditionType) {
    case LT:
    if (p.LHS.resolvedVal <= p.RHS.resolvedVal)
      ...
  }
}
```

Many database queries are *prepared*, in the sense that the predicate operands are fixed, but some of the actual values are repeatedly modified as the query is submitted over and over to the database[2]. Thus while much of the query data structure is constant from query to query, portions of it are updated during execution, so the structure is only partially invariant: some `Predicate`s are constant while others are not. However, `Satisfies` can still be specialized with respect to the constant `Predicate`s in the query, resulting in fewer loads when invoked on them.

Static annotation approaches are unable to capitalize on this opportunity. A *class-based* source code annotation, in which the `Predicate` `LHS` field is marked run-time constant, would fail since some variable `Predicate`s update their `LHS` field. Similarly, an *expression-*

[2]This technique is generally employed for security (malicious users cannot alter the structure of the query) and efficiency (the query does not need to be re-parsed each time).

based annotation, in which `p.LHS.resolvedVal` is marked constant, would also fail since some of the `Predicate`s passed to `Satisfies` are not constant.

In contrast, a dynamic invariance detector, which monitors the invariance of concrete memory locations, finds all three types of constants. Thus, just those `Predicate`s that are constant can be identified, and `Satisfies` specialized on them. Furthermore, the lack of abstraction makes individual fields or array elements distinguishable: loads from the individual constant fields of modified `Predicate`s can also be eliminated.

2.5 Maintaining Soundness: Invalidation

Runtime execution profiles are no guarantee of future behavior; if a memory location that the store profile claims is constant gets updated, any specialized traces that depend on it must be be invalidated. Thus, a sound technique is needed for (1) detecting updates to particular memory locations and (2) invalidating the corresponding specializations, ensuring that control flow is safely returned to their unspecialized versions. We discuss our solution to these two requirements below, and then describe some alternative detection strategies.

2.5.1 Detecting Updates To Assumed Invariants

During specialization, the specializer accumulates a list of the memory locations it has assumed are invariant, as well as their types. The update detector must use this information to monitor all of these memory locations for updates.

Our solution is simple: to insert write barriers in front of all stores in the program that might update any of these locations. In a naive implementation, the update detector

iterates over all stores in the program and inserts a check at each store to test the address to which it is storing against the set of locations to be monitored. If the address matches one of these locations, an invalidation is triggered for all the specialized traces that assume that location is constant.

Naturally, this approach can have a prohibitive overhead if care is not taken. Our implementation attempts to be efficient in two respects:

Reducing the number of inserted write barriers. Unlike C's, Java's type system is precise enough to eliminate write barriers for many stores.

Say a memory location to be monitored, l, corresponds to field `foo` of object class `Bar`. If `foo` was declared in a parent class `Baz`, only writes to a field named `foo` in object references that are statically subclasses of `Baz` need to have write barriers inserted[3]. Similarly, for arrays, if l is a member of a `short[]` array, only stores to `short[]` arrays need to be checked. Also, stores in constructor methods that write to `this` can be ignored, since they are necessarily storing to newly allocated objects.

Of particular concern is the insertion of write barriers into specialized code. This code is known to be very hot, and inserting too many write barriers can cause debilitating overheads. Luckily, the exact addresses of most stores in specialized code are known through constant propagation. In these cases, we can compare these addresses directly to the set of locations to monitor at specialization time and nearly always rule out the corresponding stores. If array indices are not known, array base addresses can be compared to eliminate barriers for stores to arrays that have no monitored elements.

[3]It is also possible to store to arbitrary objects via reflection. We handle this exceptional case by explicitly modifying the `java.lang.reflect` methods in the VM to notify the specializer of any stores.

Benchmark	Constant Memory Locations	Barriers Inserted	Steady-State Overhead
conv-VI	153	72	9%
conv-FI	9	5	3%
dotproduct	102	5	-1%
i-sort	50	14	6%
i-search	58	10	1%
jscheme	482	4	3%
query	84	21	2%
sim8085	25	8	12%

Figure 2.10: Invalidation detection information and overhead. The steady-state overhead compares the a specialized program with invalidation detection against the equivalent specialized program without it.

To reduce the cost of looking through every compiled method for stores, during method compilation the system notes the object types and field names to which each method stores. This information is consulted during write barrier insertion to yield a list of just the methods that need to have barriers inserted. Inserting write barriers into methods that have not yet been compiled by the VM (e.g. have not yet been invoked by the program) is deferred until those methods are compiled for the first time. This allows us to avoid recompiling most of the Java class libraries.

See Figure 2.10 for a table of the number of write barriers that had to be inserted in the benchmarks presented in this chapter.

Reducing the cost of executing the write barriers.

Our write barrier implementation inserts a hash table lookup before a store that identifies if the address to be modified points to any of the locations that the specializer assumes are invariant.

By itself, this lookup requires several arithmetic computations and memory loads,

and can interfere with cache locality. In practice, we found that the barrier overhead was too great. Thus, the specializer employs a bit in the field's enclosing object's header as a first pass to weed out stores to objects that have no invariant fields. An object's bit is set by the specializer when it makes the assumption that a field in that object is invariant. If the field is static, the bit is set in its corresponding Class object. On writes to that field, the bit of the enclosing object (or the Class object, if the field is static) is tested and only if it is set the hash table lookup is performed. (The lookup is still necessary because it is possible that the field being updated is not invariant, even though the enclosing object's header bit is set, because the object happens to contain another field that has been flagged as invariant.)

To minimize overhead on array element updates, the specializer creates a bitmask summary of the assumed-invariant array indices, as in the Diduce system [38], and inserts code to check the array index to be updated against this mask before executing the lookup. The Diduce masking procedure is conservative, so if the updated index does not match the mask, it cannot write to any invariant addresses, and the hash table lookup can be avoided. For a particular array element update, if the base address of the array is known, the specializer constructs a bitmask containing only the invariant indices of that array; if not, the bitmask is constructed from the invariant element indices of all arrays of the given type.

See Figure 2.10 for performance numbers. The detection mechanism executed with steady-state overheads of under 15%. Since this overhead is only incurred on programs that are actually specialized, and the speedup of specialized programs tends to be dramatically

larger, as shown in Section 2.7, we feel that it is suitable for a runtime environment.

2.5.2 Performing Invalidation

Once a memory location that a specialized trace assumes is invariant has been updated, that trace must be invalidated and discarded. If the trace is not on the execution stack when such an update occurs, it is easy to invalidate: the specializer assigns each trace its own boolean variable `isInvalidated`, which is set to `true` upon invalidation. The trace's dispatch is designed to check if it has been invalidated every time before invoking it, and if so to revert control to the unspecialized code. This technique does not require any recompilation upon invalidation, and has negligible overhead.

However, difficulty arises if the trace is already on the stack at the time an errant write is detected. Control must revert to unspecialized code when the trace resumes executing, regardless of where in the trace execution happens to be. We are not aware of any mechanisms in existing specialization systems that handles this case.

Our solution is relatively straightforward. During specialization, the specializer identifies all the instructions in the trace that could lead to an invalidation. It then ensures that these points are synchronized with the corresponding points in the unspecialized code so that control can immediately resume at the corresponding unspecialized points in a sound fashion if invalidation does occur. Finally, the specializer inserts conditional jumps at all of these points to check for invalidation; these jumps are identical to the check inserted at the beginning of the dispatch.

The only instructions that can invalidate a specialization are stores that require

write barriers (as determined by the write barrier insertion method described above), calls to other functions (which might have such stores), and compiler-inserted yield points (which might cause a context-switch). Thus, the specializer only inserts `isInvalidated` checks after these instructions.

Synchronization of specialized and unspecialized code is simple, as the specializer already uses the same registers where appropriate. (Since most register variables are constant-folded in a specialized trace, this technique generally adds little register pressure.) Basic blocks are split as necessary to ensure that jumps can be made after potentially invalidating instructions.

This invalidation procedure has several benefits: it ensures soundness by immediately transferring control to unspecialized code, it is easy to implement, and it does not require recompiling a specialized method upon invalidation.

2.5.3 Alternative Detection Strategies

Below we discuss a number of other potential detection techniques.

Dispatch detection. The simplest way to check for invalidation is to insert a test at the beginning of a specialized trace that compares the actual contents of the trace's assumed constant locations to the expected contents. This approach is suited for when the specialized region encompasses a CPU-intensive, memory-light computation. Consider a function

```
public BigInt[] Factor(BigInt num) { ... }
```

dispatched on *num*, in which the computation might be very expensive but the data to check (the locations corresponding to a `BigInt`, perhaps several words) can be verified very quickly. This technique is akin to simple caching.

GC-based detection. Copying collectors are already good at moving objects around in memory. If one is being used, the specializer can tell it to move objects containing assumed constant fields to special read-only pages, so that any writes to them will issue a page fault that can be trapped. This approach is very efficient for detecting writes to assumed constants, but there is the caveat that writes to other, perfectly mutable fields of the selected objects will trigger page faults as well. Thus it should be employed if the store profile indicates that these other fields are written to infrequently, or if there are only a few of them.

Mondrian hardware support. Witchel et al. [75] have introduced Mondrian memory protection, a fine-grained memory protection scheme that relies on hardware support. In this scheme, permissions are granted to memory segments as small as individual words. Using it, the specializer can grant read-only permissions to assumed constant memory locations. Whenever these memory locations are written, the memory protection scheme traps to software that can perform invalidation. An upper bound on Mondrian overhead is 9%, when every object in memory is protected; in practice, the number of objects that need to be protected is small (see Figure 2.10), so we estimate runtime overhead to be less than 5%. Transmeta already employs similar (albeit more restricted) fine-grained memory protection in its Crusoe processor [26].

2.6 Trace Creation

In this section, we describe the mechanisms used to create specialized traces, as well as some key implementation details.

Assume we have identified a suitable dispatch point instruction i and one of its hot values v, for instance the assignment of the value 2 to pc in Figure 2.1. The specialization procedure creates a specialized trace starting from i, with the assumption that i resulted in v. The procedure uses a simple benefit analysis, presented in detail in Section 2.6.1, to identify when to end the trace; the relevant values are the *net benefit* of a trace, the estimated total number of runtime cycles it will save, and the *instantaneous benefit*, an estimate of the benefit that will accrue from growing the trace further. We present synchronous and asynchronous variants of the specialization procedure.

Synchronous version. The synchronous specialization procedure adapts Dynamo-style trace creation [11]. Dynamo is a transparent dynamic optimization system that begins execution by interpreting a program. Counters are kept at loop headers, and when execution reaches a hot-enough loop header, the system starts generating optimized straight-line code alongside the code it is interpreting, stopping at a backedge. The next time execution reaches this particular point, the optimized trace is natively executed instead.

This model suggests a natural way to construct specialized traces. Traces are created synchronously over successive executions of the dispatch point. The synchronous specialization procedure is quite simple: it intercepts execution when the dispatch point i is reached; assume that i assigns v to the variable x. Like Dynamo, it then interprets the following code, creating a trace along the way. Forward branches whose conditionals are

constant are eliminated. Those that are not runtime constant assuming $x = v$ are evaluated based on the current execution state, but a fall-back jump to unoptimized code is inserted in case the outcome is different in a future execution of the trace.

This trace creation procedure differs from Dynamo's in the following respects. Traces can begin at any given program point, not just at loop headers. Constant propagation is seeded with the initial hot value, and also utilizes profile-inferred constant locations in memory. Loop backedges (backward branches) are followed and further iterations are unrolled, so that each loop iteration is specialized.

A specialized trace is terminated in one of two ways: (1) if the trace's current program point and propagated constants match the beginning of another trace, they are *linked* together: a direct jump to the other trace is issued; (2) if the instantaneous benefit falls below a threshold, usually because constant propagation becomes too intermittent, control is returned to the unoptimized code.

Lastly, unlike Dynamo, the specializer generates multiple traces at a dispatch point, one for each of its hot values. When a dispatch point is identified, a stub dispatcher is inserted that transfers control over to the trace generator if any of the hot values is detected. When a new trace is generated, the dispatcher is modified to jump directly to it when its hot value is seen again. A future time around, another hot value may be detected and another trace generated.

Asynchronous version. Due to infrastructure constraints, we implemented an asynchronous version of the specialization procedure that differs from the synchronous approach in (1) when it creates the different traces, and (2) how it resolves non-constant

branches.

Given a dispatch point and a set of hot values, the asynchronous specializer interrupts execution and creates traces for *all* of the chosen hot values at the same time. Trace linking occurs at the basic block level, and across hot value traces, reducing specialization time and code size. If a specialized version of basic block $b1$ has a jump to $b2$, and there exists a specialized version of $b2$ with the appropriate initial set of constants, a direct jump from $b1$ to $b2$ is issued, even if $b2$ is not at the start of a trace.

The specializer still eliminates conditional jumps that are fully resolvable. For those that are not, it uses an edge profile to predict the most likely branch target and continues trace construction from there.

Additional benefit ascribed to this trace from further specialization is scaled by the probability that an actual execution will take the predicted branch. For instance, if a branch b has two targets, $t1$ and $t2$, and jumps to $t1$ 60% of the time, specialization will continue at $t1$ (after inserting a fall-through jump to $t2$), but all further benefit will be scaled by .6. Thus the benefit function does not simply sum the number of optimized instructions; instead, the benefit accrued by each optimized instruction is multiplied by the current scale factor before being added. At low scale factors, even successful optimizations will accrue little benefit, since the chance of execution is low. Thus trace termination by the standard benefit criterion can result.

To avoid unrolling predictable yet unspecializable loops (such as those that perform calculations uninfluenced by the dispatch instruction), the specializer monitors the benefit accrued in each loop iteration. It stops unrolling and continues specializing beyond the loop

if the anticipated benefit falls below a specified threshold.

2.6.1 Details

Benefit function. We use a simple cost/benefit heuristic to help determine which dispatch points to select from likely candidates, and when to stop specializing a particular trace. We have not studied this heuristic in detail, and employ it primarily because it is simple and works well; further analysis and refinement is future work. A number of other specializers, such as Calpa [54] and Suganuma et al. [71], have employed more sophisticated heuristics.

Consider a trace t of a hot value v, at a dispatch point i. Its pure benefit, $Pure(t)$, is an estimate of the number of dynamic cycles it will save, using past execution frequencies to guess at the future.

$$Pure(t) = (C + wE) * count(v)$$

C is the number of inexpensive instructions, such as ALU operations, and E is the number of expensive operations, like loads and bounds checks, that have been optimized away, and w is a constant weighting factor to account for the fact that these latter instructions take longer to execute. $count(v)$ is the profiled execution count of hot value h, and is an estimate of the number of times this trace will run in the future. Predictive branching modifies this formula slightly, as explained above.

The net benefit of a trace, $Net(t)$, is its pure benefit minus the cost of recompilation (once a trace is created, it must still be compiled down to machine code), invalidation, and dispatching:

$$Net(t) = Pure(t) - R * length(t) - I(t) - count(p)$$

R is a constant recompilation factor, $I(t)$ is a function of the number of invariant memory locations that must be monitored, and $count(p)$ accounts for the cost of a dispatch: an instruction that must be executed every time the dispatch point is reached. The net benefit of a dispatch point is the sum of the net benefits of its traces.

When creating a specialized trace, we would like to know how well the procedure is currently doing, to help decide when to stop specializing. Given a window of n previous instructions in the optimized trace, we define the *instantaneous benefit*, I, as

$$I = ((C_n + wE_n) * count(v))/n - R$$

where X_n is in the number of X instructions in the last n. The instantaneous benefit is a quick estimate of the current average benefit we are receiving per instruction. In practice, we use $n = 100$.

Dispatcher creation. In our current implementation, a dispatcher consists of a series of if-else blocks, testing in turn for each of the hot values at the dispatch point, and jumping to the corresponding specialized trace if its hot value is found. These blocks are ordered in the dispatcher by the predicted frequency of occurrence of each of their hot values. Since our specializer tends to find only the first handful of hot values (nearly always fewer than 10) worthy of specialization, this simple approach seems to work well.

Algorithm discussion. The specialization algorithm has the benefit of being relatively easy to implement: there are no heavy-duty analyses, and all optimizations are performed in one forward pass. Furthermore, the specialization process, unfettered by a

fixed-size specialized region, can produce traces of different lengths for different hot values, terminating each when it individually stops being beneficial.

The algorithm creates dispatch points that are *polyvariantly specialized*: they have different specialized traces for different hot values. However, for simplicity, it does not support *polyvariant division*, in which a single specialized trace can be created and dispatched with respect to *multiple* values. DyC and other systems have shown supporting such a division to be useful in some cases [34]. We have implemented a simple extension that can specialize on multiple variables $x, y, z...$ as long as all but one are run-time constant. This extension works well in some cases but cannot, for instance, produce one trace for $x = 3$ and $y = 4$, and another for $x = 5$ and $y = 6$, since both of these variables are not run-time constant. Extending our specialization algorithm to more fully support polyvariant division is future work.

2.7 Evaluation

Methodology. The specializer presented in this chapter was implemented in the Jikes RVM 2.3.0.1 Java virtual machine. Measurements were taken on a Pentium M 1.6GHz machine with 1GB RAM running Fedora Core 3 Linux. For the specialization runs, the profiling code was sampled using the full duplication variation the Arnold-Ryder instrumentation sampling framework [8], with a sampling interval of 1000.

The applications that we benchmarked are representative of those chosen in the dynamic complication literature. In some cases we have directly translated programs benchmarked in previous research from C to Java, and in others we have taken real-world Java

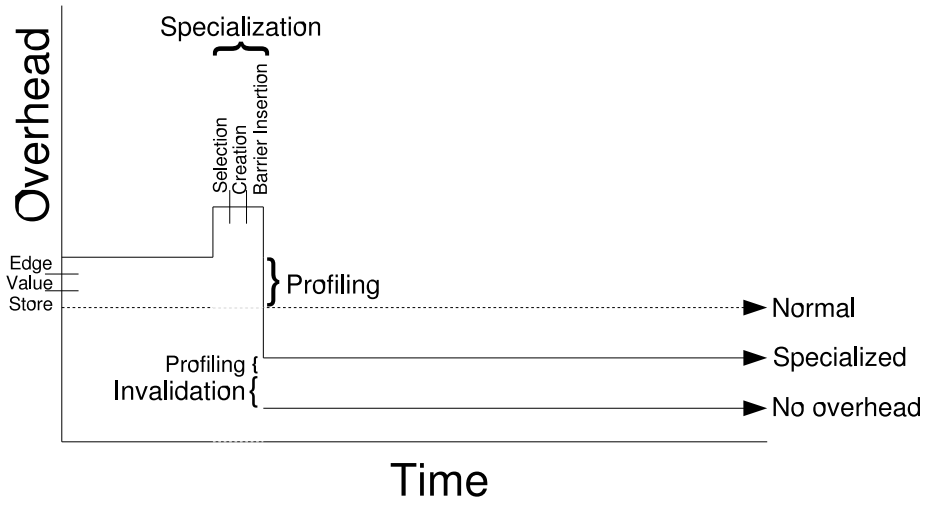


Figure 2.11: A conceptual view of the overheads involved in specialization. There is a steady-state profiling overhead, a one-time specialization overhead, and if specialization is successful, a steady-state invalidation overhead incurred by the write barriers.

programs. For many programs, specialization is not beneficial, so we also wanted to assess whether our specializer is suitable for a transparent dynamic compilation unit that can operate on any program, specializable or not. Thus, we have additionally included a benchmark, `em3d`, that is distinctively not suited for specialization. The benchmarks are described in Figure 2.6.

To ensure optimal unspecialized performance, the numbers in the results measure execution time after full initial compilation, with inlining, of the program code by Jikes's `OPT1` compiler — the highest optimization level that this version of the Jikes adaptive optimization system selects for the Linux/IA32 platform. Thus the unspecialized numbers generally represent the fastest expected performance on an unmodified Jikes RVM.

Description of results. There are a number of overheads involved in our dynamic specializer, and we have attempted to measure all of them. See Figure 2.11 for a conceptual

diagram of when these overheads come into play during a program execution, and Figure 2.12 for the actual numbers.

The specializer employs a number of profilers (described in greater detail below), each of which contributes a steady-state overhead to the execution. In practice, the overhead of all of the profilers running at once is generally less than the sum of their individual overheads, since they share a common profiling infrastructure. These overheads are shown as a percentage steady-state slowdown.

There is a one-time overhead incurred by the actual specialization process. This overhead is comprised of three distinct sections: (1) selecting a region to specialize, using the influence algorithm and then tentatively specializing on a number of candidate dispatch point instructions; (2) if a good region is found, creating specialized traces for the hot values of the selected instruction; (3) inserting write barriers and recompiling code for invalidation. These overheads are shown in seconds.

If specialization successfully completes, the program generally runs faster than it did before, but still incurs the steady-state overhead of executing the invalidation write barriers that were inserted during specialization. This overhead is shown as a percentage steady-state slowdown of the specialized program.

The "No-Overhead Speedup" row in Figure 2.12 displays the pure steady-state speedup achieved by the specialized code over a normal execution, without any of the profiling, creation, or invalidation overheads.

The "Real Speedup" rows display speedup numbers for real specialized executions, and encompass all profiling, creation, and invalidation overheads. These numbers cannot be

derived directly from the no-overhead speedup and overhead numbers, since the impact of the various overheads is dependent on the particulars of an execution: its total length, the time before the specializer was triggered, and so on. Furthermore, some percentages, such as those for invalidation, necessarily represent slowdowns of specialized programs, not the original ones.

		conv-VI	conv-FI	dotprod	i-sort	i-search	jscheme	query	sim8085	em3d
No-overhead Speedup		**215%**	**24%**	**424%**	**551%**	**571%**	**88%**	**76%**	**110%**	**0%**
Profiles	Edge	-1.0%	-1.1%	-7.6%	-0.6%	-0.2%	-2.3%	-9.4%	-0.5%	-3.9%
	Value	-2.6%	-2.6%	-15.0%	-4.5%	-4.3%	-12.7%	-15.8%	-11.2%	-4.9%
	Store	2.8%	2.6%	-2.1%	-2.8%	-2.7%	-1.6%	-6.3%	-3.2%	-1.0%
	All Simultaneously	-0.1%	-0.3%	-18.0%	-4.5%	-4.2%	-13.1%	-19.8%	-13.9%	-5.1%
Specialization	Selection	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	< 0.1s	0.1s
	Creation	0.3s	0.1s	0.1s	0.1s	0.1s	0.2s	< 0.1s	0.1s	–
	Inserting Barriers	1.5s	0.1s	0.1s	0.7s	0.6s	0.3s	0.6s	0.3s	–
	Total	1.8s	0.2s	0.2s	0.8s	0.7s	0.6s	0.7s	0.4s	0.1s
Invalidation		-9%	-3%	1%	-6%	-1%	-3%	-2%	-12%	–
Real Speedup	Short run	**153%**	**19%**	**330%**	**387%**	**401%**	**70%**	**63%**	**66%**	**-4%**
		59s/23s	59s/50s	61s/14s	60s/12s	53s/10s	59s/34s	54s/33s	59s/36s	65s/68s
	Long run	**174%**	**23%**	**417%**	**496%**	**544%**	**82%**	**71%**	**70%**	**-2%**
		601s/219s	598s/487s	603s/117s	603s/101s	527s/82s	580s/319s	544s/317s	593s/349s	525s/534s

Figure 2.12: Dynamic specialization speedups and overheads. The profile percentages measure the steady-state slowdown, before any interval doubling has occurred. The specialization numbers measure in seconds the total time it takes to construct a specialized region. The invalidation percentages measure the steady-state slowdown of the specialized program with invalidation write-barriers in place. The no-overhead and real speedups measure the change in execution speed of each program, respectively without and with all of these overheads. Execution times are rounded to the nearest second.

2.7.1 Profiling

Our specializer requires a number of profiles. They are

- An *edge profile* used to aid the influence algorithm and branch prediction.
- A *hot value profile* that collects the most frequently occurring values at potential dispatch points.
- The *store profile* we presented in Section 2.4.

Since all of these profiles must be gathered at runtime with low overhead, we used the Arnold-Ryder sampling framework [8]. In this framework, a duplicate instrumented version of each function is compiled alongside the original. The uninstrumented version is normally executed. A global counter is kept and decremented at backedges and other yieldpoints, and when it reaches zero, control is transferred to the instrumented version of the currently executing function, and samples are taken until a backedge reverts control back to the original code. The counter is then reset and normal execution resumes. Since instrumented code is run very infrequently, execution overhead is generally low and yet the resulting profiles tend to be statistically accurate.

To reduce execution time, our sampling implementation adaptively doubles the sampling interval (the starting value of the global counter) every time a specialization attempt fails. With this mechanism, programs that are specializable may incur a high profiling overhead (greater than 10%), but only for a short amount of time, before specialization occurs; the profiling overhead for programs that are not specializable quickly drops to an acceptable value.

Hot value profile details. Hot value profiling has been well studied before, as in Burrows et al. [52], so we do not discuss it in detail here. In our implementation, the hot value profile simply monitors the frequency of occurrence of the most popular values resulting from potential dispatch instructions: arguments to loads and functions, and load results. The profiler keeps a short array of value/count pairs for each profiled instruction, and employs the "Top N Value" method described in Calder et al. [17].

Overhead discussion. The edge profile and store profile have low runtime overheads. The small speedup for `convolve` with store profiling appears to be the result of a low-level compilation artifact.

We did not attempt to create an efficient hot value profiler, as such a task has been undertaken before, and instead focused on ease of implementation. For instance, the profiler is invoked via a function call rather than being inlined. Burrows et al. [52] have shown that a hot value profile can be gathered with a runtime overhead of about 10%, and proposed hardware solutions have overheads of under 2% [78]. The slowness of the hot value profiler compared to this 10% figure can be attributed partially to the large number of loads in tight loops in some of the test programs, and also to our suboptimal implementation of the profiling code. The exponential backoff in sampling interval that we employed served to keep the hot value profile overhead low for unspecializable programs.

The memory overhead for the store profile ranged from 2.2% to 7.6% with a mean of 4.8%, while the memory overhead for the hot value profile ranged from 10.8% to 30.3%, with a mean of 24.5%. This latter overhead is high primarily because all functions, whether hot or cold, are hot value-profiled in our implementation. Since only hot functions are ever

considered for specialization, an optimization that just profiles hot functions can drastically reduce the space overhead without comprimising the specializer's effectiveness. A preliminary implementation of this optimization for the hot value profile had space overheads of well under 10%.

2.7.2 Discussion of Results

In this section, we evaluate several hypotheses concerning the specializer's overall performance with respect to the benchmark data.

Does the specialization procedure work? We specialized a number of programs, from an image convolver to a Scheme interpreter executing a 500 line partial evaluator. In every case, the optimal specialization dispatch point, as determined by a manual analysis of the code, was automatically selected.

The resulting speedups are comparable to those of staged specializers like DyC [35]. In several cases, a manual analysis revealed near-optimal code; for instance, in `dotproduct`, the specializer fully unrolled the loop iterating over the (sparse) vector elements and was able to eliminate outright the 75% of the iterations in which the constant vector's element was zero. Half of the loads — the ones from the constant vector — in the other 25% were eliminated as well.

Is it suitable for a runtime environment? In all but one case, specialization time was under 1s. This overhead, along with the profiling overhead discussed in Section 2.7.1, seemed to be quickly outweighed by the much more significant speedups due to specialized code.

To warrant inclusion in a dynamic optimization system's arsenal of optimizations, the specializer should behave well on *all* programs. We ran it on `em3d`, with input parameters that made the program a bad candidate for specialization: we had it create a very large number of data objects that were visited equally often, thus rendering specializing on a small number of them ineffective. The specializer attempted and aborted three specializations, and after each failure it doubled the sampling interval. As a result, the overall slowdown was 4%.[4] This percentage is representative: we ran the specializer on numerous other unspecializable programs from SpecJVM and elsewhere, and none had a slowdown of more than 6%. We feel that this number could be made even lower with a more efficient profiling implementation.

Does it take advantage of opportunities unavailable to staged specializers? The dynamic, optimistic approach taken by our specializer allows it to exploit runtime data and execution behavior to expose optimization opportunties unavailable to an annotation-based staged specializer. We discuss three empirical results that support this claim.

The `convolve` benchmark was run in two different ways; each fixed a different argument to the convolution function. The first way, `conv-VI`, exposed a large optimization opportunity, and while the second, `conv-FI` — varying the images while fixing the matrix — did not, since the convolution matrix is generally small enough to fit into a processor cache, the specializer still created a new specialization, starting from a different dispatch point, that eliminated several computations involving the matrix for a speedup of around

[4]The steady-state profiling overheads for `em3d` were measured at the default sampling interval of 1000, before the sampling interval was ever doubled.

20%. Existing staged specializers, limited to annotating the function in just one way, would be unable to specialize on both of these usage patterns.[5]

Second, the specializer was able to optimize a semi-invariant data structure in the `query` benchmark. `query` applies an array of predicates to each element in a large dataset. We modified the benchmark to periodically update certain predicates in place. The specializer was still able to detect and optimize the constant predicates in the semi-invariant predicate array, something that a staged specializer could not do, since the variable pointing to the current predicate is only constant some of the time, and hence would be hard to annotate.

Third, we analyzed the memory behavior of `interpreter` running bubblesort to track transient constants in the form of constants embedded in the interpreted program. The specializer identified 23% of the dynamic memory loads from bubblesort's "address space" (mostly of the start and end pointers of the array to be sorted, as the algorithm looped over the elements) as constant and optimized them away, which a staged specializer could not do; this represented the removal of 9.6% of all loads in the interpreter's execution.

Is efficient invalidation checking feasible? As discussed in Section 2.5, our write barrier approach to invalidation does not require excessive overhead and is relatively easy to implement. Java's type system helps to reduce the number of barriers to be inserted. The combination of masking and using object headers does a good job of keeping barrier overhead low.

We also presented a number of other invalidation schemes that can be adopted

[5]In fact, if the function were annotated for one type of usage, and then employed at runtime in the other fashion, several useless specializations might result.

based on the properties of the virtual machine and the hardware on which the specializer is running. A hardware-based solution like Mondrian is likely to be the easiest to implement. There is some evidence that GC-based detection will also work well: 97% of all constants in these benchmarks resided in 136 entirely constant objects and arrays, making them ideal candidates for the GC method, since any writes to the read-only pages in which the GC places the objects would signify an invalidation. Thus the invalidation checking overhead for these constants would essentially be zero.

2.8 Related Work

Specialization. Program specialization is a well-studied optimization technique [22, 69, 68, 21, 41, 4, 44, 29, 35, 10, 36, 55, 48]. In its classical form, code is optimized in a source-to-source transformation. Tempo [21], DyC [35], and others used code templates to generate specialized code at runtime once constant values are known. However, they relied on programmer annotations to specify specialized regions and constant memory locations. Calpa [54] automated this process by profiling a representative input and inferring annotations. This profiling step required its own run and employed a fairly expensive annotation analysis. In some ways these staged specializers are more powerful than the one presented in this chapter in terms of pure specializing ability, for instance in supporting techniques like polyvariant division and precisely controlled loop unrolling. In others, such as in exploiting concrete heap state or per-execution runtime behavior, they are less powerful. The specializer in this chapter has the additional benefit of being fully transparent and immediately beneficial to end users. Suganuma et al. [71] implemented a form of automatic dynamic

specialization that does not use any heap constants; as a result, the system in that paper achieved speedups of 3%-6%. Zhang et al. [76] have built a value specializer with speedups of 20% on top of their dynamic optimization framework, Trident.

More recently, Gal et al. [32] have implemented a specializer called TraceMonkey in the Mozilla Firefox JavaScript interpreter. Rather than specializing on the values of local variables or heap objects, it specializes on the type of objects, using that information to avoid the object boxing and unboxing that is common among interpreters of untyped languages. Currently, our specializer cannot automatically unbox objects, since the boxing operation involves a write to the heap, which our specializer must respect. However, a minor modification to support unboxing might work as follows: during trace creation, all local variables are treated as unboxed native values. Function call arguments are boxed, and at trace exit, all live variables are boxed.

Another difference between the two specializers is that our specializer agressively unrolls loops. This approach is not compatible with a type specializer, since loop conditional values are not known. Thus, TraceMonkey treats loop bodies as the building blocks of trace creation, and stitches them together to form traces.

Dynamic optimization. The profile-and-optimize dynamic approach described in this chapter is similar to other transparent dynamic optimization systems, like Mojo [15], Hotspot [53], and others. In particular, our specializer leverages the Jikes RVM framework [7, 30] for recompiling specialized methods.

Profiling. The efficiency of our profilers rests on the Arnold-Ryder sampling framework [8]; we use it to employ a novel invariance detection profile. We use a method

suggested by Calder et al. [17] for gathering hot value data. The use of optimistic assumptions to motivate dynamic optimization was presented by Arnold and Ryder [9].

Trace creation. The main specialization algorithm's trace creation procedure draws from on-line partial evaluation techniques [64], and was inspired by Dynamo [11], although Dynamo does not use heap invariants or unroll loops when optimizing, and only produces one optimized trace per program point. Sullivan et al. [72] have tailored the Dynamo framework to optimize interpreters, although their system requires the insertion of static annotations. The influence algorithm we designed to find dispatch points approximates forward dynamic slices, which were introduced by Korel and Laski [45].

Invalidation. As far as we know, this chapter presents the first implementation and evaluation of a working automatic detection and invalidation system. Calpa [54] proposed an automated detection system by which an offline points-to analysis is used to determine where to insert invalidation checks, but we were unable to find an evaluation of this technique's overheads. DyC [35] supports manually-triggered invalidations, but does not provide a mechanism for actually performing the invalidation on a running specialized trace. Our use of the Java type system to limit the number of write barriers inserted for invalidation detection is related to the semantics-based guards used by Pu et al. to specialize operating system calls [58]. We use the bitmasking techniques employed by Diduce [38] to reduce write barrier overhead. The actual invalidation is similar in end result to on-stack replacement (OSR) techniques [18, 31]. However, OSR occurs asynchronously; the compiler compiles a version of the function custom-built for re-entry while the original version is still executing. Our invalidation mechanism must act immediately, and so we construct the

initial specialization so that invalidation can occur as soon as an offending write has been detected, and without recompilation. Another invalidation technique we suggested is based on Mondrian memory protection [75].

To the best of our knowledge, this is the first implementation of a transparent runtime specializer that uses heap data. While Sastry [67] proposed a runtime specialization system, that work relied on offline compilation techniques to emulate a runtime specializer, and had no support for invalidation. In addition, the techniques proposed in this chapter for detecting specialization points, generating traces, and linking them are simpler and lead to better specializations on the same benchmarks.

2.9 Conclusion

This chapter described the design of a transparent dynamic specializer. To the best of our knowledge, this is the first such heap-based system that is dynamic and does not rely on programmer annotations, separate profiling runs, or offline preprocessing. We presented several techniques that enable this implementation: (1) store-profile based optimistic, accurate, and fine-grained detection of heap invariance, (2) the *influence*-based dispatch identification method, (3) constant-propagation based generation of specialized traces, and (4) an efficient write barrier-based invalidation scheme.

The store profile enables detection of heap constants that existing systems cannot. Our evaluation showed that this profile can be collected at low overheads and with high accuracy. The influence metric is able to find the best dispatch points with high reliability. The invalidation mechanism operates with low overhead. The current implementation of

the specializer in Jikes RVM has low overhead in practice, accurately selects beneficial specialization points, and produces speedups of 1.2x to 6.4x on a variety of benchmarks.

Chapter 3

Automatic Incrementalization

3.1 Introduction

Type safety of modern imperative languages such as Java and C# eliminates many types of programming errors, such as buffer overflows and doubly-freed memory. As a result, algorithmic errors present a proportionately greater challenge during the development cycle. One such class of errors are data structure bugs. Many data structures bugs can be detected as violations of high-level invariants such as "the elements of this list are ordered", "no elements in this priority queue can be in that priority queue", or "in a red-black tree, the number of black nodes on any path from the root node to a leaf is the same." Verifying such invariants, however, remains non-trivial. Data structure invariants are particularly difficult for static tools to verify because static heap analysis scales poorly and current verifiers require extensive annotations.

An alternative approach is dynamic verification of invariant checks. Dynamic

checks operate on the concrete data structure and are thus typically simple to write and validate. Thanks to tools such as `jmlc` [19], dynamic checking has become more accessible to programmers. However, dynamic checks can incur a significant run time overhead, hindering the development and testing. Since checks are executed frequently and commonly traverse the entire data structure, a program with checks may run 10–100 times slower, which may be prohibitively slow for all but the most patient programmer. Consequently, dynamic checks are rarely employed, even in debugging.

This chapter introduces DITTO, an incrementalizer for a class of dynamic-data structure invariant checks written in modern imperative languages like Java and C#. We allow the programmer to write these checks in the language itself. DITTO then automatically incrementalizes such checks, rewriting them so that they only re-check the parts of a data structure that have been modified since the last check. Incremental checks typically run linearly faster than the original (about 10-times faster on data structures with 10,000 elements). We believe that the incrementalization makes dynamic checks practical in a development environment.

The goal of incrementalization is to modify an algorithm so that it computes anew only on changed input data and reuses all repeated subcomputations. Traditionally, incrementalization is designed and implemented by hand: an algorithm is modified to be aware of data modifications and to cache and reuse its previous intermediate results [60]. While hand-incrementalization can produce the desired speedups of invariant checks, manual incrementalization has several practical limitations:

- The programmer may overlook possible modifications to the data structure (as in the

infamous Java 1.1 `getSigners` bug [33]) and thus omit necessary incremental updates. The result is an incorrect invariant check that may fail to detect bugs.

- Some invariant checks may be difficult to incrementalize by hand. For example, after some effort, we gave up on incrementalizing red-black tree invariants.

- Manual incrementalization does not appear economical, as each data structure may require several checks. Programmers may also want to obtain an efficient check rapidly, for example, when writing "data-breakpoint" checks for explaining the symptoms of a particular bug.

- Perhaps most importantly, incremental code is complex and scattered throughout the program. The complexity of its maintenance may defeat the purpose of relying on invariant checks that are simple and verifiable by inspection.

Recent research by Acar et al. [1] developed a powerful general-purpose framework for incrementalization of functional programs, based on memoization and change propagation. This framework provides an efficient incrementalization mechanism while offering the programmer considerable flexibility. To incrementalize a program, the programmer (i) identifies locations whose changes should trigger recomputation; and (ii) writes functions that carry out the incremental update on these locations. The actual memoization and recomputation are encapsulated in a library. Acar's incrementalized algorithms exhibit significant speedup, so it is natural to ask how one could automate this style of incrementalization.

In this chapter, we identify an interesting domain of computations for which we develop an automatic incrementalizer. Our domain includes recursive side-effect-free func-

tions, which cover many invariants of common data structures such as red-black trees, ordered lists, and hash tables. While we support only functional checks, the checks can be executed from within arbitrary programs written in imperative languages such as Java and C#. In these languages, checks are useful to the programmer because manual verification of invariants is complicated by the fact that data structure updates can occur anywhere in the program. For the same reason, incrementalization is difficult, which should make automatic incrementalization attractive.

Properties of invariant checks allow us not only to automate incrementalization but also to offer a simple and effective implementation.

- Simplicity. An invariant check typically returns always the same result (i.e., "the check passed") and so do its subcomputations that are recursively invoked on parts of the data structure. This observation allows us to develop *optimistic memoization*, a technique that aggressively enables local recomputations to reconstruct a global result.

- Effectiveness. The local properties that establish the global property of interest are typically mutually independent and recomputation of one does not necessitate recomputation of others. For example, sortedness of a list is established from checking that adjacent elements are ordered; if an element is inserted into the list, we need to check its order only with respect to its neighbors. Independence of local computations means that incremental computation can produce significant speedups.

The main contributions of this chapter are:

1. The DITTO automatic incrementalizer for a class of data structure invariant checks

that are written in an object-oriented language.

2. A portable implementation of DITTO in Java.

3. An evaluation of Java DITTO on several benchmarks.

Section 3.2 outlines how a simple invariant check is incrementalized. Section 3.3 describes DITTO's incrementalization algorithms and Section 3.4 provides some implementation details. Section 3.5 evaluates DITTO on several small and large benchmarks. Section 3.6 discusses related work and Section **??** concludes.

3.2 Definitions and Example

In this section, we give a high-level overview of DITTO's incrementalization process. First, we define the class of invariant checks that DITTO can incrementalize.

Definition 1 *The **inputs** to a function consist of its **explicit arguments**, i.e., the values of its actual parameters; its **implicit arguments**, i.e., values accessed on the heap; and its **callee return values**, i.e., the results of function calls it makes.*

Note that implicit arguments are defined not to include locations that are read (only) by the callees of the function.

Definition 2 *A **data structure invariant check** is a set of (potentially recursive) functions that are side-effect-free in the sense that they do not write to the heap, make system calls, or escape address of an object allocated in the invariant check. Furthermore, in each function, no loop conditional or function call can depend on any callee return values.*[1]

[1]This technical restriction, described further in Section 3.3.5, is required to ensure that the original functions and their incrementalized versions have the same termination properties in the presence of optimistic memoization. This restriction can be sidestepped but we have not found it to be an impediment in practice.

```
class OrderedIntList {
  IntListElem head;

  void insert(int n) {
    invariants();
    ...
    invariants();
  }

  void delete(int n) {
    invariants();
    ...
    invariants();
  }

  void invariants() {
    if (! isOrdered(head)) complain();
  }

  Boolean isOrdered(IntListElem e) {
    if (e == null || e.next == null)
      return true;
    if (e.value > e.next.value)
      return false;
    return isOrdered(e.next);
  }
}
```

Figure 3.1: The example class `OrderedIntList` and its invariant check `isOrdered`.

Throughout this chapter, we will often assume that a check is a single recursive function. However, DITTO supports also checks composed of multiple recursive functions, such as the one in Figure 3.9. When the check contains multiple functions, we identify the check by the "entry-point" function that is invoked by the main program.

The incrementalizer memoizes the computation at the level of function invocations, so recursive checks are more efficient than iterative ones. Most iterative invariant checks can be rewritten without loss of clarity into recursive checks.

The main program has no restrictions on its behavior. We assume that invariant checks running in multithreaded programs either operate on thread-local data or are atomic to ensure data integrity during the check.

To illustrate how DITTO works, we walk through the incrementalization of a simple invariant check, `isOrdered`, shown in Figure 3.1. The invariant verifies that the list maintains its elements in sorted order. The invariant is checked at method entries and exits. The former ensures that the invariant is maintained by modifications performed from outside the class. Such modifications could occur if, say, an `IntListElem` object was mistakenly exposed to users of the class. The latter ensures that the list operation itself maintains the invariant.

The invariant check is simple and readable, but it is inefficient. In common usage scenarios, the unoptimized `isOrdered` will dominate the performance of the program. However, the check is amenable to incrementalization under most common modifications to the list. For instance, if an element e is inserted into the middle of the list, `isOrdered` needs to be re-executed only on e and its predecessor; the success of the invocation of `isOrdered`

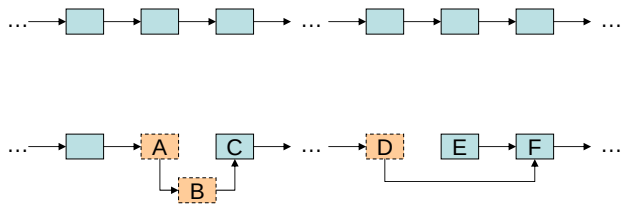


Figure 3.2: Before and after a list operation: an element is inserted, and another is deleted. The elements modified during the operation are dashed.

preceding the change guarantees the checked property for the remaining elements in the list, as they have not changed since then. This incrementalization reduces the cost of the check from the time linear in the size of the list to constant time.

Incrementalizing `isOrdered`. DITTO automatically incrementalizes `isOrdered` using the following simple process.

1. During the first execution of `invariants`, we record the sequence of recursive calls to `isOrdered`, their inputs, and their results.

2. During the subsequent execution of the main program, we track changes to memory locations that served as implicit inputs to a check. The tracking is performed with write barriers.

3. The next time `invariants` is invoked, we re-execute only the recursive invocations to `isOrdered` whose inputs have changed; we reuse memoized results for the remaining invocations. We update the memoized results so that further executions of the check can be incrementalized.

We describe these steps in detail in the context of a modification scenario, shown in Figure 3.2, where an element is inserted into the list and another element, further down

the list, is deleted. Note that we assume that the invariant check is performed only after both modifications, and not in between the two modifications.

DITTO stores all inputs for each (recursive) invocation of `isOrdered`. The function `isOrdered` has five inputs: one explicit argument, the formal parameter `e`; three implicit arguments, the fields `e.next`, `e.value`, and `e.next.value`; and one callee return value, that of the recursive call to `isOrdered`.

The modification shown in Figure 3.2 updates two fields already in the list: `A.next` and `D.next`. Based on the inputs stored in the previous execution of the check, DITTO determines that these fields served as implicit inputs to invocations `isOrdered(A)` and `isOrdered(D)`. These two invocations must be re-executed on the new input values. Since `isOrdered(A)` occurred first in the previous execution, it is re-executed first.

The re-running of `isOrdered(A)` uses the new implicit arguments, specifically the new value of `A.next`, which now points to `B`. The execution thus continues to the invocation `isOrdered(B)`. Since DITTO has not yet encountered `isOrdered` with the explicit argument `B`, it adds this new invocation to its memoization table and continues executing, reaching the recursive call to `isOrdered(C)`.

At this point, DITTO determines that (i) `isOrdered(C)` has been memoized and (ii) the implicit arguments to `isOrdered(C)` have not changed since the previous execution of the check. However, this is not sufficient to safely reuse `isOrdered(C)` because there is no guarantee that the last input to `isOrdered(C)`—the callee return value from `isOrdered(C.next)`—will return the same value as in the previous execution of the check. The danger is quite real: There is a modification to an implicit input of `isOrdered(D)` fur-

ther down the list; if `isOrdered(D)` returned a different value, this value could ultimately affect the value returned by `isOrdered(C.next)`. So, a straightforward memoization algorithm cannot safely reuse `isOrdered(C)`; instead, it must continue the re-execution until it is sure that no further callee return values might change. In our example, it would have to re-execute past `C` all the way to `D`, re-executing also all the intervening function invocations.

DITTO follows a more sophisticated algorithm. To deal with the uncertainty of callee return values, DITTO optimistically assumes that the recursive call to `isOrdered(C)` will return the same value as it did previously. This is a sensible assumption since recursive invariant check often do return the same value. (Typically, this is the "success" value.) This *optimistic memoization* strategy allows DITTO to reuse the cached result for `isOrdered(C)`, which successfully terminates the re-execution. The execution eventually returns back up to `isOrdered(A)`, which returns `true` — the same value it returned last time. Thus, the function that invoked `isOrdered(A)` need not be re-executed since all of its inputs are the same as last time; we are now done with the re-execution of the modified portion of the data structure around nodes `A` and `B`.

DITTO then re-executes `isOrdered(D)`, the second call whose implicit inputs have changed. The incrementalization then continues to `isOrdered(F)`, which is successfully reused, terminating the recursive calls. The invocation of `isOrdered(D)` evaluates to `true`, matching its previous result, so the entire recomputation ends. The invocation `isOrdered(E)` is no longer reachable in the computation and is ignored.

DITTO now returns the cached result of the entire invariant check, `true`, to the caller, `invariants()`.

Consider now the case when `isOrdered(D)` returns a value different than it did previously. (Note that our optimistic assumption is not necessarily wrong yet, as we assumed only that `isOrdered(C)`, but not `isOrdered(D)`, returns the same value as it did previously.) The new return value would be propagated from `isOrdered(D)` back up to its caller, which would be re-executed. This process would continue until either (a) a caller is reached that returns the same result that it did previously; or (b) the execution reaches the first caller, `isOrdered(head)`, and the new overall result is cached and returned. Note that if this upward propagation reaches `isOrdered(B)`, the optimistic memoization decision made when reusing `isOrdered(C)` is shown to be incorrect. In this case, `isOrdered(B)` is re-executed like the other calls during this propagation phase.

Whether the optimistic assumptions turned out wrong or not, the incrementalizer stores the new inputs and the result for each re-executed call; the memoization data for `isOrdered(E)` is garbage collected. This maintenance ensures that DITTO will be able to incrementalize the invariant check during its next execution.

Of course, data structure modifications can take on more complex forms than simple inserts and deletes. The next section describes how all possible modifications are handled in a general way, and Section 3.5 examines the performance of DITTO on invariants of considerably greater complexity.

3.3 Incrementalization Algorithm

This section presents details of our incrementalization algorithm. We start by describing the memoization cache and continue with a straightforward incrementalization

algorithm. The inefficiency of this algorithm will motivate our optimistic incrementalizer, presented next. We will conclude by explaining the steps taken when optimistic assumptions fail.

3.3.1 Computation graph

On the first invocation of an invariant check, DITTO caches the computation of the check in a *computation graph*, which records the computation at the granularity of function invocations. Between invocations of the invariant check, the graph is used to track how the main program changes the check's implicit arguments. On the subsequent invocation of the invariant check, the graph is used to identify memoized function invocations whose inputs have been changed. These function invocations are re-executed and the graph is updated; the remaining function invocations are reused from the graph. The incrementally updated graph is equivalent to re-running the invariant check from scratch on the current program state.

The computation graph contains a node for each (dynamic) function invocation performed during the execution of the check. Directed edges connect a caller with its callees. DITTO stores the graph in memory in the form of a table. A table entry, shown below, represents one node of the graph, i.e., one function invocation. We will use the terms *function invocation* and *computation node* (or node) interchangeably as appropriate.

f	explicit_args	implicit_args	calls	return_val	dirty

The entry contains six fields: f is the invoked function; *explicit_ args* is a list of values passed as actual arguments to f; *implicit_ args* is a list static and heap locations read

by the invocation; *calls* is a list of function invocations made by this function invocation, represented as links to other entries in the table; *return_val* is the return value of this invocation; and *dirty* is used during the incremental computation to mark invocations whose implicit inputs have been modified. Recall that *implicit_args* includes only the locations read by this invocation, not by its callees. The table is indexed by a pair (*f*, *explicit_args*).

DITTO constructs the computation graph by instrumenting the invariant check. The offline instrumentation diverts all invocations of an invariant check c — i.e., all calls to a functions in c from a function not in c — to the catch-all `incrementalize` runtime library function, described in detail later in this section (see Figure 3.7). For instance, the call to `isOrdered(head)` in `invariants()` in Figure 3.1 is rewritten to invoke `incrementalize()` instead.

DITTO instruments each function f in the invariant check c to record the data necessary to construct a memoization table entry. The instrumented version of the `isOrdered` function in Figure 3.1 is shown in Figure 3.3. The transformation inserts code at the beginning of f to check if this invocation has been memoized. If a table entry with the same explicit arguments already exists, the function returns with the cached result value; if not, a new entry is created and implicit arguments and the return value are recorded. The `try` and `catch` are required by optimistic memoization; their purpose is described later in this section.

In addition to recording the implicit arguments used by each function invocation, a reverse map, from heap locations (implicit arguments) to table entries, is created. This reverse map is used to determine which function invocations depend on modified heap values.

```
Boolean isOrdered(IntListElem e) {
  try {
    // creates a new entry if one doesn't exist
    MemoEntry n = getMemoEntry(isOrderedId, [e]);
    if (n.hasResult) return (Boolean) n.result;

    n.addImplicit(addressOf(e.next));
    if (e == null || e.next == null) {
      n.setResult(true);
      return true;
    }
    n.addImplicit(addressOf(e.value));
    n.addImplicit(addressOf(e.next.value));
    if (e.value > e.next.value) {
      n.setResult(false);
      return false;
    }
    n.addCall(isOrderedId, e.next);
    n.setResult(isOrdered(e.next));
    return n.result;
  } catch (Exception e) {
    throw new OptimisticMemoizationException();
  }
}
```

Figure 3.3: The instrumented version of `isOrdered()`.

See Figure 3.4 for an example initial computation graph.

Instrumentation is also used to track updates to implicit inputs. These updates can occur anywhere in the main program, so DITTO places write barriers into statements that might write those locations. The write barriers are described in further detail in Section 3.4. When an update to an implicit location is detected, function invocations whose implicit arguments have been modified are marked as *dirty*, which prevents reuse of their memoized results (see Figure 3.5).

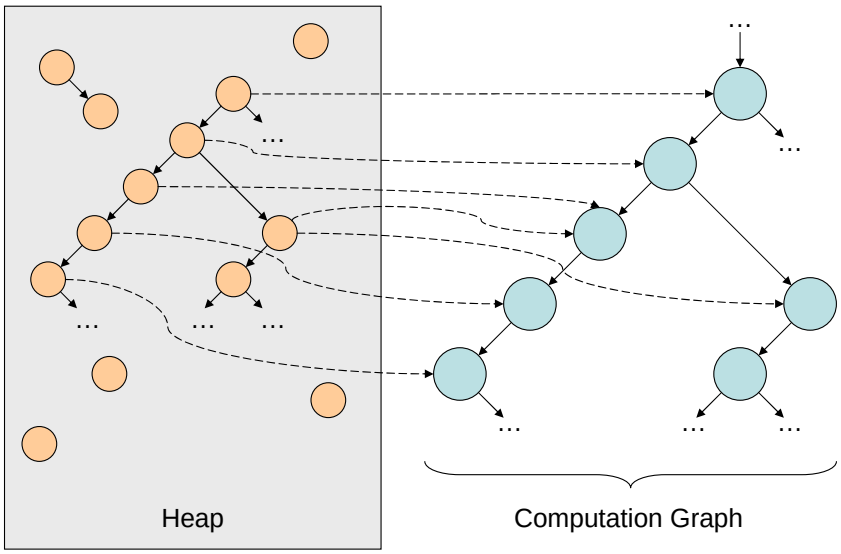


Figure 3.4: Part of the computation graph after an initial run. The dotted lines from items on the heap to computation nodes (function invocations) indicate the implicit arguments used by those nodes. Not all dotted lines are shown.

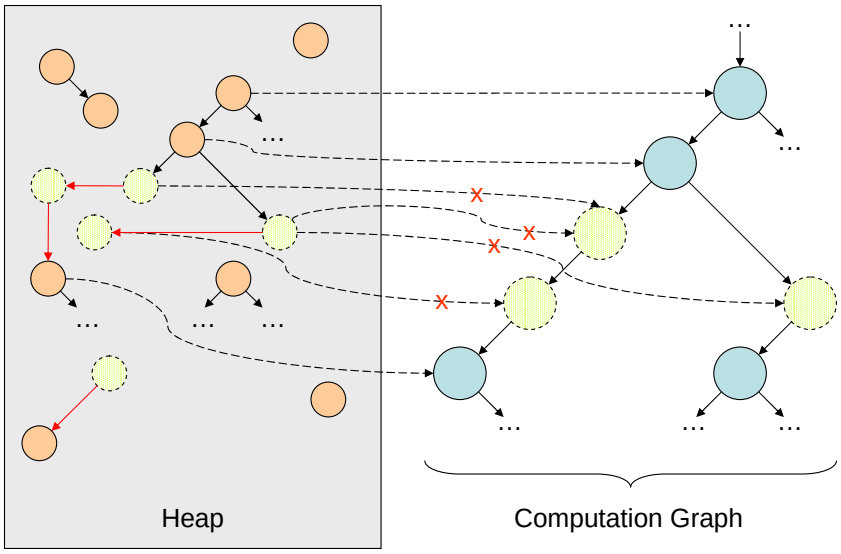


Figure 3.5: Memory locations with dashed outlines have been modified since the last execution of the invariant. All computation nodes that used these memory locations are marked as dirty.

3.3.2 Naive incrementalizer

DITTO can reuse the cached result of a function invocation if the function is invoked with identical inputs as the cached invocation. However, checking whether all inputs are identical is non-trivial. Recall that, for the purpose of memoization, a function has three kinds of inputs: explicit inputs (i.e., actual arguments); implicit inputs (i.e., values read by the function from the heap and static variables); and return values from its callees. Ideally, we want to reuse the cached result at the time when the function is invoked; but at this point we know only that the explicit arguments are identical. We may also know that the values of implicit input locations have not changed since the last invocation, but is the function going to read the same set of locations? The answer depends on the return values from the function's callees; if they differ from the previous return values, the function may read different locations and clearly cannot be reused.

A conservative rule for reuse of memoized results is to ensure that (i) the explicit arguments are identical and that there has been no change to (ii) implicit input values as well as to (iii) callee return values. (To confirm that the return values are identical, the naive incrementalizer will incrementally execute the calls, meaning that it will try to reuse as much of the calls as possible.)

Lemma 1 *Consider an invocation of function f that (i) has explicit arguments e, (ii) accesses the set of heap locations I, and (iii) invokes functions $g_1(a_1)$, ..., $g_n(a_n)$, which return values r_1, ..., r_n. The cached result for this invocation is identical to the value of $f(x)$ invoked in the current program state if the following conditions hold: (1) $x = e$; (2) the locations in I have not been modified since the memoized invocation was executed; (3) $g_1(a_1)$,*

..., $g_n(a_n)$, if invoked on the current program state, would return the same values r_1, ..., r_n as at the time of the previous invocation of $f(e)$.

The proof involves showing that the current function invocation (1) accesses the same set of locations as the cached invocation; and (2) makes identical function invocations as the cached invocation. It is easy to show that if the previous implicit locations have not been changed, the first call made by the function is identical to the first call made by the cached invocation. If this call returns the same value as previously, the function will continue accessing the same implicit input locations. Since their values have not been changed, the second call made by the function will be identical to the second call in the cached invocation. The proof then proceeds by induction.

The naive incrementalization algorithm is shown in Figure 3.6. In this code, `initial_args` are the arguments provided to the first, entry-point function call of the invariant check. `t[f,x]` represents a lookup in the memoization table of an entry with function f and explicit arguments x.

The naive incrementalizer is simple: starting from the first function invocation of the invariant check, it recursively follows the path of the computation, reusing memoized results where appropriate. However, it is very costly: in order to ascertain that child calls do in fact return the same values as in the previous execution, it requires a memoization table lookup for every function invocation in the computation, even those that are unaffected by any input modifications.

```
function incrementalize(f, initial_args)
  return memo(f, initial_args)

function memo(f, x)
  if (t[f,x] == null ||  // never been run before
      t[f,x].hasModifiedImplicitArgs())
    return exec(f, x)
  foreach (c in t[f,x].calls)
    // did the call return the same value as last time?
    old_return_val = c.return_val
    if (memo(c.f, c.explicit_args) != old_return_val)
      // memo lookup failed somewhere in c.f's call tree
      return exec(f, x)
  // conditions described in Lemma 1 hold; reuse allowed
  return t[f,x].return_val

function exec(f, x)
  // invoke f', the instrumented version of f
  return f'(x)
```

Figure 3.6: The naive incrementalizer.

3.3.3 Optimistic incrementalizer

Ideally, the incrementalizer should recompute only function invocations whose inputs have changed. But how do we determine that calls made by f would return the same values if executed in the current program state? The naive incrementalizer does so by "replaying" the sequence of all calls indirectly made by f and ensuring that all these transitive callees of f can be memoized. This process is expensive. A constant-time memoization check can be performed by time-stamping the invocation of each function and checking if any transitive callee of f had its implicit arguments modified. Such a time interval mechanism was used by by Acar et al. [1] to aid in identifying relevant functions with changed inputs.

DITTO develops what we think is a simpler mechanism based on the common property that invariant checks usually succeed. When a check succeeds, it returns a success code; the same is true for all recursive function invocations made by the check. Our optimistic assumption thus is that *a function invocation in an invariant check typically returns the same value*. This observation holds even when some of the transitive callees had their implicit inputs changed because invariants usually hold even after the data structure is modified.

The *optimistic memoization* employed by DITTO simplifies the naive incrementalizer: we optimistically reuse a cached invocation if the explicit and implicit arguments are the same; the callee return values are assumed to return the same values. The optimistic `memo()` function is shown here:

```
function optimistic_memo(f, x)
  if (t[f,x] == null ||  // never been run before
      t[f,x].hasModifiedImplicitArgs())
    return exec(f, x)
```

```
// optimistically assume that conditions
// described in Lemma 1 hold and allow reuse
return t[f,x].return_val
```

Optimistic memoization breaks dependencies of an invocation on its callees, whose return values are not yet known at the time when reuse of the invocation is attempted. This frees DITTO from having to perform the memoization lookup on many function invocations whose inputs have not changed. In other words, the benefit of optimistic memoization is that, in the common case of a successful check, we recompute only the local properties of those data structure nodes that have changed.

DITTO must of course handle the case of an incorrectly predicted optimistic value. The steps for doing so are detailed in Section 3.3.5.

3.3.4 The complete algorithm

The complete incrementalization algorithm, shown in Figure 3.7, needs to take care of two more issues: pruning of unreachable computations and recomputation in response to changed return values.

Pruning. If a computation of a check has two function invocations with modified inputs, $f(x)$ and $g(y)$, and $g(y)$ is a transitive callee of $f(x)$ then we should recompute $f(x)$ before $g(y)$. The reason is that the new computation of $f(x)$ may or may not lead to an invocation of $g(y)$. Invoking $g(y)$ could result in an exception or it could be costly (for example, node y could have moved to a different data structure on which the evaluation of the invariant check could be expensive). Thus, DITTO re-executes dirty nodes (i.e., nodes with modified implicit inputs) in a breadth-first search order (i.e., nodes closest to the root

are executed first). After each node is re-executed, the incrementalizer prunes nodes that are no longer in the computation graph; these nodes will not be re-executed.

Changed return values. When a re-execution of an invocation evaluates to a return value that differs from the cached return value, the changed value must be propagated to the caller of the recomputed invocation. DITTO tracks all nodes with differing return values and re-executes their callers in reverse breadth-first-search order, which ensures that a node is re-executed only after all its children have been re-executed (if that was necessary). This re-execution along a path continues up the graph until either (i) the return value of a re-executed ancestor evaluates to the cached value; or (ii) the root node is reached, which changes the overall result of the invariant check.

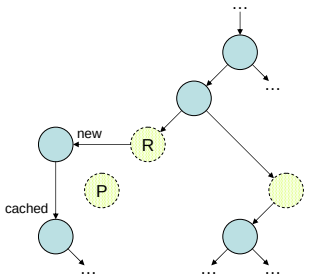
The DITTO incrementalizer is shown in Figure 3.7. In the implementation, the graph is not traversed using BFS; instead, the nodes are kept ordered using the order maintenance algorithm due to Bender, et al. [14].

An example of the algorithm in action (with pruning, optimistic memoization, and return value propagation) is shown in Figure 3.8.

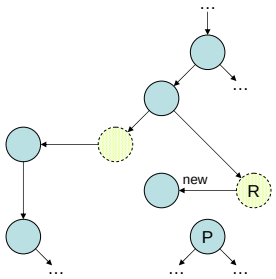
3.3.5 Optimistic mispredictions

Recall that when the optimistic incrementalizer encounters a call to (a non-dirty) invocation $g(y)$, the incrementalizer reuses its old return value without first ensuring that the $g(y)$ would return the same value in the current program state. When this optimistic assumption is wrong, the re-execution of $f(x)$, the caller of $g(y)$, may go wrong in one of three ways:

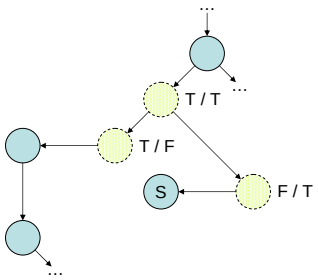
```
function incrementalize(f, initial_args)
  to_propagate = {}
  // identify memoized executions that have modified
  // implicit arguments (detected by write barriers)
  changed = get_changed_implicit_locations()
  changed_fns = map_locs_to_memo_table_entries(changed)
  // need to re-run root if arguments have changed
  if (t[f,initial_args] == null)
    changed_fns.add((f, initial_args))
  changed_fns.sort_bfs_order()
  foreach((f,x) in changed_fns)
    t[f,x].dirty = true
  foreach ((f,x) in changed_fns)
    // only re-execute if still in graph (not pruned)
    // and dirty (hasn't already been re-executed)
    if (t[f,x] != null && t[f,x].dirty)
      exec(f, x)
  propagate_return_vals()
  return t[f,initial_args].return_val

function memo(f, x)
  if (t[f,x] == null ||  // never been run before
      t[f,x].dirty)      // changed implicit_args
    return exec(f, x)
  // thanks to optimistic memoization, don't
  // need to check callee return values
  return t[f,x].return_val

function get_callers(f, x)
  // returns nodes that call f(x)

function exec(f, x)
  oldentry = t[f,x]
  // f' is the instrumented version of f
  newresult = f'(x)
  if (newresult != oldentry.return_val)
    to_propagate.add((f,x))
  foreach (c in oldentry.calls)
    if (get_callers(c.f, c.explicit_args).size() == 0)
      prune(c.f, c.explicit_args)
  return newresult

function propagate_return_vals()
  to_propagate.sort_reverse_bfs_order()
  while (to_propagate.size() > 0)
    e = to_propagate.remove(0)
    f, x, oldval = e.f, e.explicit_args. e.return_val
    newval = f'(x)
    if (oldval != newval)
      to_propagate.insert_reverse_bfs_order(
         get_callers(f,x))

function prune(f, x)
  var calls = t[f,x].calls
  t[f,x] = null
  foreach(c in calls)
    if (get_callers(c.f, c.explicit_args).size() == 0)
      prune(c.f, c.explicit_args)
```



(a) Re-execution and pruning



(b) Re-execution and pruning



(c) Changed return values

Figure 3.8: Re-execution after modification to the data structure shown in Figure 3.5. **(a)** The first dirty node, R, is re-executed. The re-execution in the node execution encounters (i) a new node, which is added to the graph, and (ii) a non-dirty node with a valid memoized value, which stops recursion early thanks to optimistic memoization. The dirty node P is pruned from the graph and will not be re-executed. **(b)** The second dirty node is re-executed. A new node is added and the non-dirty node marked 'P' and its children are pruned. Though not shown in the figure, memoization table entries are added or modified for the functions invoked in this step. The resulting computation graph reflects the changes made to the heap in Figure 3.5. **(c)** The results of re-executed nodes are compared with their old cached values. If they differ, the new results are propagated up through the graph. In this example, let the invariant check be a test for the presence of a special object S. Assume that S has moved from the left branch of the tree to the right; as a result, some node results differ ("F/T" indicates an old result of *false* and a new result of *true*), and are propagated up the graph. However, the propagation stops soon because an ancestor node's new result matches its old one.

The invocation $f(x)$ finishes evaluation but yields an incorrect result. In this scenario, no remedial action is needed. The correct return value will reach $f(x)$ during the propagation described in Section 3.3.4 and $f(x)$ will thus be re-executed with the correct return value and will produce the correct result.

The incorrect return value causes $f(x)$ to throw an exception. For example, $g(y)$ may return an object that is no longer in the data structure and may thus have invalid field values. When $f(x)$ reads those values, it may throw a divide-by-zero error or a null-pointer deference. Since this exception would not be raised in the non-incremental check, it must be prevented from reaching the main program. The code transformation described in Section 3.3.1 encloses the entire function in a try-catch block. If an exception is thrown due to a wrong optimistic assumption, the exception is caught and the execution of the function is stopped. The function will eventually be re-executed with correct inputs, as in the previous scenario. If an exception still occurs at this stage, the exception is forwarded on to the main program.

The incorrect value causes non-termination. Similar to the previous case, an incorrect return value may cause a loop or a recursion to iterate forever. The return value did not cause non-termination when $f(x)$ was executed previously because the explicit or implicit inputs were different. We offer two alternative remedial actions.

The first one, currently used by DITTO, imposes a restriction on the way DITTO-incrementalizable invariant checks must be written: *No loop conditional or function call can depend on a callee return value.* Here, *depends* includes both control- and data-dependence. Under this restriction, each loop and each call in the re-executed $f(x)$ uses only (correct)

values from the current state, and thus will not cause a spurious non-termination.

Our practical experience is that this restriction is more of a technicality than a real burden. We have yet to write a loop of any sort inside an invariant check function, and we have found it easy to overcome the function call restriction by avoiding short-circuit boolean evaluation.

To ensure that programmers are unable to violate this restriction, we have written a simple static analysis that checks for such a violation. The analysis is fairly trivial because aliasing is impossible in a side-effect-free function.

The second solution for this situation is to implement a timeout that would trigger when an optimistic execution takes far longer than it has taken historically. In this case, the invariant check would be re-executed from scratch. A benefit of this approach is that no programming restrictions are made on the function, though a cost is that its behavior may be unpredictable.

3.4 Implementation

DITTO is implemented as a Java bytecode transformation and accompanying runtime libraries. This approach does not allow for an optimized runtime implementation. For instance, the write barriers are implemented in Java, which requires two null checks and one array bounds check per barrier; an efficient JVM implementation would require far less overhead, as the barriers could be inserted at a lower level, circumventing these Java safety checks. However, the bytecode transformation approach offers the strong advantage of being as portable as Java is. It can be used with any JVM on any platform.

The implementation of DITTO supports multiple invariants per class instantiation, multiple class instantiations per class, and multiple classes. Below are specifics about some aspects of the implementation. The bytecode transformation is implemented using the excellent Javassist package [20].

Hashing of objects. In previous work on incrementalization [1], the definitions of object equality are left to the programmer. This flexibility allows the programmer to equate two objects if they differ only in fields that she knows are irrelevant to the incremental computation. Since DITTO is automatic, an all-purpose strategy is required.

DITTO's memoization table, which maps a list of explicit arguments, stored in an `Object[]`, to a particular entry that represents a function call on those arguments, is implemented as a hash table. This requires a notion of argument array equality and hashing. In terms of equality, pointer equality of `Object[]` is obviously insufficient. Instead, equality is defined as the conjunction of pointer equality for the elements (arguments) that are object references, and semantic equality for the elements that are primitive types; the hash code is defined analogously, as a combination of `System.identityHashCode()`, or `Object.hashCode()` for primitive types like `Integer` or `Boolean`. This strategy conservatively preserves semantic equality of all arguments, while preventing sharing of non-primitive types (if the same computation node were to operate on two objects, semantically equal but in different locations on the heap, and only one was updated, then the node's cached result could be incorrect for one set of arguments.) In theory, semantic equality and hashing could be applied to any immutable type.

Our benchmarks indicate that this conservative notion of equality, though not

optimally flexible, performs well in practice on DITTO's target domain.

Efficient implementation of write barriers. Since the write barriers are implemented in Java, some care must be taken to ensure reasonable performance. DITTO employs two main optimization tactics. First, during the offline bytecode transformation phase, DITTO gathers the set of fields accessed by the invariant checks it is optimizing. Write barriers are only inserted on updates to these fields, since only writes to these fields could possibly affect the implicit arguments to the invariant checks.

Secondly, each memory address caught by the barriers incurs a hash table lookup to determine what computation nodes are affected by its mutation, even if the object at that address is unrelated to any invariant checks and affects no computation nodes at all. If there are many such other writes (or if the first optimization did not sufficiently reduce the number of barriers inserted), these lookups can cause significant overhead. To combat this phenomenon, the runtime portion of DITTO keeps a reference count of dependent invariant checks in the header of each object. The write barriers are constructed to first check that the reference count is greater than zero, and only then to add its field to the list of mutated ones. The reference count for a particular object is decremented when an invariant's hash table lookup is done and the dirty nodes identified. This way, if any of its dirty nodes accesses the object again, its reference count will be incremented. If not, since the dirty nodes are the only ones that accessed it beforehand, it is no longer relevant to that invariant check and does not need to be monitored further.

In practice, the inclusion of a 'header' reference count is implemented by creating a new class `IncObject` that inherits from `java.lang.Object`, and contains an integer field

corresponding to the reference count. DITTO then sets the penultimate class in the class hierarchy of each object type used by invariant checks to inherit from this class instead of `java.lang.Object`.

Optimizing leaf calls. If a function f is invoked with arguments a that do not lead to recursion, it is often faster to compute $f(a)$ outright than to memoize it. This situation commonly occurs at the ends of data structures, when a final `null` value is reached. Thus, if all the non-primitive arguments to a function call are `null`, DITTO does not perform any cache lookups and instead runs $f(a)$ to determine the return value. In addition, small commonly used non-recursive functions, such as `hashCode()` and `size()`, are special-cased as well. In all cases, the implicit arguments to these functions, if any, are still recorded.

3.5 Evaluation

All measurements were performed on a Pentium M 1.6 GHz computer with 1 gigabyte of RAM, running the HotSpot 1.5 JVM.

3.5.1 Data structure benchmarks

We measured DITTO on several data structure benchmarks. Each data structure is instantiated at several sizes and then modified 10,000 times. We measured only small sizes (from 50 to 3,200) to reflect what we believe is common real-world usage. (Incrementalization generally produces asymptotic improvement, so arbitrary speedups can be had at large data structure sizes.) In each case, wall-clock time, including GC and all other VM and incrementalization overheads, is measured. The data structures and their modification

```
Boolean checkHashBuckets(int i) {
  if (i >= buckets.length) return true;
  boolean b1 = checkHashElements(buckets[i], i),
          b2 = checkHashBuckets(i+1);
  return b1 && b2;
}

Boolean checkHashElements(HashElement e, int i) {
  if (e == null) return true;
  return (e.key.hashCode() % buckets.length == i) &&
         checkHashElements(e.next, i);
}
```

Figure 3.9: Invariant for the hash table. The invariant is invoked as `checkHashBuckets(0)`.

patterns are described below.

If an operation requires a "random" element, it is selected at random from the set of elements guaranteed to fulfill the operation. For instance, the element for a deletion is chosen at random from the elements already in the data structure.

Ordered List. The `OrderedIntList` and its invariant `isOrdered` were described in Section 3.2. The modifications were 50% insertion of a random element, 25% deletion of a random element, and 25% deletion of the first element in the list (as in a queue).

Hash Table. The `HashTable` data structure maps keys to values, using chaining to store multiple entries in the same bucket. The invariant check, shown in Figure 3.9, verifies that no entry is in the wrong bucket. Note that the single invariant encompasses two functions. The modifications were 50% random insertions and 50% random deletions.

Red-Black Tree. We used the open-source GNU Classpath version of `TreeMap`, which implements a red-black tree in 1600 lines of Java. The invariants verify the required properties of a red-black tree, and check the following properties: (i) the tree is well-ordered

(ii) local red-black properties (e.g. a red node has black children) (iii) the number of black nodes along any path from the root to a leaf is the same. See Figure 3.10 for the code. The modifications consisted of 50% random insertions and 50% random deletions.

A red-black tree is particularly well suited to dynamic invariant checks because

1. It is a data structure with nontrivial behaviors for even simple operations such as insert and delete that are hard to "get right".
2. It has several invariants that are difficult to analyze statically but are relatively easy to write as code.

However, its complexity also challenges DITTO: a single operation can alter the data structure layout significantly, reordering, adding, and removing nodes. Additionally, two of the invariants enforce global constraints, requiring nontrivial incremental updates to the computation graph. For these reasons, we considered the red-black tree an acid test for the feasibility of DITTO.

Analysis

The results of incrementalization for these data structures at various sizes are presented in Figure 3.11. In each case DITTO successfully incrementalized the invariant, producing an asymptotic speedup over the unincrementalized version. The average speedup at 3200 elements is 7.5x.

DITTO performs well for medium to large sized data structures. However, there is some baseline overhead due to write barriers and the incrementalization data structures that have to be maintained. To more closely analyze behavior on smaller data structures, for

each structure we measured the *crossover size*, the data structure size at which it is faster to run DITTO's incrementalized version of a check than the original, all overheads considered.[2]

	Crossover size
Ordered list	≈ 250
Hash table	≈ 100
Red-black tree	≈ 200

These crossover sizes suggest that DITTO can be used as part of the development process for programs with relatively small data structures as well.

3.5.2 Sample applications

Netcols is a Tetris-like game written by a colleague in 1600 lines of Java. Jewels fall from the sky through a rectangular grid and must be made to form patterns as they land. The program keeps an array `top` of the position of the highest landed jewels in each column, and maintains the invariant that no jewels are floating – i.e. there are no empty squares below the highest spot in each column, and there are no bejeweled squares about it; see Figure 3.12 for the code.

The main event loop averaged 80ms end-to-end time with the invariant check running, noticeably sluggish. With DITTO, the event loop averaged 15ms.

JSO [43] is a JavaScript obfuscator written in 600 lines of Java. It renames JavaScript functions, and keeps a map from old names to new so that if the same function is invoked again, its correct new name will be used. However, functions whose names have certain properties or that are on a list of reserved keywords should not be renamed. Thus, we check the invariant that keys in the renaming map do not meet any exclusionary

[2] In [1], a crossover point is also mentioned, often occurring at size 1. Though our attempt to contact the author failed, we imagine that this point is measuring a different phenomenon, perhaps a theoretical crossover point without runtime overheads.

criteria. See Figure 3.13. To enable this invariant, we maintain an auxiliary list of map keys, `names`.

Figure 3.14 shows the results of feeding `JSO` JavaScript inputs of varying sizes. DITTO's incrementalized version of the check is able to mitigate much of the overhead.

3.6 Related Work

Languages such as JML [47] and Spec# [13] provide motivation for this work. These languages enable the user to write data structure invariant checks (among other specifications) directly into their code. In some cases, these checks are statically verifiable, in which case DITTO provides a complimentary solution: very small offline overhead followed by a moderate runtime overhead and verification for testing inputs, as opposed to a larger offline overhead, no runtime overhead, and verification for all inputs. On the other hand, the cases where the checks must be verified at runtime are perfectly suited to DITTO.

Software model checking [12, 28, 65] is a powerful technique for static verification. However, most model checkers do not perform well when required to maintain a precise heap abstraction, such as when verifying red-black tree invariants, often failing to verify structures of depth greater than five. Recent work by Darga et al. [25] has made progress toward verification of complex invariants, but the depth bound is still small for complex data structures and ghost fields and programmer annotations are required.

Algorithm incrementalization has been the subject of considerable research [27, 56, 57, 59, 40]; see [62] for a comprehensive bibliography of early work. Initial research often focused on hand-incrementalizing particular algorithms [60].

Liu et al. began to devise a systematic approach to incrementalization [51], culminating with recent work [50] that presented a semi-automated incrementalizer for object-oriented languages. This work differs from DITTO in two respects. First, it incrementalizes algorithms primarily through memoization (rather than a hybrid dependence/memoization solution), which may require recomputation even though true dependencies have not been modified. Second, it requires a library of hints, one for each type of input modification, that describe how the modification pertains to the incrementalization; DITTO allows for arbitrary updates.

More recently, Acar et al. [1, 2] have developed a robust framework for incrementalization that uses both memoization and change propagation. This framework offers a number of library functions with which a programmer can incrementalize functional code functions and achieve considerable speedups. Acar's work and DITTO differ in several respects.

Acar's incrementalizer operates in the context of a purely functional program in ML. Input changes and computation dependences must be specified explicitly by the programmer. The framework is general and, thanks to the functional environment, can incrementalize computations that return new objects. Dependencies are tracked at the statement level, which allows for very precise change propagation. However, to achieve this granularity, functions must be statically split into several components, so that individual statements can be executed directly. These sub-functions must then be converted to continuation-passing style.

In contrast, DITTO operates in Java. Incrementalization is done automatically via

write barriers and automatic instrumentation. DITTO operates on the domain of data structure invariant checks: recursive, side-effect-free functions. Because the rest of the program may be arbitrarily imperative, functions that return new objects are not allowed (such objects may be modified and thus are unsuitable for memoization). However, many common invariant checks can be written despite this restriction. Dependencies are tracked at the function level, which obviates the need for function splitting and CPS conversion (as well as optimizations required to elicit good CPS performance from Java VMs). The suitability of optimistic memoization for invariant checks further enables a simple implementation. Though the function-level granularity can require more code to be re-executed than necessary, invariant check functions tend to be small, and executing an entire function is often nearly as fast as identifying the few statements in that function that actually have modified dependences and rerunning just those.

Work by Cooper, Burchett, et al. [16, 23] on functional reactive programming (FRP) has tackled incrementalization from a different angle. Their work deals with providing linguistic support for programs that deal with a sequence of user inputs, which are analogous to the sequence of inputs given to DITTO's invariant checks. Like DITTO, their FRP system constructs a computation graph of the program and uses change propagation to determine what portions of the program to re-execute in response to a sequence of events. DITTO can be seen as an extension of this change propagation model that handles implicit heap inputs and works in an imperative setting. (Like Acar's work, FRP deals with functional programs and values.) Like DITTO, the FRP implementation finds the construction and maintenance of a complete dataflow graph overly cumbersome and inefficient. DITTO solves this problem

by treating individual function invocations as the smallest acceptable level of granularity in constructing its computation graph; FRP uses a more sophisticated case-by-case lowering algorithm to coalesce certain sets of dataflow graph nodes into one node.

```
void invariants() {
  if (!isRedBlack(root) || checkBlackDepth(root) == -1 ||
  ! isOrdered(root, Integer.MIN_VALUE, Integer.MAX_VALUE))
    complain();
}

Boolean isOrdered(Node n, int lower, int upper) {
  if (n == nil) return true;
  if (n.key <= lower || n.key >= upper)
    return false;
  if (n.key <=  n.left.key || n.key >= n.right.key)
    return false;
  boolean b1 = isOrdered(e.left, lower, n.key),
       b2 = isOrdered(e.right, n.key, upper);
  return b1 && b2;
}

Boolean isRedBlack(Node n) {
  if (n == nil) return true;
  Node l = n.left, r = n.right;
  if (n.color != BLACK && n.color != RED)
    return false;
  if ((l != nil && l.parent != n) ||
      (r != nil && r.parent != n))
    return false;
  if (n.color == RED && (l.color != BLACK ||
                         r.color != BLACK))
    return false;
  boolean b1 = isRedBlack(l), b2 = isRedBlack(r);
  return b1 && b2;
}

Integer checkBlackDepth(Node n) {
  if (n == nil)
    return 1;
  int left = checkBlackDepth(n.left);
  int right = checkBlackDepth(n.right);
  if (left != right || left == -1)
    return -1;
  return left + (n.color == BLACK ? 1 : 0);
}
```

Figure 3.10: Invariants for the red-black tree. `nil` is a special dummy node in the implementation that is always black.

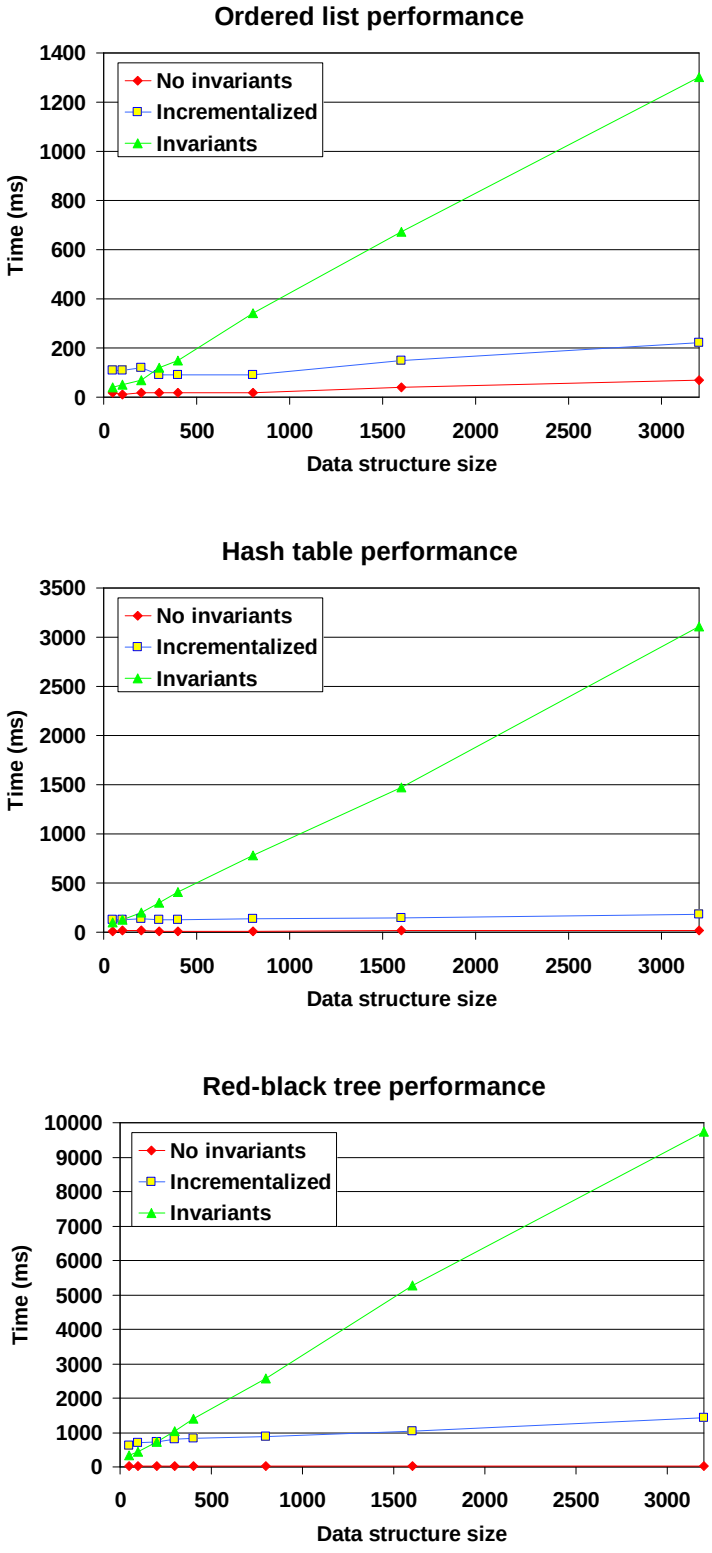


Figure 3.11: Results for data structure benchmarks. Each graph compares the performance of code with (i) no invariant checks (ii) standard invariant checks (iii) incrementalized invariant checks on different sizes of the data structure.

```
Boolean checkTop(int col) {
  if (col == width) return;
  boolean b1 = checkEmpty(col, top[col]),
          b2 = checkFull(col, top[col]-1),
          b3 = checkTop(col+1);
  return b1 && b2 && b3;
}

Boolean checkFull(int col, int row) {
  if (row == 0) return true;
  return jewels[col][row] != nullJewel &&
         checkFull(col, row-1);
}

Boolean checkEmpty(int col, int row) {
  if (row == height) return true;
  return jewels[col][row] == nullJewel &&
         checkEmpty(col, row+1);
}
```

Figure 3.12: The invariant check that verifies that a `netcols` grid has no floating jewels.

```
Boolean goodMapping(JList names) {
  if (names == null) return true;
  String s = (String) names.value;
  if (Character.isUpperCase(s.charAt(0)) ||
      Character.isDigit(s.charAt(0)))
    return false;
  boolean b1 = ! inReserved(s, 0),
          b2 = goodMapping(names.next);
  return b1 && b2;
}

Boolean inReserved(String s, int off) {
  if (off == reserved_names.length) return false;
  return s.equals(reserved_names[off]) || inReserved(s, off+1);
}
```

Figure 3.13: Invariant check for `JSO` that ensures that a protected function is not renamed.

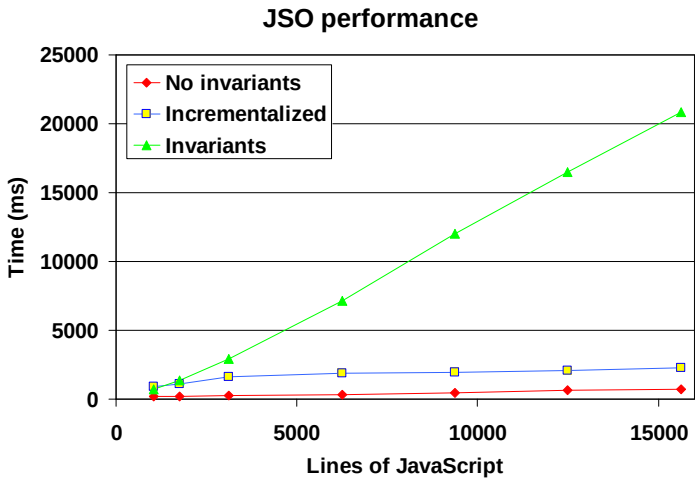


Figure 3.14: Performance numbers for `JSO`.

Chapter 4

Discussion and Conclusion

We have discussed two complex program transformations, a specializer and an incrementalizer, that are driven in large part by simple dynamic analyses. The previous chapters detailed the technical aspects of the optimizations; this brief chapter summarizes the argument made in this dissertation and discusses some additional points.

It is possible to perform simple online dynamic analyses that enable significant program transformations. Because of the constrained nature of their execution environment, online dynamic analyses are necessarily simple. Many dynamic program transformations are correspondingly simplistic: they use these simple measurements to answers simple questions, such as which function is most frequently invoked by a dynamic dispatch call. This state of affairs offers a challenge: how to construct a sophisticated program transformation that depends on a carefully chosen but limited pool of information.

In this dissertation, we have argued that if the dynamic analyses are well chosen, and a suitable domain of programs to transform is selected, sophisticated transformations

are possible. We contribute two program transformations to support this hypothesis. Our specializer uses a simple dynamic test to predict object invariance; it can speed up programs with many invariant objects by as much as 500%. DITTO dynamically constructs computation graphs to asymptotically speed up data structure invariant checks.

It is possible to trade transformative power for broader applicability. The two transformations described in this dissertation apply to specific program domains, and yield significant speedups. Tranformations fueled by dynamic analyses need not be domain-specific; the cost of increased generality is a lessening in the effectiveness of the transformation.

In additional work, we have constructed a transformation that operates on a large class of programs and yields more modest speedups. Jolt [70] is an automatic object churn remover for large-scale Java programs. *Object churn* occurs when a temporary data object is allocated on the heap and dies shortly after its creation. Due to the way object-oriented programs are structured, (a) churn is very frequent in large programs and (b) churned objects are difficult to remove via standard optimizations such as escape analysis because they often outlive their allocation context.

Jolt eliminates some object churn by identifying allocation sites that generate short-lived objects and creating a scope that includes both the allocation sites and the functions involved in the objects' use and death. Because the objects' lifetimes are now bounded by this scope, Jolt can then invoke a traditional escape analysis to eliminate their allocations. Rather than rely on static pointer and lifetime analyses, Jolt uses two very simply dynamic analyses to drive this process.

1. It monitors the value of the memory allocator's heap allocation pointer to infer how many objects have been allocated by each function and its descendants.

2. It piggybacks on the garbage collector to identify which of these objects die at the end of various function scopes.

In combination, these two analyses identify short-lived objects, where they are allocated, and where they tend to die. By placing such object allocations on the stack or in registers, Jolt removes four times as many allocations as standard optimizations and speeds up programs by as much as 15%.

Jolt's large domain (any program with object churn) and modest speedups contrast with DITTO and the specializer's small domain and large speedups, and illustrates the tradeoffs between power and generality.

Implementing dynamic analyses and transformations directly into a VM trades ease of implementation and portability for power. The specializer in this document was implemented as a JVM modification, whereas the incrementalizer was implemented as a source-to-source transformation with a runtime support library. Implementing the specializer yielded a host of positives and negatives in contrast to implementing DITTO.

Positives:

- It is powerful. It can build on the mechanisms available to the VM, including manipulating the garbage collector (done by Jolt) and the VM's object representation (done by the specializer). DITTO must respect the limits of the Java language.

- It allows for an efficient implementation. For instance, the specializer could insert

write barriers in a low level IR, whereas DITTO must implement them in Java.

Negatives:

- It requires understanding and manipulating a massive, extremely complex body of code. VMs already contain sophisticated compilers and runtime infrastructure, and it is daunting and time-consuming to identify and modify the relevant portions of a codebase containing tens or hundreds of thousands of lines of code.
- The transformation is very difficult to port. Because it relies on the specifics of a particular VM implementation, it must be re-implemented to work with another VM.
- Large-scale program transformations are difficult to debug. Our specializer had to generate thousands of lines of code. Because it was operating inside a JIT compiler, it had to use a VM-specific, poorly documented intermediate language. Because the VM is so complicated – JIT optimizations often had nonlocal effects – debugging generally meant running the entire VM and eliminating confounding variables as they became apparent. It was very difficult to establish safety and soundness.

Annotation-free implementations of complex transformations are possible. One feature that I required of the two transformations in this dissertation, as well as the third one described in this section (Jolt), is that they be automatic, requiring no user input to run. I insisted upon this requirement primarily as a matter of practicality: if an optimization requires additional programmer steps, whether in the form of programmer annotations or other input, a programmer is less likely to adopt it than if the optimization is fully automatic.

My rationale was simple: though there have been a host of powerful, effective staged or annotation-driven optimizations in the last twenty years, none are in wide use for most common languages. C, C++, and Java users rely on their compilers, sans annotations, to perform transformations, and programmers of scripting languages like Perl and Python – who view the lack of a compilation phase as a rapid development aid – are content to just invoke their language's interpreter. Optimizations that, no matter how powerful, require a manual step are simply not used by the vast majority of programmers.

Due to the complexity of the transformations attempted in this paper, making them annotation-free was a daunting requirement, and ended up being a major differentiator of this work from prior work in the area. We had to sacrifice flexibility and expressiveness. For instance,

- DyC [35], a user-driven specializer, allows the programmer to specify via annotations a number of specialization policies, such as how to cache, speculate, and dispatch. Our specializer applies one automatic set of policies to all program.
- Acar et al.'s incrementalizer [1] enables the user to specify exactly what constitutes equivalence for the purpose of memoization; DITTO conservatively relies on a combination of pointer and value equality.

In these and other ways, our transformations have had to limit their effectiveness to achieve transparency. However, I believe that the cost of this decrease in power is outweighed by the benefit of a better programmer experience, and remain confident that similar tradeoffs can be made for other complex manual optimizations to render them automatic as well.

Furthermore, given the historical unwillingness of programmers to adopt manual

transformations (as frustrating as this fact may be), I would argue that for any transformation to be considered practical for the common programmer, it should be automatic, even if such a design would yield a less powerful transformation than a manual one.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation, pages 96–107, New York, NY, USA, 2006. ACM Press.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In Symposium on Principles of Programming Languages, pages 247–259, 2002.

[3] F. E. Allen and J. Cocke. A program data flow analysis procedure. Commun. ACM, 19(3):137, 1976.

[4] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[5] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen, 1994.

[6] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F.Sweeny. Adaptive Optimization in the Jalapeno JVM. In ACM SIGPLAN Conference on Object

Oriented Programming, Systems, Languages and Applications (OOPSLA '00), October 2000.

[7] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of java. In Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 111–129. ACM Press, 2002.

[8] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In Proceedings of the 2001 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI), pages 168–179, June 2001.

[9] Matthew Arnold and Barbara G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In ECOOP, pages 498–524, 2002.

[10] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In SIGPLAN '96 Conference on Programming Language Design and Implementation, pages 149–159, 1996.

[11] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Runtime Optimization System. In Proceedings of the 2000 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI), pages 1–12, 2000.

[12] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In SIGPLAN Conference on Programming Language Design and Implementation, pages 203–213, 2001.

[13] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec-sharp programming system: An overview. `http://research.microsoft.com/specsharp/papers/krml136.pdf`.

[14] M. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002), 2002.

[15] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization, 2003.

[16] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: a static optimization technique for transparent functional reactivity. In PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pages 71–80, New York, NY, USA, 2007. ACM.

[17] B. Calder, P. Feller, and A. Eustace. Value profiling. Journal of Instruction Level Parallelism, March 1999.

[18] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In Norman 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.

[19] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, Proceedings of

the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002, pages 322–328. CSREA Press, June 2002.

[20] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03), Erfurt, Germany, volume 2830 of LNCS, pages 364–376, September 2003.

[21] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanschi. A uniform approach for compile-time and run-time specialization. In Partial Evaluation. International Seminar., pages 54–72, Dagstuhl Castle, Germany, 12-16 February 1996. Springer-Verlag, Berlin, Germany.

[22] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 145–156, St. Petersburg Beach, Florida, 21–24 January 1996.

[23] Gregory Cooper. Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language. Providence, Rhode Island, 2008. PhD dissertation, Department of Computer Science, Brown University, 2008.

[24] Craig Zilles and Gurinder Sohi. A Programmable Co-processor for Profiling. In Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7), January 2001.

[25] Paul T. Darga and Chandrasekhar Boyapati. Efficient software model checking of data structure properties. SIGPLAN Not., 41(10):363–382, 2006.

[26] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing u2122 software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In CGO '03: Proceedings of the international symposium on Code generation and optimization, pages 15–24. IEEE Computer Society, 2003.

[27] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 105–116, New York, NY, USA, 1981. ACM Press.

[28] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng, and W Visser. Tool-supported program abstraction for finite-state verification. In International Conference on Software Engineering, pages 177–187, 2001.

[29] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'c: A language for high-level, efficient, and machine-independent dynamic code generation. In Symposium on Principles of Programming Languages, pages 131–144, 1996.

[30] Bowen Alpern et al. The Jalapeno virtual machine. IBM Systems Journal, Java Performance Issue, 39(1), 2000.

[31] S. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. International Symposium on Code Generation and Optimization, pages 241–252, 2003.

[32] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Michael Bebenita, Mason Chang, Michael Franz, Edwin Smith, Rick Reitmaier, and Mohammad Haghighat. Trace-based just-in-time type specialization for dynamic languages. In PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation, New York, NY, USA, 2009. ACM Press.

[33] Hotjava 1.0 signature bug, 1997. `http://www.cs.princeton.edu/sip/news/april29.html`.

[34] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Annotation-directed run-time specialization in C. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97), volume 32, 12 of ACM SIGPLAN Notices, pages 163–178, New York, June 12–13 1997. ACM Press.

[35] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Technical Report TR-97-03-03, University of Washington, Department of Computer Science and Engineering, March 1997.

[36] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eg-

gers. The benefits and costs of DyC's run-time optimizations. ACM Transactions on Programming Languages and Systems, 22(5):932–972, 2000.

[37] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in dyc. In Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, pages 293–304. ACM Press, 1999.

[38] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. May 2002.

[39] Timothy Heil and James E. Smith. Concurrent Garbage Collection Using Hardware-Assisted Profiling. In International Sympsosium on Memory Management (ISMM), October 2000.

[40] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. ACM SIGPLAN Notices, 35(5):311–320, 2000.

[41] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial Evaluation and Automatic Program Generation. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[42] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial evaluation and automatic program generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[43] Jso. `http://shaneng.awardspace.com/#jso_description`.

[44] Todd B. Knoblock and Erik Ruf. Data Specialization. In Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, pages 215–225, Philadelphia, Pennsylvania, 21–24 May 1996.

[45] B. Korel and J. Laski. Dynamic program slicing. Inf. Process. Lett., 29(3):155–163, 1988.

[46] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. ACM SIGPLAN Notices, 28(6):56–67, 1993.

[47] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, Behavioral Specifications of Businesses and Systems, pages 175–188. Kluwer Academic Publishers, 1999.

[48] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In SIGPLAN Conference on Programming Language Design and Implementation, pages 137–148, 1996.

[49] Kevin M. Lepak and Mikko H. Lipasti. Silent stores for free. In MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, pages 22–31. ACM Press, 2000.

[50] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni Ellen Liu. Incrementalization across object abstraction. In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pages 473–486, New York, NY, USA, 2005. ACM Press.

[51] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. Science of Computer Programming, 24(1):1–39, 1995.

[52] M.Burrows, U.Erlingson, S.T.A.Leung, M.T.Vandevoorde, C.A.Waldspurger, K.Walker, and W.E.Weihl. Efficient and Flexible Value Sampling. In Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pages 160–167, November 1–3 2000.

[53] Steve Meloan. The Java HotSpot (tm) Perfomance Engine: An In-Depth Look. Article on Sun's Java Developer Connection site, 1999.

[54] Markus U. Mock, Craig Chambers, and Susan J. Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-33), pages 291–302, December 2000.

[55] Robert Muth, Scott Watterson, and Saumya Debray. Code Specialization Based on Value Profiles. In Proceedings of the 7^{th} International Static Analysis Symposium (SAS 2000), pages 340–359. Springer LNCS vol. 1824, June 2000.

[56] Bob Paige and J. T. Schwartz. Expression continuity and the formal differentiation of algorithms. In POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 58–71, New York, NY, USA, 1977. ACM Press.

[57] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. ACM Trans. Program. Lang. Syst., 4(3):402–454, 1982.

[58] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In Proceedings of the 15th ACM Symposium on Operating Systems Principles, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[59] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 315–328, New York, NY, USA, 1989. ACM Press.

[60] G. Ramalingam. Bounded incremental computation. Technical Report 1172, Univ. of Wisconsin, Madison, Computer Sciences Dept., 1210 West Dayton St., Madison, WI 53706, USA, 1993.

[61] G. Ramalingam. Data flow frequency analysis. In SIGPLAN Conference on Programming Language Design and Implementation, pages 267–277, 1996.

[62] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 502–510, New York, NY, USA, 1993. ACM Press.

[63] Erik Ruf. Topics in Online Partial Evaluation. PhD thesis, Stanford University, 1993.

[64] Erik Ruf and Daniel Weise. Opportunities for online partial evaluation. Technical Report CSL-TR-92-516, 1992.

[65] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, Proc. 9th International Conference on Computer Aided Verification (CAV'97), volume 1254, pages 72–83. Springer Verlag, 1997.

[66] S. Subramanya Sastry, Rastislav Bodik, and James E. Smith. Rapid profiling via stratified sampling. In Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA-01), volume 29,3 of Computer Architecture News, pages 278–289, New York, June 2001. ACM Press.

[67] Subramanya Sastry. Techniques for Transparent Program Specialization In Dynamic Optimzers. PhD thesis, University of Wisconsin, Madison, March 2003.

[68] U. P. Schultz, J. L. Lawall, and C. Consel. Specialization patterns. In Proceedings of the 15^{th} IEEE International Conference on Automated Software Engineering (ASE 2000), pages 197–206, Grenoble, France, September 2000. IEEE.

[69] Ulrik Schultz, Julia Lawall, Charles Consel, and Gilles Muller. Towards automatic specialization of Java programs. In R. Guerraoui, editor, Proceedings ECOOP'99, LCNS 1628, pages 367–390, Lisbon, Portugal, June 1999. Springer-Verlag.

[70] Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pages 127–142, New York, NY, USA, 2008. ACM.

[71] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio

Nakatani. A dynamic optimization framework for a java just-in-time compiler. In OOPSLA, pages 180–194, 2001.

[72] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In IVME '03: Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators, pages 50–57. ACM Press, 2003.

[73] Robert Endre Tarjan. Fast algorithms for solving path problems. J. ACM, 28(3):594–614, 1981.

[74] F. Tip. A survey of program slicing techniques. Journal of programming languages, 3:121–189, 1995.

[75] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian Memory Protection. In Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), October 1–3 2002.

[76] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. An event-driven multithreaded dynamic optimization framework. In Parallel Architectures and Compilation Techniques (PACT), 2005.

[77] X. Zhang and R. Gupta. Cost effective dynamic program slicing, 2004.

[78] Craig B. Zilles and Gurindar S. Sohi. A programmable co-processor for profiling. In HPCA, pages 241–, 2001.