Runtime Specialization With Optimistic Heap Analysis

Ajeet Shankar
University of California, Berkeley
aj@cs.berkeley.edu

S. Subramanya Sastry
University of Wisconsin, Madison
sastry@cs.wisc.edu

Rastislav Bodík
University of California, Berkeley
bodik@cs.berkeley.edu

James E. Smith
University of Wisconsin, Madison
jes@ece.wisc.edu

ABSTRACT

We describe a highly practical program specializer for Java programs. The specializer is powerful, because it specializes optimistically, using (potentially transient) constants in the heap; it is precise, because it specializes using data structures that are only partially invariant; it is deployable, because it is hidden in a JIT compiler and does not require any user annotations or offline preprocessing; it is simple, because it uses existing JIT compiler ingredients; and it is fast, because it specializes programs in under 1s.

These properties are the result of (1) a new algorithm for selecting specializable code fragments, based on a notion of influence; (2) a precise store profile for identifying constant heap locations; and (3) an efficient invalidation mechanism for monitoring optimistic assumptions about heap constants. Our implementation of the specializer in the Jikes RVM has low overhead, selects specialization points that would be chosen manually, and produces speedups ranging from a factor of 1.2 to 6.4, comparable with annotation-guided specializers.

Categories and Subject Descriptors

D.3.m [**Programming Languages**]: Miscellaneous

General Terms

Algorithms, Languages, Performance

Keywords

Dynamic optimization, partial evaluation, program analysis, specialization

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 [2, 8, 13, 14, 17, 21, 22, 26, 28, 31, 35, 36, 43, 44, 45]. 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 [23].

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 [14, 21, 31], or heavyweight program analysis [35]. 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 paper 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 of their 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 [35], 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 paper is the first fully transparent specializer to use heap data. Suganuma et al. [45] 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 paper, 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 paper is organized as follows. Section 2 is an overview of the system, and presents three main challenges of dynamic specialization: region selection, heap invariance detection, and invalidation. Sections 3, 4, and 5 discuss our solutions to the three principle challenges, and Section 6 describes some details about region creation. Finally, we present an experimental evaluation of our work in Section 7. Related work is discussed in Section 8.

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.

2.1 Specialization Model

Classical specialization is applied in scenarios in which a program P is re-executed with a part of its input unchanged [27]. 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 [13] and DyC [21], 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) [39] that is further simplified by specializing traces created in the spirit of the Dynamo specializer [9]. 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 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 over-approximates the benefit (see Section 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.

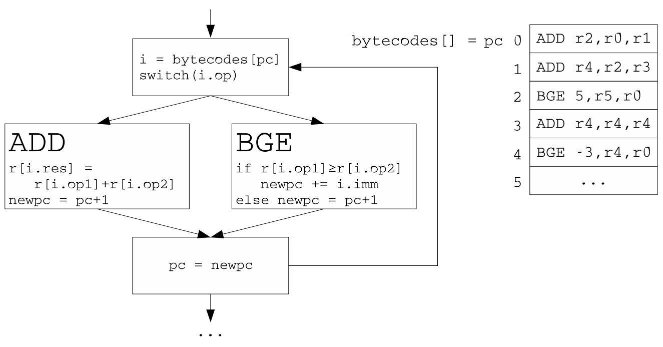

Figure 1: A program to be specialized: a simple interpreter (left) and the set of bytecodes it is to interpret (right).

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 4). We have found this profile to be sufficiently accurate. 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 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 [6, 11, 15, 25, 33, 41].

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 5.

2.2 Example

In this section, we illustrate the specialization procedure with a specific example. Figures 1 and 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 3, on the function and its associated dynamic execution information. The algorithm estimates the number of dynamic instructions that will follow each instruction in the function and selects those with the most following instructions:

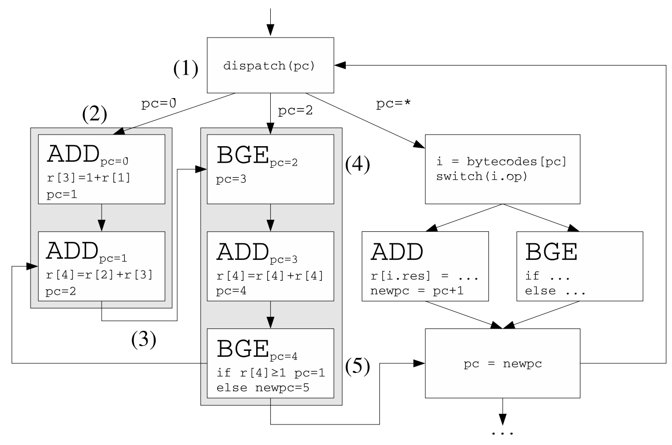

Figure 2: The interpreter in its specialized incarnation. The two shaded columns are specialized traces, while the diamond on the right is the original code.

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 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.

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 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 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 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 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 [18]. 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 [4, 5, 19], so we do not address these issues in this paper.

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 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 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.

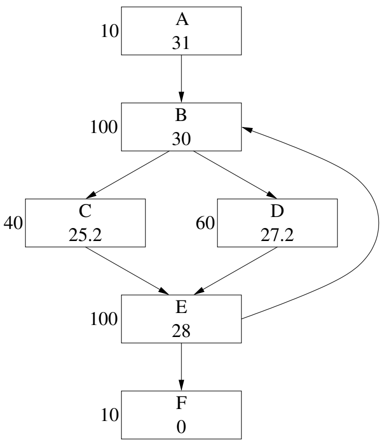

Figure 3: Influence numbers for a sample control flow graph. The number to the left of each block is its dynamic execution count, and the number inside is the influence.

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 [48] 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 [51], 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 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 $e_1e_2...e_n$ 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:

$$occurrence(t) \equiv t \text{ is followed on an execution of } f \qquad (1)$$

$$\mathbf{Pr}[occurrence(t)] = \prod_{\text{edge } e=(m,n)\in t} \frac{count(e)}{count(m)} \qquad (2)$$

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

$$\mathbf{Pr}[reach(m,s)] = \sum_{\text{edge } e=(m,n)} \frac{count(e)}{count(m)} \cdot \begin{cases} 1 & \text{if } n = s \\ \mathbf{Pr}[reach(n,s)] & \text{otherwise} \end{cases} \qquad (5)$$

$$\mathbf{E}[len(m)] = 1 + \sum_{\text{edge } e=(m,n)} \frac{count(e)}{count(m)} \mathbf{E}[len(n)] \qquad (6)$$

$$influence(i) = \mathbf{Pr}[reach(start,i)] \cdot \mathbf{E}[len(i)] \qquad (7)$$

Figure 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.

$$\mathbf{E}[num(m)] = \frac{count(m)}{count(start)} \qquad (8)$$

$$\mathbf{E}[slen(m,s)] = 1 + \sum_{\text{edge } e=(m,n)} \frac{count(e)}{count(m)} \cdot \begin{cases} 0 & \text{if } n = s \\ \mathbf{E}[slen(n,s)] & \text{otherwise} \end{cases} \qquad (9)$$

$$influence(i) = \mathbf{E}[num(i)] \cdot \mathbf{E}[slen(i,i)] \qquad (10)$$

Figure 5: Alternate 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.

We define the *influence* of an instruction i with respect to a particular trace t as the length of the subtract from the first occurrence of i along t until the end of the trace:

$$influence_t(i) \equiv length(e_k ... e_n) \qquad (3)$$

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:

$$influence(i) \equiv \sum_{t \in EES} \mathbf{Pr}[occurrence(t)] \cdot influence_t(i) \qquad (4)$$

See Figure 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 of influence from Equation 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 7, Figure 4): the probability of ever reaching i during an invocation of f (*reach*, defined in Equation 5), and the expected length from i to the end of the function once i is reached (*len*, defined in Equation 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 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 9). The influence of i is the product of these two expectations (Equation 10).

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 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.

The systems of equations in Figures 4 and 5 can be solved via Ramalingam's data flow frequency analysis framework [38]. 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 [47] runs in $O(e \log v)$ time, and the Allen-Cocke interval analysis algorithm [1] 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 Appendix A.

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 7 compares influence against the other heuristics mentioned above on the actual Java programs described in Figure 6. Each of the other metrics successfully ranked the optimal dispatch 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.

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. Alternately, 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

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 [21]	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 6: Description of benchmarks and their inputs.

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, we would make it

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

While a number of static analyses [3, 30] 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 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 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.

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 runtime. 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.

[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 [32].

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-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.

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 alternate detection strategies.

[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).

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.

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 8 for a table of the number of write barriers that had to be inserted in each of this paper's benchmarks.

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.

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

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 [24], 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 8 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 7, we feel that it is suitable for a runtime environment.

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.

5.3 Alternate 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. [49] 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 8), 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 [16].

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 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 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 [9]. 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.

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 [35] and Suganuma et al. [45], 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 [20]. 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.

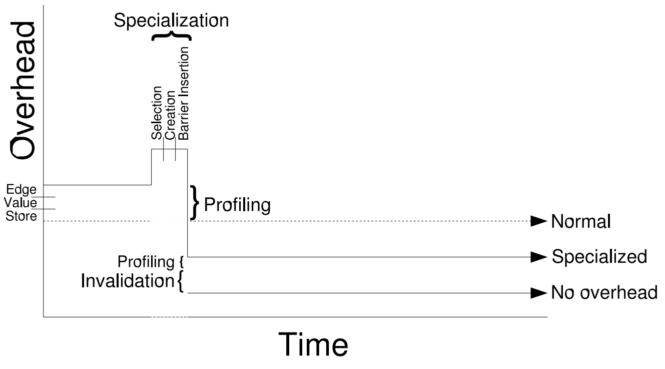

Figure 9: A conceptual view of the overheads involved in specialization. There is a steady-state profiling overhead, a one-time specialization overhead, and if specialization is successful, a steady-state invalidation overhead incurred by the write barriers.

7. EVALUATION

Methodology. The specializer presented in this paper 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 [6], 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 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 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 9 for a conceptual diagram of when these overheads come into play during a program execution, and Figure 10 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 10 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.

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 4.

Since all of these profiles must be gathered at runtime with low overhead, we used the Arnold-Ryder sampling framework [6]. 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. [33], 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. [11].

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. [33] 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% [52]. 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%.

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 [21]. 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 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

		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 10: 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.

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 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.

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

[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.

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 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 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.

8. RELATED WORK

Specialization. Program specialization is a well-studied optimization technique [14, 44, 43, 13, 26, 2, 28, 17, 21, 8, 22, 36, 31]. In its classical form, code is optimized in a source-to-source transformation. Tempo [13], DyC [21], 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 [35] 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 paper 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 paper has the additional benefit of being fully transparent and immediately beneficial to end users. Suganuma et al. [45] 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. [50] have built a value specializer with speedups of 20% on top of their dynamic optimization framework, Trident.

Dynamic optimization. The profile-and-optimize dynamic approach described in this paper is similar to other transparent dynamic optimization systems, like Mojo [10], Hotspot [34], and others. In particular, our specializer leverages the Jikes RVM framework [5, 18] for recompiling specialized methods.

Profiling. The efficiency of our profilers rests on the Arnold-Ryder sampling framework [6]; we use it to employ a novel invariance detection profile. We use a method suggested by Calder et al. [11] for gathering hot value data. The use of optimistic assumptions to motivate dynamic optimization was presented by Arnold and Ryder [7].

Trace creation. The main specialization algorithm's trace creation procedure draws from on-line partial evaluation techniques [40], and was inspired by Dynamo [9], although Dynamo does not use heap invariants or unroll loops when optimizing, and only produces one optimized trace per program point. Sullivan et al. [46] 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 [29].

Invalidation. As far as we know, this paper presents the first implementation and evaluation of a working automatic detection and invalidation system. Calpa [35] 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 [21] 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 [37]. We use the bitmasking techniques employed by Diduce [24] to reduce write barrier overhead. The actual invalidation is similar in end result to on-stack replacement (OSR) techniques [12, 19]. 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 [49].

To the best of our knowledge, this is the first implementation of a transparent runtime specializer that uses heap data. While Sastry [42] 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 paper for detecting specialization points, generating traces, and linking them are simpler and lead to better specializations on the same benchmarks.

9. CONCLUSION

This paper 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.

Acknowledgements

We thank Brian Fields, William McCloskey, Manu Sridharan, and Umesh Shankar for feedback on earlier drafts of this paper, as well as the anonymous reviewers for their helpful comments. We especially thank David Mandelin for his help in formalizing the linear-time influence algorithm.

This work is supported in part by the National Science Foundation, with grants CCF-0085949, CCR-0105721, CCR-0243657, CNS-0225610, CCR-0326577, an award from University of California MICRO program, the Okawa Research Award, a National Defense Science and Engineering Graduate Fellowship, as well as donations from IBM and Intel. This work has also been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. The views expressed herein are not necessarily those of DARPA or IBM.

10. REFERENCES

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. The benefits and costs of DyC's run-time optimizations. *ACM Transactions on Programming Languages and Systems*, 22(5):932–972, 2000.

[23] 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.

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

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

[26] 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).

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

[28] 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.

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

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

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] G. Ramalingam. Data flow frequency analysis. In

SIGPLAN Conference on Programming Language Design and Implementation, pages 267–277, 1996.

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

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

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

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

[49] 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.

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

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

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

APPENDIX

A. LINEAR-TIME INFLUENCE ALGORITHM

We present an algorithm for computing the influence of an instruction n that is linear in the number of instructions 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.

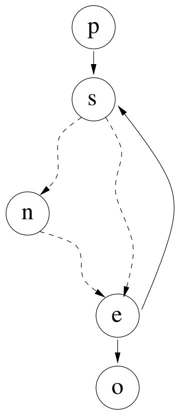

Figure 11: A loop. The dotted edges represent arbitrary acyclic control flow.

The difficulty arises if n is in a loop. Consider a standalone loop, as in Figure 11. The loop can have arbitrary branches and so on inside it, as long as it does not have any 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.)

$$\mathit{influence}(n) = \frac{\mathit{count}(p)}{\mathit{count}(\mathit{start})} F(n)L(n) \tag{11}$$

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 11 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 $\frac{\mathit{count}(\mathit{backedge})}{\mathit{count}(e)}$.

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.

A.1 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,

$$F(n) = \sum_{i=0}^{\infty} ((1 - f(n))b)^i \cdot f(n) \tag{12}$$

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

$$F(n) = \frac{f(n)}{1 - (1 - f(n))b} \tag{13}$$

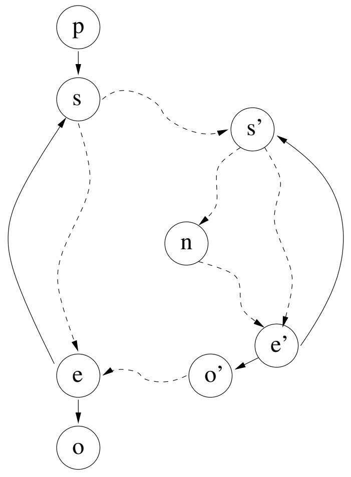

Figure 12: Nested loops. The inner loop variables have been primed.

A.2 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.

$$L(n) = r(n) + (1-b)\sum_{i=0}^{\infty} ilb^i \tag{14}$$

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

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

$$L(n) = r(n) + \frac{lb}{1-b} \tag{15}$$

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.

A.3 Handling Nested Loops

Consider a properly formed nested loop, as in Figure 12. 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.:

$$l' + (1-b')\sum_{i=0}^{\infty} il'b'^i \tag{16}$$

$$= l' + \frac{l'b'}{1-b'} \tag{17}$$

$$= \frac{l'}{1-b'} \tag{18}$$

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

$$\begin{aligned} F(s')L(s') &= \frac{f(s')}{1-(1-f(s'))b'}(r(s') + \frac{l'b'}{1-b'}) && (19)\\ &= \frac{1}{1}(l' + \frac{l'b'}{1-b'}) && (20)\\ &= \frac{l'}{1-b'} && (21)\end{aligned}$$

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.