Runtime Specialization With Optimistic Heap Analysis

Ajeet Shankar
UC Berkeley
aj@cs.berkeley.edu

Rastislav Bodík
UC Berkeley
bodik@cs.berkeley.edu

S. Subramanya Sastry
UW Madison
sastry@cs.wisc.edu

James E. Smith
UW Madison
jes@ece.wisc.edu

ABSTRACT

Runtime specialization is an optimization process that can provide significant speedups by exploiting a program's runtime state. Existing specializers employ a staged model that statically identifies specialization code regions and constant heap locations, and then optimizes them at runtime once the constants are known. This paper describes a dynamic program specializer that is transparent, and thus does not require any static components such as programmer annotations or pre-runtime analysis. The specializer uses (1) a novel store profile to identify constant heap locations that a staged system cannot, such as constants present in an interpreted program. This information is used in tandem with (2) a new algorithm for finding specialization starting points based on a notion of influence to create specializations. (3) An automatic invalidation transformation efficiently monitors assumptions about heap constants, and is able to deactivate specialized traces even if they are on the stack. An implementation of the specializer in Jikes RVM has low overhead in practice, selects specialization points that would be chosen manually, and produces speedups of 1.2x to 6.3x on a variety of benchmarks.

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* is a related, more aggressive strategy by which hot portions of code are heavily optimized under the assumption that frequently occurring and infrequently updated runtime values are constant; when these values turn up again, the optimized code is invoked [13, 40, 39, 12, 25, 2, 26, 16, 20, 7, 21, 33, 28, 32, 41]. For certain classes of programs, in which a few popular values consistently dictate execution behavior, employing this technique can result in marked speedups, up to 5x in some cases [22]. Interpreters, raytracers, and database query executors are all classes of programs that exhibit good specialization opportunities.

So far, specialization has 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 [28, 20, 13], or heavyweight program analysis [32]. This offline step has the additional disadvantage of forcing staged specializers to abstract away from the concrete state of a particular program run — its exact memory layout and execution details — 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 as it is running. This specialization technique has the following novel combination of properties:

- *It employs optimistic heap analysis.* Our dynamic specializer uses an efficient concrete heap profile to *optimistically* and *precisely* detect the invariance of individual heap locations. Thus 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 rapidly identifies specialization regions.* The specializer combines execution information gathered at runtime with a novel linear-time algorithm based on a new notion of instruction *influence* to automatically 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 automatically monitors assumed constants and invalidates specializations when they are modified.* To safely reap the benefits of optimistic invariance detection, the optimistic assumptions must be monitored throughout program execution. Thus, the specializer employs a low-overhead invalidation system that (a) detects when assumed constants have been updated and then (b) safely invalidates the corresponding specializations, even if they are currently on the execution stack.

These three features enable the specializer to operate fully transparently at runtime. 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 [32], 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.

Suganuma et al. [41] have constructed a fully dynamic specializer, but unlike 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. To the best of our knowledge, the system presented in this paper is the first fully transparent specializer to use heap data.

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 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 then 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:

- Optimistic and accurate fine-grained detection of heap invariants by a store profile.
- Fast identification of good specialization points via a new *influence* metric.
- An automatic mechanism for invalidating specializations when assumed invariants are modified.
- An implementation of this system that runs transparently on the Jikes RVM.

An evaluation of the system and its components on a series of benchmarks indicates that it has low overhead, finds beneficial specialization points, and produces speedups of 1.2x to 6.3x.

This paper is organized as follows. Section 2 is an overview of the system, and Sections 3 and 4 discuss the methods of execution profiling and specialization selection and creation that it uses. Optimistically detected invariants must be checked; we describe our checking mechanism and several alternative strategies in Section 5. Finally, we present an experimental evaluation of our work in Section 6. Related work is discussed in Section 7.

2. OVERVIEW

This section presents the key idea behind our dynamic analysis, an overview of the steps of our technique, and some examples illustrating its novel features.

Key idea: specialization through dynamic analysis. In order to develop a dynamic specializer, we rely on dynamic program analysis, which we use to answer all five questions underlying our model of program specialization:

1. Which variables exhibit frequently repeating values? Any code using these variables could be specialized for each of these hot values.
2. Which parts of heap data structures are (semi-)invariant? These memory locations can be considered run-time constants.
3. Given the answers to these two questions, what parts of the program should be specialized?
4. How does this specialization occur?
5. To maintain soundness, how does the specializer detect when assumed-invariant data structures are modified, so that it can invalidate any corresponding specializations?

We solve the first two questions with two forms of value profiling: we collect (1) a *hot value profile*, which computes the most frequently occurring values at each program point, and (2) a *store profile*, which computes the set of memory addresses that have been written to. Thanks to recent advances in *sampling-based profiling*, these profiles can be collected with high accuracy, yet with overheads sufficiently low for dynamic optimizers [5, 14, 37, 24, 30, 10].

We defer discussion of the third question until after we describe the specialization process.

Specialization model. A *specialization* starts at a *dispatch point* instruction p, which assigns to a variable, say x. If p frequently assigns the value v to x, we say p "has" or "results in" hot value v.

Each specialization contains several code *traces*, one for each hot value resulting from the dispatch point. Each trace is created using the constant propagation method described below, and is similar to a trace in the Dynamo system [8]. After specialization, at the dispatch point execution is rerouted to the appropriate trace by a *dispatcher* that compares the result of the instruction with the specialized hot values. If the result is not among the hot values, control is restored to the original code.

We turn a constant propagator into a trace creator by letting it consult the two profiles, essentially as follows: given a starting program point p that assigns hot value v to variable x, the specializer performs standard constant propagation starting from p, with a few modifications: (1) it assumes x is a constant equal to v; (2) it unrolls loops where appropriate; (3) it evaluates load instructions on the concrete memory state of the program at the time of specialization. This third modification discovers memory invariance: when the constant propagator encounters a load whose address a is a run-time constant, it consults the store profile. If, according to the profile, the memory location a has not been updated, the specializer optimistically assumes that its contents are also run-time constant; the load is eliminated and a's contents are propagated further as runtime constants.

However simple, this profile-based specializer-driven identification of memory invariance is both the most novel and

most powerful part of our specializer: it allows the specializer to generate code that is specialized not only with respect to the hot value v, but also with respect to the invariant part of the heap. Since the store profile monitors individual concrete locations, the invariance detection is as least precise as than any static pointer analysis working on an abstract heap, and generally as accurate; we present a further analysis in Section 2.2.

Allowing the constant propagator to evaluate memory-based instructions gives it the power to implement a host of optimizations with little effort. In addition to removing many memory loads, it can eliminate array bounds checks, null checks, typecast checks, and resolve instanceof operators and dynamically dispatched calls naturally during the course of propagation.

When generating each trace, the specializer unrolls loops either deterministically, when the conditional turns out to be a run-time constant, or speculatively, when it does not but the control flow path can be predicted reliably. Specialization is terminated when too few instructions are being optimized away, or when the likelihood of execution of the code path being specialized becomes too low, due to too many speculative branch predictions.

Since each hot value may have a different effect on memory accesses and loop unrollings, the termination point and thus the size of each specialization trace varies, although the traces always start at the dispatch point.

At the end of each trace, control usually reverts back to the corresponding point in the original code; however, specialized traces are linked together with direct jumps where appropriate, avoiding the dispatcher.

Finding dispatch points. The key to identifying where to specialize is finding a dispatch instruction whose hot values most *influence* the specialization procedure, by enabling the specializer to do the greatest amount of beneficial constant propagation and loop unrolling. We have found that a simple property that approximates a forward dynamic slice for a basic block in linear time, detailed in Section 4.1, is generally sufficient to identify the most influential instructions accurately. Once a select few candidate instructions have been identified, each one is tentatively specialized to a limited degree, and the most promising one is selected as the dispatch point and fully specialized. Our measurements show that this technique is fast enough for a runtime environment.

Invalidation. Since the specializer cannot be sure that the memory locations it assumes are constant will not change, it must ensure that if these locations are updated, specialization traces that rely on them are invalidated. Our specializer employs a write barrier-based invalidation detection mechanism that uses language properties to greatly reduce the number of barriers to be inserted. The specialized traces are constructed in such a way that even if an invalidation occurs while the specialized code is being executed, control immediately reverts to unspecialized code. We describe this system and several alternative detection strategies in Section 5.

Implementation. The entire specialization process runs completely independently from program execution to execution. During a given execution, profiles are continuously gathered. When a method becomes suitably hot, it is specialized. If a beneficial dispatch point is found, specialized code is generated and the method is recompiled and re-inserted into the execution stream.

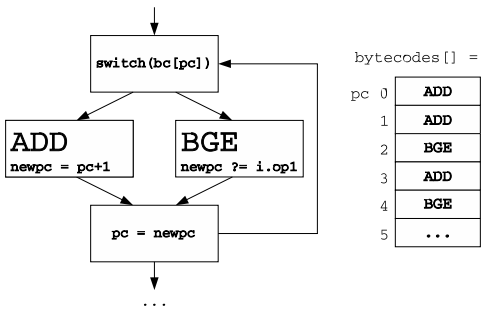

Figure 1: A simple interpreter (left) and its input (right).

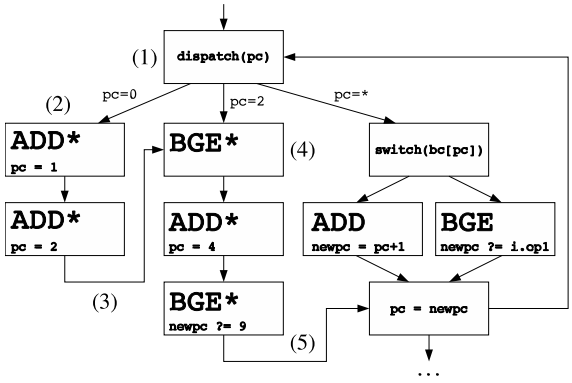

Figure 2: The interpreter in its specialized incarnation. $X*$ indicates a specialized version of basic block X. The leftmost two columns are two specialized traces, while the diamond on the right is the original code.

Our dynamic specializer is implemented in the Jikes RVM Java virtual machine [17]. 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 [3, 18, 4], so we do not address these issues in this paper.

2.1 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. If the first is greater than or equal to the second, it sets *pc* to a prespecified bytecode index; otherwise, *pc* is just incremented. Before specialization, the process of interpreting each bytecode involves seven or eight loads, eight safety checks (null pointer, array bounds), several arithmetic operations, a conditional or store, and a jump.

During execution, the specializer identifies the interpreter as a hot method. The influence algorithm indicates that the assignment to *pc* is a very influential instruction. Since the hot portions of the *interpreted* program happen to be at bytecodes 0 and 2, the hot value profile that has been accumulating data since the start of execution indicates that

0 and 2 are two commonly occurring values of *pc*. The specializer creates two traces, one for each of these values, and a dispatcher (1) that jumps to the corresponding trace or falls back to the original code as appropriate.

Each trace contains optimized straight-line code. Since the store profile indicates that *bytecodes* is constant, the specializer eliminates all loads and safety checks associated with it, and folds many of the constant values. For instance, the first specialized `ADD` block (2) has only two loads and no safety checks. Since the profile is optimistic, the elimination of these instructions is perfectly safe until *bytecodes* is modified; thus the specializer must now monitor it for updates, as described below.

The following `ADD` block is automatically unrolled in-line, since the value of *pc* is known and propagated. Unrolling could continue further, but since there is already a compatible trace being developed for $pc = 2$, the traces are linked together (3), eliminating the need for further code generation or an additional dispatch on subsequent executions.

The second trace begins with a `BGE` (4). It normally would require a conditional jump based on the values of two heap locations, but the profiler identifies these locations as constants. The conditional thus fully resolves to the fall-through block, and unrolling continues. (If it did not, but the result was nevertheless predictable from a basic block profile, unrolling would still continue at the predicted target, but an additional failsafe jump would be added back to the unspecialized code.)

After specializing several more interpreter loops, the propagator either fails to predict the next jump or determines that further code generation would be fruitless, and issues a jump back to the unspecialized code (5). To detect if heap locations that have been assumed constant by the specialization process (such as *bytecodes*) are ever updated, write barriers are inserted at memory store instructions that the type system indicates might update these locations.

The specialized traces are compiled down to machine code and inserted into the execution stream. During the next interpreter loop iteration, if *pc* happens to result in a hot value (which, based on past behavior, should be likely), one of the specialized traces will be invoked.

2.2 Benefits of Dynamic Invariance Detection

As mentioned above, a principal facet of this dynamic specialization approach is the profile-based inference of heap constants. Here we explore this facet in greater detail. Consider the following classes of constants.

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

The more of these constants a specializer can accurately infer, the more of the run-time state it can use in optimization, and the better a job it will do.

Compile-time constants can be optimized by normal static compilers. Run-time constants include, for instance, the bytecodes fed to an interpreter: data that we know will not change, but whose values we can only access at run-time. This class of constants provides the bulk of optimization opportunities for current staged specializers: during their off-line phase, they can annotate these constants and specialize with respect to them at run-time. Dependent 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 prime 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 such tools generally do not have static access to that particular (arbitrary) interpreted program. Identifying these constants enables the interpreted program to be specialized, not just the interpreter.

Consider another example: a database query processor that iterates over the boolean predicates in queries, 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*: 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[1]. 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 – even though not all `Predicate`s passed to it are constant – removing many loads when invoked on them.

Static annotation approaches are unable to capitalize on this situation. 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 `this.LHS.value` is marked constant, would also fail since some of the `Predicate`s passed to `Satisfies` are not constant.

To a dynamic invariance detector, monitoring the invariance of concrete memory locations, there is no difference between a run-time constant and a dependent constant (or a compile-time constant, for that matter); at any given moment, they are all identifiable. 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 to the individual constant fields of modified `Predicate`s can also be eliminated.

3. PROFILING

In this section, we analyze the profiles gathered by the specializer. It requires two main profiles:

[1]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).

- a *hot value profile* that collects the most frequently occurring values at potential dispatch points.
- a *store profile* that optimistically identifies constant memory locations.

Since both of these profiles must be gathered at run-time with low overhead, we used the Arnold-Ryder sampling framework [5]. 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 minimal, around 5%, and yet the resulting profiles tend to be statistically accurate.

3.1 The Hot Value Profile

Hot value profiling has been well studied before, as in Burrows et al. [30], so we do not discuss it in detail here. The hot value profile simply keeps track of the frequency of occurrence of the most popular values resulting from potential dispatch instructions: arguments to loads and functions, and load results. The profile keeps a short array of value/count pairs for each profiled instruction, and employs the "Top N Value" method shown in Calder et al. [10].

3.2 The Store Profile

The goal of the store profile is to indicate to the specializer which memory locations are constant. Ideally, the specializer would like to know for each address a

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

If so, then a is assumed by the profile to remain variant in the future; if not, then it has only been initialized so far and we can consider it constant.

However, since we are employing a sampling-based profiler to keep overhead low, we cannot track every store; instead, we only monitor one out of every 1000 or so. Given this restriction, the store profile just remembers the set of addresses it has seen updated, and we employ the following simple approximation to $var(a)$:

$w(a)$: has any store to a been observed by the profiler?

If so, then a has almost certainly *not* been constant, since it must have been updated enough to be detected by the profiler (or just very unlucky); if not, then we make the optimistic assumption that a is indeed constant.

Implementation. The store profile monitors the addresses of all memory updates via a hash table. 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.

Evaluation of precision. This sampling-based optimistic approach to invariance detection has the benefit of being extremely simple to implement. However, a possible downside is that it might be too optimistic, marking many objects as constant that in fact get updated later on. These incorrect assumptions can lead to an excessive number of invalidations, rendering the specialization process ineffective.

Benchmark	Store	Hot Value	Both
`convolve`	-2.8%	2.6%	2.8%
`dotproduct`	2.1%	15.0%	16.3%
`interpreter`	2.8%	4.5%	6.9%
`jscheme`	1.6%	12.7%	13.1%
`query`	6.3%	15.8%	18.2%
`sim8085`	3.2%	11.2%	19.2%

Figure 3: Steady-state execution time overhead of profiling benchmark applications in percent run-time slowdown. The sampling interval was 1000.

Benchmark	Store	Hot Value	Both
`convolve`	7.2%	22.7%	30.4%
`dotproduct`	4.5%	30.3%	31.1%
`interpreter`	7.6%	26.2%	30.2%
`jscheme`	2.3%	27.0%	29.8%
`query`	2.2%	10.8%	13.1%
`sim8085`	3.7%	29.8%	39.1%

Figure 4: Steady-state memory overhead of profiling benchmark applications in percent increase in average heap size. The sampling interval was 1000.

To assess the accuracy of the store profile, we devised an experiment to determine the fraction of constants claimed by the profile that actually turned out to be constant for the remainder of program execution.

Let L be the set of *all* memory locations read by the program up through when the specializer consulted the store profile as it attempted to resolve loads during specialization. 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 $\dot{S}$ be the set of locations detected as written to by the store profile up to this point. $\dot{C} := L - \dot{S}$, then, is the set of memory locations the store profile reported as constant.

Let W be the set of *all* existing locations written to from this point to the end of program execution. $C := L - W$ is then the set of locations that actually were constant from specialization onward.

We define the accuracy of the store profile as $|C \cap \dot{C}|/|\dot{C}|$: the fraction of claimed constants that really were never modified by the end of the program.[2]

We instrumented 12 Java programs to compute this value. The accuracy was 95.6%: that percentage of locations in $\dot{C}$, the set of all claimed constants, remained constant, indicating that (1) locations that start out constant overwhelmingly tend to remain constant, and (2) variable locations tend to get updated frequently enough to be observed by the profiler. We feel that these observations provide a reasonable basis to employ the store profile, as few invalidations should occur as a result of its predictions. Of course, it is still possible for particular programs to manipulate memory in such a way as to deceive the profiler.

3.3 Profiling Overhead

The steady-state overhead of employing our hot value and store profiles on a set of benchmarks is detailed in Figures 3 and 4. For a description of the benchmarks, see Section 6.

[2]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 [29]. Thus this definition under-approximates the true accuracy of the store profile.

The store profile has reasonable runtime overheads of around 5%. 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 function calls rather than being inlined. Previous work by Burrows et al. [30] has 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% [47]. The slowness of the hot value profile 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.

In addition to providing a better profiler implementation, a number of explicit strategies could be used to further decrease this overhead. To reduce execution time, one technique we adopted in our implementation is to adaptively increase the sampling interval for those programs for which specialization does not seem likely (say, after several failed specialization attempts). This technique generally keeps overheads under 10%; while we did not employ it for the benchmarks in this section, we did for the overall evaluation (Section 6). Furthermore, a hardware-based profiling approach, or one that performs a more sophisticated placement of sampling checks, should also decrease execution overhead. Next, to decrease the memory overhead in an adaptive compilation system, hot value profiles could be collected only for functions already identified as hot and scheduled for recompilation, since those functions generally are the best candidates for specialization. (The bulk of memory overhead is due to hot value profile arrays, so reducing the set of hot value profiled functions should proportionately reduce the memory overhead as well.)

In addition to these two main profiles, we gather basic block execution frequencies. This profile is trivial to gather in most virtual machines, and our sampling based implementation has negligible performance and memory overhead when run in conjunction with the other two profiles. From this profile, we use a simple algorithm, omitted here, to approximate an edge profile that is used during branch prediction and by the dispatch point-finding algorithm described in Section 4.1.

4. DYNAMIC SPECIALIZATION

In this section, we describe the mechanisms used to create specialized traces, and then use these mechanisms to motivate an effective yet costly dispatch point selection algorithm and a practical approximation of it. Finally, we describe key implementation details.

Assume we have identified a suitable dispatch point instruction p 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 p, with the assumption that p resulted in v. The procedure uses a simple benefit analysis, presented in detail in Section 4.2, 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 [8]. 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 p is reached; assume that p 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 loop headers. Constant propagation is seeded with the initial hot value, and utilizes profile-inferred constant locations in memory; see Section 2. Loop backedges (backward branches) are followed and further iterations are unrolled, so that each loop iteration is specialized.

A specialization 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 ex-

ecution 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.

4.1 Identifying Dispatch Points

Given a hot function, we would like to identify the best dispatch point from which to specialize. A simple algorithm for doing so is to simulate the specialization procedure on each instruction in the function (using that instruction's hottest values to seed traces) without generating any actual code, and choose the one that results in the greatest net benefit.

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.

However, ascertaining the best dispatch instruction without even speculatively specializing on it is difficult, since a dispatch's benefit depends on the number of concrete heap constants its specialized region addresses — a number that is heavily dependent on the particular seed hot values fed to the constant propagator and the subsequent operations on them.

Thus, our specializer considers a small number of candidate instructions (in our implementation, five), speculatively 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 places the best dispatch instructions among this candidate set. We tried several simple heuristics.

Hotness. Instructions are ordered by a combination of execution frequency and hot value consistency (simply the sum of the profiled number of occurrences of its top ten hot values), since the specialization traces of these instructions will be executed very frequently.

This approach fails because it does not accurately predict the cumulative effect of an instruction on subsequent instructions. For instance, instructions deep inside loops often appear to be good candidates by this metric, whereas dispatching on an instruction outside the loop is often more beneficial.

Domination. Instructions are ordered by the number of instructions in the control-flow graph they dominate; 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 instructions. For instance, loop variable updates (e.g. `i++`) can be good specialization candidates but generally are not dominators.

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: simply try 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.

The `pc = newpc` instruction in Figure 1 would not be ranked highly by either of these heuristics, even though it is very influential to the execution of the program.

All of these heuristics are "brittle" in the sense that they only find good dispatch points if they are provided with functions with a particularstatic 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 p and a value v, the forward dynamic slice is the set of dynamic instructions affected when the value computed by p is v; see Tip [44] for a good summary.

The motivation behind using slices is that instructions that contribute to v's specialization trace's benefit are from p's forward dynamic slice: the ones that happen to resolve to loads of constant values as a result of p evaluating to v. Thus, computing a forward dynamic slice for each of an instruction's hot values yields a reliable over-approximation of the potential benefit of specializing on that instruction.

However, computing such a slice is very costly; even the most efficient algorithms require extensive preprocessing [46], a luxury we cannot afford in a purely runtime environment. Since regardless of the metric we use, we plan to speculatively specialize its top few dispatch point candidates anyway, we are more than willing to trade off the precision of a slicing algorithm for an *efficient* approximation that can consistently suggest the best candidates.

Thus we make a key simplifying assumption: we expand the definition of forward slice to include all of the instructions in CFG that dynamically follow instruction p. This lets us approximate a dynamic slice without regard to particular values or dataflow behavior: we simply need to compute the number of dynamic instructions that follow p. Additionally, we compute the same slice for all instructions in the same basic block.

Formally, consider a function f's control-flow graph accompanied by 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 simply a sequence of edges $e_1e_2...$ from the start of the function to the end. Let $count(x)$ be the dynamic execution count of graph component x. Assuming branch independence, each trace t can be assigned a probability of occurrence, by taking the product of the probabilities of its branch choices:

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

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

where $count(a)$ is a's dynamic execution count. Together, these traces and their probabilities constitute f's *expected execution set*, or *EES*.

We define the *influence* of a basic block b with respect to

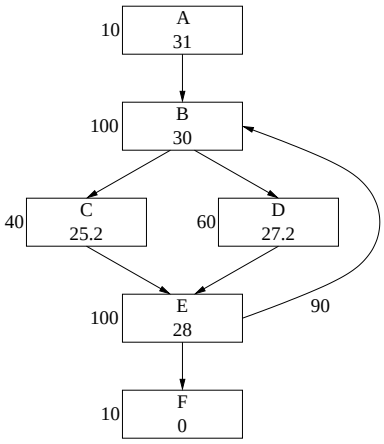

Figure 5: Basic block influences numbers for a sample control flow graph. The number to the left of each block is its dynamic execution count, and the number inside is the influence.

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

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

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

Figure 6: Equations for computing the influence of a basic block b as the solution of two systems of linear equations. *start* is the function's entry basic block.

a particular trace t as the length of the path from the first occurrence of b along t until the end of the function:

$$influence_t(b) \equiv length(suffix(t, b)) \quad (3)$$

where $suffix(t, b)$ means the portion of trace t from the first instance of b to the end.[3]

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

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

See Figure 5 for a sample influence computation. The influence of 30 for B means that an average of 30 basic blocks follow the first execution of B on a given function invocation. Note that the influence of basic block 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* basic blocks 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 naive computation of influence from Equation 4 infeasible because the EES can be infinite in size.

[3]Throughout this discussion, we assume for clarity that basic blocks are each of unit size. The notion of influence is trivially generalized in our implementation to accurately compute influence for basic blocks of different sizes.

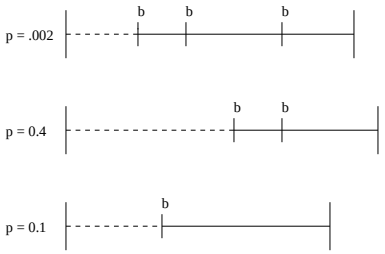

Figure 7: Some sample traces. The solid lines to the right of each leftmost b count toward b's influence, weighted by each trace's probability of execution. We can also compute influence by viewing each trace as a series of shorter paths between individual executions of b.

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

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

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

Figure 8: Alternate equations for computing the influence of a basic block b as the solution to a single system of linear equations. *start* is the function's entry basic block.

Instead, we can recast the influence of b as the combination of two problems: the probability of ever reaching b during an invocation of f, and the expected length from b to the end of the function once b is reached (Figure 6). Equations 5 and 6 are both recursive definitions that produce systems of linear equations.

We can use dynamic execution count data to simplify the problem further, to solving just one system of linear equations, by directly computing the expected number of times b is executed *per invocation* of f. We can then view each trace as a series of shorter paths starting from b, each of which terminates when it reaches b again or (in the case of the last short path) the end of the function (Figure 7), and define a system of equations to compute the expected length of such a path (Figure 8). The influence of b is the product of these two expectations.

The systems of equations in Figures 6 and 8 can be solved via Ramalingam's data flow frequency analysis framework [35]. 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 [43] 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 somewhat more complicated but truly linear 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

Benchmark	Total	Hot	Dom	First-n	Inf
`convolve`, image	39	34	1	1	1
`convolve`, kernel	39	33	2	2	2
`dotproduct`	8	5	1	1	1
`interpreter`, both	52	3	46	52	4
`jscheme`	29	26	1	1	1
`query`	20	12	1	1	1
`sim8085`	86	3	31	21	3

Figure 9: 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.

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.

For actual Java programs, influence appears to succeed at identifying good dispatch instructions both accurately and consistently; see Figure 9.

4.2 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.

Consider a trace t of a hot value v, at a dispatch point p. 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 specialization 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 only find 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 specialization region, can produce traces of different lengths for different hot values, making each one only as long as it needs to be.

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 [19]. 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.

5. INVALIDATION

Runtime execution profiles are no guarantee of future behavior; if a memory location that the store profile claims is constant gets updated, any specialization 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.

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 specialization 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.

Benchmark	Constant Memory Locations	Barriers Inserted	Steady-State Overhead
`convolve`, image	153	72	9%
`convolve`, kernel	9	5	3%
`dotproduct`	102	5	-1%
`interpreter`, bsort	50	14	6%
`interpreter`, search	58	10	1%
`jscheme`	482	4	3%
`query`	84	21	2%
`sim8085`	25	8	12%

Figure 10: Invalidation detection information and overhead. See Figure 11 for descriptions of the benchmarks. The steady-state overhead compares the a specialized program with invalidation detection against the equivalent specialized program without it.

Unlike C, 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[4]. 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 10 for a table of the number of write barriers that had to be inserted in the benchmarks presented in this paper.

Reducing the cost of executing the write barriers. Our write barrier implementation uses a bit in the Jikes RVM object header. An object's bit is set by the specializer when it makes the assumption that a field in that object is invariant. On writes to that field, the bit of the enclosing object is tested and if it is set, an invalidation occurs. For arrays, an additional check is made to determine if the particular index that is being updated matches one that is considered invariant.

For updates to static fields, for which there is no enclosing object, the specializer inserts a hash table lookup before the store that identifies if the address to be modified, a, points to any of the locations that the specializer assumes are invariant. By itself, this lookup requires several arithmetic computations and memory loads, and can interfere with cache locality. In practice, we found that the barrier overhead was too great.

To rectify this, the specializer creates a bitmask summary of all assumed-invariant static addresses, as in the Diduce system [23], and inserts code to check a against this mask before executing the lookup. The Diduce masking procedure is conservative, so if a does not match the mask, it cannot write to any invariant addresses, and the hash table lookup can be avoided. Unfortunately, a single mask tends to offer insufficient protection: as it is widened to handle all of the invariant static addresses, it generally matches against many other static addresses as well. To allay this problem, the specializer employs multiple masks. When a new invariant static address is discovered, the closest matching mask is widened to accommodate it. (Often no widening is needed at all, as invariant memory locations tend to be locally bunched, even if the set of addresses as a whole is diverse.) Tests against all of the masks are inserted at the barrier; if a matches none of them, then the lookup can be skipped. We found that using three masks works well in practice.

See Figure 10 for performance numbers. The detection mechanism executed with steady-state overheads of under 15%, which we feel is suitable for a runtime environment, given the overall specialization speedups shown in Section 6.

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 for handling 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 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

[4]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.

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 specialization 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 employed if the store profile indicates that these other fields are written to infrequently, or if there are only a few of them.[5]

Mondrian hardware support. Perhaps the most natural solution is to use hardware. Witchel et al. [45] 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 11), 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 [15].

[5] An additional caveat of using a copying collector with a specializer is that the collector needs to be told about constant memory addresses embedded in specialized code, so that it can update them as it moves objects between semispaces.

6. 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 512MB RAM running Red Hat Linux 9. For the specialization runs, the profiling code was sampled using the full duplication variation the Arnold-Ryder instrumentation sampling framework [5], with a sampling interval of 1000.

The benchmark results are presented in Figure 11.

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.

The execution times for the specialized runs also encompass all specialization costs, including profiling overhead, specialization and recompilation times, and all aspects of invalidation, including write barrier insertion and runtime overhead.

Each benchmark was run in a short trial, to assess the impact of the specialization overhead, and a long trial, to get a sense of the asymptotic speedup; while the time measurements include all overheads, we also present the specialization time separately in the table. We have evaluated many of the system's components in previous sections; here 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 [20]. 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 the 75% of the iterations in which the constant vector's element was zero outright. 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 3, seemed to be quickly outweighed by the much more significant speedups due to specialized code.

To warrant inclusion in a dynamic optimization system's arsenal of optimizations, the specializer should behave well on *all* programs. We ran it on `em3d`, with input parameters that made the program a bad candidate for specialization: we had it create a very large number of data objects that were visited equally often, thus rendering specializing on a small number of them ineffective. The specializer attempted and aborted three specializations, and after each failure it doubled the sampling interval. As a result, the overall slowdown was 4%. This percentage is representative: we ran

Benchmark	Input	Speedup		Specialization Time
		Short run (unspec/spec)	Long run (unspec/spec)	
`convolve` Transforms an image with a matrix; taken from the `ImageJ` toolkit	fixed image, various matrices	**2.30x** 62s/27s	**2.73x** 618s/226s	0.9s
	various images, fixed matrix	**1.15x** 62s/54s	**1.18x** 621s/526s	0.2s
`dotproduct` Converted from C version used in DyC [20]	sparse constant vector	**3.80x** 57s/15s	**4.75x** 563s/119s	0.2s
`interpreter` Interprets simple bytecodes	bubblesort bytecodes	**4.21x** 59s/14s	**5.72x** 589s/103s	0.5s
	binary search bytecodes	**4.58x** 55s/12s	**6.26x** 545s/87s	0.6s
`jscheme` Interprets Scheme code	partial evaluator	**1.69x** 59s/35s	**1.80x** 580s/322s	1.6s
`query` Performs a database query; from DyC	semi-invariant query	**1.60x** 56s/35s	**1.71x** 544s/318s	0.3s
`sim8085` Intel 8085 Microprocessor simulator	included sample program	**1.43x** 60s/42s	**1.46x** 595s/407s	0.4s
`em3d` (intentionally unspecializable) Electromagnetic wave propagation	-n 10000 -d 100	**0.96x** 65s/68s	**0.98x** 525s/534s	0.2s

Figure 11: Dynamic specialization speedups and costs. Specialization time details the total time taken during specialization, including the time needed to insert write barriers for invalidation. The time for `em3d`, a representative unspecializable program, represents three failed specialization attempts.

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 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 exposed a large optimization opportunity, and while the second — 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 15%. Existing staged specializers, limited to annotating the function in just one way, would be unable to specialize on both of these usage patterns.[6]

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

Third, we analyzed the memory behavior of `interpreter` running bubblesort to track dependent 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 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 be essentially zero.

7. RELATED WORK

Specialization. Program specialization is a well-studied optimization technique [13, 40, 39, 12, 25, 2, 26, 16, 20, 7, 21, 33, 28]. In its classical form, code is optimized in a source-to-source transformation. Tempo [12], DyC [20], and others used code templates to generate specialized code at runtime once constant values are known. However, they relied on programmer annotations to specify specialization regions and constant memory locations. Calpa [32] 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,

[6] 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.

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. [41] 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%.

Dynamic optimization. The profile-and-optimize dynamic approach described in this paper is similar to other transparent dynamic optimization systems, like Mojo [9], Hotspot [31], and others. In particular, our specializer leverages the Jikes RVM framework [17, 4] for recompiling specialized methods.

Profiling. The efficiency of our profilers rests on the Arnold-Ryder sampling framework [5]; we use it to employ a novel invariance detection profile. We use a method suggested by Calder et al. [10] for gathering hot value data. The use of optimistic assumptions to motivate dynamic optimization is presented in Arnold and Ryder [6].

Trace creation. The main specialization algorithm's trace creation procedure draws from on-line partial evaluation techniques [36], and was inspired by Dynamo [8], although Dynamo does not use heap invariants or unroll loops when optimizing, and only produces one optimized trace per program point. Sullivan et al. [42] 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 [27].

Invalidation. As far as we know, this paper presents the first implementation and evaluation of a working automatic detection and invalidation system. Calpa [32] 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 [20] supports manually-triggered invalidations, but does not provide a mechanism for actually performing the invalidation on a running specialization 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 [34]. We use the bitmasking techniques employed by Diduce [23] to reduce write barrier overhead. The actual invalidation is similar in end result to on-stack replacement (OSR) techniques [18, 11]. 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 [45].

To the best of our knowledge, this is the first implementation of a transparent runtime specializer that uses heap data. While Sastry [38] 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.

8. 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 6.3x on a variety of benchmarks.

9. 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] S. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement, 2003.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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).

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

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

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

1992.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

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

[45] 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.

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

[47] 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 a basic block n that is linear in the number of basic blocks in the graph. We exploit the fact that the graph is reducible to precompute closed-form summaries of the needed expectation properties for loops.

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, say 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.

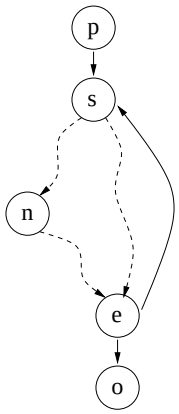

Figure 12: A loop. The dotted edges represent arbitrary acyclic control flow.

The difficulty arises if n is in a loop. Consider a standalone loop, as in Figure 12. 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.)

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

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

To compute $F(n)$ and $L(n)$, we first need to describe some properties of the loop. See Figure 12 for a sample loop. s is the loop's entry node, and e is the loop's exit node.

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

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

f(n). Probability of reaching n from s on *one* arbitrary loop iteration (i.e. without reaching s again). Computed like l above.

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

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

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

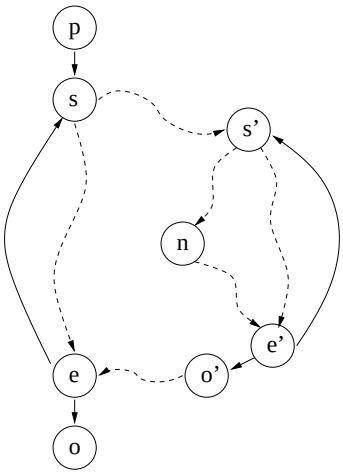

Figure 13: Nested loops. The inner loop variables have been primed.

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 13. 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 these properties have already been computed.

Computing the influence of nodes that are in the outer loop but not in the innner 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.:

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

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

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

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

$$F(s')L(s') \quad = \quad \frac{f(s')}{1-(1-f(s'))b'}(r(s') + \frac{l'b'}{1-b'}) \tag{19}$$

$$= \quad \frac{1}{1}(l' + \frac{l'b'}{1-b'}) \tag{20}$$

$$= \quad \frac{l'}{1-b'} \tag{21}$$

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

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

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

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

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

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

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

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