JOLT: Lightweight Dynamic Analysis and Removal of Object Churn

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

Matthew Arnold
IBM T.J. Watson Research Center
marnold@us.ibm.com

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

Abstract

It has been observed that component-based applications exhibit *object churn*, the excessive creation of short-lived objects, often caused by trading performance for modularity. Because churned objects are short-lived, they appear to be good candidates for stack allocation. Unfortunately, most churned objects escape their allocating function, making escape analysis ineffective.

We reduce object churn with three contributions. First, we formalize two measures of churn, *capture* and *control* (15). Second, we develop lightweight dynamic analyses for measuring both *capture* and *control*. Third, we develop an algorithm that uses *capture* and *control* to inline portions of the call graph to make churned objects non-escaping, enabling churn optimization via escape analysis.

JOLT is a lightweight dynamic churn optimizer that uses our algorithms. We embedded JOLT in the JIT compiler of the IBM J9 commercial JVM, and evaluated JOLT on large application frameworks, including Eclipse and JBoss. We found that JOLT eliminates over 4 times as many allocations as a state-of-the-art escape analysis alone.

Categories and Subject Descriptors D.3.4 Processors [*Programming Languages*]: Optimization

General Terms Algorithms, Performance

Keywords Churn, allocation optimization, Java, virtual machine, selective optimization, escape analysis, inlining

1. Introduction

Large-scale applications are often built on application frameworks, such a Sun's J2EE and IBM's Eclipse. These frameworks employ many reusable components and third-party libraries. To allow composition with other components in an application, components typically provide a simple, general interface, which often necessitates construction of many intermediate objects that are quickly discarded. This phenomenon is known as *object churn* (15; 23).

Object churn is harmful for several reasons. First, it puts pressure on the garbage collector. Second, if temporary objects appear to require synchronization, it inhibits parallelization. Third, construction of temporary objects may require unjustified computational overhead, as fields in these overly general objects might be used only once or not at all.

Churn-inducing code can be optimized manually, for example with code refactoring. Dufour *et al*. (15) developed a hybrid static/dynamic analyzer that uses notions of object *capture* and *control* to guide programmers to problematic program fragments. To eliminate the identified churn, a programmer may need to change the data structure representation. Because churn is pervasive (23), such refactoring may involve several components, which may not be economical. Churn may also involve libraries for which source code is not available, preventing refactoring.

This paper explores automatic reduction of object churn as might be performed in a JIT compiler. On first sight, the problem of object churn removal appears to be solved: escape analysis (10; 12; 29) identifies objects that can be stack allocated, or even promoted to registers via scalar replacement, and subsequent dead value analysis may be able to remove useless object fields; existing JIT compilers contain advanced implementations of these techniques.

However, we observed that object allocation optimizations based on escape analysis do not perform well on component-based applications, largely because many short-lived objects escape their allocating functions. In a typical scenario, objects are passed up the stack via object factories or functions that compute intermediate values. While the escape analyses reported in literature usually eliminate over 20% of allocations on test programs (10; 12; 29) , our measurements on large component-based applications indicate that even a sophisticated escape analysis performed on function boundaries in a high-performing commercial JVM generally eliminates fewer than 10% of allocations (see Section 6).

Another factor limiting the effectiveness of escape analysis is the stringent budget available to the JIT compiler. The VM must select methods for compilation and choose which optimizations to apply to each method. A common heuris-

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

tic used by profile-guided compilers is to select optimizations based on execution frequency (or "hotness"), but such a heuristic may overlook many of the tens of thousands of functions in a component-based applications that exhibit object churn but are not very hot. In addition, the hottest methods are not necessarily involved in object churn. Excessive inlining, which improves churn elimination, can negatively impact performance (8), particularly if applied unjudiciously in large framework-based applications.

Figure 1 illustrates these concerns with an example of a standard library function used in many business applications. The code exhibits typical object churn: objects created in functions called by `divide` are immediately used in `divide` and subsequently discarded. Since these functions are not often among the hottest functions in a program, escape analysis may not be performed on them. Even if it was, the analysis would find that the objects escape their allocating functions. This problem can be alleviated with inlining, but we observed that several of the called functions are overlooked by a standard inliner, because it considers them too large and is unaware of the potentially large stack allocation benefit that could result.

In summary, there are two main obstacles to the elimination of object churn from component-based applications using a JIT compiler:

1. The JIT lacks the budget to perform allocation optimizations on all functions and, conversely, sufficient information to determine which subset of functions would benefit from allocation optimizations.
2. Short-lived objects often escape their allocating functions, defeating traditional escape analysis.

JOLT, the system described in this paper, deals with these two issues by (a) using a lightweight dynamic analysis to identify *churn scopes*, which are subgraphs of the call graph with significant churn; and (b) performing selective function inlining to make allocation sites in churn scopes amenable to traditional allocation optimizations.

Our solution for short-lived escaping objects is motivated by the empirical observation of Dufour *et al.* (15) that many escaping objects are eventually contained. Consistently across large programs, over 50% of objects did not escape from their allocating method or one of its three ancestors in the call chain at the time of object allocation. The churn scopes computed by JOLT encapsulate live ranges of short-lived objects. The JIT compiler's inliner is instructed to inline methods in the churn scope, making them visible to escape analysis. With this profile guidance, the JIT's optimizer can better select and optimize targets from among the thousands moderately hot functions that typically exist in a large-scale application.

JOLT optimizes the code in Figure 1 in three steps. First, the JOLT dynamic analysis detects that `divide()` captures many objects and hence makes `divide()` the start of a churn scope, meaning that some subgraph (of the call graph) rooted at `divide()` may be worth optimizing. Second, JOLT identifies what subgraph that is. Although `divide()` makes many calls, JOLT's inliner, still observing an inlining budget, will inline into `divide()` those calls that aid in eliminating churn. These inlined calls become part of the churn scope. Finally, JOLT invokes the escape analysis on this scope, even if the `divide` function were not hot enough to justify this level of optimization using traditional JIT heuristics.

This paper makes the following contributions:

- We formalized two measures of churn locality, *capture* and *control* (15). Together, they form the basis for identifying areas of optimizeable object churn (Section 2).
- We developed efficient algorithms for measuring *capture* and *control* at run time. The algorithms are designed for a VM with contiguous memory allocations and thread-local heaps (Section 3), a common configuration in modern VMs.
- We developed an alternate algorithm for approximating these measures in any VM with a garbage collector (Appendix A).
- We developed an inlining algorithm that uses the results of these churn analyses to expose profitable allocation contexts, called *churn scopes*, to standard JIT compiler allocation optimizations (Section 4).
- We implemented JOLT for Java in a development version IBM's J9 JVM (Section 5). Our evaluation of this system on several large-scale applications (Section 6) shows that our transformation increases 4-fold the rate of object stack allocation.

2. Capture and Control Analysis

The ultimate goal of our dynamic analysis is to compute *churn scopes*, subgraphs of the static call graph that encapsulate lifetimes of many objects; the functions comprising each churn scope are inlined and handed over to escape analysis. Clearly, the program as a whole represents a churn scope that maximizes contained object lifetimes; it is, however, usually too large too large to inline and optimize. Therefore, in Section 4 we give heuristics that seek to balance the amount of encapsulated lifetimes and the scope size.

In this section we take the first step towards computing churn scopes. We formalize *capture* and *control*, two program properties that we use to compute churn scopes in Section 4. We motivate capture and control with how one would compute churn scopes. In this section we assume that churn scopes are rooted at some call graph node and contain all nodes reachable from this root; this restriction is relaxed in Section 4. This section first defines capture and control for a dynamic call tree, and then does so for the static call graph.

2.1 Definitions

Consider an execution trace E of a program P composed of a set of functions F. The trace E is a sequence of events corresponding to the start and end points of function invo-

```
public BigDecimal divide(BigDecimal val, int newScale, int roundingMode)
         throws ArithmeticException, IllegalArgumentException {
  ...
  valIntVal = valIntVal.multiply(BigInteger.valueOf(10).pow(-power));              // 3 objects
  ...
  BigInteger dividend = intVal.multiply(BigInteger.valueOf(10).pow(power));        // 3 objects
  BigInteger parts[] = dividend.divideAndRemainder (valIntVal);                    // 2 objects
  ...
  BigInteger posRemainder = parts[1].signum() < 0 ? parts[1].negate() : parts[1];  // 1 object
  valIntVal = valIntVal.signum() < 0 ? valIntVal.negate () : valIntVal;            // 1 object
  int half = posRemainder.shiftLeft(1).compareTo(valIntVal);                       // 1 object
  ...
  // (valueOf most likely using constant here)
  unrounded = unrounded.add (BigInteger.valueOf (sign > 0 ? 1 : -1));              // 1 object
  ...
  return new BigDecimal (unrounded, newScale);
}
```

Figure 1. An example of object churn from the the GNU Classpath implementation of the standard Java Class Library method `BigDecimal.divide()`. The calls shown here each return either `BigDecimal` or `BigInteger` objects, which are immediately used and then discarded. The number of churned object allocations is shown next to each call.

cations and object lifetimes. The events are identified with unique timestamps. For each function $f \in F$, let f_i denote the ith invocation of f in E. Let $start(f_i)$ denote the beginning of the ith invocation of f_i and $end(f_i)$ the time when the function returns. For an object o allocated in E, $start(o)$ gives the time of o's allocation and $end(o)$ marks the end of o's lifetime. Our implementation conservatively defines the end of the lifetime as the time at which o becomes unreachable.

A function invocation c_j is a child of invocation f_i if f_i directly invokes c_j. The set $children(f_i)$ gives all children of f_i. We have

$$c_j \in children(f_i) \Rightarrow \\ start(f_i) < start(c_j) < end(c_j) < end(f_i)$$

Child invocations cannot overlap, so we have

$$c_j, c_k \in children(f_i) \wedge j \neq k \Rightarrow \\ end(c_j) < start(c_k) \vee end(c_k) < start(c_j)$$

The $children$ relation on invocations defines the dynamic call tree. The static counterpart, defined over functions in F, defines the static call graph: a function c is a *child* of $f \in F$ if there are i, j such that f_i invokes c_j.

We define $alloc(f_i)$ as the set of all objects allocated by f_i and its descendents. We define $escape(f_i)$ as the subset of $alloc(f_i)$ composed of objects that escape f_i:

$$alloc(f_i) := \{o \,|\, start(f_i) < start(o) < end(f_i)\}$$
$$escape(f_i) := \\ \{o \,|\, start(f_i) < start(o) < end(f_i) < end(o)\}$$

A *dynamic churn scope* of the function invocation f_i is a subtree of the dynamic call tree rooted at the node f_i; the churn scope contains all nodes reachable from node f_i. A *static churn scope* of a function f is a subgraph of the call graph rooted at the node f that contains all nodes reachable from the node f.

2.2 Capture

Our first property uses the notion of *capture* introduced by Dufour, *et al.* (15). We define the *capture* of a function invocation f_i as the set of all objects allocated by f_i and its descendents whose lifetimes end before f_i returns.

$$capture(f_i) := alloc(f_i) \setminus escape(f_i)$$

The *capture* of a function invocation f_i is an indicator of object churn: if f_i or its descendents create many short-lived objects, these objects are likely captured by f_i. In the context of the optimization that we have in mind, $capture(f_i)$ gives an upper bound on the number of object allocations that can be eliminated if the optimization selects f_i as the root of a churn scope.

Though f_i's scope may have significant optimizeable object churn, we want to balance optimization benefit with its cost, influenced primarily by the size of the scope. First, we are interested in whether choosing an ancestor of f_i might expose even more churn. Second, we want to determine whether there is a smaller scope with comparable optimization benefit. Such a scope may be rooted at one of f_i's children.

The second question is answered in the following subsection. To answer the first question, we use the capture rate, *%capture*, which normalizes capture to the number of object

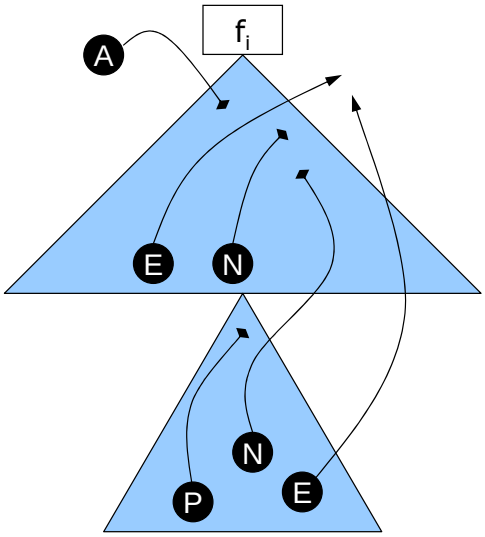

Figure 2. Object lifetime behavior of a function invocation f_i and one of its child calls. The circles denote allocation events. The objects marked E, P, and N were are allocated by f_i or its child. The objects marked E remain reachable past the exit of f_i and are considered *escaped*. The object marked P is unreachable at f_i's exit and is considered *captured*. The objects marked N are also captured but have an additional property: they are live during f_i's execution, not just during its descendants' execution, and are thus considered *controlled* by f_i. (The object marked A was live before f_i's execution but is unreachable after it; it is considered *absorbed* and is relevant to Section A.3.)

allocations:

$$\%capture(f_i) := \frac{|capture(f_i)|}{|alloc(f_i)|}$$

The higher the value of $\%capture(f_i)$, the less attractive it is to grow the scope to the parent of f_i, because most objects are already captured by f_i.

2.3 Control

To determine whether it may be more profitable to root the scope at a child of f_i, rather than at f_i, we formalize *control*, which indicates the level of "object encapsulation" provided by f_i. We define $control(f_i)$ as those objects allocated by f_i and its descendents whose lifetimes end in f_i but not in any of its children. The objects controlled by f_i include in particular objects that escape the immediate children of f_i but do not escape f_i. The invocation f_i controls these objects in the sense that it uses them in its computation (including passing these objects among its children), but they do not survive past its return. Control is an indicator of churn: If f_i controls few of the objects that it captures, it may not be a profitable root of a churn scope.

The set of objects controlled by f_i is defined as follows.

$$control(f_i) := capture(f_i) \setminus \bigcup_{c_j \in children(f_i)} capture(c_j)$$

We also define the control rate, denoted *%control*:

$$\%control(f_i) := \frac{|control(f_i)|}{|alloc(f_i)|}$$

The control rate can be used to identify suitable churn scopes. Imagine that we are looking for a suitable churn scope by moving the churn scope root down the call graph. In this case, the control rate *%control* acts as a sentinel: a high value of $\%control(f_i)$ suggests that shrinking the scope from f_i to one of f_i's children would deprive a significant number of objects controlled by f_i of the context that bounds their lifetimes. The control rate thus complements *%capture*, which acts as a sentinel when moving the root upwards (we may want to place the root high enough to capture most of the allocated objects). For a diagram of *control* and *capture*, see Figure 2.

2.4 Aggregating Dynamic Information

We have defined *capture* and *control* for dynamic function invocations, whose $children$ relation defines the dynamic call tree. Since JOLT needs to select scopes on the static call graph, we also define *capture* and *control* for static functions by aggregating the dynamic values. Note that *capture* and *control* for function invocations range over sets of objects. To enable statistical aggregation, we define their static counterparts, $\overline{capture}(f)$ and $\overline{control}(f)$, as the mean over the cardinalities of these sets. (Instead of comptuing set cardinalities, we sometimes compute the aggregate memory footprint of these objects.) Given a function f with invocations $f_1, ..., f_n$, we define $\overline{capture}(f)$ and $\overline{\%capture}(f)$ as

$$\begin{aligned} \overline{capture}(f) &:= \frac{1}{n}\sum_{i=0}^{n} |capture(f_i)| \\ \overline{\%capture}(f) &:= \frac{1}{n}\sum_{i=0}^{n} \%capture(f_i) \end{aligned}$$

The static values $\overline{control}(f)$ and $\overline{\%control}$ are defined analogously, as the mean of the dynamic values $control(f_i)$ and $\%control(f_i)$, respectively.

$$\begin{aligned} \overline{control}(f) &:= \frac{1}{n}\sum_{i=0}^{n} |control(f_i)| \\ \overline{\%control}(f) &:= \frac{1}{n}\sum_{i=0}^{n} \%control(f_i) \end{aligned}$$

It may seem that we could save work by computing the static control values from the static capture values, as follows:

$$\overline{control}(f) := \overline{capture}(f) \setminus \bigcup_{c \in children(f)} \overline{capture}(c)$$

where the $children$ relation is given by the static call graph. This formula would avoid computing the dynamic values $control(f_i)$ during profiling. The price, however, is the loss of context sensitivity present in the dynamic values $control(f_i)$. Because the dynamic values are computed on the dynamic call tree, each invocation is analyzed in its calling context. Since $\overline{control}(f)$ is computed from the dynamic control values, it reflects only the behavior of children of f when they are called from f, ignoring them when they are called from other functions.

3. Computing Capture and Control

JOLT computes *capture* and an approximation of *control* with low-level mechanisms typically available in modern virtual machines. Specifically, our dynamic analysis exploits the fact that (i) virtual machines contain a tracing garbage collector; (ii) memory managers often allocate objects contiguously, as in a copying collector; and (iii) heaps are typically thread-local, i.e., they service allocation requests from a single thread. For JVMs that do not meet the assumption of thread-local heaps (TLHs) and a contiguous allocator, we have devised an alternate set of algorithms. These analyses, given in Appendix A, approximate *capture* and *control* with any garbage collector.

In this section, we first show how to compute $alloc(f_i)$ and $escape(f_i)$ with a few simple measurements; as described in Section 2, these values can be used to compute $capture(f_i)$, $\%capture(f_i)$, $\overline{capture}(f)$, and $\overline{\%capture}(f)$. We then describe a difficulty with computing $control(f_i)$ with a garbage collector, and present an approximation of $\overline{control}(f)$ that circumvents this difficulty.

3.1 Computing *alloc* and *escape*

In a JVM with contiguous allocations and thread-local heaps it is straightforward to compute $alloc(f_i)$, the total size of objects allocated by f_i. Each thread has a pointer to the beginning of the free space in its TLH. When a function allocates an object, the free space pointer is incremented by the size of the object, and the intervening space is returned to the function to be used by that object. We use the function $tlhp(f_i, t)$ to denote the value of the free space pointer at time t in the thread executing f_i.

To compute $alloc(f_i)$, we determine by how much the free space pointer has moved between $start(f_i)$ and $end(f_i)$. Since allocations are contiguous and thread-local, this value gives the total size of objects allocated by f_i. The range of addresses of these objects is

$$\langle tlhp(f_i, start(f_i)), tlhp(f_i, end(f_i))\rangle$$

This range is valid if no garbage collection occurred during f_i. If it did, our sampling profiler discards the measurement.

To compute *escape*, JOLT invokes the tracing garbage collector at the end of f_i and passes it the allocation range computed above. As the GC traverses live objects on the heap, it checks whether the object's address falls within the allocation range of f_i. Objects in $alloc(f_i)$ that are still alive at f_i's exit comprise $escape(f_i)$. Figure 3 illustrates this computation.

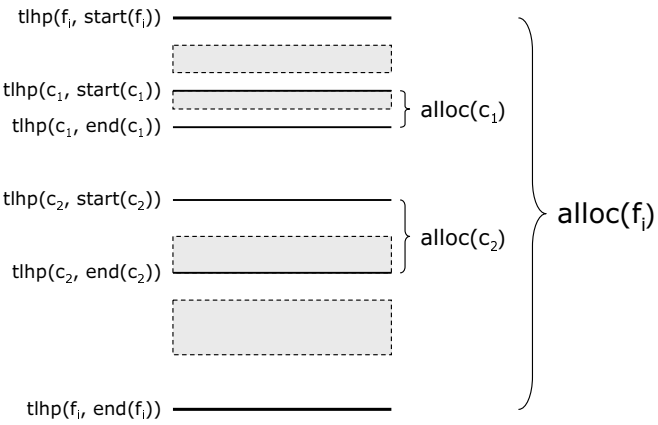

Figure 3. A view of the portion of a thread-local heap relevant to a function invocation f_i, illustrating the information that a thread-local heap allocation pointer provides in computing churn analyses. By tracking the position of this pointer (shown at the left of the heap) as the program execution progresses, JOLT computes *alloc* for the function and each of its children. Furthermore, by running a garbage collection at the end of f_i, JOLT identifies the set of objects that are still live, $escape(f_i)$ (shown here in gray). The remaining objects, in white, constitute $capture(f_i)$. Finally, the gray objects in each child allocation region $alloc(c_j)$ provide a lower bound on $escape(c_j)$.

3.2 Computing $\overline{control}$

Recall that to compute $\overline{control}$

$$\overline{control}(f) \quad := \quad \frac{1}{n}\sum_{i=0}^{n} control(f_i)$$

we require a value for $control(f_i)$ for each invocation f_i, which in turn relies on the value of $escape(c_j)$ for each child c_j of f_i. Unfortunately, we cannot compute $escape(c_j)$ for each child c_j of f_i using the method described in the preceding subsection. Since the collector is copying, invoking it at the end of c_j will rearrange the heap, invalidating measurements for f_i and its other children. Thus, to compute $escape(f_i)$, we can afford only one garbage collection during the execution of f_i, at the end of f_i. To circumvent this problem, JOLT samples the value $escape(c_j, f_i)$, which is the value of $escape(c_j)$ when c_j executed as a child of f_i.

In more detail, JOLT computes $\overline{control}(f)$ not from the dynamic values $control(f_i)$, but from static information that preserves context-sensitivity whenever possible. The approximation uses static call graph edge frequencies *freq*

to weight the data from each of its static child calls:

$$\overline{control^*}(f) = \overline{capture}(f) - \sum_{c \in children(f)} freq(f,c) \cdot cap_f^*(c)$$

The value $freq(f,c) \cdot cap_f^*(c)$ approximates the total objects captured by c when c was called from f. The value $cap_f^*(c)$ is computed in one of three ways, in order of preference:

Content-sensitive sampling of c in f. This approach measures the behavior of c in the context of f. After f has been sufficiently sampled to establish a stable value for $\overline{capture}(f)$ (in practice, we use six samples), JOLT alters its behavior the next time f is scheduled for sampling. Rather than sampling $capture(f_i)$, JOLT randomly selects a child call c_j from f_i's execution. It then profiles that child call, computing $capture(c_j)$ by invoking the garbage collector at the end of c_j and storing this information as the calling-context-sensitive value $capture(c_j, f_i)$. During the summarization of f, if such a context-sensitive sample of c is available, the specific value $capture(c_j, f_i)$ is summarized to $\overline{capture}_f(c)$, which cap_f^* then uses to compute $\overline{control}(f)$ as per the formula above.

Context-insensitive value for c. If no such context-sensitive value is available for a given child c, JOLT attempts to use the context-insensitive value $\overline{capture}(c)$ that is computed as the mean of all available samples of $capture(c_j)$. This approach is less accurate, because it does not consider the context-sensitive behavior of c when called by f.

Upper-bound value of $capture(c_j)$. Finally, in the rare case that no samples of $capture(c_j)$ have been taken, JOLT falls back to computing an upper bound for $capture(c_j)$. This is done by computing a lower bound on $escape(c_j)$. During the initial profiles of f, when the garbage collector is being invoked at the end of f_i, JOLT passes the collector the allocation range not only for f_i but also for each of its children (see Figure 3). Then, during the collector's marking phase, if a live object happens to fall within the allocation range of f_i, the GC checks further to see whether the object also lies within the allocation range of c_j (see Figure 4 for pseudocode). If so, the object escapes c_j since it escapes its caller f_i. Because $alloc(c_j)$ is measured precisely, this lower bound on $escape(c_j)$ yields an upper bound on $capture(c_j)$. Though this upper bound is not very accurate (specifically, it cannot determine how many objects escaping c_j are controlled by f_i), it is inexpensive to obtain and serves to bound the error when no other information about c is available. As with the context-sensitive sampling, this upper bound on $capture(c_j)$ is summarized to an upper bound on $\overline{capture}_f(c)$, which cap_f^* uses in the computation of $\overline{control}(f)$.

4. Selecting Churn Scopes For Optimization

This section describes the heuristics used to select and optimize *churn scopes*, the sets of functions encompassing object churn that are transformed into a single function via inlining and then handed to the optimizer. In the first step, JOLT uses the dynamic analyses described in the previous sections to find the root of a scope. In the second step, JOLT inlines certain descendents of the root, depending on their benefit and cost to the optimization, in order to expose allocation sites to a traditional escape analysis. This algorithm is based on several approximations, but it has the benefit of being very simple to implement, and has good results in practice; see Section 6.

```
function mark_object(object o) {
  ...
  function f = currently_profiled_function;
  if (f != null && o >= tlhp(f, start(f)) &&
      o < tlhp(f, end(f))) {
    add_to_escaped(f, o);
    foreach(c in f.child_calls) {
      if (o >= tlhp(c, start(c)) &&
          o < tlhp(c, end(c)) {
        add_to_escaped_with_context(c, f, o);
    }
  }
  ...
}
```

Figure 4. Pseudocode for JOLT's *escape* algorithm during the GC's live object marking step.

4.1 Step 1: Finding the Root of the Scope

A *churn scope* is a subgraph of the static call graph that has a single root from which all nodes in the scope are reachable. This step finds a suitable root of the scope. JOLT uses three of the analyses from previous sections in conjunction to select scopes that are likely to result in beneficial optimization. JOLT selects scope roots that

- have a high *capture* value, which ensures that there are many churned objects in the scope, justifying the expense of optimizing it.
- have a high *%capture* value, which indicates that the scope would not gain many newly captured objected were it to be grown to include a caller of the root function: most of the allocated objects are being captured already.
- have a high *%control* value, which indicates that many of the captured objects are live at the scope's root. Thus, the scope would lose these captured objects were it to shrink to a child of the root, since the controlled objects would escape that scope.

When a function f has accumulated a set number of sampled profiles, a summary is generated using the techniques described in Section 3. JOLT evaluates the churn scope rooted at f by summing the rankings of $\overline{capture}(f)$, $\overline{\%capture}(f)$, and $\overline{\%control}(f)$ against all currently sam-

pled functions. (No functions are ranked until 100 functions have been summarized.) If f's summed ranking is in the top 1% of all functions, then it is selected as an churn scope root, and its scope is scheduled for churn optimization.

4.2 Step 2: Inlining Descendents

Ideally, we would inline into the churn scope root f all functions reachable from it. The resulting single function would encapsulate all of its captured objects' lifetimes, which would allow standard escape analysis to optimize them. Unfortunately, excessive inlining causes well-known problems, and JOLT therefore selects descendents of the root such that an inlining budget is not exceeded. To aid in explaning this selection process, we view the static call graph rooted at f as a call tree, by conceptually removing any cycles, and duplicating any vertices in the resulting DAG that have multiple in-edges. JOLT's strategy for selection is based on two observations.

Our first observation is that for the escape analysis to identify and remove the churn, we need to inline into the root f only those functions that allocate objects captured by f; this will make them optimizable by the escape analysis. Functions without captured allocations can be given a lower priority for inlining into f.

Our second observation helps us deal with the loss of precision that occurs when we aggregate capture information from the dynamic call tree onto the static call graph. Ideally, to decide whether a function g should be inlined into the root f, we want to know how many objects allocated by g are captured by f. This seems to require maintaining information for all pairs of reachable call graph nodes, which we do not track for reasons of efficiency. To avoid this overhead, we observe than f was selected as the root because of its high *%capture* value, which means that *most* objects allocated in f's scope are captured by f. Thus, we simplify the problem with the assumption that any allocation site is worth inlining, since it most likely allocates captured objects.

Since inlining every function with an allocation site often exceeds the maximum permitted size bound, we need to select the most profitable functions. We prefer to inline functions that allocate the most objects; this policy gives the largest number of allocations the chance of optimization. We phrase our final problem as follows:

Given as input a call tree rooted at f, with the benefit of each function in the tree equal to the number of allocations it contains, what is the subtree rooted at f with the greatest benefit that respects an inlining size bound k?

We show by reduction from the classic Knapsack problem that this problem is NP-Hard. The inputs to Knapsack are a set of items, each with a cost and a benefit, and the output is the subset of these items with the maximum benefit that does not exceed a given cost bound.

We represent each of the input items to Knapsack by a function whose size is the item's cost and which contains as many allocations as the item's benefit. Each of these functions is added as a child of a root function f to form a call tree. The size bound is set to the Knapsack input's cost bound. The solution to our problem yields a subtree containing those items that maximize the total benefit while respecting the bound, which solves the Knapsack problem.

```
totalCost := 0
S := set of initial candidate inlining decisions
while (totalCost < LIMIT && S != emptyset)
  choose inlining decision D from S with the largest
     benefit/cost ratio
  if (totalCost + cost(D) < LIMIT)
    inline D
    totalCost += cost(D)
    add children of D in tree to S
  S = S \ { D }
```

Figure 5. The inlining algorithm, based on an approximation of Knapsack, used by JOLT.

We observe, based on this reduction, that our problem is the Knapsack problem with the added restriction that for any item to be selected, its ancestors in a provided tree must be selected as well. Thus, we use an efficient approximation of Knapsack, discussed in the context of inlining in Arnold, *et al.* (5), and presented in a modified version in Figure 5, to solve it.

The initial inputs to this approximation are f's children. The algorithm chooses the input with the greatest benefit/cost ratio to inline first, and subsequently adds that input's children as new inputs, repeating this process until the cost limit is reached. Since it inlines from the root node f down the tree, the restriction of requiring ancestor nodes to be inlined is implicitly satisfied.

However, the algorithm with its stated inputs is ill-suited for maximizing the number of inlined allocations from the whole tree, since it greedily favors local maxima. Consider the chain of function calls $f \rightarrow g \rightarrow h$. If g has no allocations, it has no benefit, and the greedy algorithm would not inline along the $f \rightarrow g$ edge, even if h has many allocations.

To address this problem, JOLT uses the *alloc* value from its dynamic analyses to provide the Knapsack approximation with more holistc cost and benefit values. We change the benefit of each input n to be $alloc(n)$ — the number of allocations that occur in the entire subtree starting at n — and the cost to be the total size of the functions in the subtree. In the above example of $f \rightarrow g \rightarrow h$, the benefit of g, then, is not just the number of allocations it contains, but the number of allocations its subtree (including h) contains. Thus, even at f we can choose to inline g, knowing that further along its subtree are functions with many allocations.

This use of *alloc* is imprecise in the sense that it assumes that an entire subtree's cost and benefit will be acquired

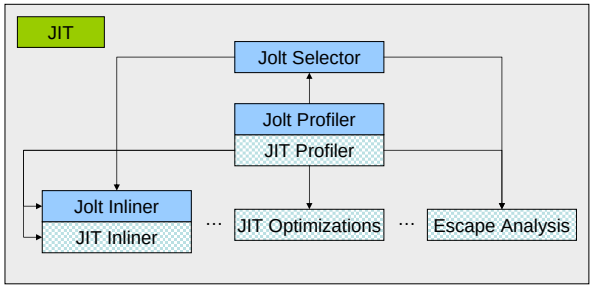

Figure 6. A schematic of the JOLT implementation. The profiler gathers data for the analyses (described in Section 2) much like a normal VM profiler does, via sampling. It passes this information on to a selector, which rates each function. If a function's aggregate rank is in the top 1% of all profiled functions, it is selected for optimization. The JOLT inliner (Section 4) then uses smart inlining (aided by the standard JIT profiler) to expose as many allocations as possible. The standard escape analysis is also prodded to run on the expanded function, even if it might not have otherwise.

when the subtree's root is selected for inlining, when in fact only part of the subtree may be inlined, depending on the inlining budget and what other inlining candidates are available.

Once the churn scope has been selected and a subset of its functions is inlined into the root function f, traditional escape analysis is performed on the expanded f, eliminating the captured allocations therein.

5. Implementation in J9

In this section we describe implementation details. JOLT is implemented in a development version IBM's J9 JVM. See Figure 6 for a diagram of the JOLT architecture.

Analysis implementation. To reduce overhead, full-duplication sampling (7) is used to gather runtime profiles. Thus, JOLT's profiler only runs during a function's slow path. JOLT optimizes slow-path overhead in two ways. First, it does not instrument leaf calls (since they are served well by normal JIT optimizations), and does not instrument around child calls to functions which themsleves have no calls and no allocations (and thus no objects to capture). Note that even though these functions and calls may not be profiled directly, their behavior is still captured by the statistics taken by the calling function. Second, JOLT only instruments functions when they are compiled at the intermediate optimization level. Thus, the hottest functions accrue some samples, and then have no instrumentation at all when they are recompiled by the JIT at a higher optimization level. After 10 samples, a function is considered for churn optimization.

Naturally, error is introduced by sampling data rather than recording statistics from all function invocations. This tradeoff is well documented (7): a lower sampling interval enables quicker convergence toward unsampled data at the cost of higher overhead. A judicious choice of interval can affect the error as desired.

Note that this error does not affect any individual sample on an invocation f_i; it only affects the aggregate statistics for function f because not all of its invocations are tracked.

Though the analysis summaries are defined in terms of numbers of objects, JOLT's implementation actually computes the size of those objects in bytes. This decision makes implementation easier, since it requires only computing the size of heap ranges (a simple subtraction), rather than walking the heap to count objects.

Inlining implementation. JOLT's inliner is implemented alongside the standard VM inliner. Edge and basic block frequencies used in dynamic allocation counts are gathered from the VM's built-in profiler. In the implementation, the call graph explored starting from a function f is bounded at depth 6. Though we describe the inlining algorithm in Section 4.2 as operating on a call tree, due to the greedy nature of the algorithm JOLT does not need to explicitly convert the call graph to a tree. Instead, it retains the call graph and only lazily duplicates nodes as necessary when their parents are chosen for inlining.

6. Evaluation

We evaluated JOLT on a number of large and popular framework-based benchmarks. These are: an implementation of TPC-W (2) running atop the JBoss application server (16); the Eclipse benchmark found in the DaCapo benchmark suite (9); the JPetStore e-commerce application running atop the Spring application framework (1) (the most popular framework listed on sourceforge.net); and SPECjbb2005, a three-tiered client/server benchmark. We also evaluated it against the DaCapo benchmark suite.

6.1 Methodology

Measurements were made with a developmental version of IBM's J9, a leading commercial VM, on a dual Xeon 2.8 Ghz machine with 1 GB of RAM running Red Hat Linux with a 2.6.9 kernel. Baseline numbers were measured by running the VM in its default configuration.

Each result was obtained by taking the median steady-state performance of 5 program runs. The DaCapo benchmarks, including Eclipse, were version 2006-10 and were run with the parameters `-n 10 -s large`; the time of the tenth run was taken as the steady-state. Unfortunately, the developmental version of our VM was unable to execute the `jython` benchmark from the DaCapo suite. Load for the Petstore application was generated by the `petload` program (3), running on another server. Two 100-second executions of `petload` were performed, with the second taken to be steady-state. The TPC-W workload was generated with the implementation's accompanying remote browser program; it was executed on another server with the relevant parameters `-TT 0.0 -RU 100 -MI 100 -GETIM false`

Benchmark	Comp. Time	Runtime/ Throughput	% Speedup	# Objs Eliminated	% Objs Eliminated	Improvement Over Base
Eclipse						
Baseline	53s	94.7s		5.3m/1.2b	0.4%	
Inlining	75s	98.0s	-3.5%	9.6m/1.2b	0.7%	2.0x
Selection	58s	93.2s	1.6%	20.5m/1.2b	1.7%	3.8x
JOLT	62s	90.4s	4.8%	23.6m/1.2b	1.9%	4.6x
JPetstore on Spring						
Baseline	61s	810 *		1.2m/169m	0.7%	
Inlining	66s	790 *	-2.5%	1.6m/170m	1.0%	1.3x
Selection	64s	816 *	0.7%	2.2m/166m	1.3%	1.9x
JOLT	77s	828 *	2.2%	4.1m/166m	2.5%	3.5x
TPCW on JBoss						
Baseline	34s	98.9 *		4.8k/198m	0.0%	
Inlining	44s	100.7 *	1.8%	896k/193m	0.5%	192x
Selection	36s	97.2 *	-1.7%	5.7m/191m	3.0%	$1.3x10^3$
JOLT	43s	101.5 *	2.6%	8.0m/186m	4.3%	$1.7x10^3$
SPECjbb2005						
Baseline	11s	20173 *		25.5m/267m	9.6%	
Inlining	18s	20196 *	0.1%	24.0m/241m	9.9%	1.0x
Selection	18s	22233 *	10.2%	43.1m/238m	18.1%	1.9x
JOLT	20s	22828 *	13.2%	81.2m/253m	32.1%	3.4x
DaCapo						
`antlr` Baseline	15s	6.6s		39.2k/85.2m	0.0%	
`antlr` JOLT	21s	6.7s	-1.5%	1.7m/83.2m	2.0%	44.4x
`bloat` Baseline	19s	81.4s		82.7m/1.1b	7.3%	
`bloat` JOLT	28s	70.9s	14.8%	375m/1.1b	33.8%	4.6x
`chart` Baseline	16s	21.3s		20.7m/896m	2.3%	
`chart` JOLT	21s	20.1s	6.0%	47.0m/905m	5.2%	2.3x
`fop` Baseline	5s	4.2s		120k/10.5m	1.1%	
`fop` JOLT	8s	4.1s	2.4%	1.0m/9.6m	10.6%	9.3x
`hsqldb` Baseline	22s	11.4s		0/97.2m	0.0%	
`hsqldb` JOLT	29s	11.4s	0.0%	4.7m/103m	4.6%	∞
`luindex` Baseline	14s	9.2s		2.1m/114m	1.8%	
`luindex` JOLT	16s	8.6s	7.0%	4.9m/111m	4.4%	2.4x
`lusearch` Baseline	47s	11.6s		9.1m/395m	2.3%	
`lusearch` JOLT	49s	10.6s	9.4%	173m/405m	42.7%	18.6x
`pmd` Baseline	19s	20.3s		163m/1.1b	15.5%	
`pmd` JOLT	28s	20.5s	-1.0%	164m/1.1b	15.5%	1.0x
`xalan` Baseline	41s	39.1s		955k/732m	0.1%	
`xalan` JOLT	37s	38.3s	2.1%	9.8m/734m	1.3%	10.2x

Table 1. Steady-state performance numbers of JOLT on several benchmarks. Note that compilation time was measured over the entire run and thus may exceed the single steady-state benchmark time. Performance numbers with a * indicate higher is better. Number precision is discarded only for display; full precision was used in computing percentages and ratios.

`-CUST 144000 -ITEM 10000`. The SpecJBB2005 benchmark was configured to run from 1 to 4 warehouses, with a ramp up period of 60 seconds and a measurement period of 60 seconds. The throughput with four warehouses was taken to be the steady-state performance.

6.2 Results and Analysis

The results are presented in Table 1. For each benchmark, *Baseline* shows the steady-state performance of the original, unmodified VM, while JOLT shows the steady-state performance of the full JOLT optimizer, including profiling overhead. The *Inlining* and *Selection* configurations are discussed later in this section.

% Speedup is computed from the ratio of a configuration's runtime to the Baseline runtime. *# Objs Eliminated* displays two numbers, X/Y, where X is the number of dynamic object allocations eliminated of Y total dynamic object allocations during an entire program run; *k* signifies thousands, *m* millions, and *b* billions. The *Improvement Over Base* column shows the ratio of the number of objects eliminated by a particular configuration to the number of objects eliminated by the Baseline configuration.

In every benchmark, JOLT was able to eliminate more allocations than escape analysis alone. The increase ranged from just over 1x to ∞ (in the case where escape analysis was unable to remove any objects at all), with a median of 4.6x. Note that due to our measurement methodology, the numbers measured for the evaluation represent eliminated object allocations, whereas JOLT's optimizer attempts to eliminate the maximum number of *bytes* allocated. Thus the percentage of allocated bytes eliminated by JOLT is likely to be greater.

Performance also improved under JOLT, with an average speedup of 4.8% for all applications, 5.7% for the component-based applications, and a max speedup of 14.8% for `bloat`.

The compilation overhead ranged from -10% with `xalan` [1] to 82% with SPECjbb2005, with an average of 32%. We feel that this increase in the relatively constant-factor overhead of compilation is acceptable in the context of long-running programs that may execute from tens of minutes to days.

A surprising result is how few allocations escape analysis is able to remove on these large-scale programs. Though JOLT is able to improve upon these numbers, there remains a significant opportunity for further optimization that we are continuing to investigate.

JOLT is composed of two primary mechanisms, a set of dynamic analyses and an inliner. It is possible that JOLT's empirical results are more due to one or the other of these; for instance, it might be possible to do just as well with only a dynamic analyses-driven scope selector and then a standard inliner. We evaluated this hypothesis on the component-based benchmarks as follows. In addition to comparing the default VM eliminations against the JOLT eliminations, we also measured two other configurations: (a) the analyses were used to select churn scopes, which were then fed to the default VM inliner, rather than the JOLT inliner, before escape analysis ("Selection") (b) no churn scopes were selected, and instead all hot functions were optimized using the JOLT inliner ("Inlining"). The results are shown in Table 1.

[1] For reasons we could not fully diagnose, the JOLT run performed less compilation than the baseline run.

Benchmark	Profiling Overhead
Eclipse	0.8%
JPetstore/Spring	1.1%
TPCW/JBoss	1.7%
SPECjbb2005	1.2%
`antlr`	2.3%
`bloat`	-0.2%
`chart`	-1.6%
`fop`	2.6%
`hsqldb`	2.7%
`luindex`	0.5%
`lusearch`	-0.4%
`pmd`	1.5%
`xalan`	0.3%

Table 2. Profiling overheads for computing the capture and control analyses.

The Inlining configuration tended to slow the program down; however, it did allow for more allocation elimination, possibly because every hot method fed to the escape analysis had far more allocation sites present. The Selection configuration generally performed right at the baseline. Neither exceeded the combined JOLT configuration on any of the four benchmarks, which seems to indicate that both are contributing to its performance.

We report the overhead of the capture and control analysis profiler in Table 2. The overhead numbers measure the total overhead over the full long-running execution. The profiler sampled one function invocation of every 100,000. See Section 5 for details on how functions were profiled. The speedups for several benchmarks are possibly due to the profiler's behavior of recompiling a function once it has taken enough measurements, to remove the sampling code. The cost of this approach is reflected in compilation time (Table 1), but the aggressive recompilation of frequently executed methods may be improving performance.

7. Related Work

Mitchell, *et al*. (23) identified the object churn and excessive computation pervasive in component-based software. Dufour, *et al*. (15) informally explored the notions of *capture* and *control*, and used a hybrid analysis to aid a programmer in finding areas of object churn. Our work strives to address

these same problems by automatically identifying and optimizing churn in a virtual machine.

JOLT is prototyped in a leading production VM that performs profiling and adaptive recompilation similar to other virtual machines (4; 6; 18; 19; 25; 28). Methods begin executing in an interpreter, and hot methods are profiled and promoted to higher levels of optimization using a JIT compiler. For the aggressive dynamic analysis, JOLT employs the Arnold-Ryder sampling framework (7) to keep overhead low.

Escape analysis (14; 26) (as pertains to Java (10; 12; 29)) is a critical component in JOLT's optimization scheme. Whaley and Rinard (29), Gay and Steensgaard (17), and Blanchet (10) have described extensions to their escape analyses that can detect object capture 1 and arbitrarily many functions up the call stack, respectively. However, these extensions either are not benchmarked or do not perform well in practice (10), possibly due to the large amount of duplicated context necessary to eliminate each captured object.

Since then, advances have been made in the sophistication and aggressiveness of escape analysis in JIT compilers (21) and the transformations that eliminate object allocation, such as lazily reallocating eliminated allocations if necessary (22). Escape analyses can be used directly by JOLT in its optimization procedure as they become available in state-of-the-art VMs. Although these escape analyses use interprocedural analysis to identify objects that do not escape from non-inlined callees, unlike JOLT they do not use interprocedural analysis to identify callees containing key allocations, which thus must be inlined for the escape analysis to be effective.

Many papers have used profiling information to guide inlining (5; 8; 11; 20). Scheifler (27) first reduced a size-bounded inlining problem to Knapsack. Like JOLT, several works have used inlining not as an end in itself (as a call overhead reduction) but as a means to enable other optimizations. EDO (24) inlined hot exception paths so that thrown exceptions do not have to walk the call stack. Dean and Chambers (13) decided between multiple inlining options based on the benefits accrued by optimizations applied to the inlined method's body.

8. Conclusion

In this paper, we have presented JOLT, a fully-automatic online churn optimizer. It selects scopes to optimize via a novel lightweight dynamic analysis based on the notions of *control* and *capture*. Churn elimination is achieved by using combined dynamic and static analysis to guide inlining decisions, making the resulting compilation unit more amenable to escape analysis. The resulting code is fed to a state of the art JIT optimizer that performs escape analysis along with other standard optimizations.

Acknowledgements

We thank Nick Mitchell for several early discussions on object churn, Gary Sevitsky for his feedback on a draft of this paper, and Bill McCloskey for help with his `petload` tool. We are grateful to the anonymous referees for their helpful comments. This work was supported in part by the National Science Foundation with grants CCF-0085949, CNS-0326577, and CNS-0524815, an NSF Graduate Fellowship, a generous gift from IBM Corporation, the IBM Open Collaborative Research project, the AF-TRUST project, and the University of California MICRO program .

References

[1] Spring framework. `http://www.springframework.org/`.

[2] TPC-W NYU. `http://cs.nyu.edu/~totok/professional/software/tpcw/tpcw.html`.

[3] Personal communication with William McCloskey, October 2007.

[4] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Dong-Yuan Chen, Anwar Ghuloum, Vijay Menon, Brian Murphy, Mauricio Serrano, and Tatiana Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1):19–31, February 2003.

[5] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 52–64, New York, NY, USA, 2000. ACM.

[6] 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.

[7] 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.

[8] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. *SIGPLAN Not.*, 32(5):134–145, 1997.

[9] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM.

[10] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented pro-*

gramming, systems, languages, and applications, pages 20–34, New York, NY, USA, 1999. ACM.

[11] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[12] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, New York, NY, USA, 1999. ACM.

[13] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 273–282, New York, NY, USA, 1994. ACM.

[14] Alan Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 157–168, New York, NY, USA, 1990. ACM.

[15] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 118–128, New York, NY, USA, 2007. ACM.

[16] Marc Fleury and Francisco Reverbel. The JBoss extensible server. In Markus Endler and Douglas Schmidt, editors, *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *LNCS*, pages 344–373. Springer-Verlag, 2003.

[17] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *th International Conference on Compiler Construction (CC'2000)*, volume 1781. Springer-Verlag, 2000.

[18] Nikola Grcevski, Allan Kilstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[19] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 38(11):187–204, November 2003.

[20] Owen Kaser and C. R. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 24(2):55–72, 1998.

[21] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 111–120, New York, NY, USA, 2005. ACM.

[22] Thomas Kotzmann and Hanspeter Mossenbock. Run-time support for optimizations based on escape analysis. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Computing (ECOOP) 2006*, 2006.

[24] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. Edo: Exception-directed optimization in java. *ACM Trans. Program. Lang. Syst.*, 28(1):70–105, 2006.

[25] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM)*, pages 1–12, April 2001.

[26] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 116–127, New York, NY, USA, 1992. ACM.

[27] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20(9):647–654, 1977.

[28] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 36(11):180–195, November 2001. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

[29] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206, New York, NY, USA, 1999. ACM.

APPENDIX

A. Algorithms for Approximating Capture and Control in Any VM

The algorithms for computing *capture* and *control* presented in this paper rely on two virtual machine features: a contiguous object allocator and thread-local heaps.

Though most production VMs support these features, not all VMs do, and in this section we present an alternate set of algorithms that can approximate *capture* and *control* on just about any VM. The only requirement is that the VM have (a) a garbage collector that can report the number of live objects it has found after a collection and (b) an allocator that can count the number of allocations it has made.

Naturally, with these two simple primitives, we aim to approximate *capture* and *control* rather than to compute them exactly. We do this by reducing the problem from tracking the reachability of individual objects as they are allocated to simply tracking differences in the number of reachable objects present in the program as the execution progresses. Below, we describe the two major steps of this approximation, and then discuss the error the approximation introduces.

A.1 Aggregation

We begin with the assumption that we can approximate the two simple notions defined in Section 2.1, *alloc* and *escape*, from the two VM primitives described above; the actual approximation is described in Section A.2. Our first step toward an approximation is to sacrifice object-level precision in favor of aggregation. In other words, we forego the knowledge of, say, exactly which objects are captured by a particular function in favor of knowing how many such objects there are.

Recall the definitions of *capture* and *control* in terms of *alloc* and *escape*. Here, we show that these definitions hold if we replace sets with their cardinalities.

LEMMA 1.

$$|capture(f_i)| = |alloc(f_i)| - |escape(f_i)|$$

$$|control(f_i)| = |capture(f_i)| - \sum_{c \in children(f_i)} |capture(c)|$$

This computation of cardinality from the primitives *alloc* and *escape* is precise, even in the presence of set subtraction. In other words, we prove that we only lose the knowledge of what objects are in each set; we still know the exact size of the sets.

Proof. Consider some sets $t, s_1, s_2, \cdots$ and a set subtraction $t \setminus \cup_i s_i$. We would like to show that $|t \setminus \cup_i s_i| = |t| - \sum_i |s_i|$.

For each s_i, then, we must show that (a) $s \subseteq t$ (or else subtracting the cardinality will undercount the result) and (b) $\forall_{j.i \neq j} s_i \cap s_j = \emptyset$ (or else the sum of the cardinalities of the s_is will be overcounted, and the result undercounted).

For $capture(f_i)$, we must only show that $escape(f_i) \subseteq alloc(f_i)$. Consider an object o. $o \in escape(f_i) \implies start(f_i) < start(o) < end(f_i) \implies o \in alloc(f_i)$.

We expand $control(f_i)$ to

$$control(f_i) = (alloc(f_i) \setminus escape(f_i)) \setminus \bigcup_{c \in children(f_i)} capture(c)$$

Consider an object o and a child c of f_i. By the definitions of *capture*, *child*, and *alloc*, we know that $o \in capture(c) \implies start(c) < start(o) < end(c) \implies start(f_i) < start(o) < end(f_i) \implies o \in alloc(f_i)$.

Thus, we have

$$\begin{array}{rcll} escape(f_i) & \subseteq & alloc(f_i) & \text{(as above)} \\ \forall_{c \in children(f_i)} capture(c) & \subseteq & alloc(f_i) & \end{array}$$

To prove null intersection, we first compare the *escape* term to each of the *capture* terms, and then compare the *capture* terms to each other. Again, consider an object o and an arbitrary child call c.

By the definitions of *escape*, *capture*, and *child*, $o \in escape(f_i) \implies end(f_i) < end(o) \implies end(c) < end(o) \implies o \notin capture(c)$.

Now consider two children, c_1 and c_2. By the definition of *child*, w.l.o.g. assume $end(c_1) < start(c_2)$. Then $o \in capture(c_1) \implies end(o) < end(c_1) \implies end(o) < start(c_2) \implies o \notin capture(c_2)$, and $o \in capture(c_2) \implies start(c_2) < start(o) \implies end(c_1) < start(o) \implies o \notin capture(c_1)$. This yields

$$\begin{array}{rcl} \forall_{c \in children(f_i)} capture(c) \cap escape(f_i) & = & \emptyset \\ \forall_{c,d \in children(f_i).c \neq d} capture(c) \cap capture(d) & = & \emptyset \end{array}$$

This completes the proof. $\diamondsuit$

A.2 Runtime Values

It remains to approximate *alloc* and *escape* for each function in a program at runtime given the two memory management primitives described at the beginning of the Appendix, which we call *objcount* and *reachcount*.

objcount(t): the number of objects allocated from time 0 to time t

reachcount(t): the number of reachable objects at time t

Objcount is obtained by keeping a running counter of object allocations, and gives us the exact cardinality of *alloc*.

$$|alloc(f_i)| = objcount(end(f_i)) - objcount(start(f_i))$$

Reachcount is obtained by running the garbage collector and counting the number of live objects it finds, and serves as an approximation for *escape*.

$$|escape(f_i)| \cong reachcount(end(f_i)) - reachcount(start(f_i))$$

The error introduced by this approximation is discussed in Section A.3.

For a particular function invocation f_i, we compute *objcount* and *reachcount* at its entry and exit. These data points are sufficient to compute $capture(f_i)$ and $control(f_i)$. A full-duplication sampling profiler is used to gather samples without running an excessive number of garbage collections.

f_i is also instrumented to compute *objcount* and *reachcount* additionally at the beginning and end of any child calls it makes, yielding $|alloc(c)|$ and $|escape(c)|$ for each child call. This additional information yields $control(f_i)$, as per the defintion in Section 2.3.

A summary of the runtime approximations of the analyses described in Section 2 is shown in Figure 7. Unlike the algorithm presented in the body of this paper, this approximation generates *control* values for individual function invocations f_i. Thus, the formal definition of $\overline{control}$ can be used to compute function averages.

A.3 Error

The approximation of *capture* and *control* detailed above introduces error in two ways, discussed here.

***Escape* approximation.** In Section A.2, we approximated *escape* by measuring the difference in the number

$$
\begin{aligned}
|capture^*(f_i)| &= (objcount(end(f_i)) - objcount(start(f_i))) - (reachcount(end(f_i)) - reachcount(start(f_i))) \\
|\%capture^*(f_i)| &= \frac{|capture^*(f_i)|}{objcount(end(f_i)) - objcount(start(f_i))} \\
|control(f_i)^*| &= |capture^*(f_i)| - \sum_{c \in children(f_i)} |capture^*(c)| \\
|\%control^*(f_i)| &= \frac{|control^*(f_i)|}{objcount(end(f_i)) - objcount(start(f_i))}
\end{aligned}
$$

Figure 7. A summary of the analysis approximations used by the generalized system described in the Appendix. We use x^* to denote the approximate version of x. To compute values for a static function f, we compute the median values over all available dynamic datapoints f_i.

of reachable objects between function beginning and function end. However, this difference actually accounts for two phenomena: locally allocated objects that escape increase the difference, but objects allocated before the function start that are no longer reachable at function exit *decrease* the difference. We call these latter objects *absorbed* by f_i. Let $\Delta reachcount(x, y) = reachcount(y) - reachcount(x)$.

$$absorb(f_i) := \{o | start(o) < start(f_i) < end(o) < end(f_i)\}$$

$$\Delta reachcount(start(f_i), end(f_i)) = |escape(f_i)| - |absorb(f_i)|$$

Thus the error introduced by this approximation is exactly equal to the number of absorbed objects. From this we can compute the error for the analyses. We use x^* to denote the approximate version of x.

$$|capture^*(f_i)| = |capture(f_i)| + |absorb(f_i)|$$

$$|\%capture^*(f_i)| = |\%capture(f_i)| + \frac{|absorb(f_i)|}{|alloc(f_i)|}$$

$$|control^*(f_i)| = |control(f_i)| + |absorb(f_i)| - \sum_{c \in children(f_i)} |absorb(c)|$$

$$|\%control^*(f_i)| = |\%control(f_i)| + \frac{|absorb(f_i)| - \sum |absorb(c)|}{|alloc(f_i)|}$$

To mitigate some of this error, an implementation of JOLT using these approximation algorithms can make use of a simple observation. Consider the case in which $\Delta reachcount(start(f_i), end(f_i)) = x$. If every object allocated in f_i's churn scope were captured, $x = 0$ (no more objects are reachable at the end of f_i than at the beginning). On the other hand, if $x < 0$, at least x objects must have been absorbed by f_i, to account for the fact that the number of reachable objects has decreased. JOLT can thus put a lower bound on the number of absorbed objects for any f_i whose $\Delta reachcount$ is negative, and use this lower bound, for instance, by subtracting it directly from the approximation of *capture*, reducing the error.

This situation may arise infrequently, though, since *absorbed* can be masked by escaping objects. Now imagine that *reachcount* were computed after every instruction $m_{1..k}$ in f_i. Then, as in with the end of f_i, if for any $n \leq k$, $\Delta reachcount(start(f_i), m_n) = x$ where $x < 0$, at least x objects must have been absorbed between the start of f_i and that instruction. Each instruction, then, provides an opportunity to place a lower bound on absorbed.

Of course, JOLT cannot calculate *reachcount* after every instruction in f_i, since that would require a very large number of GCs. However, it does have some existing GC data points already being gathered, around the child calls that f_i makes, as described in Section A.2. Thus, JOLT can compute $\Delta reachcount$ between the start of f_i and each other profiled point. If any of these deltas is negative, that places a lower bound on $absorbed(f_i)$, which is then used to reduce the error.

We measured absorbed objects with runtime instrumentation on the SPECjbb2005 benchmark. For the functions selected by JOLT for optimization, the ratio of absorbed objects to allocated objects was 1:8, indicating that the error is likely to be low in practice for functions amenable to churn optimization.

Multithreading. The black-box approach that leads to the error in the *escape* approximation also introduces error in another way. Consider the computation of $alloc(f_i)$. A global allocation counter is kept, and the difference is taken between this counter's value at the end of f_i and its value at the beginning of f_i. In a single-threaded program, this result is exact, since any intervening allocations must have occurred within f_i and its children. However, if the program execution has multiple threads, the global counter will be incremented by any allocations that occur in other threads that might have been run between the start and end of f_i. Similarly, liveness counts may be erroneously inflated or deflated by behavior in other threads. Of course, the magnitude

of this error is largely dependent on the specific behavior of these threads.

One way to overcome this error is to discard obvious outliers in the profile data. These outliers generally indicate extra-thread behavior rather than anomalous function behavior. (This intuition is supported by the very quick convergence of profile data in the thread-local algorithms.) Once outliers are discarded and the deviation in profile values is low, it becomes possible to gauge the true behavior of sampled functions, even in the presence of multiple threads.