

A Scalable Technique for Characterizing the Usage of Temporaries in Framework-intensive Java Applications*

Bruno Dufour
Dept of Computer Science
Rutgers University
dufour@cs.rutgers.edu

Barbara G. Ryder
Dept of Computer Science
Rutgers University
ryder@cs.rutgers.edu

Gary Sevitsky
IBM T.J. Watson Research
Center
sevitsky@us.ibm.com

ABSTRACT

Framework-intensive applications (e.g., Web applications) heavily use temporary data structures, often resulting in performance bottlenecks. This paper presents an optimized blended escape analysis to approximate object lifetimes and thus, to identify these temporaries and their uses. Empirical results show that this optimized analysis on average prunes 37% of the basic blocks in our benchmarks, and achieves a speedup of up to 29 times compared to the original analysis. Newly defined metrics quantify key properties of temporary data structures and their uses. A detailed empirical evaluation offers the *first* characterization of temporaries in framework-intensive applications. The results show that temporary data structures can include up to 12 distinct object types and can traverse through as many as 14 method invocations before being captured.

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures; D.3.4 [Programming languages]: Processors

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

Dataflow analysis, escape analysis, program understanding, performance, framework-intensive applications, Java

1. INTRODUCTION

The increasing complexity of tasks accomplished by software has led to the proliferation of large *framework-intensive* applications. For example, Web applications are typically built by integrating numerous layers of middleware, libraries and frameworks; however while this eases development effort, this reliance on code reuse comes at a cost. Passing data from one framework layer to another often requires expensive operations such as wrapping and unwrapping objects. This problem is exacerbated by the fact that

developers are unfamiliar with the underlying frameworks and libraries used by their applications and unaware of the cost of API calls. As a result, framework-intensive applications do an extraordinary amount of work to accomplish simple tasks. For example, the conversion of a single date field from a SOAP data source to a Java object can require as many as 268 method calls and the generation of 70 objects [21]. Much work involves the creation and initialization of *temporaries*, short-lived objects that are created as the by-product of some computation. *Object churn*, that is the excessive usage of temporaries, is a new, but common problem in framework-intensive applications, that can degrade performance dramatically. The combined cost of allocating, initializing and ultimately garbage collecting temporaries can dominate execution time. In extreme cases, object churn can result in almost continuous calls to the garbage collector, effectively halting execution progress because of the amount of temporary storage being used and released over short time intervals.

The sophisticated optimizations employed by modern just-in-time compilers (JITs) are unable to ameliorate object churn. In particular, many JITs now employ an escape analysis to stack-allocate temporary objects, yet high garbage collection rates still remain in many framework-intensive systems. Costs are high even with improvements in memory management. Determining why large numbers of temporaries are created by these systems is the first important question that needs to be addressed. In addition, object churn cannot be alleviated solely by optimization of individual frameworks, because often the temporaries are passed in calls across framework boundaries.

The goal of our research is to enable a characterization of temporaries in framework-intensive applications and of the program regions that create and use these temporaries. In addition to quantitative characterization, we want to aid program understanding of specific temporary structures and their uses. The ultimate aim of obtaining this information is to provide a deeper understanding of object churn, in order to devise the appropriate actions for ameliorating the problem. This may be accomplished through focused global optimizations, best practices for framework API design and usage, and/or better diagnosis and assessment tools for framework-intensive applications

There are two aspects of the behavior of framework-intensive systems that make studying temporaries difficult. First, temporary creation and usage is often not localized to a single method, but involves multiple methods, each contributing a few allocations or making use of temporary objects allocated elsewhere. Second, temporary objects often appear as part of larger temporary data structures. In such cases, understanding the purpose of a single object requires studying its role within a data structure. Existing profiling tools like *Jinsight* [10] and *ArcFlow* [1] focus on the allocating

*This work was funded in part by IBM Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-59593-995-1 ...\$5.00.

methods of individual objects, and therefore do not provide information regarding how objects are used.

In previous work [11], object lifetime information, computed using blended escape analysis, allowed summarization of the usage of temporaries. First, by identifying which instances are used only locally within a region of the program, we can approximate the set of temporaries. We can identify program regions that are the top users of temporaries and rank them by the number of temporaries allocated within them. Second, because escape analysis subsumes an interleaved points-to analysis, it allows us to reason about how object instances become connected during execution. We can group the temporaries by their connectivity into data structures. We show how this information can enable characterization studies as well as better understanding of specific temporary structures.

Framework-intensive applications are a challenge to existing analysis techniques. Purely static analyses, accomplished through examination of code without execution, suffer problems of insufficient scalability and/or precision for answering behavioral questions for these systems. Purely dynamic analyses, accomplished through judiciously placed instrumentation in source code or byte-code, or by probing the JVM run-time system, introduce too much execution overhead and possibly perturb execution behavior. Limiting the amount of information collected at runtime severely impacts the usefulness of the analysis.

Previously we proposed blended analysis as an alternative to a purely static or dynamic analysis [11]. Blended analysis performs an interprocedural static analysis on a calling structure obtained by profiling an application. The goal of a blended analysis is to obtain precision comparable to that of a purely dynamic analysis while keeping the run-time overhead as low as possible. In other words, blended analysis pushes the most expensive part of the computation to an offline static analysis that uses the results of a previous lightweight dynamic analysis. This is essential for analysis of real-world, deployed applications, for which slowing down execution by more than a few percent is unacceptable. Blended analysis also reduces the amount of work performed by the static analysis by only analyzing the calling paths in the code that were exercised at runtime, and eliminates the problems of how to handle dynamic class loading and reflective method calls, as these are captured in the execution trace(s).

Previous blended analyses used dynamic information to restrict the static analysis to the actually executed methods. In this paper we present a refinement that allows the blended analysis to *filter out unexecuted code at the granularity of basic blocks* rather than methods. We show that this optimization significantly improves both analysis scalability and the precision of the computed results. We use an optimized blended escape analysis to perform a detailed investigation of the nature and usage of temporaries in four benchmarks obtained from two framework-intensive applications: *Trade 6*, a financial simulation Web application, and the Eclipse JDT Compiler, an incremental Java compiler. We have designed new metrics to capture the effects of the pruning technique on the scalability and precision of the analysis. We also have new metrics that enable the *first* automated characterization of the behavior of temporaries in such benchmarks. The results provide unique insights on the complexity of temporary data structures, the way they are used in typical scenarios, and the challenges faced in helping developers to understand and fix performance problems due to temporaries.

In our previous work [11], we described the blended analysis paradigm and presented an original blended escape analysis algorithm with postprocessing. Some simple aggregate measures of the escape behavior of objects were calculated using *Trade 6*, but dis-

appointingly, were not useful for explaining the behavior of temporaries. In addition, previously there was no characterization of the regions where temporary objects were used, nor of the temporary data structures themselves.

Thus, this paper makes the following major contributions:

- An *optimized blended escape analysis algorithm* that prunes away unexecuted basic blocks in methods, achieving increased precision and scalability attested to by empirical experiments. The pruning technique is generally applicable to any static analysis used in the blended analysis paradigm, as it uses light-weight dynamic information (i.e., executed calls and object creations) to prove the infeasibility of basic blocks in the executions to be analyzed.
- *New metrics* for blended static and dynamic analyses that quantify key properties related to the use of temporary objects. By combining static and dynamic information, we have defined a new data structure abstraction that supports a rich characterization of temporary data structures, a first step towards dealing with the performance problems they incur.
- *Initial empirical findings* that characterize the nature and usage of temporary objects in representative, framework-intensive Java applications. Our analysis enables the location of regions of excessive temporary usage in framework-intensive applications.

In Section 2, we give some background on blended escape analysis and discuss novel optimizations that improve its scalability. In Section 3 we present our metrics, along with our data and the results of our empirical study. In Section 4 we discuss some related work and close with our conclusions in Section 5.

2. ANALYSIS REFINEMENTS

Blended analysis [11] is a tightly coupled combination of dynamic and static analyses, in which the dynamic analysis determines the program region to which the static analysis will be applied. In our previous work on blended analysis, the dynamic analysis obtained a calling structure of a program on which a subsequent, interprocedural static analysis was performed. Blended analysis aims to capture detailed properties of a single execution or set of executions. This means the analysis is *unsafe* because it does not summarize the behavior of all possible executions, like a standard static analysis [18]. Nevertheless, blended analysis is well suited for program behavior understanding tasks, since they usually require very detailed information about a given execution (e.g., a run that exhibits performance problems).

Blended analysis offers many advantages compared to a purely static or dynamic analysis. First, blended analysis limits the scope of the static analysis to methods in the program that actually were executed, thus dramatically reducing the cost of a very precise static analysis by reducing its focus, allowing achievement of high precision over an interesting portion of the program. Second, blended analysis only requires a lightweight dynamic analysis, thus limiting the amount of overhead and perturbation during execution.

Our first instantiation of blended analysis was a blended escape analysis [11]. In this section, first we briefly present background information on escape analysis, and in particular summarize our previous work on blended escape analysis. Second, we discuss two optimizations that significantly lower analysis cost, improve scalability and increase precision. The first optimization uses declared type information to improve the computed results; the second uses dynamic information about executed calls and object allocations to reduce the work of the intraprocedural analysis.

```

1 public X identity(X p1) {
2     return p1;
3 }

5 public X escape(X p2) {
6     G.global = p2;
7     return p2;
8 }

10 public void f() {
11     X inst;
12     if (cond)
13         inst = identity(new Y());
14     else
15         inst = escape(new Z());
16 }

```

Listing 1: Example program

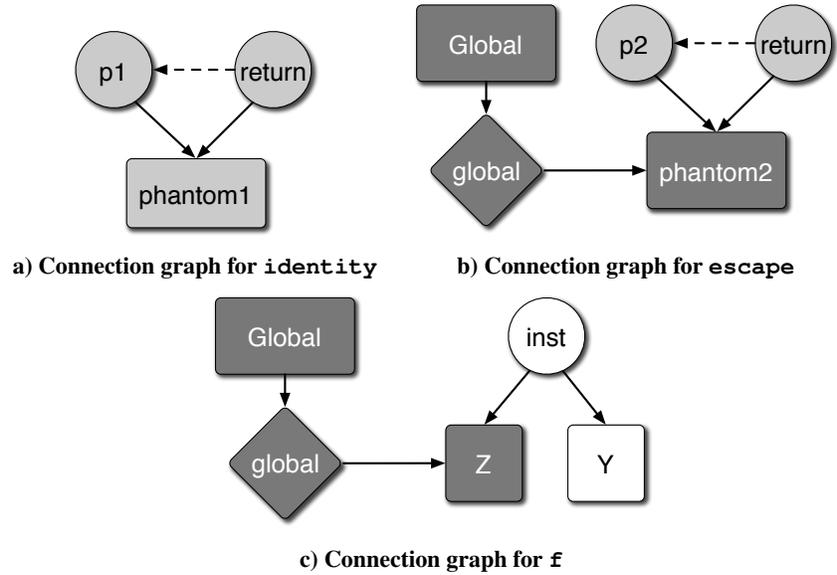


Figure 1: Summary connection graphs for methods in the example program

2.1 Blended Escape Analysis

Escape analysis. Escape analysis computes bounds on the reachability of objects. An object is said to escape a method m if it is reachable beyond the lifetime of an invocation of m during which it is created. Similarly, an object escapes a thread t if it is reachable at any point from a reference outside of t . Escape analysis has traditionally been used to compute those objects that are addressable only during the lifetime of their allocating method or within their allocating thread. The former allows objects to be allocated on the run-time stack rather than in the heap, thus reducing heap fragmentation and garbage collection overhead; the latter enables optimizations to avoid costly synchronization operations.

Escape analysis examines assignments and uses of references to compute an *escape state* for each object. Each object can be assigned one of three possible escape states: *globally escaping*, *arg escaping* or *captured*. An object is marked globally escaping when it becomes globally reachable (e.g., by being assigned to a *static* field). Objects that are reachable through parameters or that are returned to caller methods are labeled *arg escaping*. Objects that don't escape are marked as *captured*. During the analysis, a given object can have different escape states in different methods along a call path in the program; however, all objects eventually either globally escape or become captured. We refer to the final escape state of an object as its *disposition*.

Choi *et al.* [7] defined a context-sensitive, flow-sensitive [24] escape analysis algorithm. It computes a *connection graph* for each method in a call graph that summarizes connectivity between objects and stores their escape states. Connection graph nodes represent objects, reference variables or fields, and edges represent points-to relationships between them. Intuitively, escape states are propagated backwards along call paths in the call graph, starting from leaf methods.

For example, consider the code shown in Listing 1. Method `identity` just returns its parameter. The escape analysis algorithm computes the connection graph for this method as follows. First, a node is created to represent the parameter `p1`, along with a *phantom* object node to represent all objects that could be passed to `identity` through `p1`. An edge is created between `p1` and `phantom1` to indicate that `p1` can point to `phantom1`. Because `p1` is an ex-

ternal reference, it is initially marked as being arg escaping (light gray). Next, the return statement on line 2 causes a `return` node to be created (and marked arg escaping because returned objects arg-escape the method). An edge is also added between the return node and `p1` to indicate that any object pointed to by `p1` can be returned from `identity`. Finally, the graph is simplified by making each reference node (e.g., `return`, local variable or parameter node) point directly to all objects that are transitively reachable from it. Redundant edges are also removed during this process, and escape states are propagated along the remaining edges. In this example, the `return` node is made to point directly to `phantom1`, and the now redundant edge between `return` and `p1` is removed (shown as a dashed edge in Figure 1). Propagating escape states causes `phantom1` to be marked arg escaping. Figure 1 a) shows the final connection graph for `identity`.

Method `escape` is similar to method `identity` except that it additionally assigns its parameter to a global field, causing it to escape globally. The initial connection graph for `escape` contains nodes `p2` and `phantom2` (similarly to the previous example). The field assignment statement on line 6 causes the field node `global` to be created, and made to point to `phantom2`. Note that for simplicity, static field accesses are modeled as instance fields of a global object that is initially marked as globally escaping (dark gray). The return statement for method `escape` is processed in the same way as in `identity`. When the resulting connection graph is simplified, however, note that `phantom2` is marked as globally escaping, as shown in Figure 1 b).

Method `f` makes calls to both `identity` and `escape` demonstrating how previously computed connection graphs are used at a call site. The escape analysis for `f` proceeds by first creating a node for local variable `inst`. When analyzing the `if` branch of the conditional statement, the analysis first creates an object node for `Y`. A mapping is then established between the parameters of the callee and the arguments in the caller. For instance, `phantom1` is mapped to `Y` in this case. A similar mapping is established between the return value and the `inst` variable. Processing the call to `identity` results in the addition of an edge between `inst` and the `Y` object in the connection graph for `f`. Note that arg escaping objects in the callee are marked as non-escaping in the caller during this process,

causing object Y to correctly appear as non-escaping (white) in the connection graph for f . The `else` branch of the conditional statement is processed similarly, and the final connection graph for f is shown in Figure 1 c). Note that object Z is marked as globally escaping since it was mapped to the globally escaping `phantom2` when processing the call to `escape`.

Blended escape analysis. As mentioned previously, we used a blended version of the Choi *et al.* escape analysis to approximate the *effective lifetime* of an object, that is, the time from object allocation to its last use [11]. We use IBM’s *Jinsight* tool to generate a dynamic call graph used as input to the blended escape analysis. The *Jinsight* profiler is routinely used within IBM for performance diagnosis. We extended the tool to generate various kinds of dynamic calling structures from existing traces. This ensures that our technique can easily be integrated in the normal performance understanding workflow.

In framework-intensive applications, object allocations frequently occur in low-level library methods that are used in many different contexts. We therefore extended the original analysis to maintain a distinct escape state for each object at every method in the call graph. This allows our blended escape analysis to distinguish between different escape behaviors along individual paths in the call graph. As will be shown, this additional information is useful for understanding program behavior and data manipulation, in contrast to previous uses of escape analysis.

Postprocessing connection graphs. The precision of the information in the connection graphs can be further improved by a mapping onto a program representation that retains richer calling context information than the dynamic call graph used in the escape analysis. A *dynamic calling context tree (CCT)* [2] is a context-sensitive calling structure in which method invocations are differentiated based on their call chain prefix. In other words, two invocations of the same method are considered equivalent if and only if they are the result of the same sequence of method calls starting at the program entry point. Note that the presence of recursion is a special case that introduces cycles in a CCT. In contrast, a call graph is a context-insensitive calling structure where the same node represents all invocations of a given method. For brevity, we use the term *context* to refer to a CCT node and simply refer to a call graph node as a *method*.

The postprocessing algorithm effectively overlays information from the connection graphs onto the CCT. This serves two main purposes: it provides more fine-grained information about instances at each context, and it allows behaviors that were merged in the blended escape analysis to be disambiguated. Note that a similar gain could be achieved by using different choices of calling structures for a blended analysis. As future work, we will study the impact of varying the level of context sensitivity in the calling structure representation on the cost and precision of the blended analysis.

The postprocessing phase of the blended escape analysis manipulates both static and dynamic object abstractions at the same time. For clarity, we refer to the dynamic abstraction of an object as an *instance* (i.e., an object that was dynamically allocated at runtime) and reserve the term *object* to denote abstract objects used by the static analysis, which are allocation sites in our analysis.

The postprocessing algorithm generates a *reduced connection graph* for each context in the CCT. The reduced connection graph for a given context contains only nodes of the original connection graph that correspond to abstract objects with *visible* instances in this context. An instance is visible at a given context if its allocation has been observed and the instance was not captured lower in the CCT. Each reduced connection graph node is decorated with

its associated instances. This is achieved by propagating visible instances backwards in the CCT.¹ The reduced connection graph also elides edges other than those that represent points-to relationships between objects through their reference fields.

The reduced connection graphs are used for two purposes. First, they allow the number of instances captured at each context to be computed, thus enabling the identification and ranking of the calling contexts according to their usage of temporaries. Second, they summarize the connectivity of objects at a given context using a simple and easily accessible representation. In practice, we have found that reduced connection graphs, unlike the original raw connection graphs, provide a good level of abstraction for understanding and manual exploration of temporary structures.

2.2 Optimized blended analysis

Declared Types. Due to the conservative nature of static analysis, it is common for edges that violate type assignment rules to be created in the connection graphs. To address this issue, we modified the escape analysis to take advantage of knowledge of declared types; thus, type-inconsistent edges are never added to the connection graph.² This optimization is well-known, and has been shown to significantly increase the precision and to reduce the execution time cost of points-to analysis [17].

Basic Block Pruning. Most applications only execute a very small portion of their source code during a single execution. Blended analysis exploits this observation by using a dynamic calling structure as a basis for the static analysis. This ensures that only methods that were executed are visited by the analysis, thus reducing the amount of interprocedural propagation. However, even methods that were exercised during an execution typically contain a significant number of unexecuted instructions. Based on this observation, we developed a new technique that employs the dynamic information collected in order to reduce the amount of intraprocedural work of the static analysis and to improve precision.

Our technique works by pruning a basic block from the control flow graph of a method if it can be shown that the block was never executed. Unexecuted basic blocks are identified using two kinds of dynamic information for each method, observed calls and allocated types of instances. The dynamic calling structure contains a list of observed targets for each executed method. We also annotate all nodes in the calling structure with a list of observed allocated types collected during profiling. Any basic block that contains a call site that does not match any observed target, or that contains an object allocation that did not execute, can be marked as unexecuted. The control flow graph (CFG) for this method can then be pruned by removing any path from entry to exit that includes at least one basic block that never executed. Pruning a path in the CFG removes all basic blocks that are not shared between this path and any other possibly executed path.

The amount of information collected at runtime is limited by our choice of profiler. While *Jinsight* provides the full calling context for each object allocation and method invocation, it does not record enough information to determine which allocation sites or call sites correspond to these events. This limitation of the profiler requires conservative assumptions to be made in cases where a given call or allocation could have originated from multiple sites in the same method.³ In such cases, we safely assume that all matching sites

¹Cycles in the CCT are handled in the propagation by fixed point iteration.

²In our experiments, we used declared types both in our original and pruned algorithm (described next).

³To address this limitation, we are investigating additional ways to build calling structures using lightweight bytecode instrumentation.

were potentially executed. This may force our analysis to consider unexecuted code, but it ensures analysis of all executed code.

Pruning unexecuted blocks is a technique that is generally applicable to any blended analysis. It is particularly compelling in the case of a flow-sensitive analysis that typically requires state to be maintained at each basic block in the CFG. Each pruned basic block therefore translates directly into memory savings in addition to reducing the overall amount of work to be performed by the analysis. Experimental results show that our pruning technique results in a significant scalability gain in our blended escape analysis. We defer a full discussion of these results to Section 3.4.1.

3. METRICS AND FINDINGS

In this section, we first present an overview of the metrics we define, and the sources of imprecision in measurement on blended analyses. We next describe the four benchmarks used in our empirical data gathering. Finally, we define each metric, discuss what it illustrates on the benchmark suite, and present our findings and interpretations. We illustrate key points with an actual example solution, obtained from one of the applications.

3.1 Metrics - Overview

Our blended analyses uncover as yet unexplained characteristics of framework-intensive applications. Therefore, new metrics needed to be defined to measure the effectiveness of the analysis and the data usages observed.

There were three major measurement goals for our new metrics: (i) to determine the effectiveness of the pruning technique on the blended analysis algorithm, (ii) to characterize the usage of temporary data structures in our 'typical' framework-intensive benchmarks, and (iii) to characterize these temporary data structures themselves. Each metric addresses one or more of these goals. Metrics covering goal (i) are comparative measures of the original versus the pruned blended algorithm. For example, we measure improvements in execution time and in the size of the program representations used for analysis; these are improvements in analysis *scalability*. Measuring the categorization of objects as captured or escaping allows estimation of *precision* improvements. Metrics covering goal (ii) capture properties of the execution related to the usage of temporaries. These include the escape categorization of an object, which can be mapped to its corresponding dynamic instances, and the distance from allocation to capture for each object. Metrics covering goal (iii) quantify the complexity of temporary data structures in terms of the numbers of instances contained, the number of types contained, the complexity of the object interconnections, etc.

3.2 Sources of imprecision

Ideally, the results of a blended analysis would perfectly capture the properties of an execution. In practice, however, there are two sources of imprecision that affect its results: static and dynamic.

Static analysis is often required to make conservative assumptions to ensure a safe solution. For example, in escape analysis objects may be conservatively classified as escaping when they are in fact captured, but the analysis is not precise enough to see this. Similarly, objects may appear to be reachable from a reference in a connection graph, when this can not occur during program execution. This imprecision in static analysis stems from the fact that it is impossible to determine in general the infeasibility of an arbitrary path in a static program representation [18]. We term this *static imprecision*.

Dynamic analysis also contributes imprecision to the analysis results. Dynamic imprecision occurs because either the level of

detail found in the execution trace adversely affects the precision of the analysis, or the aggregation of the program trace into a more scalable program representation results in loss of precision about the calling context of the data.

In the first case, as explained in Section 2, *Jinsight* does not include allocation site or call site information in the traces it generates. This requires conservative assumptions to be made when mapping target invocations to potential call sites and instances to possible allocation sites. Moreover, because multidimensional arrays in Java are represented as arrays of arrays, *Jinsight* only reports allocations of single dimension array types. Without information about allocation sites, it is therefore not possible to disambiguate between two instances of type `Object []` when, for example, a given method can allocate both `char [][]` instances and `int [][]` instances. Finally, *Jinsight* is sometimes unable to resolve array types correctly; such allocations are then reported as arrays of a special *unknown* type. Our analysis must therefore conservatively assume that any array type matches an array of *unknown* type.

In the second case, in blended analysis the execution trace is aggregated into a call graph (or a CCT), before being used by the static analysis. This aggregation can conflate some behaviors that never occur together in practice, by making some unexecuted call paths appear to be feasible. We term either of these cases *dynamic imprecision*.

3.3 Experimental Setup

For our experiments, we used two well-known framework-intensive applications: *Trade* and *Eclipse*. Our escape analysis is built using the WALA analysis framework.⁴ To obtain complete call graphs from the trace, all experiments were performed with an IBM JVM version 1.4.2 with the JIT disabled in order prevent method inlining at runtime. Note that different JIT implementations may provide more fine-grained control over the specific optimizations performed by the JIT, and may allow inlining to be disabled without requiring the JIT to be turned off completely.⁵ Our test machine is a Pentium 4 2.8 GHz machine with 2 GB of memory running the Linux kernel version 2.6.12.

Trade 6. We used version 6.0.1 of the *Trade* benchmark running on *WebSphere* 6.0.0.1 and *DB2* 8.2.0.⁶ The way in which the *Trade* benchmark interfaces with the *WebSphere* middleware can be configured through parameters. We experimented with three configurations of *Trade* by varying two of its parameters: the run-time mode and the access mode. The run-time mode parameter controls how the benchmark accesses its backing database: the *Direct* configuration uses the *Java Database Connectivity (JDBC)* low-level API, while in the *EJB* configuration database operations are performed via *Enterprise Java Beans (EJBs)*.⁷ The access mode parameter was set to either *Standard* or *WebServices*. The latter setting causes the benchmark to use the *WebSphere* implementation of web services (e.g., SOAP) to access transaction results. All other parameters retained their default values.

Each of the three benchmarks was warmed up with 5000 steps of the built-in scenario before tracing a single transaction that retrieves a user's portfolio information from a back-end database into Java objects. Our analysis was applied to the portion of that transaction that retrieves nine holdings from a database. The warm-up phase is necessary to allow all necessary classes to be loaded and caches

⁴<http://wala.sourceforge.net/>

⁵Instrumentation-based profiling techniques generate accurate call graphs even in the presence of inlining.

⁶*Trade*, *WebSphere* and *DB2* are available to academic researchers through the IBM Academic Initiative.

⁷*Trade 6* uses the EJB 2 framework.

Benchmark	Alloc'd Types	Alloc'd Instances	Methods	Calls	Max Stack Depth
<i>Direct/Std</i>	30	186	710	4,484	26
<i>Direct/WS</i>	166	5,522	3,308	127,794	53
<i>EJB/Std</i>	82	1,751	1,978	60,936	62
<i>Eclipse</i>	168	53,191	1,411	1,081,927	53

Table 1: Benchmark characteristics

Benchmark	Pruned BBs	Running time (h:m:s)		Speed-up
		Orig	Pruned	
<i>Trade Direct/Std</i>	38.8%	0:00:22	0:00:11	2.0
<i>Trade Direct/WS</i>	36.0%	3:01:52	0:19:31	9.3
<i>Trade EJB/Std</i>	41.0%	6:49:54	0:13:50	29.6
<i>Eclipse JDT</i>	30.9%	43:13:20	2:01:39	21.3
Average	36.7%			15.6

Table 2: Pruning effects

to be populated. Tracing the benchmark in a steady state is more representative of the behavior of real Web applications.

Eclipse JDT Compiler. We experimented with the Eclipse JDT compiler by tracing a single regression test from the *XLarge* test suite. We ran the *XLarge* suite using JUnit and traced the execution of the eighth test in the suite, which compiles a complete Java file. Because a new compiler is instantiated before each test is executed, the Eclipse JDT trace does not correspond to a steady state of the application. For example, required classes were loaded during the execution of the test.

Because the *Trade* application consists of a relatively small user code that interacts with a large amount of framework and library code, the three configurations of the same application have very different properties and behavior in practice. Therefore, we use these three configurations as different benchmarks, as have other researchers [27]. These differences are confirmed by the data in Table 1 that presents benchmark characteristics. Columns 2 and 3 show the total number of distinct types that were allocated and the total number of instances (i.e., observed object allocations), respectively. The last three columns show the total number of distinct methods executed, the total number of method invocations and the maximum depth of the call stack during execution. The results clearly show a large variation in the characteristics of each benchmark, illustrating the differences between the libraries used by the three *Trade* benchmarks. The results also attest to the complexity of these framework-intensive benchmarks. Note that when the observed scenario in *Trade Direct/WS* runs, it allocates 166 types of objects and experiences call stack depths of over 50.

3.4 Empirical Results and Interpretation

In this section we define new metrics that are useful for computing the effects of our algorithm optimizations, identifying temporary objects and data structures, and characterizing them and their uses. For each metric, we present empirical data from our benchmark suite, and discuss specific findings and observations based on the results. We believe these to be the *first* such descriptive metrics for temporaries in framework-intensive Java applications.

3.4.1 Pruning Effects

In order to measure the impact of the pruning technique on the scalability of the analysis, we compute two metrics:

Metric 1: Pruned basic blocks

The percentage of basic blocks in the entire application that were marked as unexecuted and therefore pruned away.

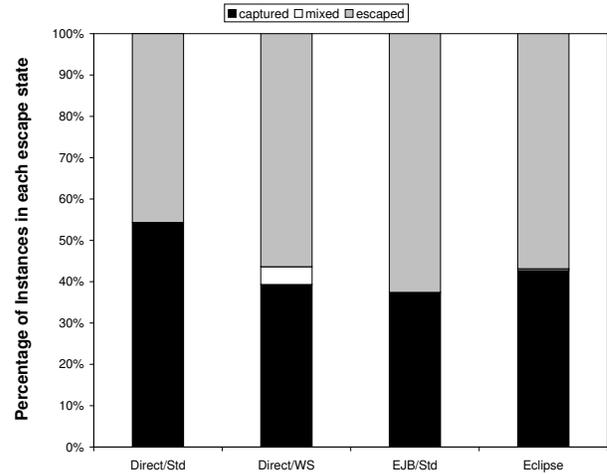


Figure 2: Disposition breakdown (by instances)

Metric 2: Execution time

The amount of time required to compute the escape analysis results. To show improvements between algorithms and make the results comparable across different benchmarks, this metric only includes the analysis phase of the algorithm (i.e., excludes the time required to perform common operations such as reading the dynamic call graph from a file or outputting the analysis results).

Table 2 shows the percentage of pruned basic blocks for all benchmarks. Our technique pruned on average 37% of all basic blocks in an application. Since most flow-sensitive analyses need to associate state with each basic block, this pruning directly translates into a significant reduction of the memory footprint of the representation of the application. Also, removing basic blocks implies that the algorithm has less code to analyze, and thus can be expected to run faster.

In order to study the effect of pruning on the overall scalability of our blended escape analysis, we recorded the total time required to perform the analysis for each benchmark. Table 2 shows the running times for both the original and pruned analyses. Pruning has a clear impact on the analysis time for all benchmarks, and achieves speedups (i.e., with respect to the original algorithm) of between 2 and 29, with an average speedup of 15.6.⁸ Note that in the remaining sections, for metrics whose specific purpose is characterization, we only report the results of the pruned algorithm.

3.4.2 Disposition

Recall from Section 2 that our blended escape analysis assigns an escape state to each object at every node in the call graph. Every object also receives a *disposition*, or final escape state. The disposition of an object induces the disposition of its corresponding instances (i.e., as determined by the postprocessing). Without dynamic imprecision, every instance would either globally escape or be captured. However, dynamic imprecision sometimes introduces ambiguity regarding the path in the dynamic CCT traversed by an instance. In such cases, the postprocessing algorithm is forced to label some instances as both escaping and captured, a state henceforth referred to as *mixed*.

We compute two metrics that relate to disposition:

Metric 3: Disposition breakdown (by instances)

The percentage of instances whose disposition is *globally escap-*

⁸We are working on optimizing our implementation, and expect to reduce both the unpruned and pruned analysis times significantly.

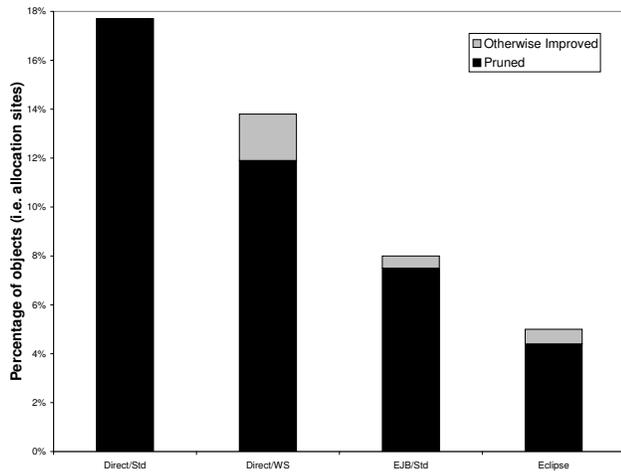


Figure 3: Disposition improvement (by objects)

ing, captured or mixed. This metric is used to characterize the usage of temporaries.

Figure 2 shows disposition breakdown results for all the benchmarks for both algorithm versions. Less than 5% of instances fall in the mixed category across all benchmarks, showing that a vast majority of objects can be categorized as either captured or globally escaping, even in the presence of dynamic imprecision. More importantly, Figure 2 also shows that about 43% of all instances never escape globally, thus indicating that temporaries account for a significant portion of all instances.

Metric 4: Disposition improvement (by objects)

The percentage of objects whose disposition is improved by the pruning algorithm. The disposition of an object is considered to be improved (i) if its corresponding allocation site is found to be unexecuted and is pruned away, or (ii) if the object is assigned a more precise disposition. Note that an object that was labeled as globally escaping by the original algorithm may have its disposition improved to mixed, if it is shown to be captured on at least one path in the calling structure. This metric is used to show precision improvements due to CFG pruning.

Figure 3 displays the disposition improvement results tallied by objects. For each benchmark, the bottom portion of the bar represents objects corresponding to allocation sites that the pruning technique marked as unexecuted that were therefore pruned away. The top portion of the bar shows the percentage of objects for which the pruned analysis computed a more precise disposition. The results show that between 5% and 17.7% of the objects benefit from the pruning algorithm. Identification of unexecuted allocation sites is responsible for 86% to 100% of the improvements, and it is clearly the most effective aspect of the pruning algorithm. However, Figure 3 also shows that a small number of objects are assigned a more precise disposition, up to 2% in the case of the *Trade Direct/WS* benchmark.

3.4.3 Capturing depth

The capturing depth metric is a measure of the nature of the individual regions in the program calling structure that use temporaries.

Metric 5: Capturing depth

The capturing depth of an instance is the length of the shortest acyclic path from its allocating context to its capturing context. An instance may have multiple capturing depths if it is captured by more than one context; in this case, the instance contributes

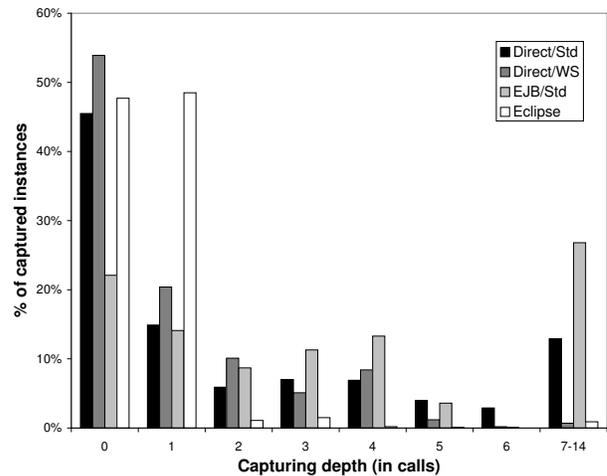


Figure 4: Capturing depth histogram

to the overall capturing depth distribution once for each depth. This metric is used to characterize the usage of temporaries.

In essence, capturing depth denotes a lower bound on the number of method calls during which an instance is live; as such, it helps to describe the program region that uses the instance. Deeper regions define larger subtrees in the calling structure (i.e., each subtree has at least $d + 1$ contexts for a capturing depth d). Temporaries that are constrained to small regions may be easier targets for compiler optimizations. Temporaries that belong to deep regions are likely to cross framework boundaries and may require interprocedural code specialization optimizations. Deeper regions also make it more difficult for developers to identify the source of potential performance problems. For these reasons, we are interested in knowing how the use of temporaries is distributed in a given application.

Figure 4 shows the distribution of capturing depths for all benchmarks (i.e., depth of 1 means 1 call). For the *Trade* benchmarks, between 25% and 64% of instances are captured more than 1 call away from their allocating method. Therefore, a local escape analysis (i.e., within a single method) is likely to be ineffective at identifying temporaries, even if one level of inlining is used. An interprocedural escape analysis is necessary.

The capturing depth metric also shows that, for some benchmarks temporary usage is very complex. For example, in the *Trade EJB/Std* benchmark, more than 26% of instances are captured 7 or more calls away from their allocating method. Note that temporaries can also be passed down transitively to callees, so the capturing depth only represents a lower bound on the number of methods involved in manipulating a particular instance.

3.4.4 Illustrative Example

Figure 5 shows an example of a CCT context that captures instances allocated at many different calling depths. The figure shows the escape behavior of a single call to the `getConnection` method of the J2EE data source layer. Data sources are an abstraction above the database access layer, enabling such features as connection pooling and precompiled query caching. This example is typical of how the layering of frameworks can cause a simple function, in this case obtaining the use of a locally cached connection, to lead to the costly initialization of complex temporary structures.

In order to access the connection, a `Subject` authentication structure is built, using Java's standard security framework. The figure shows that this is a temporary structure, not visible beyond `getConnection`. The `Subject` object is allocated five calling levels

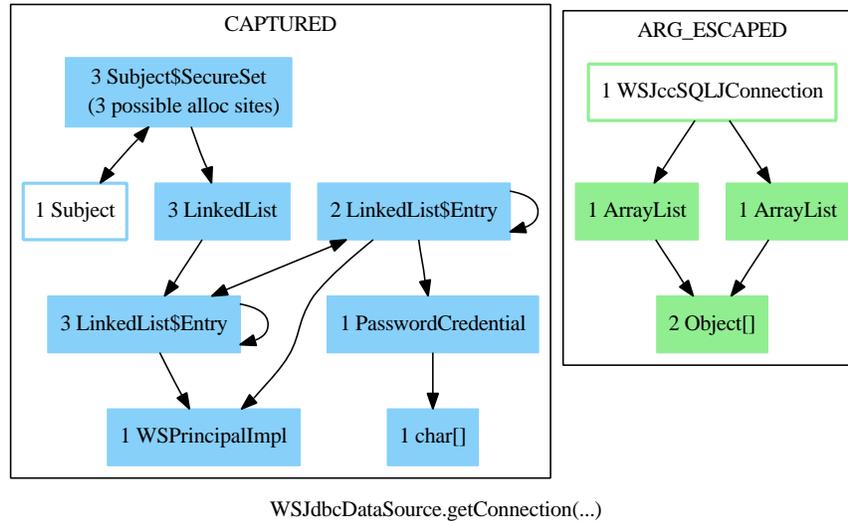


Figure 5: From *Trade Direct/Std*: two data structures created when obtaining a data source connection.

away from `getConnection`. Its internal infrastructure is allocated even further away, the result of additional division of labor. For example, the `Subject` constructor allocates three instances of an internal set class, each of which delegates its storage to a `LinkedList`. Each of these always allocates a sentinel `LinkedList$Entry`, now eight call levels from where the structure creation was initiated. Note that such low-level library objects may appear in other contexts as well, thus illustrating the importance of maintaining distinct escape behavior for a given object at different contexts.

3.4.5 Concentration

The previous metric, capturing depth, is a descriptive measure of the individual program regions that use temporaries. Here we look at how the object churn costs are distributed across such regions in the application scenarios we are analyzing. We would like to understand whether object churn behavior is typically concentrated in a few regions, or is spread out across many regions. This information can guide us toward solutions, for example, showing whether diagnosis tools that help a user find a few hot regions of object churn would be sufficient, or if problems are so widespread they can only be handled by automated optimizations or better API design practices.

Metric 6: Concentration

The concentration metric reports the percentage of captured instances that are explained by $X\%$ of the top capturing methods. This metric uses percentages in order to be comparable across benchmarks. This metric is used to characterize the usage of temporaries.

Figure 6 shows the results for each of the benchmarks, reporting concentration using 5%, 10% and 20% for X . The results indicate that *about half of the temporaries (on average) are explained by the top 5% of the capturing methods*. In the case of the *Eclipse JDT Compiler*, the top two capturing methods account for over 92% of the captured instances! This is largely due to the fact that this benchmark first populates a cache by loading and parsing classes from the disk. Other benchmarks show a different concentration of temporaries. For example, in the case of *Trade EJB/Std*, 34% of the instances are explained by the top 5% of the capturing methods, or 2 out of the 44 capturing methods. Note that our analysis already

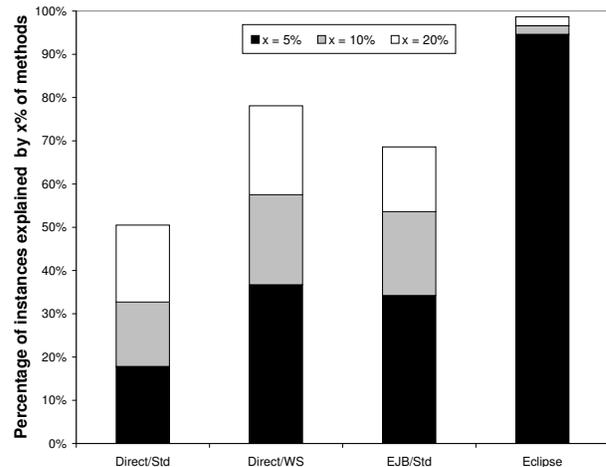
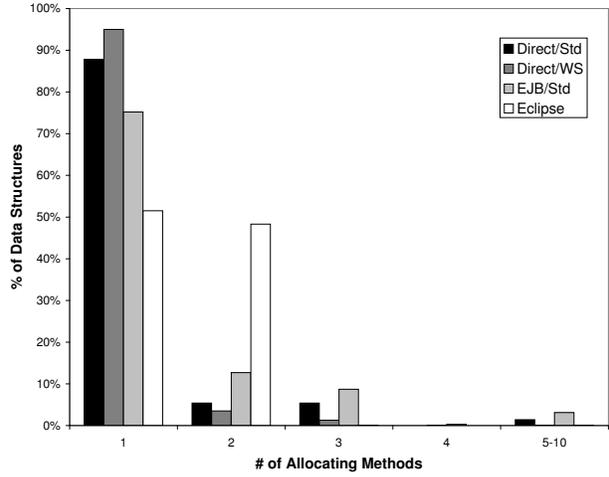
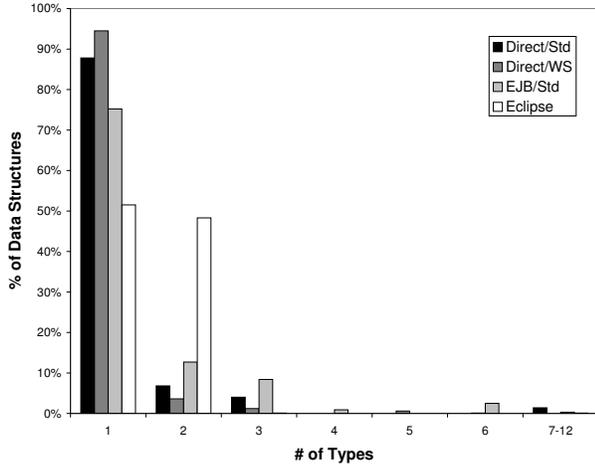


Figure 6: Concentration

focuses attention from 1979 observed methods down to just 44 capturing methods. In all benchmarks, the top 20% of methods explain the majority of captured instances (between 50.5% and 98.7% for *Trade Direct/Std* and *Eclipse JDT Compiler*, respectively). In *Trade EJB/Std*, the top 20% of capturing methods explain 68% of the instances.

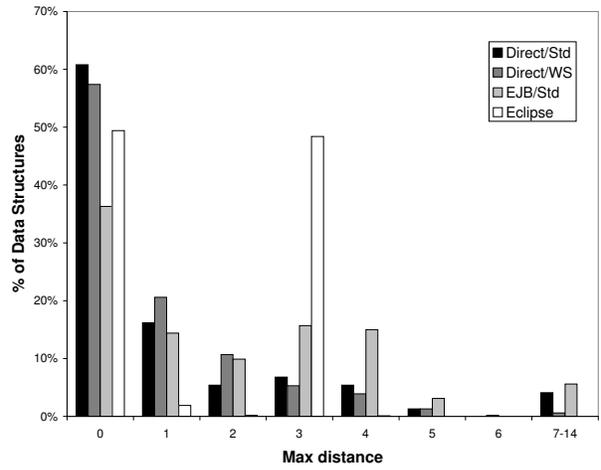
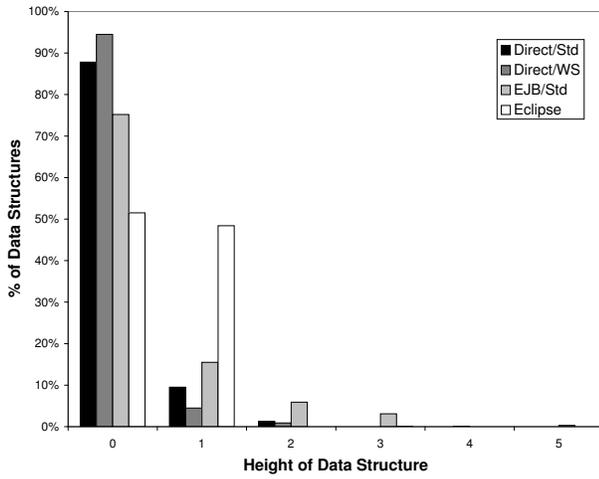
3.4.6 Complexity of Data Structures

Temporaries often are organized into complex data structures in order to perform a given task. Such structures can be expensive to create because they involve both the cost of allocation and initialization of constituent instances and the cost of linking instances together into a data structure. For this reason, we are interested in characterizing the complexity of temporary data structures in a given application, using the reduced connection graphs to identify these temporary structures. By calculating the number of instances in a data structure, we can estimate the savings possible through optimization of its usage. Certain optimizations may be aimed at temporary structures as a whole, such as pulling constant structures



a) # of types

b) # of allocating methods



c) Height of data structure

d) Maximum capturing distance

Figure 7: Complexity of data structures (by occurrences)

out of a loop, pooling similar structures that have expensive construction for reuse, or specializing structures for a specific usage pattern.

In order to characterize temporary data structures, we need a clear definition of what constitutes a data structure in the reduced connection graph of a calling context. A *data structure* in the reduced connection graph of a calling context in the CCT consists of a root (i.e., an object with no incoming edges)⁹ and those nodes reachable from it that have the same escape state. Because we are interested in characterizing temporaries, we compute metrics only over captured data structures.

Given the dynamic imprecision in the profiling data about executed allocation sites of the same type within the same method, we chose to merge objects in the reduced connection graph that are indistinguishable dynamically. This means that a set of objects representing allocation sites of the same type in the same method with the same connectivity in the reduced connection graph are merged. This transformation prevents many of the issues of double-counting instances when they cannot be mapped to a unique allocation site. In all our discussions that follow, we use the term *object* to refer to

⁹For strongly connected components in the connection graph, we select a node with no back edges as the root.

an object in the reduced connection graph *after* this merging transformation has been applied.

Because every object has a corresponding set of associated instances, we can compute of the number of *occurrences* of a data structure (i.e., the number of instances of its root) as well as the total number of instances comprising the data structure (i.e., the sum of the number of instances for each object in the data structure).

Note that a given object may appear as part of multiple data structures. For instance, a `char[]` array that is created as part of a `String` may, at a later time, become part of a `StringBuffer` object as a result of a `toString()` operation.

We investigate the complexity of data structures and characterize them by computing the following four metrics:

Metric 7: # of types

The number of distinct object types in each data structure. The more types a data structure contains, the more complex it is.

Metric 8: # of allocating methods

For each data structure, the number of distinct methods that allocate instances that are part of this data structure. The complexity of a data structure increases with the number of allocating methods.

Metric 9: Height of data structure

The length of the longest acyclic path in the reduced connection graph from a given data structure root to any other object in the data structure. The complexity of a data structure increases with its height.

Metric 10: Maximum capturing distance

The maximum capturing depth of any instance in the data structure. This metric calculates the longest capturing call chain corresponding to an instance contained in the data structure. Note that the capturing distance (like the capturing depth metric) computes a lower bound on the number of calls traversed during the use of the data structure, since data structures may be passed down to callees as well during execution.

All four metrics can be reported in aggregate form over all data structures (i.e., by occurrences) or, alternatively, over all instances in data structures (i.e., by instances). Intuitively, metrics computed by occurrences capture properties of the data structures themselves. Metrics computed by instances aim to answer questions regarding the importance of certain data structures weighted by the number of the instances they explain. For example, metrics computed by instances could be useful to determine how profitable a specific compiler optimization could be.

Figure 7 (a-d) presents the results for all the data structure metrics. The figure shows that there is a *strong similarity* between the number of types (a), the number of allocating methods (b), and the height of data structures (c). This correlation suggests that most methods creating part of a data structure allocate instances of a single type during execution. Figure 7 also shows that the *Trade EJB/Std* benchmark uses more complex temporary structures than the other benchmarks. For example, more than 12% of its temporary data structures have 3 or more types, as compared to at most 5% for the other benchmarks.

These results might at first suggest that most data structures are relatively simple in structure; however, the instance-weighted metrics (not shown) reveal a more nuanced picture. For *Trade EJB/Std*, 28% of captured instances occur in data structures containing 6 or more types. Even in the simplest benchmark, *Trade Direct/Std*, 25% of the instances are from temporary structures with at least 3 types. Finally, Figure 7 d) reveals a complex pattern of usage of temporary structures. For example, in *Trade EJB/Std*, 49% of structure occurrences (with 73% of the instances) contain at least one instance allocated at least 2 calls away from the data structure's capturing context. Even the smaller *Trade Direct/Std* shows similar usage, with 23% of data structure occurrences (with 43% of the instances) containing instances allocated at least 2 calls away from their use.

The example in Figure 5 shows a captured `Subject` structure that has a height of six and consists of instances of seven distinct types, allocated by six different methods. (In the figure, the inferred root of each structure is drawn with a white background). A temporary structure this complex points to an expensive initialization process. The trace indeed shows that initialization of this structure requires 109 method calls. Looking beyond the region we analyzed, the trace also shows that each transaction calls `getConnection` five times, thus initializing and throwing away five occurrences of this same `Subject` structure, all representing the same user and password information. This suggests the possible optimization of saving the `Subject` structure within each transaction, or perhaps even across transactions for the same user. It may also suggest that the Java authentication infrastructure consider using a simpler data structure, for example one specialized for the common case of a single user id and password. These are pos-

sibilities that would not be apparent by looking at each object in isolation.

Figure 5 reveals an additional cost of accessing a connection: the creation of an occurrence of `WsJccSQLJConnection`, a wrapper structure introduced by the data source layer. The reduced connection graph of a higher-level CCT context (not shown) reveals that this structure is temporary as well, and that this context additionally makes use of a similar temporary structure for wrapping compiled queries. Note that one purpose of the data source layer is to amortize the cost of using database resources. Yet it is often the case that additional resources are expended just to make use of such an abstraction layer. The data structure metrics point to costly initialization for these wrapper structures, since each is the result of three or more allocating methods. This suggests that a higher-level optimization is possible, saving the data source connection within each transaction (since the data source is fixed). This would avoid the construction of the `Subject` and wrapper structures.

Finally, from a program understanding standpoint, viewing `WS-JccSQLJConnection` and its infrastructure as a single data structure, as in Figure 5, helped reveal another anomaly. Upon examination of the role of the two `ArrayLists` within a connection wrapper, it was discovered that the same field was erroneously populated with a new `ArrayList` twice: once by the concrete class, and once by its superclass.

4. RELATED WORK

The blended analysis paradigm clearly is related to all existing static and dynamic analyses. Given space limitations, this section will focus only on the most relevant areas of related research: (i) studies of framework-intensive software, (ii) previous combinations of static and dynamic analyses, and (iii) previous characterizations of the use of temporaries in large Java codes.

Studies of framework-intensive systems. Previous analyses of framework-intensive applications used dynamic analysis to diagnose and optimize performance problems and to aid in understanding the data structures used.

Ammons *et al.* [3], built the dynamic analysis tool *Bottlenecks* to explore execution profiles to find performance bottlenecks. Experiments with *Bottlenecks* on *Trade 3*, the *SPECjAppServer2002* and *XML*, demonstrated the complexity of these frameworks in terms of their calling structure, by measuring the maximum and mean depths of call paths (i.e., 77 max, 34 mean depth) and out-degree of method nodes in the dynamic call graph (i.e., 74 max, 1.89 mean degree).

Srinivas *et al.* [27] designed a dynamic analysis technique that identifies 'interesting method invocations', that is, those that account for a specified cumulative percentage of execution cost, in components selected by the user. The technique was tested successfully on e-commerce applications built on *Websphere* and on parts of the *Eclipse* IDE.

Mitchell *et al.* [21], constructed a characterization of the runtime behavior of framework-intensive systems, by combining dynamic analysis with manual inspection of source code. This characterization was used to organize the aggregation of operation costs in terms of method calls and object creations. The emphasis was on developing high-level abstractions of behavior that allow the recognizable grouping of observed method calls to better understand their function and their cost.

Each of the above works introduces a novel way to summarize a complex execution, in order to highlight a small number of regions to study. In our work we use object lifetime information to identify expensive regions of temporary usage. Recent work by Shankar *et al.* [26] also employs object lifetime information to address regions

of object churn. This work uses dynamic sampling to identify regions where it is profitable for JIT compilers to target aggressive inlining, and thus widen the scope of stack allocation and related optimizations.

Combined static and dynamic analyses. Static and dynamic analyses have been combined to solve a wide range of problems, including an early overview paper [12]. Typically, the static analysis results direct where the dynamic analysis should be applied (e.g., the placement of code instrumentation by a compiler). In contrast, the blended analysis paradigm uses dynamic analysis results to guide a subsequent static analysis, that may itself use dynamic information to achieve greater precision. Here, we will only discuss examples of the latter combination (i.e., dynamic analysis followed by static). Another difference in combined analyses is seen in the close or loose coupling between the different types of analyses used. Blended analysis is closely coupled; other analysis combinations work in a more pipelined or loosely coupled fashion, with the results of one analysis providing the input to the next analysis phase.

Gupta *et al.* [16] used dynamic information – observed breakpoints at branches and procedure calls/returns – to prune infeasible control flow while calculating a static slice to explain program behavior for a specific execution. Similarly, in model checking C programs, Groce *et al.* interpreted failure traces by identifying a subset of executions consistent with the trace, and then slicing the code while eliminating portions that were inconsistent with the trace, thus potentially increasing the precision of the slice [15]. These uses of dynamic analysis to enhance the precision of a subsequent static analysis are similar to the approach of blended analysis.

Mock *et al.* [22] designed a static slicing algorithm for C programs which used observed dynamic points-to relations instead of computed static points-to relations when forming the slices. Their findings were disappointing as they only improved the static slices for programs with function pointer references which could thereby be exactly resolved. In this case, the dynamic information didn't improve the precision of the static slicing of C programs.

Orso *et al.* [23] designed a change impact analysis algorithm that given a program change at c , (i) calculates the set of tests which execute c (i.e., dynamic information) and (ii) calculates the forward slice using c as the slicing criterion. The results of (i) and (ii) are intersected to form the 'impact set', the set of nodes which are affected by the program change at c . Here, the static analysis improves the relevance of the dynamic analysis information reported, but the combined result is better than either result viewed singly.

Godofroid *et al.* performed a symbolic execution (i.e., static analysis) on a test execution path (i.e., dynamic analysis), in order to use the path condition constraints to generate test cases that would explore alternative paths [14]. If the path condition constraint problem is not solvable, random concrete values are substituted for symbolic values to allow solution. This main idea has been expanded by Sen *et al.* to form a basis for *concolic testing* methodologies which use both symbolic execution and substitution of concrete values when necessary [25]. The similarity to blended analysis is that a dynamic execution path is being explored by a static analysis (i.e., symbolic execution).

Recently there have been several explorations of loosely coupled combined analyses. The *Check 'n' Crash* tool [8] provides dynamic testing of the errors/warnings reported by *ESC/Java* [13], in order to filter out false positives. This process consists of a static analysis whose results are checked by a subsequent dynamic analysis. Similarly, *DSD-Crasher* [9] runs *Daikon*, a dynamic analysis tool that finds program invariants, to provide additional assumptions to *Check 'n' Crash* to help it generate the tests to check the errors

reported by *ESC/Java*. This newer process consists of: dynamic analysis, static analysis, dynamic analysis.

Tomb *et al.* [28] try to find errors in programs similarly through a combination of loosely coupled static and dynamic analyses. A variably interprocedural symbolic execution analysis is used to explore Java program state on interprocedural paths expanded to a specific max call depth associated with each method. For program states that might result in a run-time exception, the associated constraints gathered by this analysis are then solved to find test inputs that will expose the possible error on a program execution. Empirical experiments demonstrate the utility of this approach.

Artzi *et al.* described various pipelined combinations of static and dynamic mutability analyses for Java method parameters [4]. Empirical results obtained showed that a pipelined combination analysis can exceed the accuracy of a more complex static analysis.

Characterizations of data structures in Java codes.

A key contribution of blended escape analysis is its ability to characterize the use of temporaries in framework-intensive systems. Mitchell [19] developed a set of graph transformations that allow identification of the most important data structures of a framework-intensive system from an execution heap snapshot. Clever graph reductions were applied repeatedly to obtain the key data structure relations, in terms of the nodes in a backbone of a graph that represents thousands of actual object instantiations. Mitchell and Sevtitsky further classified the data structures in these applications as *healthy* or not, depending on the amount of actual data contained and the structural overhead of storing that data [20]. In this work the summarization mechanism used is based on object ownership and structural properties, deduced from the results of a dynamic analysis. In contrast, in our work data structures are summarized by their lifetime as reflected by their escape state, and their connectivity, both derived from a blended escape analysis.

Blackburn *et al.* compute dynamic metrics about objects and their lifetimes to validate the diversity of the DaCapo benchmark suite of Java programs, as compared to the SPEC Java benchmarks [5]. Object lifetimes in the benchmarks are measured by heap examination during garbage collections. The unit of 'distance' in a data structure here is heap cross-generational distance between objects in the same data structure, whereas the distances associated with data structures in our studies correspond to the levels of calling context separating allocation and capture.

Buytaert *et al.* [6] estimated the possible payoff of avoiding object creation in Java programs, by eschewing the need for temporaries through refactoring. Temporaries were identified by instrumenting the JVM to keep information about each object's allocation site and its last use. The VM-specific methodology was only tested on small benchmarks, so its utility on complex, framework-intensive codes is not clear.

5. CONCLUSIONS

The growing popularity of framework-intensive applications has led to a new kind of performance problem associated with the creation, initialization and use of temporary data structures. Understanding the contributing factors to excessive use of temporaries is critical to being able ultimately to fix these performance problems. Addressing this issue, we have presented an optimized blended escape analysis with improved precision and increased scalability, demonstrated by our empirical data. We have defined new metrics that specifically characterize both the usage and complexity of temporary data structures; we have applied them to four framework-intensive benchmarks and reported our findings. Further, we have demonstrated the effectiveness of our approach by explaining a problematic scenario from one of our benchmarks.

6. REFERENCES

- [1] W. Alexander, R. Berry, F. Levine, and R. Urquhart. A unifying approach to performance analysis in the Java environment. *IBM Systems Journal*, 1:118–134, 2000.
- [2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 85–96. ACM Press, 1997.
- [3] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *Proc. of the European Conf. on Object-oriented Prog. (ECOOP)*, 2004.
- [4] S. Artzi, M. D. Ernst, D. Glasser, and A. Kiezun. Combined static and dynamic mutability analysis. In *Proc. of the 22nd IEEE/ACM Intl. Conf. on Automated Soft. Eng.*, pages 104–113, 2007.
- [5] S. Blackburn, R. Garner, C. Hoffmann, A. Khan, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. V. Dincklage, and B. Wiederemann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Prog. Syst., Langs, and Appls (OOPSLA)*, pages 169–190, Oct. 2006.
- [6] D. Buytaert, K. Beyls, and K. D. Bosschere. Hinting refactorings to reduce object creation in Java. In *Proceedings of the fifth ACES Symposium*, pages 73–76, 1 2005.
- [7] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. on Prog. Lang. and Sys.*, 25(6):876–910, 2003.
- [8] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. of the 27th Intl Conf. on Soft. Eng. (ICSE)*, 2005.
- [9] C. Csallner and Y. Smaragdakis. DSD-crasher: A hybrid analysis tool for bug finding. In *Proc. of ACM SIGSOFT Intl Symp. on Soft. Test. and Anal. (ISSTA)*, pages 245–254, 2006.
- [10] W. DePauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlassides, and J. Yang. Software visualization: State of the art survey. In *LNCS 2269*, 2002.
- [11] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07: Proc. of the Intl Symp. on Soft. Test. and Anal.*, pages 118–128, 2007.
- [12] M. Ernst. Static and dynamic analysis: Synergy and duality. In *Proc. of the Wksp on Dyn. Anal.*, 2003.
- [13] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 234–245, 2002.
- [14] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 2005.
- [15] A. Groce and R. Joshi. Exploiting traces in program analysis. In *Proc. of Intl Conf. on Tools and Algs for the Constr. and Anal. of Sys. (TACAS)*, 2006.
- [16] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Trans. on Soft. Eng. and Meth.*, 6(4), Oct. 1997.
- [17] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *12th Intl Conf. on Comp. Const. (CC'03)*, volume 2622 of *LNCS*, pages 153–169, April 2003.
- [18] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.
- [19] N. Mitchell. The runtime structure of object ownership. In *Proc. of the European Conf. on Object-oriented Prog. (ECOOP)*, 2006.
- [20] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Prog. Syst., Langs, and Appls (OOPSLA)*, pages 245–260, Oct. 2007.
- [21] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *Proc. of the European Conf. on Object-oriented Prog. (ECOOP)*, 2006.
- [22] M. Mock, D. Atkinson, C. Chambers, and S. Eggars. Improving program slicing with dynamic points-to data. In *Proc. of the 10th Symp. on the Found. of Soft. Eng.*, 2002.
- [23] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of European Soft. Eng. Conf. and ACM SIGSOFT Symp. on the Found. of Soft. Eng. (ESEC/FSE'03)*, Helsinki, Finland, September 2003.
- [24] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *invited paper in the Proc. of the Twelfth Intl Conf. on Comp. Constr.*, pages 126–137, April 2003.
- [25] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the Conf. on the Found. of Soft. Eng.*, 2005.
- [26] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM Press, 2008. (To appear).
- [27] K. Srinivas and H. Srinivasan. Summarizing application performance from a components perspective. In *Proc. of the ACM SIGSOFT Conf. on Found. of Soft. Eng.*, pages 136–145, September 2005.
- [28] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *Proc. of the ACM SIGSOFT Intl Symp. on Soft. Test. and Anal. (ISSTA)*, pages 97–107, Jul. 2007.