Blended Analysis for Performance Understanding of Framework-based 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

This paper defines a new analysis paradigm, *blended program analysis*, that enables practical, effective analysis of large framework-based Java applications for performance understanding. Blended analysis combines a *dynamic* representation of the program calling structure, with a *static* analysis applied to a region of that calling structure with observed performance problems. A blended escape analysis is presented which enables approximation of object effective lifetimes, to facilitate explanation of the usage of newly created objects in a program region. Performance bottlenecks stemming from overuse of temporary structures are common in framework-based applications. Metrics are introduced to expose how, in aggregate, these applications make use of new objects. Results of empirical experiments with the *Trade* benchmark are presented. A case study demonstrates how results from a blended escape analysis can help locate, in a region which calls 223 distinct methods, the single call path responsible for a performance problem involving objects created at 9 distinct sites and as far as 6 call levels away.

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

Commercial object-oriented programs are commonly built by integrating numerous layers of libraries and frameworks. These *framework-based* systems may be web server applications (e.g., using J2EE, web services, and portal frameworks), or client applications (e.g., using Eclipse). In these systems, understanding performance problems can be difficult for a number of reasons. Typically, problems are not localized in a few hot methods; more often there are patterns of problematic activity spanning many frameworks, the combined result of design choices in each framework [20]. To the user seeking to understand its performance, an application resembles an *iceberg*, where only a small portion of the code is familiar, and performance consequences are hidden under many layers of unfamiliar code below. The scale of activity (e.g., number of method calls, number of objects created) adds to the difficulty of understanding even simple features. In one study of framework-based applications [24], in the simplest version of a stock trading application, a simple transaction that reads 10 records from an external database required 28,747 calls, at average calling depth of 39. In our experience, industrial applications are often even more layered;[1] thus, locating performance problems and understanding them well enough to ameliorate them, a difficult task even for experts, requires analysis-based tool support.

Motivating problem. *Object churn* (i.e., the creation of a high volume of temporary objects) is one factor in many performance problems in framework-based systems. In addition to object allocation and garbage collection costs, object initialization is a source of significant execution activity. It is common for temporaries to occur as mini-data structures (i.e., groups of connected objects), built up with much effort only to be thrown away shortly thereafter. Initialization may involve long chains of calls across multiple frameworks, and may include the creation of *subordinate* temporaries, used only for the construction of slightly longer-lived data. [2] In a typical example from an industrial application we analyzed, a loop reads a list of dates; each iteration builds a temporary Java `SimpleDateFormat` to parse each input, even though they are all in the same format. Each `SimpleDateFormat` is a structure of 9 or more instances from 7 distinct classes; building one costs 511 calls and creates 16 subordinate temporaries in the process.

Current profiling tools (e.g., *Jinsight* [9], *HPROF* [3], *ArcFlow* [1]), provide some valuable information about new objects, such as the context in which they are created and whether they survive a garbage collection; however, they provide limited information about how these objects are used. Knowing what happens to a new object, particularly specific information about its *effective lifetime* (i.e., its dynamic scope)[4] can be helpful for understanding performance in

*This work was funded in part by IBM Research.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

[1] Note: Sevitsky is a member of a group at IBM Research that has diagnosed performance problems in dozens of framework-based industrial applications.

[2] We informally term temporaries *subordinate*, to mean their only purpose is to facilitate the initialization of other temporaries.

[3] http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

[4] Effective lifetime means the period between the object's creation and its last use during an execution. This is distinct from a common

expensive regions. In the previous example, knowing that each `SimpleDateFormat` structure is only used within one iteration of the loop would suggest reuse through memoization or caching. Furthermore, knowing which objects have effective lifetimes bounded by the construction of each `SimpleDateFormat` would let us classify them and their initialization code as a secondary problem, to be studied only if the initial problem cannot be fixed.

Blended analysis paradigm. In this paper we present a new analysis paradigm for performance understanding of framework-based applications. This paradigm is called *blended* analysis, because dynamic analysis is used to obtain the calling structure of a particular execution of interest and then a static analysis is performed on that calling structure to obtain more detailed semantic information relevant for performance understanding. We examine one execution, rather than all executions, because we are addressing the performance problems of a particular execution. By applying a static analysis to a dynamically derived calling structure, we derive further information about what may have occurred in that execution, without incurring execution-time overhead, and avoiding the problems presented to static analysis by the use of reflection and dynamic class loading. Our hypothesis is that *blended analysis will enable a more precise and scalable analysis for performance understanding at an acceptable cost, in comparison to a purely static analysis (i.e., too imprecise) or a purely dynamic analysis (i.e., too costly because sampling will not provide sufficient precision).*

Blended escape analysis. As a first test of the hypothesis, we have designed a *blended escape analysis* that will enable explanation of regions of program execution subject to object churn. Escape analysis is a technique for approximating the effective lifetime of objects. It computes if and how newly created objects become visible beyond the method in which they were created. There are many existing static escape analyses, [8, 6, 7, 25, 11, 5], but for framework-based applications, achieving scalability and precision with them is a challenge, given the degree of software layering, the reliance on collections and reflection, and the prevalence of long chains of calls that massage data [17, 20]. Unfortunately, we cannot only look at the application code, since for performance understanding we need to see all layers that contribute to a problem, and problems are often caused by interactions between the frameworks. A purely dynamic approach, collecting object effective lifetime data at a finer granularity, is prohibitively expensive. Simple schemes to reduce the amount of data collected, such as filtering out libraries or aggregating by class, are of limited value because they lose valuable context. A blended escape analysis offers a way to avoid these difficulties.

There are many choices of dynamic information for input to a blended escape analysis. In our study we chose a call graph aggregated from more detailed information captured during a run, and used static allocation sites as our object abstraction. In framework-based applications, it is common for low-level libraries to generate objects whose eventual usage varies according to the calling context. To address this, our approach retains distinct escape state information at each calling context where an object is visible, unlike a traditional escape analysis. This allows a user to study specific usages of common objects.

In a typical use of our technique, the user starts by identifying a scenario (e.g., a web transaction), and a known expensive region that she would like to better understand. Blended escape analysis consists of (i) running the scenario to collect the calling context data for the specified region, and (ii) then running our escape analysis on that calling structure. Finally, object escape information is made available to the user as annotations on the calling context information.

We have validated the approach on *Trade*[5] a web server benchmark for retail brokerage applications. The multiple configurations of *Trade*, using different combinations of commonly used frameworks, mirror the variety of architectures found in real-world web server applications. For each of four configurations, we run a single transaction and analyze several expensive regions that perform different types of functions. To evaluate the analysis results, both for describing the object escape properties of code regions and for identifying areas of object churn, several new metrics are introduced.

There are many ways that the blended escape analysis results could be used to aid performance understanding. First, by providing an upper bound on object effective lifetimes, the analysis can sort out which objects are *temporary with respect to specific calling contexts*. This is useful for pointing out regions that make the most use of temporaries, as in the above `SimpleDateFormat` example. Second, by identifying objects that do escape a given context, the analysis can help explain the construction of longer-lived data. Third, escaping objects can shed light on what a section of unfamiliar code was intended to produce. Finally, a reduced version of the points-to information built during the course of the analysis can expose how new objects are grouped into larger structures at each calling context.

We have only begun to explore possibilities for making the results available to a user, and we demonstrate one such approach. We start with a calling context tree (*CCT*) [2], aggregated from the information gathered during the run. In our example we assume the user explores this data with an existing performance analysis tool, and has identified a smaller, suspect region to study. Because our blended escape analysis is based on a static object abstraction, we can further refine its results by using dynamic object allocation information collected during the run, to reflect the instances actually allocated at each CCT node in the region, and the paths on which these allocations occurred. This allows us to annotate the CCT with the number of instances captured at each node, and thus highlight for the user the calling contexts that are key users of temporaries. In our example we show how in a region of 476 CCT nodes, we can quickly find the single node responsible for using most of the temporaries. We show how escape and connectivity information at just a few nodes can help the user identify an optimization opportunity involving objects created at 9 distinct allocation sites, as far as 6 call levels away.

Summary. The main contributions of this paper include:

- a new analysis paradigm for blending static and dynamic analyses for performance understanding to achieve high precision for acceptable and practical cost,
- an instantiation of the paradigm in a blended escape analysis that extends previous techniques by calculating contextual escape information for an object in each method it reaches,
- an empirical study of the object escape properties for several regions of the framework-intensive *Trade* benchmark, and
- a demonstration of novel uses of escape analysis for performance understanding, by refining escape results to expose how new objects are used.

Overview. In Section 2, we give some background on escape analysis, and in Section 3 we describe our blended analysis technique, including some of the challenges of aligning the static and

definition of object lifetime used in garbage collection, that is, the period between object creation and collection by a GC.

[5] https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=trade6

dynamic program representations. In Section 4, we present details of our blended escape analysis and the implementation choices made. In Section 5 we describe our experiments, the metrics collected, and the results of our study. In Section 6 we show an example of how the results of the analysis can aid performance understanding. In Section 7 we discuss related work and close with our conclusions in Section 8.

2. ESCAPE ANALYSIS

Escape analysis computes an approximation of the effective lifetime of an object. It has been used traditionally for compiler optimizations requiring either information about an object (i) escaping a method invocation or (ii) escaping an allocating thread. The former (i) allows the object to be *stack-allocated*, (i.e., allocated on the run-time stack rather than in the heap), reducing heap fragmentation as well as average garbage collection time. The latter (ii) can be used to avoid expensive synchronization operations.

When an object is allocated, the run-time stack contains the method invocations that form its *allocation context*. If during execution, that object can be accessed beyond the lifetime of an invocation of a method f in its allocation context, then we say that the object *escapes* f. Alternatively, an object is *captured* by method *g*, if *g* is in its allocation context and the object cannot be accessed beyond the lifetime of the invocation of *g*.[6]

Escape analysis examines the values of references to determine the escape behavior of objects in a program. Each object therefore has an associated escape state: *globally escaping*, *non-escaping* or *escaping through parameters and/or return values (arg-escaping)*. For example, an object is globally escaping if it is referenced by a static field, non-escaping if it only can be referenced within a method through local reference variables, or arg-escaping if it escapes transitively through an argument or return value. In this last case, the object will eventually either be captured by some other method (i.e., become non-escaping) or will globally escape.

Several escape analysis algorithms have been proposed in the literature. They can be divided into two main categories: set-based and dataflow algorithms.

Set-based algorithms. Set-based algorithms use set constraints as a mechanism to compute escape information. Three set-based escape algorithms have been proposed for the Java language. All of them are designed for speed over precision and are both context- and flow-insensitive. Moreover, they associate escape state with references rather than objects. Bogda and Hölzle [7] proposed an algorithm for synchronization removal that is based on escape analysis. Their analysis is performed in two stages: the first stage identifies references that get stored in the heap, while the second stage narrows down the set of references to those that allow local objects to escape. Gay and Steensgaard [11] proposed an algorithm that relies on assigning a *fresh* status to a new object, which is then propagated to methods and references, and used to compute escape information. Beers *et al*. [5] have proposed an escape analysis algorithm specifically designed to have its results encoded as class file attributes and used at runtime by the Just In Time (JIT) compiler. Their algorithm uses two passes: the first computes approximations of run-time types for references, and the second uses the types computed in the first pass to find *captured variables*, (i.e., variables through which objects do not escape).

Dataflow algorithms. The dataflow escape algorithms are commonly accepted as being more precise than the set-based algorithms but also are more expensive. Two dataflow algorithms have been proposed in the literature: one by Whaley and Rinard [25] and one by Choi *et al*. [8]. Both algorithms build a representation of the relationships between references in a program (similar to points-to analysis) and associate escape state information with an *abstract object*. Each abstract object represents the set of possible objects created at runtime at an allocation site. These two algorithms differ in their treatment of strong updates and their representation of data structures (i.e., object aggregates).

The algorithm by Choi *et al*. relies on *connection graphs* to represent relationships between references and abstract objects. Connection graphs contain two main kinds of nodes: *object nodes* representing abstract objects and *reference nodes* corresponding to variables or fields in the program. Nodes are decorated with their escape property.

Example of escape analysis. The example in Figure 1 illustrates the Choi *et al*. escape analysis. The code builds a linked list from an array of primitives and then performs a linear search on it. The type of data stored in the list depends on command line parameters passed to the program and thus cannot be determined statically. Figure 2 shows the static call graph for this application.

Escape analysis proceeds in a bottom-up manner on the call graph. A connection graph is generated at each call graph node to represent a summary of the relevant data structures at that node and the (current) escape state of abstract objects. Cycles in the call graph are handled by iterating until the solution converges. Objects created before a method invocation can be introduced into that method through parameter-argument associations. Such objects are represented by *phantom* nodes in the connection graph that act as placeholders, and play a key role in mapping information from callees to caller during the interprocedural analysis.

Figure 3 shows the connection graph for the `makeList(int[])` method. Rectangular nodes correspond to abstract objects, diamond-shaped nodes to fields, round nodes to variables (or return values) and dashed boxes to phantom nodes. For example, the node labeled *P1* corresponds to the array of integers passed to `makeList(int[])` as a parameter. The method contains two allocation sites, `S1` and `S2`, which are represented by object nodes of the same name. The 1-limited[7] analysis makes the *S2* node point back to itself through its `next` field. The *S2* node also points to the *S1* node through its `payload` field, because information from the connection graph of the `Node(Data, Node)` constructor called at *S2* has been merged into the connection graph for `makeList`. Finally, the *return* node points to the return value, a `Node` object. Note that the connection graph for `makeList(char[])` is the same shape, except it contains a node for a `char[]` instead of an `int[]`.

After the connection graph has been built, escape states are propagated along its edges. Because the *data* and *return* nodes are initially arg-escaping, this state is propagated to all nodes that are transitively reachable from them. Therefore, all nodes in the connection graph are marked as arg-escaping, as shown.

Figure 4 shows the connection graph for the `main` method. The analysis starts by creating the phantom object *P3* for the `args` parameter. It then creates an object node for the array of integers allocated at statement *S3* and marks it initially as non-escaping. Information from the connection graph for the `makeList(int[])` method is then merged into the connection graph for `main` by mapping the actual argument of the call to the corresponding parameter node in the callee's connection graph. Therefore, *S3* is mapped to *P1*. Because `makeList` does not make *P1* escape globally, the es-

[6] The same terms can be used to describe the relation between objects and their allocating execution threads.

[7] In points-to analysis, recursive data structures must be represented by finite summaries of their possible shape. A standard representation uses *k-limiting* which retains the first *k* elements of a recursive structure and approximates the rest by summary nodes[16].

```
public class ListExample {
    static Node global_node;

    public static Node find(Node head, Data data) {
        for (Node n = head; n != null; ) {
            if (n.payload.equals(data)) return n;
            n = n.next;
        }
        return null;
    }

    public static Node makeList(char[] data) {...}

    public static Node makeList(int[] data) {
        if (data == null) { return null; }

        Node head = null;
        for (int i = data.length - 1; i >= 0; i--) {
S1:         Data d = new IntData(data[i]);
S2:         head = new Node(d, head);
        }
        return head;
    }

    public static void main(String[] args) {
        Node list; Data key;

        if (args.length == 0) {
S3:         list = makeList(new int[] {0,1,2,3,4,5});
S4:         key = new IntData(3);
        } else {
S5:         list = makeList(new char[] {'a', 'b', 'c'});
S6:         key = new CharData('a');
        }
        global_node = find(list, key);
    }}

interface Data { }
class IntData implements Data {...}
class CharData implements Data {...}
class Node {...}
```

Figure 1: Code listing for escape analysis example

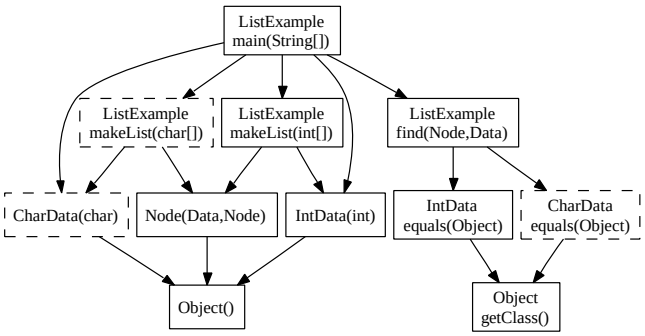

Figure 2: Call graph for `ListExample`

cape state of *S3* does not have to be updated. The connection graph for `makeList(char[])` is merged similarly.

The return value of each call to `makeList` is a reference to a `Node` that itself is used as an actual argument in the call to `find` in `main`, (i.e., *S2, S2'*). After processing the call to `find`, the analysis merges the connection graph of `find` with that of `main`. This results in an edge between the `global_node` field and each of the return values from the calls to `makeList`. The processing of the call to `find` creates the cross edges between the two `payload` fields and their associated *S1* objects, because the algorithm is not context-sensitive, and therefore, cannot separate effects which occur on different calls to `find` [22].

For ease of implementation, all static fields of classes referenced are treated as instance fields of the singleton global object *G*, shown

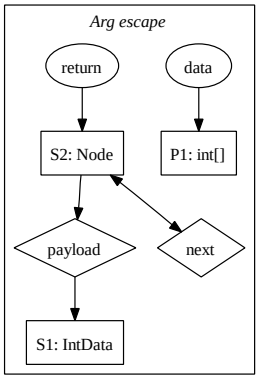

Figure 3: Connection graph for `makeList(int[])` method

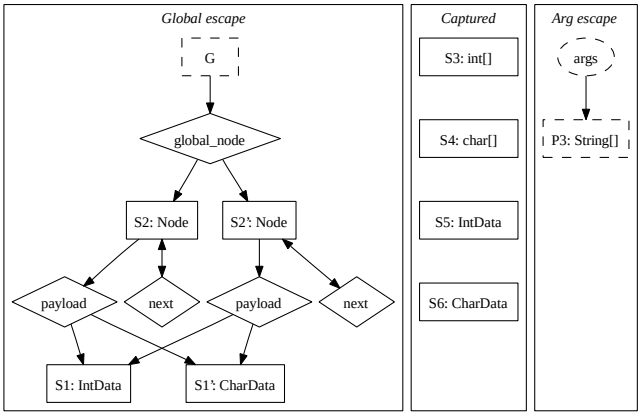

Figure 4: Connection graph for `main` method

as a phantom node. The *G* object is initially marked as globally escaping; after propagation, all objects reachable from the `global_node` field are marked as globally escaping as well.

3. BLENDED ANALYSIS

This section describes *blended analysis*, a new analysis paradigm that performs an interprocedural static analysis on a calling structure obtained through dynamic analysis, thus capturing properties of a single execution. There are practical uses for an analysis of even a single execution. When confronted with poor performance in an execution, we need to know more about that particular execution to understand which performance problems have occurred. When debugging, we are concerned with the specific execution that resulted in an error.

The goal of a blended analysis is to achieve the precision of a fully dynamic analysis (to understand a particular execution) for much less cost. Often, obtaining all needed information dynamically is prohibitively expensive or impossible. For example, computing escape analysis information dynamically would require tracking all object allocations and pointer updates. Studies such as [15] have shown that obtaining such information precisely can slow down the execution of a program by as much as two orders of magnitude. Limits on the amount of overhead that can be tolerated while profiling a deployed application in a production environment make this approach impossible to use in practice. Blended analysis offers an alternative by allowing a static analyses to be performed off-line to gather the needed information, based on an easy to collect record of the execution. The rest of this section discusses the blended anal-

ysis paradigm, its novel aspects as well as the new challenges it raises.

Calling structure. Interprocedural static analysis requires information about the possible callees at each analyzed call site. In traditional analysis, this information is usually computed in the form of a call graph. Call graphs can be built with varying degrees of precision (e.g., [13]). In a blended analysis, the calling structure is obtained from an execution trace; therefore, virtual dispatch can be resolved exactly. The execution trace is often represented as a tree structure in which each edge represents a call. Call trees are typically very large even for relatively short program runs, but fortunately, such detailed traces contain more information than may be required. Smaller calling structures such as *call graphs* or *Calling Context Trees (CCTs)* [2] can be easily obtained by aggregating nodes in the call tree. Techniques also exist to collect CCTs directly at runtime (e.g., [26]). Sometimes, a problematic transaction or a *scenario* (i.e., a partial transaction with specific functionality) is identified in advance and the execution trace is limited to that part of the application. In this case, the calling structure is restricted to that part of the program to be examined by the analysis.

A dynamically obtained calling structure must be modified to become amenable to static analysis. For instance, calls to static class initializers appear in the trace as part of the class loading mechanism. We represent them in the call graph as program entry points, as in static analysis. In contrast, calls to run-time support code such as the class loader or garbage collector are seen explicitly in dynamic traces, but have no associated call site in bytecode. They are currently discarded from the calling structure used in blended analysis.[8]

As discussed above, when tracing a scenario, the full calling structure for the execution is not known, and the analysis has to be performed on a partial calling structure that often does not include all natural entry points for the application (e.g., *main*). The analysis therefore must be started from arbitrary methods that often have reference parameters. While it is possible to ignore such parameters and model them as *phantom object* references (i.e., references to objects created outside of the scope of the entry methods), it is more desirable to handle such objects in a more precise way. Therefore, a root method is artificially created and used to invoke other non-natural entry point methods with appropriate parameters. Declared types are not sufficient to appropriately synthesize parameters, since they often correspond to non-instantiatable types (e.g., interfaces and abstract classes). Therefore, dynamic information from the execution trace is used to compute a set of types for each parameter, from which we synthesize its corresponding objects.

Dynamic language features. Dynamic class loading and reflection are typically difficult problems for a traditional static analysis. A common approach to handling them is to require user input specifying the set of all possible classes that can be loaded at runtime and the set of all possible targets at each reflective call site. Blended analysis does not require this effort, because at runtime the set of loaded classes can be recorded. Even dynamically generated classes that do not exist statically (e.g., those generated by the Java Virtual Machine to handle certain reflective features such as proxy classes) can be recorded. The static analysis component of the blended analysis therefore has access to all loaded classes. Similarly, targets of reflective calls can be recorded.

Limitations. Since the dynamic calling structure represents only calls that occur during a single execution (or set of executions) of a given application, the analysis is *safe only for that given execution*, rather than all possible executions as in traditional static analysis.

[8]This is an area for future exploration. In some applications, we have observed costly object churn hidden in class loaders.

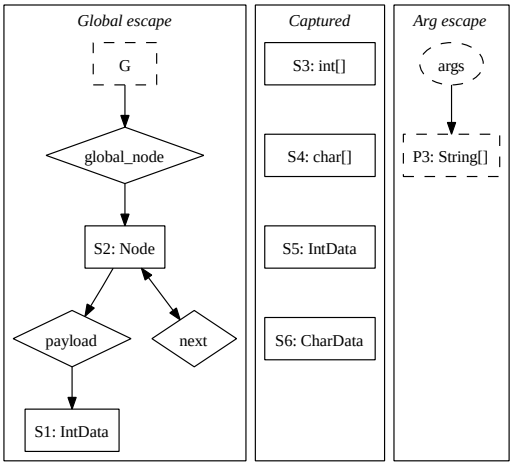

Figure 5: Connection graph for `main` with dynamic call graph

In addition, we make the common assumption of software testing, that the execution is deterministic and thus, repeatable, so that what we learn from this execution can be used to predict what will occur on subsequent executions with the same input.

4. BLENDED ESCAPE ANALYSIS

In this section we present a specific instance of the blended analysis paradigm, *blended escape analysis*, and describe its implementation framework and its limitations. We defined and implemented a blended version of the escape analysis by Choi *et al*. [8]. As mentioned in Section 2, this analysis associates escape information with abstract objects. In order to be able to study *how* objects escape in addition to *which* objects escape, we modified the analysis to keep track of a *distinct escape state* for each object at each node in the calling structure, rather than keeping only one escape state per abstract object. Thus, the escape states of an abstract object can be examined along a path in the calling structure; especially in layered applications, it may be valuable to know that an abstract object escapes on one path in the calling structure but is captured on another path.

Example. If the code from Figure 1 is invoked without any command line arguments, it can be easily observed that the `else` part of the branch in `main` is not executed. Therefore, the call graph for the application will be the same as the one presented in Figure 2 with the exception of the `char`-related nodes (shown in dashed boxes).

Figure 5 shows the corresponding connection graph obtained by a blended analysis for the `main` method. The `global_node` field now only points to a list of `IntData` objects. Note that objects *S5* and *S6* are present in the connection graph because the `else` branch of the `if` statement still is analyzed even though it was not executed.[9]

Implementation. We implemented a blended escape analysis comprised of two components: one for dynamic analysis and the other for static analysis. The dynamic analysis component is a modified version of *Jinsight*, a software visualization tool based on the Java Virtual Machine Profiling Interface (JVMPI), extended to export aggregated calling structures from execution traces. We turn off the just-in-time compiler (JIT) for the profiled application, in order to avoid confusion in the dynamic call graph, for example, caused by method inlining.[10] While a more efficient tracing

[9]As future work we plan to explore pruning control flow graphs using dynamic knowledge of the calls that were executed.

[10]We plan to explore relaxing this requirement in future work.

technique could be achieved, this *Jinsight*-based approach has the advantage of allowing the results from our blended analysis to complement the information that is already available to the *Jinsight* user. Our tool is currently able to generate dynamic call graphs and CCTs in addition to dynamic call trees. Because it relies on the JVMPI, *Jinsight* does not provide information pertaining to call sites within a method. We therefore assume that any call site which matches an observed target method in signature and type is a potential invocation of that method.

Our tool allows a user to generate the calling structure for an entire trace or only part of a trace. The generated calling structure can also contain information about dynamic object allocations at each node. While this information is not currently used by the blended analysis, it is relevant to refining analysis results as discussed in Section 6. The modified *Jinsight* tool also monitors class loading to record classes that may not be available statically.

The static component of the blended analysis framework is implemented on top of the IBM *WALA* analysis framework.[11] This component can use static call graphs built using the *WALA* infrastructure or dynamic calling structures generated from our modified version of *Jinsight*. This design permits future experiments with different levels of calling context in the analysis. Models for native methods have been hand-coded. Our current implementation uses allocation sites as a representation of abstract objects, but could be easily adapted to use other representations such as allocation information from the actual run.

To handle reflection, we hand-coded specification templates for reflective methods that are automatically instantiated for an observed target of a reflective call. These specifications make reflective calls appear exactly like regular method calls, so that this component also can be used to perform a safe static analysis on a static call structure expanded with these reflective calls.

Limitations. Recall that as a blended analysis, our blended escape analysis uses a calling structure derived from one execution; therefore, the static analysis escape results derived are only *safe* with respect to that particular execution. In this implementation, dynamic information is used to prune the possible interprocedural targets of a virtual call; other pruning uses are possible (see Section 6). Within each method, the static analysis is flow-sensitive, with no assumptions made about branches taken. All static allocation sites within reachable methods in the calling structure are assumed to be possibly executed. This implementation used a call graph: use of other calling structure aggregations is possible.

5. EXPERIMENTS

Experimental Setup. Experiments with blended escape analysis were performed on the *Trade* 6.0.1 financial transactions benchmark running on *WebSphere* 6.0.0.1 and *DB2* 8.2.0.[12] The *Trade* benchmark defines parameters that determine how it interfaces with the *WebSphere* middleware. We experimented with four configurations of *Trade* by varying two of the 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). 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, WSDL and UDDI) to access transaction results. All other parameters retained their default values. The four configurations, having varying degrees of layering, reflect the variety seen in real-world web server applications. The scenarios we tested implement identical functionality, regardless of the configuration. Note that in the web services cases there are separate threads simulating a web services client and server; we analyzed portions of both threads together, so that the functionality we capture is comparable with the other configurations.

Data Points. For all four configurations, the benchmark was warmed up with 5000 steps of the built-in scenario in order to allow for class loading and the population of caches. We then traced a single login transaction using our modified version of *Jinsight*. This transaction performs a number of functions. We chose three major subtasks of the transaction as our scenarios for analysis:

- ***login*** : Authenticates a user using information stored in a back-end database. In the process it updates the database with the last login time.
- ***getHoldings*** : Retrieves a user's portfolio information from a back-end database into Java objects. The user's data was 9 holdings records.
- ***jsp*** : Formats the user's portfolio as a web page using the Java Server Pages (JSP) technology. It also retrieves and formats market summary data via a nested JSP. Note that for this scenario, only three of the four configurations could be run while ensuring comparable functionality.

Metrics. We define four metrics to capture aspects of the escape behavior in aggregate, for a given scenario under study.
Distribution of escaping object states: For each abstract object, we compute its final escaping state. Each object can either globally escape on all paths in the calling structure, be captured on all paths, or sometimes be captured and sometimes globally escape. We report the percentage of objects that fall within each category.
Distribution of allocating nodes: For each node in the calling structure that contains at least one allocation site, we compute the total number of abstract objects created at that node and the number of those objects that are marked as non-escaping at that node.
Depth of escaping path: For those abstract objects that eventually escape globally along some path in the calling structure, we compute the length of the shortest path from the calling structure node where the object is allocated to a node from which it escapes.
Depth of capturing path: For those abstract objects that are eventually captured on all paths in the calling structure, we compute the length of the shortest path from the calling structure node where the object is allocated to a node where it is captured.

Results. Table 1 shows a comparison of the size of each scenario and configuration. Clearly the *EJB* and *WebServices* configurations introduce additional complexity over *Direct*. The combined effect of these additional frameworks, in the *WebServices* configuration, is not completely additive, however, suggesting that *EJB* and *WebServices* make use of common frameworks. Interestingly, the *getHoldings* and *login* scenarios have very similar values for the number of invoked methods, number of allocated types, number of abstract objects and stack depth. While only *login* makes database updates, the primary effect appears to be the shared usage of database access code. The additional instances reflect the fact that *getHoldings* receives more records from the database, an effect that is magnified in the more layered configurations. The *jsp* scenario performs a different task, which leads to a noticeable difference in these characteristics.

[11] *WALA* is available under the Eclipse public license from http://wala.sourceforge.net.

[12] *Trade*, *WebSphere* and *DB2* are now available to academic researchers through the IBM Academic Initiative.

Scenario	Config.	Methods	Invocations	Instances Allocated	Instance Types	Abstract Objects	Max Stack Depth
get-Holdings	*Dct-Std*	710	4,484	186	30	549	26
	Dct-WS	3,308	127,794	5,522	166	2,517	53
	EJB-Std	1,978	60,936	1,751	82	1,834	62
	EJB-WS	4,479	184,288	7,088	210	3,747	72
login	*Dct-Std*	752	3,497	131	30	571	26
	Dct-WS	3,296	59,301	3,185	160	2,484	49
	EJB-Std	1,991	14,983	551	77	1,812	49
	EJB-WS	4,447	70,757	3,584	198	3,670	66
jsp	*Dct-WS*	3,647	195,379	7,346	178	2,726	66
	EJB-Std	2,336	250,715	7,511	96	2,027	67
	EJB-WS	4,680	425,946	14,026	219	3,847	66

Table 1: Size comparison of all scenarios and configurations

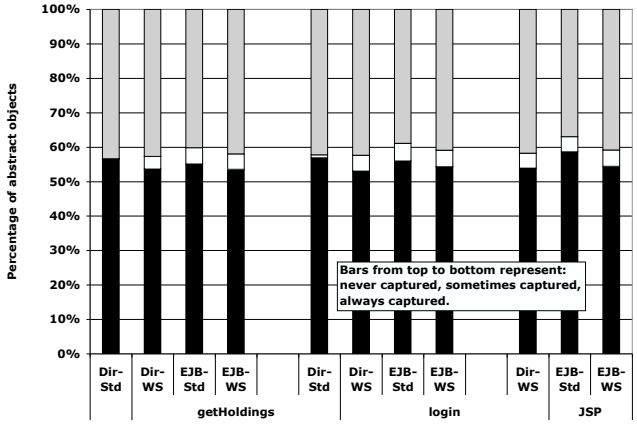

Figure 6: Breakdown of abstract objects by escape state

Distribution of escaping object states: Figure 6 shows the breakdown of abstract objects according to their final escape state on paths in the call graph. It is clear that most objects eventually globally escape or are captured (i.e., observe the black and grey parts of the bars). Also, on average there are more captured objects than escaping objects, indicating a significant use of temporaries in these scenarios. Interestingly, there is little variation between these categories over all 11 data points, despite the significant variations in underlying framework implementations.

Distribution of allocating nodes: Figure 7 a) shows the distribution of allocating call graph nodes in *getHoldings*, according to the number of abstract objects they allocate. This graph shows that most methods allocate few objects. There are, however, a few methods that allocate a surprisingly high number of objects (e.g., one method allocates 46 objects in the *EJB* configurations). Figure 7 b) shows the capturing behavior of allocating methods. It can be observed that most allocating methods capture a small number of the objects they allocate; on average only about half of the allocated objects are captured in their allocating method.

Depth of escaping path: Figure 8 shows the depth of escaping path results for all four benchmark configurations in the *getHoldings* scenario. The depth of escaping path metric demonstrates significant differences between configurations. For instance, the *Direct-Standard* configuration, which is the simplest of all four, has higher frequencies for longer shortest escape depths than the other three configurations. The total number of allocated objects in *Direct-Standard* is much lower than in the other configurations. These two observations may mean that the complex configurations introduce more short-lived objects than *Direct-Standard*. On the other hand, because of the conservative nature of the escape analy-

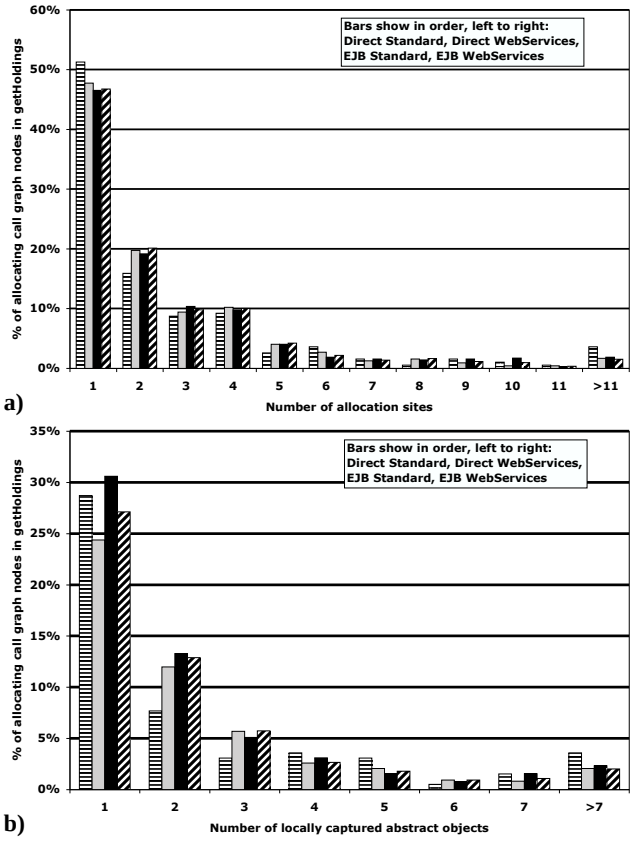

Figure 7: Allocations in *getHoldings*

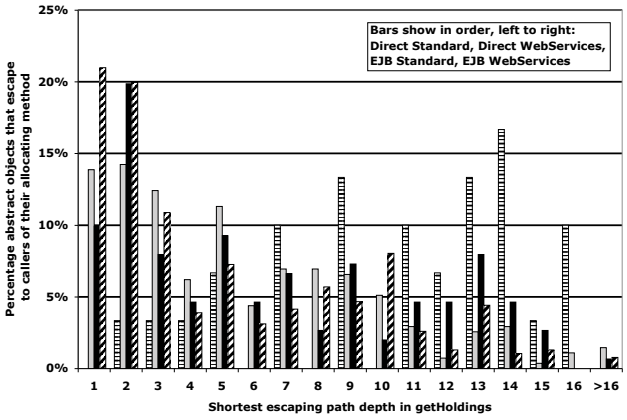

Figure 8: Shortest escaping path depth

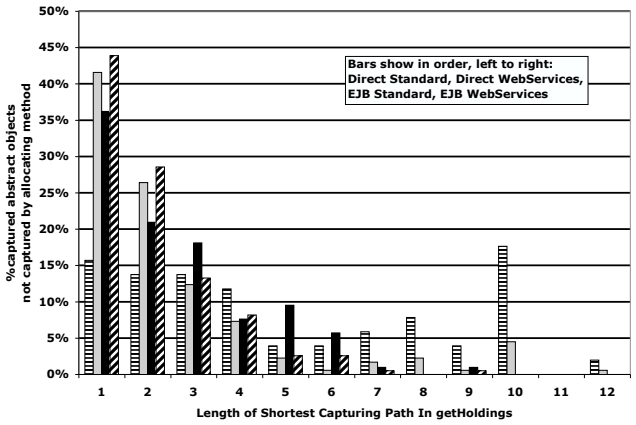

Figure 9: Shortest capturing path depth

sis, complex data structures are more likely to be marked as globally escaping than simple data structures. This is more likely to happen earlier on escape paths than for simpler data structures.

Only the *Direct-WebServices* and *EJB-WebServices* configurations have escaping path depths that are longer than 40. This suggests that the *WebServices Trade* configuration introduces much longer call chains to build data structures through the additional layers of frameworks that it uses. The results for *login* (not shown) are very similar to those for *getHoldings*. It may be that uses of these frameworks are stylized which causes this similarity; this is a subject for future investigation. However, the results for the *jsp* scenario (not shown) exhibit a significant variation from the previous scenario for objects that have escaping depths of at least 6 nodes. This suggests that the *jsp* scenario either uses more or different layers of framework.

Depth of capturing path: Figure 9 shows the results for the depth of capturing path metric for all four benchmark configurations in the *getHoldings* scenario. This metric has much smaller depth values than the escaping depth metric, which suggests that escaping objects used as part of persistent data structures, migrate through more framework layers than captured objects, which are more likely to be used as temporaries. As previously, the *getHoldings* and the *login* scenario have very similar behavior. While the results for the *jsp* scenario (not shown) only differ significantly for capturing depths of at least 6, they exhibit some variation for depths as low as 2. This illustrates again that the *jsp* scenario differs from the other two scenarios in terms of its framework usage.

6. PERFORMANCE UNDERSTANDING WITH BLENDED ANALYSIS

In this section we demonstrate a use of blended escape analysis to aid performance understanding. In our usage scenario we assume the user is exploring dynamic information with a tool such as *Jinsight* or *ArcFlow*, to identify suspect regions. The results of the blended escape analysis are used to provide additional insight into a region. Since our blended escape analysis is based on a static object abstraction, we first refine our results, using the dynamic information, to reflect allocations that actually occurred. This postprocessing step aids understanding by removing extraneous objects from consideration at each calling context, and, more importantly, by providing instance counts with each escape state, so the user can assess the magnitude of a potential problem.

Our aim is to help the user understand the usage of temporary data, to identify areas that can be optimized. First, by computing the number of instances captured at each calling context, we can guide the user toward regions that make the heaviest use of temporaries. We can also expose the connectivity of temporaries, to enable their understanding as data structures rather than individual objects. We can also allow the user to browse individual details to understand the disposition of particular objects at each calling context. In the rest of this section, we first describe how we postprocess the blended escape analysis results using dynamic allocation information. We then show how this information can enable discovery of an optimization opportunity.

		Abstract Objs		CCT Contexts		
					Capturing	
Scenario	**Config.**	**Total**	**Seen**	**Total**	**Static**	**Post.**
get-Holdings	*Dct-Std*	549	61	1,473	348	17
	Dct-WS	2,517	503	18,267	3,919	322
	EJB-Std	1,834	183	8,089	2,223	71
	EJB-WS	3,747	606	25,012	5,848	373
login	*Dct-Std*	571	64	2,057	467	28
	Dct-WS	2,484	486	18,567	3,975	328
	EJB-Std	1,812	195	9,303	2,549	86
	EJB-WS	3,670	598	25,864	6,093	381
jsp	*Dct-WS*	2,726	579	19,990	4,329	393
	EJB-Std	2,027	268	9,074	2,497	130
	EJB-WS	3,847	678	25,278	5,937	428

Table 2: Effect of postprocessing with dynamic allocation information

Reduced connection graphs. For the postprocessing stage, we illustrate our approach using as input a CCT, decorated with the number and types of instances allocated at each context. This information is derived from the original trace. For each context node in the CCT we can then derive a *reduced connection graph*, reflecting what occurred in the run during the lifetime of the calls that that context represents. We derive this from the connection graph the blended escape analysis computed for the corresponding method. First, we annotate each node in the connection graph with the number of instances it could represent. These are the instances allocated during the lifetime of the calls represented by this context, excluding those captured along every path from the allocation site to this node. At the same time, we remove connection graph nodes representing objects that could not have been visible in this context, because either no allocation was observed during this context's lifetime, or any allocations that did occur were through paths that captured the object. We then summarize each reduced connection graph with the count of instances having each escape state. We can use this to highlight calling contexts that capture any instances (and that capture a large number of instances). Finally, for ease of understanding, we simplify the connectivity in the reduced connection graphs, by eliding links other than those relating Java objects to one another.

Table 2 shows the result of running the algorithm on each of our eleven cases. The rightmost two columns show the number of capturing CCT contexts we identify, first without and then with the postprocessing step. By postprocessing with the CCT of dynamic allocations we are able to reduce the number of contexts to consider by at least an order of magnitude in all cases.

Note that the CCT we use in our illustration is a finer aggregation than the call graph used by the blended escape analysis. Therefore, for different CCT nodes representing the same method in the code, the reduced connection graphs will differ, reflecting the way allocations actually occurred in different contexts. Note also that it is possible to use finer or coarser aggregations at this stage; these choices have an impact on the accuracy of the instance counts. Evaluating these tradeoffs is an area for further exploration.

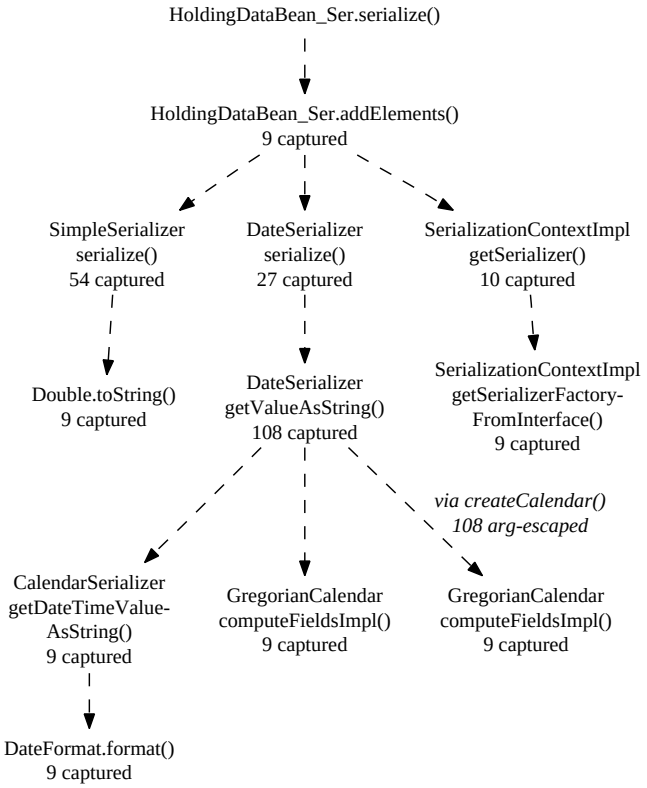

Figure 10: CCT for `HoldingDataBean_Ser.serialize` showing only the capturing contexts

Example: finding and understanding areas that use temporaries. For our illustration we will explore the CCT for a portion of the *getHoldings* scenario, in the *Direct-WebServices* configuration. This region, headed by the method `HoldingDataBean_Ser.serialize()`, is responsible for formatting stock holding records into the response portion of a SOAP message. Nine such records were serialized in this run, each via one call to `serialize()`. In the process, 290 new instances of 14 distinct classes were created. We would like to understand how these instances were used.

Using the CCT and the results of the blended escape analysis on this run, we build the reduced connection graph for each CCT node. With these graphs we can compute the number of instances captured at each CCT node. The CCT rooted at `serialize()` represents calls to 223 distinct methods, and has maximum depth of 20 from `serialize()`. Of the 476 CCT nodes in this region, there were only 11 that captured any instances. In Figure 10 we show a reduced version of the CCT, eliding everything but those few capturing nodes. We annotate each node with the number of instances it captures. In total, of the 290 instances allocated, 262 were captured.

We can see that the largest number of new instances were captured in `DateSerializer.getValueAsString()`. Every holding record has a number of fields, including one field with the purchase date. This code is formatting that date field. We can explore the details of the objects captured there, using the reduced connection graph shown in Figure 11. Each box represents an abstract object; objects are arranged in clusters according to escape state. Each object is annotated with the number of its instances visible at this context. As we would expect for code whose purpose is formatting, we see a number of `Strings` arg-escaping.

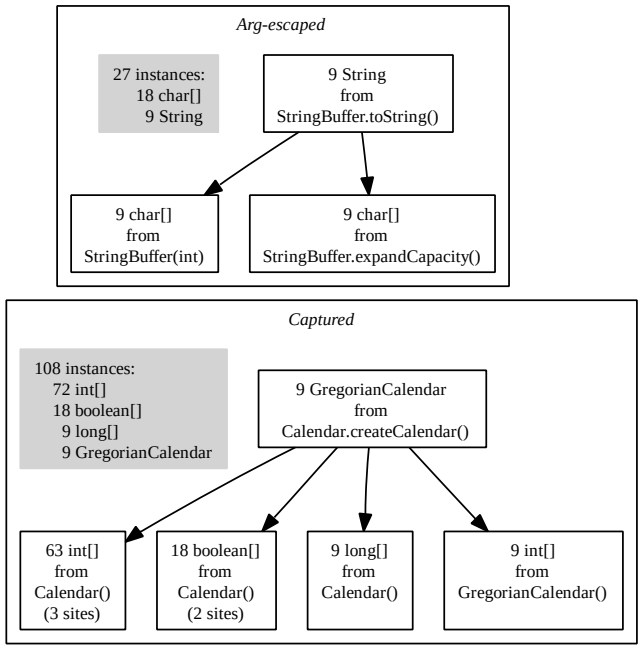

Figure 11: Reduced connection graph for `DateSerializer.getValueAsString` showing escape state and connectivity of instances

In the cluster showing captured objects, the connectivity suggests that 9 `GregorianCalendar` *structures* are captured here, with 99 associated arrays. This suggests that `getValueAsString()` is causing a temporary `GregorianCalendar` structure to be built in order to process the date field of each of the nine holding records. `GregorianCalendar` is a Java library class used to convert times between milliseconds and human-oriented units such as months, days of the week, or minutes. The fact that this structure consists of so many distinct objects suggests a complex initialization process.[13] An optimization would be to cache the `GregorianCalendar`, possibly per thread. Before making this decision, we would like to better understand what's involved in creating these structures.

Exploring our original CCT, we find that `DateSerializer.getValueAsString()` calls `GregorianCalendar.createCalendar()` (noted in Figure 10). Its reduced connection graph (not shown) reveals that these `GregorianCalendar` structures arg-escape that method. In the figure we can see that below `createCalendar()` is a node, `computeFieldsImpl()`, that captures nine additional instances (`int` arrays in this case). These subordinate temporaries are "byproducts" of the construction of the `GregorianCalendar`, and can be counted as a cost of that construction. In total then, we would save the construction of 117 new instances, by reusing an existing calendar constructed earlier.

Discussion. By focusing on where objects are used, rather than on where they are allocated, we are able to aggregate disparate events into a single summary for the user to focus on. The 117 temporaries in the above example were allocated at 9 distinct sites across 4 methods, some as far as 6 levels from the context under consideration for optimization. A single summary helped highlight the reason for 45 percent of the temporaries in this example.

[13]The connectivity information is a conservative approximation. One challenge in blended analysis is to explain to the user the degree of certainty of different aspects of the results. In this example, the library does in fact build this complex structure each time.

We note also that many of the allocation sites, particularly for lower-level framework objects, are called from many paths in the larger *getHoldings* CCT, with varied escape behavior. This is typical of applications which make heavy use of frameworks. For example, 540 `Strings` were allocated in `StringBuffer.toString`, and are potentially captured at 42 different nodes in the call graph for *getHoldings*. Yet in our example region, just 9 instances were allocated; all arg-escaped from the context we were studying, and were captured by the calling context, `DateSerializer.serialize`. By postprocessing the blended escape analysis using the CCT, and by retaining distinct escape information along different paths in the blended analysis, we were able to provide results relevant to the region we were studying. The postprocessing step also allowed us to remove extraneous information. In the above example, the original connection graph for `getValueAsString()` showed that either a temporary `BuddhistCalendar` or `GregorianCalendar` may have been used; the reduced connection graph shows that we need only consider the `GregorianCalendar`.

In summary, by utilizing additional dynamic information, the postpass is able to filter the static analysis results for the user, thereby improving precision. A comparison of the third and fourth columns in Table 2 shows that in any given run, object creations occurred at only a small percentage of the allocation sites. One area for further study is to understand the degree to which each component of the additional dynamic information – the object creations versus their contexts – contributes to the improvement in precision. Furthermore, it is possible that the blended escape analysis algorithm could employ additional dynamic information, such as the object instances created, to render the static analysis more precise (i.e. closer to a dynamic analysis). Specifically, we would like to use knowledge of created object instances to prune the intraprocedural paths explored during the static analysis. We will examine this variant of blended escape analysis in future work.

7. RELATED WORK

Since the research in this paper can be related to much existing static and dynamic analyses, space limitations force a focus on the two most relevant areas of research: studies of framework-intensive systems and examples of previous combined static/dynamic analyses.[14] In Section 2, related escape analysis algorithms have already been described. There also are dynamic analyses of object lifetime (e.g., [23]), which differ in their goal from our work, as they are aimed at approximating the last use of an object in order to optimize garbage collection, whereas our analysis finds a usage region for an object in the call graph. Moreover, these analyses have not been shown as scalable to framework-based applications, the focus of our work.

Studies of framework-based systems All the work summarized in this section shares our goal: to better understand the performance of framework-based applications. However, these previous studies all are based solely on dynamic analysis, rather than a combination of static and dynamic analyses, and they primarily concentrate on identifying areas with performance problems, either in the program calling structure or in data structures in dynamic heap snapshots. The last study reported in this section combines dynamic information with manual inspection, to categorize data transformation operations. In contrast, blended escape analysis is a combined static and dynamic analysis which extracts semantic data about object usage in a framework-based application to help explain performance difficulties. The specific performance problem addressed here, object churn, was not studied in this previous work.

Ammons *et al*. [3], presented a dynamic analysis tool, *Bottlenecks* which helps a user explore execution profiles of framework-based applications in order to find performance bottlenecks. Fourteen bottlenecks were found by examining two different versions of *Trade3* running on *Websphere*. This study showed the complexity of the calling structure of these applications by measuring max and mean depth of call paths and out-degree of call nodes. Srinivas *et al*. [24] presented a dynamic analysis technique which identifies interesting method invocations, those that account for a specified cumulative percentage of execution cost, in components selected by the user. The paper considers how to summarize execution costs in a meaningful way in framework-based codes, using a combination of *base cost* (i.e., the cost of an invocation minus the cost of its callees) and *cumulative cost* (i.e., the cost of an invocation plus the cost of its callees). The technique was tested successfully on e-commerce applications and on parts of the *Eclipse* IDE.

Two tools have been developed at IBM Research to interpret dynamic heap snapshots of framework-based programs, for aiding understanding of program memory usage, especially for longer-lived data. *Leakbot* [19], an automated and scalable memory leak detection tool, finds the data structures in two successive execution heaps obtained early in execution, identifies those data structures which are likely to be leaking by using structural and temporal properties, and then selectively tracks objects in those data structures, allowing identification of potentially leaking structures. *YETI* is a tool for identifying and summarizing key data structures in a heap snapshot [18]. *YETI* derives an object reference graph for the heap, to show all existing relationships between objects. Clever graph reductions are applied to highlight the key structural relations; these produce a *backbone* of the reduced graph that represents thousands of objects, but contains only tens of nodes.

Mitchell *et al*. [20] presented a new characterization of the runtime behavior of framework-based systems, obtained by combining dynamic analysis with manual inspection of source code. The authors present a decomposition of execution events as a hierarchy of dataflow diagrams, showing the flow of logical data through a series of physical transformations. This decomposition is used to organize the aggregation of operation costs in units of method calls and object creations. The emphasis of this paper is on developing high-level abstractions of behavior that organize the observed method calls into identifiable and recognizable groupings, to better understand their function and their cost.

Combined static and dynamic analyses There have been previous combinations of static and dynamic analyses for solving a wide range of problems, including an early overview paper on this topic [10]. Typically, the static analysis results are used to direct where the dynamic analysis should be applied. In contrast, the blended analysis paradigm uses the dynamic analysis results to enable more precise and focused static analysis. Given space limitations, we restrict our remarks to those previous combined analyses similar in their use of dynamic analysis to our blended analysis.

Gupta *et al*. [14] used dynamic information – observed breakpoints and procedure calls/returns – to prune infeasible control flow while calculating a static slice to explain program behavior for a specific execution. Breakpoint information can be used to distinguish feasible predicate branches in control flow graphs. Procedure calls/returns enable more accurate calculation of interproce-

[14]Note: we do not discuss static or dynamic algorithms for shape analysis of the heap for two reasons: (i) the shape of data structures is a byproduct of our connection graphs, but the analysis focus is on effective object lifetimes for a particular execution, (ii) static shape analyses, which face scalability challenges when applied to framework-based applications, report data structures possible in the heap over all executions, information not fine-grained enough to aid understanding of performance problems in these codes.

dural paths during static slicing. Groce *et al*. also used dynamic information to prune infeasible control flow, while worked on improving model checking for C programs. The authors interpreted failure traces of events 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 [12]. The use of dynamic analysis to enhance the precision of a subsequent static analysis in these two papers is similar to the approach in blended analysis, but the code under analysis is not framework-based nor object-oriented.

Mock *et al*. [21] substituted dynamic points-to information for static points-to relations, to restrict the dependences used in static slicing to those observed during debugging, and thus increase the utility of the slices obtained. However, experimental findings were disappointing because slices only improved where function pointer references could be exactly resolved. This work, similar to blended analysis, restricts the static analysis to a particular execution, in that known values for function pointers are used.

Artzi *et al*. [4] presented a combined (or staged) static and dynamic parameter mutability analysis for Java, in which different analysis phases are executed in a loosely-coupled pipeline. Communication between the stages is through the derived mutability data. This work differs from blended analysis in terms of the coupling between the static and dynamic analyses, which is through the data solution being computed, rather than the program representation. Nevertheless, it is similar in presenting a framework within which specific analysis choices can be instantiated and thus, different static/dynamic analysis combinations tried. Although the blended escape analysis presented here has only two analysis phases, the blended paradigm allows for additional analysis passes.

8. CONCLUSIONS

The growing use of libraries and frameworks in commercial software has led to an increase in runtime complexity. Understanding performance problems in these layered applications can be difficult. To aid in this task, we have defined a new analysis paradigm that blends static and dynamic techniques. We have designed a blended escape analysis for approximating object effective lifetimes, to help explain how temporary structures are built and used. Experiments with several scenarios in *Trade* have yielded aggregate results on the use of newly created objects in layered systems. We have also shown how effective lifetime and connectivity information obtained from a blended escape analysis, in combination with additional dynamic information, can aid in identifying performance bottlenecks in a particular application.

Acknowledgements. We would like to thank Edith Schonberg and Nick Mitchell for providing valuable feedback and ideas, as well as the anonymous reviewers for their helpful comments.

9. REFERENCES

[1] W. P. Alexander, R. F. Berry, F. E. Levine, and R. J. 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 fbw 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. Ernst, D. Glasser, and A. Kiezun. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2006-065, MIT CS and AI Lab., Sep. 2006.

[5] M. Q. Beers, C. H. Stork, and M. Franz. Efficiently verifiable escape analysis. In *Proc. of the European Conf. on Object-Oriented Prog. (ECOOP)*, pages 96–122. Springer, 2004.

[6] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Prog. Sys., Lang. and Appl. (OOPSLA)*, pages 20–34. ACM Press, 1999.

[7] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Prog. Sys., Lang. and Appl. (OOPSLA)*, pages 35–46. ACM Press, 1999.

[8] 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.

[9] W. DePauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Software Visualization: State of the Art Survey, LNCS 2269*, 2002.

[10] M. Ernst. Static and dynamic analysis: Synergy and duality. In *Proc. of the Wshp on Dynamic Analysis (WODA)*, 2003.

[11] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proc. of the Int'l Conf. on Compiler Const. (CC)*, pages 82–93. Springer-Verlag, 2000.

[12] A. Groce and R. Joshi. Exploiting traces in program analysis. In *Proc. of Int'l Conf. on Tools and Algs for the Construction and Analysis of Systems (TACAS)*, 2006.

[13] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. on Prog. Langs and Sys.*, 23(6):685–746, 2001.

[14] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Trans. on SE and Meth.*, 6(4), Oct. 1997.

[15] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanovic. Generating object lifetime traces with Merlin. *ACM Trans. on Prog. Lang. and Sys.*, 28(3):476–516, 2006.

[16] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1982.

[17] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 48–61, 2005.

[18] N. Mitchell. The runtime structure of object ownership. In *Proc. of the European Conf. on Object-Oriented Prog. (ECOOP)*, 2006.

[19] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. of the European Conf. on Object-Oriented Prog. (ECOOP)*, 2003.

[20] 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.

[21] M. Mock, D. Atkinson, C. Chambers, and S. Eggars. Improving program slicing with dynamic points-to data. In *Proc. of the Conf. on the Foundations of SE (FSE)*, 2002.

[22] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proc. of the 12th Int'l Conf. on Compiler Constr.*, pages 126–137, April 2003.

[23] R. Shaham, E. Kolodner, and M. Sagiv. Estimating the impact of heap liveness information on space consumption in java. In *Proc. of the Int'l Symp. on Memory Management*, 2002.

[24] K. Srinivas and H. Srinivasan. Summarizing application performance from a components perspective. In *Proc. of the Conf. on the Foundations of SE (FSE)*, pages 136–145, September 2005.

[25] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Prog. Sys., Lang. and Appl. (OOPSLA)*, pages 187–206. ACM Press, 1999.

[26] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 263–271, 2006.