PRACTICAL ANALYSIS OF FRAMEWORK-INTENSIVE APPLICATIONS

by

BRUNO DUFOUR

A DISSERTATION SUBMITTED TO THE

GRADUATE SCHOOL – NEW BRUNSWICK

RUTGERS, THE STATE UNIVERSITY OF NEW JERSEY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE

WRITTEN UNDER THE DIRECTION OF

BARBARA G. RYDER

AND APPROVED BY

New Brunswick, New Jersey

January 2010

ABSTRACT OF THE DISSERTATION

Practical analysis of framework-intensive applications

by BRUNO DUFOUR

Dissertation director:

Barbara G. Ryder

Many modern applications (e.g., web applications) are composed of a relatively small amount of application code that calls a large number of third-party libraries and frameworks. Such framework-intensive systems typically exhibit different characteristics from traditional applications. Current tools and techniques are often inadequate in analyzing applications of such scale and complexity. Approaches based on static analysis suffer problems of insufficient scalability and/or insufficient precision. Purely dynamic analyses, introduce too much execution overhead, especially for production systems, or are too limited in the information gathered.

The main contribution of this thesis is a new analysis paradigm, *blended analysis*, combines elements of static and dynamic analyses in order to enable analyses of framework-intensive applications that achieve good precision at a practical cost. This is accomplished by narrowing the focus of a static analysis to a set of executions of interest identified using a lightweight dynamic analysis. We also present an optimization technique that further reduces the amount of code to be analyzed by removing infeasible basic blocks, and leads to significant increases in scalability and precision of the analysis. We contribute *Elude*, a publicly available framework for blended analysis of Java programs.

We demonstrate the usefuless of blended analysis in practice by applying it to *object churn*, a common problem in framework-intensive applications caused by the excessive usage of temporary objects. We present a set of new metrics to characterize the usage and complexity of temporaries. We use an instantiation of the blended analysis paradigm, blended escape analysis, to compute these metrics for a set of real framework-intensive applications. Using these results we perform a detailed analysis of temporaries in these applications. We also use our technique to identify a set of problematic scenarios in a commercial application.

Acknowledgments

I would like to thank my advisor, Barbara Ryder, for her constant encouragement throughout my studies at Rutgers, her guidance and her commitment to providing a great learning environment. I am also grateful to Gary Sevitsky for sharing his invaluable experience, for his stimulating discussions, and his constant dedication to our collaboration.

I extend my thanks to the members of Intelligent Application Analysis group at the IBM T.J. Watson Research Center, in particular Nick Mitchell and Edith Schonberg for their suggestions and their support, as well as other IBM Researchers who volunteered their time and ideas for this project: Marco Pistoia, Omer Tripp, Julian Dolby and Stephen J. Fink.

I would also like to thank Jan Wloka for his invaluable input and his friendship, as well as the other Prolangs members at Rutgers and Virginia Tech: Weilei Zhang, Chen Fu, Xiaoxia Ren, Ophelia Chesley, Marc Fisher II, as well as Shrutarshi Basu and Luke Marrs for being the first external users of *Elude*.

This research was funded in part by IBM Research.

I would like to thank my family and friends to whom I am indebted for their much appreciated support and encouragements. A very special thanks to my fiancée, Arzoo, for her unconditional love, support, encouragement and patience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The increasing complexity of tasks accomplished by software has led to the proliferation of large *framework-intensive* applications. These applications are typically built by integrating numerous layers of middleware, libraries and frameworks with a relatively small amount of application code written by a developer. While it eases development effort, this reliance on code reuse comes at a cost. Developers are usually unfamiliar with the complex interactions between the various layers of libraries and frameworks used by their applications, and are therefore unaware of the impact of these libraries on their applications. As a result, framework-intensive applications often exhibit problems that are difficult to identify and eliminate. For example, one common performance bottleneck in framework-intensive applications is *object churn*, that is the excessive creation of short-lived objects that are the byproduct of some computation. Object churn can dramatically degrade the performance of applications due to the combined cost of allocating, initializing and ultimately garbage collecting temporaries. This problem is exacerbated by the fact that it is seldom localized in a few methods, but rather spans many layers of unfamiliar, complex framework code. Identifying and eliminating such problems requires a precise understanding of the behavior of an entire system. Tool support is therefore required to assist developers in improving the overall quality of their software.

Framework-intensive systems are an unexplored domain for program analysis, differing sharply in scale and structure from the usual compiler benchmarks currently used in analysis research. These 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 insufficient precision for answering behavioral questions for these systems. For instance, the common use of dynamic class loading and reflective mechanisms in Web applications forces static analyses to make worst-case assumptions about the possible behavior of a program. Purely dynamic analyses, accomplished through judiciously placed instrumentation in source code, bytecode or by probing the JVM runtime system, introduce too much execution overhead, especially for production systems, or are too limited in the information gathered. Limiting the amount of information collected at runtime severely impacts the usefulness of the analysis. The main contribution of this thesis is a new analysis paradigm, *blended analysis*, that avoids individual weaknesses of pure static and dynamic analyses by combining them in new ways.

There are two main approaches for combining static and dynamic analysis techniques:

- Static analysis can be used to determine where dynamic analysis should be triggered in an attempt to minimize the runtime overhead. There has been some work in the literature exploring this alternative. The most common use of this strategy involves using a dynamic analysis to collect facts about a program that could not be determined statically.

- Dynamic analysis can be used to precisely capture the scope of a problem, and focus a static analysis on the areas of interest based on one or more concrete executions of the application. We call analyses that follow this alternative *blended analyses* because they commonly rely on a close coupling of the individual analysis components. This combination of static and dynamic analyses is relatively unexplored.

In both cases, feedback loops can be introduced between the static and dynamic analyses. For example, the results of a blended analysis can be further refined by post-processing them with additional dynamic information from a more precise representation of the dynamic calling structure.

The present dissertation develops the thesis that *blended analysis enables a more precise and scalable analysis of framework-intensive applications 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).

1.2 Contributions

The main goal of this work is to provide useful, detailed analysis results for systems whose complexity and scale thwart automated analysis today. This thesis makes five main contributions.

First, we define a new analysis paradigm for blending static and dynamic analyses to achieve high precision at a practical (runtime) cost. Blended analysis 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. Blended analysis offers many advantages compared to a purely static or dynamic analysis:

- It limits the scope of the static analysis to calling paths in the code that were exercised at runtime, thus dramatically reducing the cost of a very precise static analysis. Reducing the focus of the static analysis allows achievement of high precision over an interesting portion of the program.

- It only requires a lightweight dynamic analysis, thus limiting the amount of overhead and perturbation during execution. This is essential for analysis of real-world, deployed applications, for which slowing down execution by more than a few percent is unacceptable.

- It eliminates the problems of how to handle inherently dynamic features like dynamic class loading and reflective method calls, as these can be captured by the dynamic analysis.

Second, we present a novel optimization technique that uses lightweight dynamic information (i.e., executed calls and object creations) to prove the infeasibility of basic blocks in the executions to be analyzed. Such unexecuted basic blocks can then be ignored by the static analysis. We show in Chapter 6 that this optimization results in significantly lower analysis cost, improved scalability and increased precision.

Third, we provide *Elude*, a publicly available software framework for blended analysis. This new tool can lower the barriers to further investigations into framework-intensive applications by providing a flexible platform to implement new blended analyses. *Elude* consists of two components: a lightweight dynamic analysis tool as well as a static analysis engine. *Elude* leverages WALA[1], a popular open-source static analysis infrastructure for Java developed by IBM. A number of existing analyses are already implemented with WALA; these analyses could easily be made blended using *Elude*, or used as building blocks in new blended analyses.

Fourth, we provide an instantiation of the blended analysis paradigm in a blended escape analysis. This analysis identifies temporary objects by computing bounds on their *effective lifetime* (i.e., the period between the object's creation and its last use during an execution). The blended escape analysis is tailored to perform a detailed characterization of the nature and usage of temporary objects in framework-intensive Java applications. It demonstrates the usefulness of the blended analysis paradigm to study real problems such as object churn in framework-intensive applications. The details of the analysis are discussed in Chapter 4.

Finally, we define a set of new metrics that quantify key properties related to the use of temporary objects. These metrics are computed from the results of the blended escape analysis. By combining static and dynamic information, we define new data structure abstraction that supports a rich characterization of temporary data structures. We then use these metrics to perform a detailed study of the temporaries in a set of representative benchmarks. We present the result of our findings in Chapter 6.

[1]`http://wala.sourceforge.net`

1.3 Thesis organization

The remainder of this dissertation is organized as follows. We begin with a survey of the related work in the next chapter. Chapter 3 presents the blended analysis paradigm and the optimization technique in details. Chapter 4 discusses background information on escape analysis, and then presents our blended escape analysis. Chapter 5 provides a description of the metrics used for studying the characteristics of temporaries in framework-intensive applications. We use these metrics in Chapter 6 to perform a detailed study of temporaries in real benchmarks. Finally, Chapter 7 concludes this work, and suggests future directions for research.

Chapter 2

Related work

Blended analysis is related to a large body of existing research. We therefore focus our discussion of the related work on two major topics. In Section 2.1, we present work that shares our goal to study framework-intensive applications. In Section 2.2, we survey existing combinations of static and dynamic analyses.

2.1 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.* [ACGS04] built the dynamic analysis tool *Bottlenecks* to explore execution profiles to find performance bottlenecks. *Bottlenecks* supports examining call-tree profiles in two ways: to find expensive call sequences and also to find call sequences which are more expensive in one call-tree profile than in another. 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). Fourteen bottlenecks were found by examining two versions of *Trade3* running on Websphere that differed by whether or not security was enabled. Through optimization, a 23% improvement in throughput with security enabled was realized.

Srinivas *et. al.* [SS05] 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. The main problem addressed was how to summarize execution costs in framework-intensive codes in a meaningful manner. Srinivas *et. al.* used 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).

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* [MS03], 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* [Mit06] is a tool for identifying and summarizing key data structures in a heap snapshot. It 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. The goal is to aid user performance understanding by uncovering the key data structures in the system.

Mitchell *et. al.* [MSS06] constructed a characterization of the run-time behavior of framework-intensive systems, by combining dynamic analysis with manual inspection of source code. Execution events were presented as a hierarchy of dataflow diagrams, showing a series of logical and physical data transformations. Next the types of input to these transformations were classified as either *carriers* (i.e., essentially inputs) or *facilitators* (i.e., auxiliary data for the transformation). 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. Their technique exposes the ways in which framework-intensive applications expend their resources massaging data.

Zhao *et. al.* [ZSZ+09] have studied the impact of excessive object allocations on the scalability of a number of Java benchmarks to a high number of CPU cores. Specifically, they were concerned with the bus write traffic caused by the garbage collected memory management strategy. They found that zeroing the memory of new allocated objects (as required by the Java memory model) combined with a large memory footprint can saturate the write bus and cause significant performance degradation.

Recent work by Shankar *et. al.* [SAB08] addresses object churn in the context of a production just-in-time (JIT) compiler. They use sampling profiles of object lifetimes to identify program regions, *churn scopes*, that encapsulate the lifetime of many objects. Object lifetime is computed by monitoring allocations inside the JVM, and forcing garbage collections at key points during execution to determine which objects are still reachable. For a given method f, their technique uses the notions of *capture* (i.e., the fraction of objects allocated during the lifetime of f that do not escape it) and *control* (i.e., the fraction of objects allocated during the lifetime of f that are captured in f but not in any of its children). By using heuristics that carefully balance these parameters, they were able to target agressive inlining in the JIT, thereby significantly improving the effectiveness of the standard escape analysis optimizations. While we share a goal of addressing object churn with the work by Shankar *et. al.*, both approaches are different and complementary. We focus on performance understanding techniques that allow developers to make high-level changes to their applications, thus reducing the number of temporary objects and more importantly their costly initialization. In contrast, Shankar *et. al.* are concerned with optimizing the allocation and reclamation costs associated with temporary objects by improving the efficiency of optimizations like stack allocation.

Xu *et. al.* [XAM+09] have designed a new runtime technique that identifies bloat in framework-intensive applications caused by excessive copying of data between objects. Performance defects were found and fixed in a number of benchmarks with great success; the resulting runtime improvements ranged from 9% to 30%, and the number of object creations was reduced by up to 65%. Xu *et. al.* modified the IBM *J9* production Java Virtual Machine (JVM) to perform a detailed monitoring of the execution at the level of executed instructions. They report execution overheads ranging from 1000% to 6400%. Their technique also requires a *shadow heap* for bookkeeping, thereby doubling the memory requirements for an execution. In constrast, blended analysis aims to minimize the time and space overhead of the dynamic analysis in order to avoid perturbing the execution of the application.

Tripp *et. al.* [TPF+09] have devised a static taint analysis algorithm for Java that specifically targets web applications. Their analysis uses a variety of models that are tailored to the applications under study, e.g., for reflective calls, Enterprise Java Beans (EJBs), Java Server Pages (JSPs), and other popular frameworks. Tripp *et. al.* also used a prioritization scheme that focuses the analysis on the subset of code that is likely to participate in taint propagation. This is a static analogue to blended analysis, since the static analysis is confined to a specific region of interest. Note that if the analysis is not able to complete (e.g., because the time budget is exhausted or it runs out of memory), then the results are unsound. This reinforces our claim that for some problems, unsound results like those provided by a typical blended analysis are useful.

2.2 Combinations of static and dynamic analyses

Static and dynamic analyses have been combined to solve a wide range of problems. In an early paper, Ernst [Ern03] discussed various ways in which both techniques can complement each other.

Typically, static analysis has been used to determine where the dynamic analysis should be applied. For example, static analysis can be used to minimize the cost of instrumentation inserted into the program (e.g., [MHM98, RRH02, vPG02]). 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. 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.* [GSH97] 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.* [GJ06] 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. While their goal is similar to the goal of blended analysis, the code under analysis is not framework-intensive, as they apply this technique to small C programs that they are model checking. These uses of dynamic analysis to enhance the precision of a subsequent static analysis are similar to blended analyses. One key difference between hybrid slicing and our work is the amount of dynamic information that has to be collected at runtime. Hybrid slicing requires instruction-level traces whereas our blended analysis requires only lightweight method-level tracing.

Mock *et. al.* [MACE02] designed a static slicing algorithm for C programs which used observed dynamic points-to relations – a lower bound on the static points-to relations – instead of computed static points-to relations when forming the slices. The static slices obtained did not capture all possible execution time behavior, but were suitable for their debugging task. 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 did not improve the precision of the static slicing of C programs. Because the dynamic analysis results were applied to obtain smaller slices, but not necessarily more precise slices, so this combination of static and dynamic analyses is different than that of Gupta *et. al.*.

Orso *et. al.* [OAH03] 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. Essentially, the forward slice tracks all things dependent on c, and the intersection with the trace selects all the things which could have been affected by c on this execution. Here, the static analysis improves the relevance of the dynamic analysis information reported, but the combined result is better than either result viewed in isolation.

Godefroid *et. al.* [GKS05] 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. 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.* [SMA05] to form a basis for *concolic testing* methodologies which use both symbolic execution and substitution of concrete values when necessary. The similarity to blended analysis is that a dynamic execution path is being explored by a static analysis (i.e., symbolic execution). The single-path symbolic execution technique has recently been extended by Saxena *et. al.* [SPMS09] to use a set of concrete executions in conjonction with the symbolic execution in order to get achieve greater precision in the presence of loops.

Beyer *et. al.* [BHT08] have described a formalism that allows different explicit and symbolic analyses to be composed and their precision adjusted dynamically based on the computed results. For instance, an explicit analysis can keep track of variables or heap locations precisely, and introduce predicates or symbolic values when the size of the state being tracked exceeds some predefined limits. This work is a generalization of the previous approach by Godefroid *et. al.*, because the precision of any of the composed analyses can be increased or decreased at any point and with arbitrarily fine-grained control (e.g., select which variables to track explicitly rather than apply the concrete execution to all variables).

Recently there have been several explorations of loosely coupled combined analyses. The *Check 'n' Crash* tool [CS05] provides dynamic testing of the errors/warnings reported by *ESC/Java* [FLL+02], in order to filter out false positives. This process consists of a static analysis whose results are checked by a subsequent dynamic analysis. *DSD-Crasher* [CS06] additionally runs *Daikon*, a dynamic analysis tool that finds likely program invariants, to provide additional assumptions to *Check 'n' Crash*. These assumptions help the tool to generate the tests to check the errors reported by *ESC/Java*. This newer process introduces a new dynamic analysis phase before the combined (static followed by dynamic) analysis used in [CS05].

Tomb *et. al.* [TBV07] tried 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 maximum 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.* [AEGK07] described various pipelined combinations of static and dynamic mutability analyses for Java method parameters. Empirical results obtained showed that a pipelined combination analysis can exceed the accuracy of a more complex static analysis.

Inoue *et. al.* [IKN09] described a technique for C that combines a dynamic call path profile with the result of a static analysis that recovers the source code structure from compiled binaries. Their technique uses a looser coupling than blended analysis, but nevertheless shares many of the same concerns. For example, Inoue *et. al.* have to address the effects of optimizing compiler transformations when mapping the dynamic information to source code. Similarly, blended analysis has to address transformations performed by the Just-in-time (JIT) compiler. This issue will be addressed in more details in Chapter 3.

Sinha *et. al.* [SSG$^+$09] described a technique for fault localisation and repair that uses stack traces from faulty concrete executions to guide a static dataflow analysis. The analysis proceeds backward from the point where an exception was raised in order to identify the statement that is responsible for the incorrect assignment that caused the exception. The dynamic information here serves a similar purpose as with blended analysis, but it defines the program region to be analyzed more loosely since the faulty assignments do not necessarily belong to methods that appear in the provided stack trace.

Chapter 3

Blended analysis

There are many problems that require a precise understanding of one or more concrete executions of a program. For example, when confronted with poor performance in an execution, it is necessary to understand more about this particular execution in order to locate the performance bottleneck. When debugging, we often want to study a particular execution that is known to cause an error. Traditionally, dynamic analysis has been used to solve these problems. In many cases, however, obtaining all needed information dynamically is prohibitively expensive or impossible. For instance, tracking pointer relationships between objects at runtime would require tracking all object allocations and pointer updates. Studies such as [HBM+06] and [XAM+09] have shown that obtaining such information precisely can dramatically impact the execution time of a program, and report slowdowns ranging from several times to two orders of magnitude. Such slow downs not only perturb the execution of programs and thus the quality of the information that is collected, but they also render such dynamic analyses useless for production systems. The goal of blended analysis is therefore to enable analyses of such systems that achieve a precision comparable to that of a dynamic analysis while minimizing the runtime cost. This chapter presents the blended analysis paradigm, its novel aspects as well as the new challenges it raises.

3.1 Paradigm

Traditional static analysis is usually concerned with computing results that are valid for all possible executions of a program. While there are a large number of potential uses for general results computed using a static analysis, there are also many practical applications for an analysis of even a single execution. For instance, 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.

Blended analysis is a new analysis paradigm that performs a static analysis on a subset of an application that is determined using dynamic analysis. Blended analysis is therefore designed to capture properties of a finite set of concrete executions rather than all possible executions. By limiting the scope of a static analysis to the executed portion of the code, it is possible to analyze programs for which traditional static analysis does not scale. In addition, excluding unexecuted code from the analysis leads to results that more accurately reflect the observed behavior of the program, and often translates into an increased precision compared to a traditional all-static analysis.

The blended analysis paradigm allows the analysis design space to be explored. Additional dynamic information can be passed from the dynamic analysis component to the static analysis. For example, an analysis could use dynamic method execution frequencies to rank results in terms of relevance, rather than relying on purely static heuristics. Additionally, the results of a blended analysis could be used to direct a subsequent dynamic analysis. There are many potential uses for such a technique. For instance, when a blended analysis determines that it needs more information about an unexplored portion of the program, it could signal the dynamic analysis to collect the necessary data. Alternatively, a dynamic analysis could leverage the results of a blended analysis to determine which program regions should be the focus of a more detailed investigation.

3.2 Safety

The theoretical goal for most analyses is to compute an *ideal solution* that precisely accounts for all possible executions of a program. In practice, however, this computation is undecidable, and analysis designers need to resort to computing subsets or supersets of the ideal solution. A static analysis is considered *safe* if it always computes a solution that is valid for all possible executions of a program (i.e., a superset of the ideal solution). Since blended analysis is only concerned with a finite set of concrete executions of a given application, it is not generally safe for all executions. If the static analysis component of a given analysis is safe, however, the blended analysis is safe *for the particular execution(s) being considered*.

Figure 3.1 compares various analysis paradigms in terms of their safety property. Notice that unlike dynamic analysis, blended analysis does not always compute a proper subset of the ideal solution. This is due to the conservative approximations made by the static analysis component of a blended analysis. On the other hand, blended analysis is able to capture behavior that an unsafe static analysis can miss. For example, many static analyses

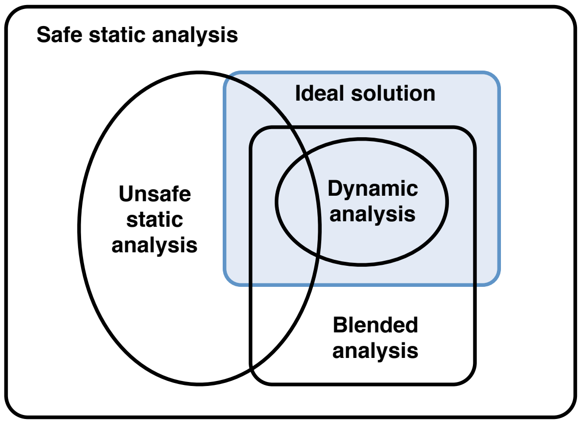


Figure 3.1: Comparison of safety property of various analysis paradigms

are unsound in the presence of reflection in order to avoid making worst case assumptions that lead to scalability issues and loss of precision. In contrast, blended analysis has access to precise dynamic information about the reflective behavior of an application, and is able to replace problematic reflective calls with equivalent code in a way that is safe for the executions being considered.

3.3 Calling structure

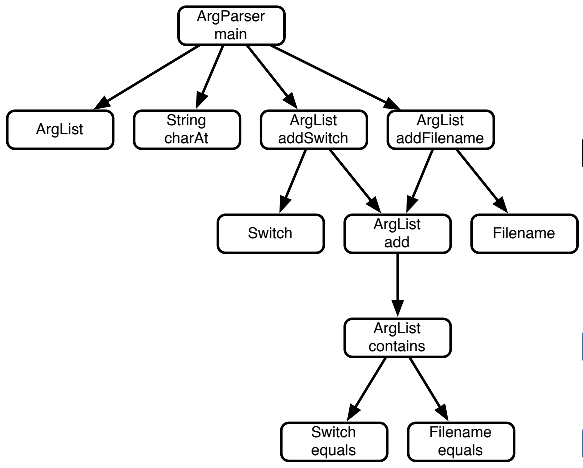
Interprocedural static analysis typically explores a program by following calls between methods, either forward (i.e., from caller to callee) or backward (i.e., from callee to caller). This requires knowledge about the possible target methods at each call site. The vast majority of static analyses obtain this information by building a *call graph* from the analyzed code. A call graph is a directed graph in which nodes represent methods in a program, and edges represent calls between methods. A basic call graph comprises a single node for each method in the program. It is however possible for multiple nodes in the call graph to correspond to the same method. Such nodes are said to represent

```
public class ArgParser {
   public static void main(String[] args)
   {
     ArgList theList = new ArgList();

     for (String arg: args) {
       if (arg.charAt(0) == '-') {
         theList.addSwitch(arg);
       } else {
         theList.addFilename(arg);
       }
     }
   }
}

interface Argument {
  public boolean equals(Object obj);
  public int hashCode();
}

class Switch implements Argument {
  ...
}

class Filename implements Arguments {
  ...
}

class ArgList {
  private Node head;
  private Node tail;

  public void addSwitch(String name)
  {
    this.add(new Switch(name));
  }

  public void addFilename(String name)
  {
    this.add(new Filename(name));
  }

  public void add(Argument arg)
  {
    if (!contains(arg)) {
      Node n = new Node();
      n.argument = arg;
      if (head == null) {
        head = tail = n;
      } else {
        tail.next = n;
        tail = n;
      }
    }
  }

  private boolean contains(Object obj)
  {
    Node n = head;
    while (n != null) {
      if (obj.equals(n.argument)) {
        return true;
      }
      n = n.next;
    }

    return false;
  }

  private class Node {
    public Object argument;
    public Node next;
  }
}
```

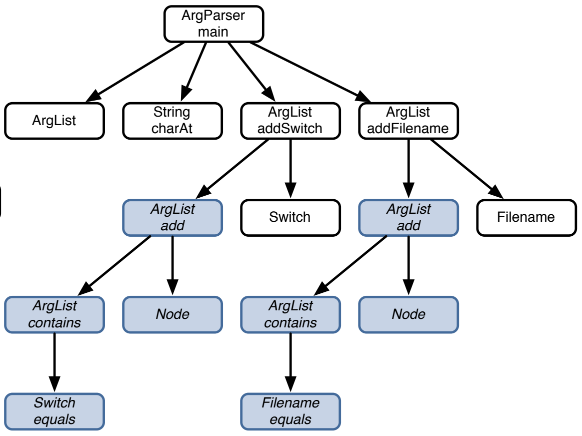
Figure 3.2: A simple command-line parser example

that method in different *contexts*. Examples of common contexts include the caller of a method [GDDC97] or its associated receiver object [MRR05]. Because of its importance to static analysis, call graph building is a heavily studied topic. Many call graph building algorithms have been defined, each with particular tradeoffs between cost and precision (e.g., [DGC01, TP00, LH03, LH06]).

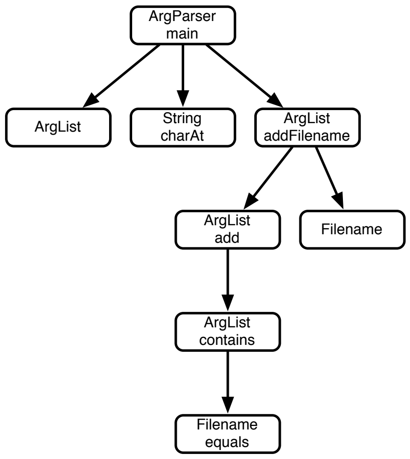
Consider the simple program shown in Figure 3.2. This application performs rudimentary command-line parsing: for each argument passed to it, it creates either a `Switch` object or a `Filename` object, which is then inserted in a linked list of parsed arguments



(a) Static context-insensitive call graph



(b) Static context-sensitive call graph



(c) A possible dynamic call graph

Figure 3.3: Calling structures for the `ArgParser` example

after checking for duplicates. The context-insensitive, static call graph for this program appears in Figure 3.3a. It captures all of the possible calls in the program (e.g., `main` can call four different methods), but it also introduces some imprecision. Specifically, because both `addSwitch` and `addFilename` call `ArgList.add`, the call graph makes it appear that calls to `addSwitch` can lead to calls to `Filename.equals`, and that calls to `addFilename` can lead to calls to `Switch.equals`. Clearly, this is not the case. A more precise call graph which maintains calling context, such as the one shown in Figure 3.3b, addresses that problem at the expense of a larger call graph representation.

Observe that the graph shown in Figure 3.3b contains more nodes due to the added context (as shown by the highlighted nodes).

In a blended analysis, the call graph only contains calls to methods that were executed, and excludes calls to all other methods. When used to drive a static analysis, the dynamic call graph ensures that only executed methods get analyzed. This allows a blended analysis to analyze a potentially much smaller portion of the code, and thus compute its results using less resources. For instance, consider once again the example program in Figure 3.2. Assuming that only filenames are passed to the program, the resulting dynamic call graph would look like that in Figure 3.3c. Specifically, calls to `addSwitch` and `Switch.equals` are no longer present because they did not happen during the execution being considered.

The dynamic call graph in blended analysis is obtained from an execution trace. The trace is often represented as a *call tree*. Call trees are similar to call graphs except that they contain a distinct node for each invocation of a given method at runtime. They are typically very large, even for relatively short program runs. Fortunately, such detailed traces often contain more information than may be required. More concise representations of the calls can easily be obtained by aggregating nodes in the call tree. For example, a basic (context-insensitive) call graph can be derived from a call tree by merging all nodes that represent invocations of the same method. Other aggregation schemes that represent calls more accurately are also possible. For instance, *Calling Context Trees (CCTs)* [ABL97] are a popular way to represent dynamic calls. The key idea behind CCTs is to differentiate between calls of a method based on the full invocation stack that resulted in the call. CCTs offer a much more precise representation of the calls in a program than a basic call graph, but without the prohibitively high cost of full call trees. While techniques exist to collect CCTs directly at runtime (e.g., [ZSCC06]), it is easy to generate them from an existing call tree by aggregating nodes that share the same method sequence from the beginning on the trace.

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. For blended analysis, we promote them to program entry points, as in static analysis. Similarly, calls to runtime 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. In certain cases, however, it could be desirable to analyze this code. For example, class loaders can perform costly operations and lead to performance problems. Such code can be promoted to program entry points (as it is done with static initializers) in order to ensure that it will be analyzed. Alternatively, synthetic call sites can be added in the corresponding bytecode, thereby enabling the analysis of the runtime support code while preserving its calling context.

Sometimes, a transaction or a *scenario* (i.e., a partial transaction with specific functionality) of interest is identified in advance and the execution trace is limited to that part of the application. For example, if a specific set of inputs to a transaction are known to trigger a performance problem, then it is often desirable to trace this transaction in isolation. The calling structure is therefore restricted to that part of the program that was exercised at runtime. Because the program region to be examined by the analysis does not include all natural entry points (e.g., `main`), the analysis must be started from arbitrary methods that often have reference parameters. While some analyses are unaffected by the presence of objects created outside of their scope, others require information about these objects. Currently, 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 corresponding objects are synthesized.

3.4 Dynamic language features

Dynamic class loading and reflection are typically difficult problems for a traditional static analysis. They commonly force analysis designers to make worst-case assumptions or require user input to specify the set of all possible classes that can be loaded and all possible targets at each reflective call site. Blended analysis does not require this effort because the necessary information can be recorded during the execution of the program. For instance, a profiler can easily collect classes as they are loaded. Even dynamically generated classes that do not exist statically (e.g., those generated by the JVM 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, and is able to operate under closed-world assumptions.

Dynamically obtained calling structures contain explicit behavioral information at reflection points in the program. By using this information, it is possible to eliminate the need for these assumptions. Specifically, bytecode can be synthesized at each reflection point to capture the observed dynamic behavior precisely in terms of method calls and allocated objects. As there are relatively few reflective methods in the Java standard library, it is easy to manually construct individual models. Model creation is discussed in more details in Section 3.6.

3.5 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 collected dynamic information 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 all basic blocks that (i) are found on a path from entry to exit that contains at least one unexecuted basic block and (ii) are not shared between this path and other, possibly executed paths.

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 Chapter 6.

3.6 Elude

We have implemented *Elude*, a publicly available framework for blended analysis of Java programs. Figure 3.4 shows the overall architecture of the tool. *Elude* is composed of two main components: a dynamic analysis component based on the *Jinsight* profiler [DJM+02] and a static analysis component based on the WALA framework [WALA]. Each component is discussed in more detail next.

3.6.1 Dynamic analysis

The dynamic analysis component of *Elude* is responsible for two main tasks: collecting classes that are loaded by the Java virtual machine and generating dynamic calling struc-

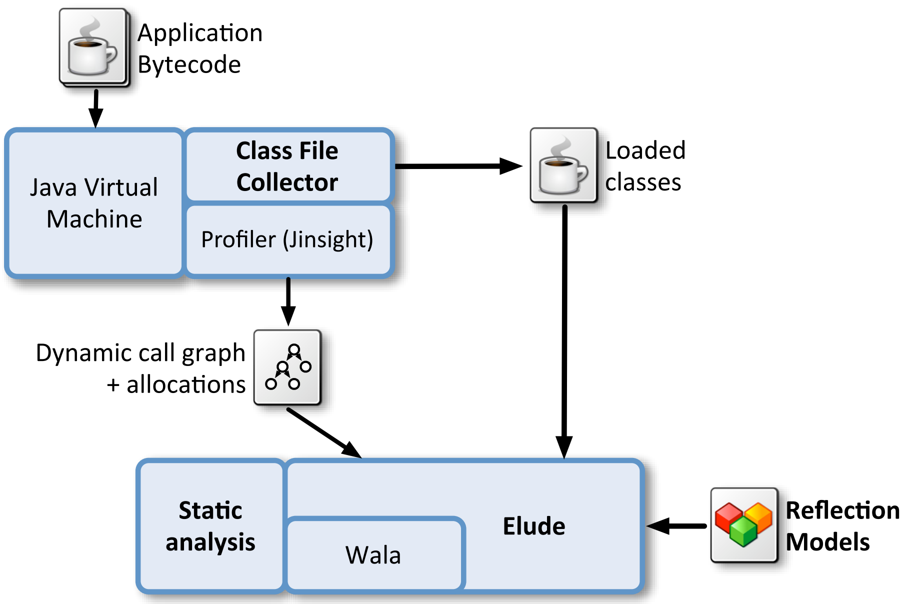


Figure 3.4: The *Elude* framework

tures used by the static analysis component. The class file collector uses the Java instrumentation capabilities to intercept class loading and write the data to disk. The dynamic calling structures are generated from traces collected by the *Jinsight* profiler. This profiler is routinely used within IBM for performance diagnosis. Using *Jinsight* to generate dynamic calling structures offers two main advantages. First, it ensures that our technique can easily be integrated in the normal performance understanding workflow, thus lowering the cost of adoption. Second, extending the tool to generate dynamic calling structures from existing, unmodified traces provides easier access to data from real production applications. Currently, the tool supports the generation of context-insensitive call graphs as well as other context-sensitive calling structures such as CCTs. Because multiple calling structures can be generated from a single trace, the modified *Jinsight* tool does not require rerunning the program in order to experiment with various levels of aggregation in calling structures.

Our choice of profiler also limits the amount of information that can be collected at run-time. 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. In such cases, we safely assume that any matching site was potentially executed. This may force our analysis to consider unexecuted code, but it ensures analysis of all executed code. Building a specialized profiler for blended analysis (e.g., using lightweight bytecode instrumentation) could address this limitation, and is a possible direction for further exploration.

Note that because *Jinsight* is a proprietary IBM product and not publicly available, we have chosen to decouple *Elude* completely from the profiler by using a simple textual format for its call graphs. Any existing profiler capable of recording dynamic information that is suitable for generating a dynamic calling structure should be trivial to extend to produce *Elude*-compatible call graphs.

3.6.2 Static analysis

The static analysis component in *Elude* is built on WALA, an open source static analysis framework. *Elude* receives as input a dynamic calling structure as well as a set of classes to analyze from which *Elude* generates WALA-compatible structures. Therefore, any WALA analysis can be converted into a blended analysis.[1]

In order to properly align static and dynamic information, *Elude* needs precise behav-

[1]Some modifications to the analysis may be required.

```
@Model(
 "java.lang.reflect.Array.get(Ljava/lang/Object;I)Ljava/lang/Object;"
)
public static Object java_lang_reflect_Array_get(Object array,
       int index) {
   return ((Object[]) array)[index];
}
```

Figure 3.5: A Java model

ioral models of the parts of the program for which no bytecode is available (e.g., native methods). To this end, *Elude* contains a complex modeling infrastructure that allow models to be specified using different formalisms. For instance, models can be generated automatically by providing Java code that has comparable behavior. Figure 3.5 provides an example of such a high level model specification for a reflective array access operation. Specifying models using Java code allows for the very quick development of new models. Although *Elude* currently contains a set of models for the IBM 1.4.2 JRE as well as the IBM J9 1.5 JRE, these models must be updated with new Java releases, or for different JVM vendors. Note that for maximum flexibility in specifying models, *Elude* also supports creating models by directly building a WALA-compatible 3-address code representation.

Elude also supports parameterized model specifications for which code is generated based on dynamic input. Such specifications are particularly well suited for handling reflection. Figure 3.6 shows a simple parameterized model specification for the reflective creation of array objects. The model code first obtains a list of array types that were allocated at runtime, and then proceeds to generate code that generates arrays of each of these types. Note that Java annotations are used extensively as part of the modelling infrastructure to simplify the specifications as well as improve readability. In this case, the model is applied

```
@Model({
 "com.ibm.jvm.ExtendedSystem.newArray(Class,int,Object):Object",
 "java.lang.reflect.Array.newArray(Class,int):Object"
})
public MethodSummary makeArrayFactoryModel(ReflectionEnvironment env,
      CGNode caller, IMethod target) {
   // Get the dynamic information for the modeled method
   ENode targetNode = env.getUniqueTargetNode(caller, target);
   Set<TypeReference> allocTypes = env.getAllocatedTypes(targetNode);

   SummaryBuilder builder = new SummaryBuilder(target);
   int[] dims = new int[] { builder.getParameter(1) };

   // Add an allocation for each array type
   BitSet values = new ArrayBitSet();
   for (TypeReference type: allocTypes) {
      int result = builder.addNewArray(type, dims);
      values.add(result);
   }

   // Create a new variable for the return value
   // that contains all created arrays
   int result = builder.addPhi(values.toArray());
   builder.addReturn(result);

   // Create the parameterized model
   return builder.build();
}
```

Figure 3.6: A parameterized model

to two different methods, `ExtendedSystem.newArray` and `Array.newArray`.

Chapter 4

Blended escape analysis

4.1 Motivation

Understanding performance problems in framework-intensive applications 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 [MSS06]. 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 [SS05], 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 an average calling depth of 39. Clearly, tool support is required to help developers understand the behavior of their applications.

Much of the work performed by framework-intensive applications involves the creation and initialization of *temporaries*, short-lived objects that are created as the by-product of some computation. The excessive usage of temporaries, known as *object churn*, is a new but widespread problem in framework-intensive applications. The combined cost of allocating, initializing and ultimately garbage collecting temporaries can dominate execution time, and degrade performance dramatically. 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.

Despite the sophisticated optimizations used by modern just-in-time (JIT) compilers, object churn remains a problem in many commercial applications. In particular, even attempts to stack-allocate temporary objects at runtime are typically insufficient for eliminating object churn. Temporary object costs are high even with improvements in memory management such as the use of *nursery* space in the garbage collector for the allocation of short-lived objects. While the cost of allocating physical memory for an object may be low, the initialization of this object is often much more expensive. In addition, object churn cannot be alleviated solely by optimization of individual frameworks, because often the temporaries are passed in calls across framework boundaries.

In order to address object churn, it is necessary to understand why temporaries are created and how they are used within an application. 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. Current profiling tools such as *Jinsight* [DJM+02], *HPROF* [HPROF], *Arcflow* [ABLU00], and even commercial products like *YourKit* [YourKit] and *CodePro* [CodePro] are inadequate for identifying object churn. While they provide some valuable information about new objects, such as the context in which they are created and whether they survive a garbage collection, these tools offer limited information about how these objects are used.

We show the usefulness of the blended analysis paradigm by obtaining characterizations of temporaries in framework-intensive applications and of the program regions that create and use these temporaries. Specifically, by using a blended escape analysis to identify which objects are local to a program region, the set of temporaries can be approximated. Program regions can then be ranked in terms of the number of temporaries that they contain. Moreover, because escape analysis subsumes points-to analysis, this technique allows us to reason about how object instances become connected during execution, and thus study temporaries as elements of larger data structures manipulated by the program. This information allows temporaries to be grouped by their connectivity into data structures, and can ultimately enable characterization studies as well as better understanding of specific temporary structures. 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 specialization optimizations, best practices for framework API design and usage, and/or better diagnosis and assessment tools for framework-intensive applications.

4.2 Background

Escape analysis computes bounds on the dynamic scope of objects. It was first proposed to enable compiler optimizations such as stack allocation of objects, which reduces heap fragmentation and garbage collection overhead by allocating object on the run-time stack rather than the heap, and synchronization removal, a technique that avoids costly synchronization operations when code can be shown to be thread-safe. The former requires information about objects that escape a particular method invocation; the latter necessitates knowing which objects escape their allocating thread. More formally, 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 from a reference at any point at a point in execution outside of t. 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*.

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

4.2.1 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 [BH99] 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 [GS00] proposed an algorithm that relies on assigning a *fresh* status to a new object, which is then propagated to methods and references, and is used to compute escape information. Beers *et. al.* [BSF04] have proposed an escape analysis algorithm specifically designed to have its results encoded as class file attributes and used at runtime by the 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).

4.2.2 Dataflow algorithms

Dataflow algorithms compute escape information using a set of equations that describe the effect of each node in the control flow graph of a method on the solution. These equations are evaluated iteratively until the solution reaches a stable state (*fixed point*). This process is repeated for each method in the analyzed part of the application. 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 [WR99] and one by Choi *et. al.* [CGS^{+}99, CGS^{+}03]. 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). Both algorithms are context-sensitive and flow-sensitive [Ryd03].

The algorithm by Choi *et. al.* relies on *connection graphs* to represent relationships between references and abstract objects. The analysis proceeds in a bottom-up manner on the call graph; cycles in the call graph are handled by iterating until the solution converges. 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. Connection graphs contain two main kinds of nodes: *object nodes* representing abstract objects and *reference nodes* corresponding to variables or fields in the program. Each node is decorated with its current escape property. 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 callee to caller during the interprocedural analysis.

```
interface X {...}
class Y implements X {...}
class Z implements X {...}
class G {
   public static Object global;
}

public class EscapeExample {
   public static X identity(X p1) {
      return p1;
   }

   public static X escape(X p2) {
      G.global = p2;
      return p2;
   }

   public static void f() {
      X inst;
      if (...)
         inst = identity(new Y());
      else
         inst = escape(new Z());
   }
}
```

Figure 4.1: A simple example for escape analysis

4.2.3 Example of escape analysis

The example in Figure 4.1 is designed to illustrate the main features of the escape analysis algorithm by Choi *et. al.*; the corresponding call graph appears in Figure 4.2. The example code contains four leaf methods in its call graph: `identity`, `escape`, and the constructors for classes `Y` and `Z`. Assuming that `Y` and `Z` are default constructors, only `identity`, `escape` and `f` are interesting from the point of view of escape analysis. Consider the `identity` method. The analysis first creates a reference node in the connection graph

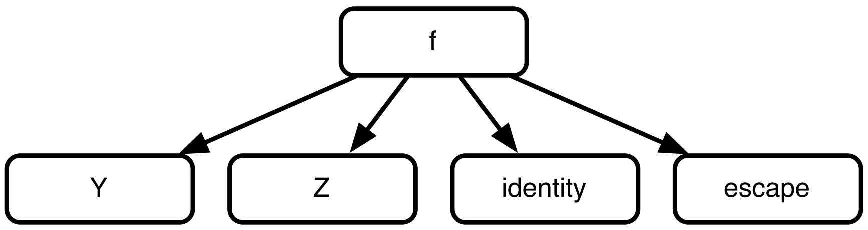
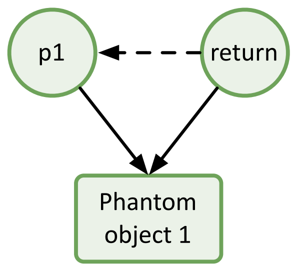


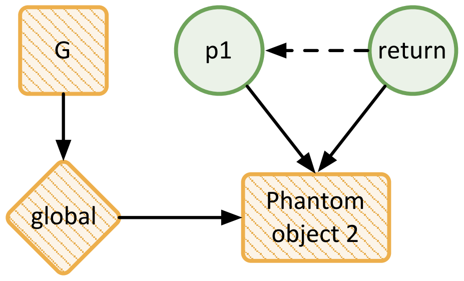
Figure 4.2: Call graph for the escape analysis example

corresponding to the `p1` parameter. Because the object pointed to by `p1` is external and therefore unknown at this point, the analysis models it with a *phantom* object node. When processing the return statement, another reference node is created for the return value of the method, and made to point to `p1`. At the end of the dataflow computation for the method, all reference nodes are made to point directly to objects that are transitively reachable from them, and edges between reference nodes are removed. The final connection graph for the `identity` method appears in Figure 4.3a. Note that both reference nodes are marked as arg-escaping the method, as indicated by their 🟩 pattern. The analysis proceeds similarly for the `escape` method. A reference node is created in the connection graph for `p2`, and is made to point to a new phantom node representing the external object passed as argument. In this case, however, the argument is assigned to a static field. The `G` object and its `global` field are added to the connection graph and marked as globally escaping. The analysis then proceeds as before. The final connection graph for `escape` is shown in Figure 4.3b. Note that escape states are ultimately propagated along connection graph edges, causing the phantom object node to also be marked as globally escaping (as indicated by the ▨ pattern).

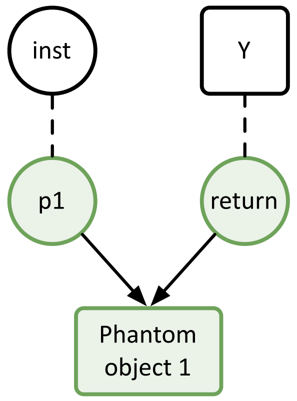
After the analysis has processed all leaf methods, it can process the `f` method using the previously computed information. First, it creates a reference node for the `inst` variable.



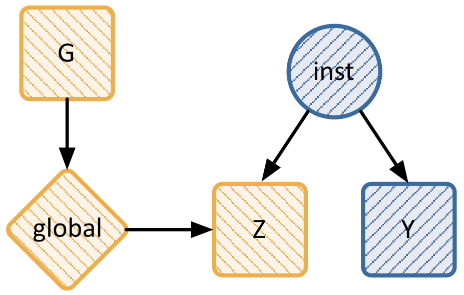
(a) Connection graph for `identity`



(b) Connection graph for `escape`



(c) Mapping the connection graph for `identity` into `f`



(d) Connection graph for `f`

Figure 4.3: Connection graphs for the escape analysis example

Each branch of the conditional statement is then analyzed sequentially. In the `true` branch, a new object node is created for the allocation of a `Y` object. The call is then analyzed by mapping actuals to the parameters in the callee (i.e., mapping `Y` to `p1`), and processing assignments from return values if applicable (i.e., assigning the result of `identity` to `inst`). The mapping process, illustrated in Figure 4.3c, results in an edge being added in the connection graph from `inst` to `Y`. Note that arg-escaping objects are marked as captured in the caller at this point (as indicated by their ▨ pattern). The other branch is analyzed similarly. The final connection graph for `f` appears in Figure 4.3d. Note that the `Z` object is marked as globally escaping since it is referenced by the `global` field, while the `Y` object is never used and thus remains captured.

4.3 Blended escape analysis

We implemented a blended version of the Choi *et. al.* escape analysis using *Elude*. The details of the experiments performed with the tool will be described in Chapter 6. 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 in Chapter 6, this additional information is useful for understanding program behavior and data manipulation, in contrast to previous uses of escape analysis.

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 declared types; thus, type-inconsistent edges are never added to the connection graph. This optimization is well-known in points-to analysis, and has been shown to significantly increase the precision and to reduce the execution time cost [LH03].

4.3.1 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. Recall from Chapter 3 that a dynamic calling context tree (CCT) 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. In contrast, a call graph is a context-insensitive calling structure where the same node represents all invocations of a given method. Note that the presence of recursion is a special case that introduces cycles in a CCT.

The post-processing 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 CCT node, 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 a different choice of calling structures for a blended analysis, albeit at a higher cost for the analysis itself due to the larger size of context-sensitive calling structures as compared to context-insensitive call graphs. Studying the impact of varying the level of context sensitivity in the calling structure representation on the cost and precision of blended analysis is an interesting research direction, and is left as future work.

The post-processing phase of the blended escape analysis manipulates both static and dynamic object abstractions at the same time. For clarity, in this discussion 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 (i.e., allocation sites in our analysis).

For each context node in the CCT, the postprocessing algorithm generates a *reduced* *connection graph* from the connection graph computed by the blended escape analysis for the corresponding method. The reduced connection graph more accurately represents the escape behavior at that node. To construct the reduced connection graph, each node in the original connection graph is first annotated with the number of instances that it possibly represents. 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. We also 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. This is achieved by propagating visible instances backwards in the CCT.[1] Note that because paths in the CCT are more precise representations of call behavior than the paths used to compute the blended escape analysis, it is possible for objects to be captured earlier than indicated by the escape analysis results, thereby no longer being visible at a context for which the original connection graph contained a matching object. 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. We therefore use a single object to represent all allocation sites of the same type in the same method. We refer to these objects as *merged abstract objects*. This transformation prevents many of the issues of double-counting instances when they cannot be mapped to a unique allocation site. Finally, for ease of understanding, we simplify the connectivity in the reduced connection graphs, by eliminating field information and eliding links other than those relating objects to one another (such as links from ref-

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

erence variables and parameters). Figure 4.4 shows an example of a reduced connection graph generated by the post-processing algorithm. The graph was obtained from *Trade*, a financial benchmark for the Websphere Application Server.

The reduced connection graphs are used for two purposes. First, they allow the number of instances captured at each CCT node to be computed, thus enabling the identification and ranking of the nodes according to their usage of temporaries. Second, they summarize the connectivity of objects at a given CCT node 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.

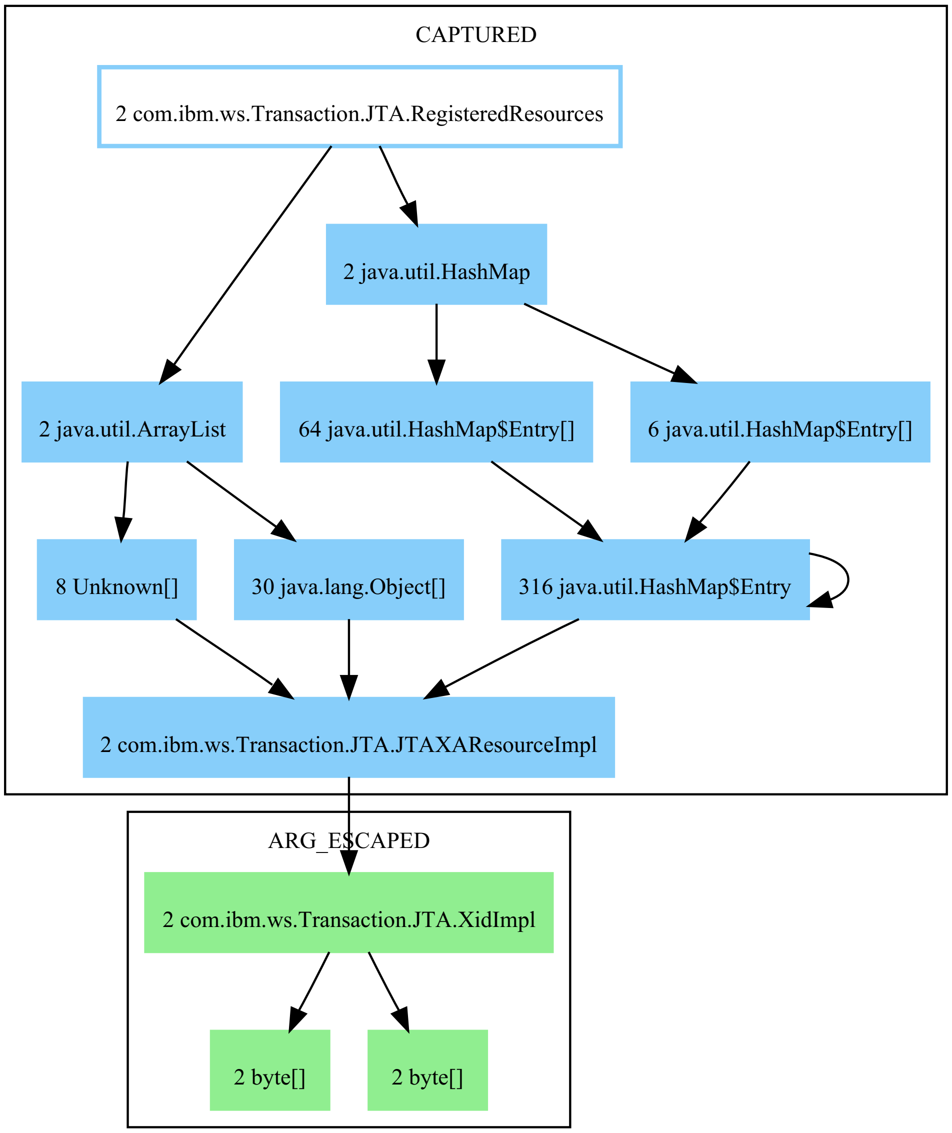

CAPTURED
2 com.ibm.ws.Transaction.JTA.RegisteredResources
2 java.util.HashMap
2 java.util.ArrayList
64 java.util.HashMap$Entry[]
6 java.util.HashMap$Entry[]
8 Unknown[]
30 java.lang.Object[]
316 java.util.HashMap$Entry
2 com.ibm.ws.Transaction.JTA.JTAXAResourceImpl
ARG_ESCAPED
2 com.ibm.ws.Transaction.JTA.XidImpl
2 byte[]
2 byte[]

Figure 4.4: Example of reduced connection graph for `XATransactionWrapper.enlist` in *Trade* showing a large number of captured instances. Each node in the graph shows the type of object represented along with the number of instances of that object observed at runtime.

Chapter 5

Metrics

The blended analysis presented in Chapter 4 exposes previously unexplored characteristics of framework-intensive applications. In order to evaluate the effectiveness of the blended analysis paradigm to analyze and understand the behavior of real framework-intensive applications, it was necessary to develop new metrics that quantify key aspects of the algorithm and the behavior of framework-intensive applications with respect to temporaries. This chapter presents our new metrics along with a detailed discussion of the various factors that affected their design.

5.1 Measurement goals

There are three major measurement goals for the new metrics:

(i) To determine the effectiveness of the control-flow graph (CFG) pruning technique on the blended analysis algorithm,

(ii) To characterize the usage of the temporaries in framework-intensive applications,

(iii) To characterize the nature of the 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, and focus on improvements in analysis *scalability* and *precision*. Metrics covering goal (ii) capture properties of the execution related to the usage of temporaries. Such properties include, for example, the escape categorization of an object as either captured or escaping, or the distance from allocation to capture for each object. Metrics covering goal (iii) quantify the complexity of temporary data structures in terms of the objects they contain and their interconnections. Note that the first goal applies to all blended analysis algorithms, while the other two are specific to our blended escape analysis algorithm.

5.2 Design factors

Designing useful metrics requires careful consideration of many factors. In this work, we are concerned with three main factors that influenced the design of our metrics.

5.2.1 Comparability

Some metrics are intrinsically comparative measures. For instance, metrics that measure the scalability improvements due to CFG pruning are by definition comparing the pruned and unpruned versions of a blended analysis. Other metrics could be designed to be compared across several benchmarks, or different executions of the same benchmark. In all cases, it is important to ensure that the properties that are measured can be meaningfully compared. Consider a metric that measures the percentage of allocation sites that only create temporary (i.e., captured) objects. This metric can meaningfully be compared across different benchmarks in order to assess the relative importance of temporaries in different applications, or to identify likely targets for code optimizations. The same metric would be much less useful for comparing pruned and unpruned analysis results for a single benchmark, as the absolute number of call sites is subject to change due to the removal of unexecuted basic blocks.

5.2.2 Object abstraction

The nature of a blended analysis introduces a duality in object representations. The execution trace contains information about dynamically allocated object instances. Static analysis, however, typically uses object abstractions such as allocation sites. Because information computed for a static abstract object can be mapped to the corresponding dynamic instances in the trace, a metric pertaining to objects can be defined to use either static abstract objects or dynamic object instances as object representation. There are advantages to each representation, and the choice between them is often influenced by the intended use for the metric. Static abstract objects can be useful for characterization metrics or program understanding tasks, for example. Dynamic instances are often useful to find performance bottlenecks or to estimate the profitability of compiler optimizations, among other uses.

5.2.3 Data structure abstraction

In object-oriented systems, and framework-intensive applications in particular, individual objects are often organized into more complex data structures in order to perform a given task. Understanding the behavior of framework-intensive applications therefore requires understanding the data structures that they create. In order to study data structures, a clear definition of what constitutes a data structure is required. In this work, we define data structures in terms of objects in the reduced connection graphs computed by our blended escape analysis. More precisely, a data structure is a root object in the reduced connection graph for a method (i.e., an object with no incoming edges or whose only incoming edges are back edges), along with all objects reachable from the root. The rich information associated with reduced connection graph objects allows a wide array of properties to be computed for each data structure, such as the distinct object types in a data structure, the set methods that contributed objects to a data structure, or the capturing path of a data structure, to name only a few. Because every object has a corresponding set of associated instances, we can also compute 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).

5.3 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 [MR90]. 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 3.6, *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*.

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 discussions that follow, we use the term *object* to refer to an object in the reduced connection graph *after* this merging transformation has been applied.

5.4 Metric definitions

This section defines a set of 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. A detailed empirical evaluation using these metrics is deferred until the next chapter.

5.4.1 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. Because flow-sensitive analyses like our blended escape analysis require state to be kept for each basic block while analyzing a given method, pruned basic blocks translate into time and memory savings for the analysis.

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).

5.4.2 Disposition

Recall from Chapter 4 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 post-processing). 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 post-processing 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 escaping*, *captured* or *mixed*. This metric is used to characterize the usage of temporaries: a large proportion of captured instances indicates that an application is probably making an excessive use of temporaries, and is a prime subject for object churn. This metric could alternatively be computed using objects rather than instances in order to compute the proportion of allocation sites that only allocate temporaries, and therefore provide an upper bound on the number of allocation sites that could be targeted by compiler optimizations aiming to ameliorate object churn.

Metric 4: *Disposition improvement (by objects)*

The percentage of objects (i.e., allocation sites) 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.

5.4.3 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 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.

5.4.4 Concentration

The previous metric, capturing depth, is a descriptive measure of the individual program regions that use temporaries. The concentration metric looks at how the object churn costs are distributed across such regions in the application scenarios we are analyzing. The goal of this metric is 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. In this work, we used 5%, 10% and 20% as values for X. This metric is used to characterize the usage of temporaries.

5.4.5 Complexity of data structures

Temporaries often are organized into complex data structures that can be expensive to create not only due to the cost of allocation and initialization of the constituent instances, but also because of 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. Because we are interested in characterizing temporaries, we compute metrics only over captured data 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 out of a loop, pooling similar structures that have expensive construction for reuse, or specializing structures for a specific usage pattern.

We investigate the complexity of data structures and characterize them by computing the following five 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: *# of merged abstract objects*

The number of merged abstract objects in each data structure. Recall that because of dynamic imprecision, we use *merged abstract objects* as an approximation of allocation sites. This metric is therefore an approximation of the number of allocation sites that contribute to a data structure. Each merged abstract object represents all allocation sites with the same allocated type in a method (or CCT context). The complexity of a data structure increases directly with increases in this metric.

Metric 10: *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 11: *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 five data structure 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.

Chapter 6

Empirical evaluation

Chapter 5 has introduced a set of new metrics that aim to capture specific properties related to the usage of temporaries in framework-intensive applications. This chapter presents empirical results for each metric obtained from a set of representative framework-intensive applications, including well-established benchmarks as well as commercial applications. Section 6.1 discusses each of the benchmarks in detail. Section 6.2 presents the metric results along with an interpretation of the relevant findings. Section 6.3 presents a specific case study that demonstrates the usefulness of our approach for understanding object churn and the usage of temporaries, using one commercial application as a subject.

6.1 Experimental setup

For our experiments, we used four framework-intensive applications: *Trade*, *Eclipse*, *Jazz* and a private commercial application (*CDMS*). Our escape analysis is built using the WALA analysis framework.[1] 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 to prevent method

[1]`http://wala.sourceforge.net/`

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.[2] Our test machine was a Pentium 4 2.8 GHz machine with 2 GB of memory running the Linux kernel version 2.6.12.

6.1.1 Trade 6

We used version 6.0.1 of the *Trade* benchmark running on Websphere 6.0.0.1 and DB2 8.2.0. [3] The way in which the *Trade* benchmark interfaces with the Websphere middleware can be configured through several parameters. We experimented with four 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)*.[4] 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 four 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

[2]Instrumentation-based profiling techniques generate accurate call graphs even in the presence of inlining.
[3]*Trade*, Websphere and DB2 are available to academic researchers through the IBM Academic Initiative.
[4]*Trade* 6 uses the EJB 2 framework.

Benchmark	**Allocated Types**	**Allocated Instances**	**Methods**	**Calls**	**Max Stack Depth**
Direct-Std	30	186	710	4,484	26
Direct-WS	166	5,522	3,308	127,794	53
EJB-Std	82	1,751	1,978	60,936	62
EJB-WS	210	7,088	4,479	184,288	72
Eclipse	168	53,191	1,411	1,081,927	53
Jazz	181	9,470	2,547	170,311	86
CDMS	254	62,066	3,144	1,495,192	50

Table 6.1: Benchmark characteristics

all required classes to be loaded and caches to be populated. Tracing the benchmark in a steady state is more representative of the behavior of real Web applications.

Because the *Trade* application consists of a relatively small user code that interacts with a large amount of framework and library code, the four configurations of the same application have very different properties and behavior in practice. Therefore, we use these four configurations as different benchmarks, as have other researchers [SS05]. These differences are confirmed by the data in Table 6.1 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 four *Trade* benchmarks. The results also attest to the complexity of these framework-intensive benchmarks. Note that when the observed scenario in *Direct-WS* runs, it allocates 166 types of objects and experiences call stack depths of over 50.

6.1.2 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.

6.1.3 Jazz server

Jazz is a scalable collaborative development platform that integrates a number of features including work item management directly in the development process. Jazz uses a client-server architecture, where the clients are integrated development environments (IDEs) based on Eclipse and the server is based on Websphere. We traced version 1.0M5a of the Jazz server while a user was executing a query to list all open work items assigned to him. The query returned a single item. In order to ensure a steady state of the application, we performed the query a first time, then performed a different query (in order to avoid possible caching optimizations), and finally traced another execution of the original query. We focused the analysis on the part of the trace that fetches the query results from the database.

6.1.4 Commercial document management server (CDMS)

This application is a commercial platform that provides management and workflow for documents. We traced a part of the test suite responsible for storing 10 documents in the database. There is no processing of the documents themselves; rather, the system stores metadata about each document in the database. As this software is proprietary and was made available to us under a strict non-disclosure agreement, it is not possible to disclose more information about it. Note that in the rest of the discussion, all package names for classes in this application have been changed to `cdms`.

6.2 Empirical Results and Interpretation

In this section we present empirical data for metrics presented in Chapter 5, and discuss specific findings and observations based on the results.

6.2.1 Pruning effects

Recall from Section 3.5 that calls and object allocations performed by a method at run-time are used to identify unexecuted basic blocks within a method, which are then removed from the control-flow graph (CFG) of a method and therefore not analyzed. In order to measure the effectiveness of the basic block pruning on the escape analysis, we analyzed each benchmark twice: first with all pruning disabled, then with pruning enabled. We measure the impact of the pruning technique on the scalability of the blended escape analysis in terms of three different characteristics: pruned basic blocks, execution time and precision improvements.

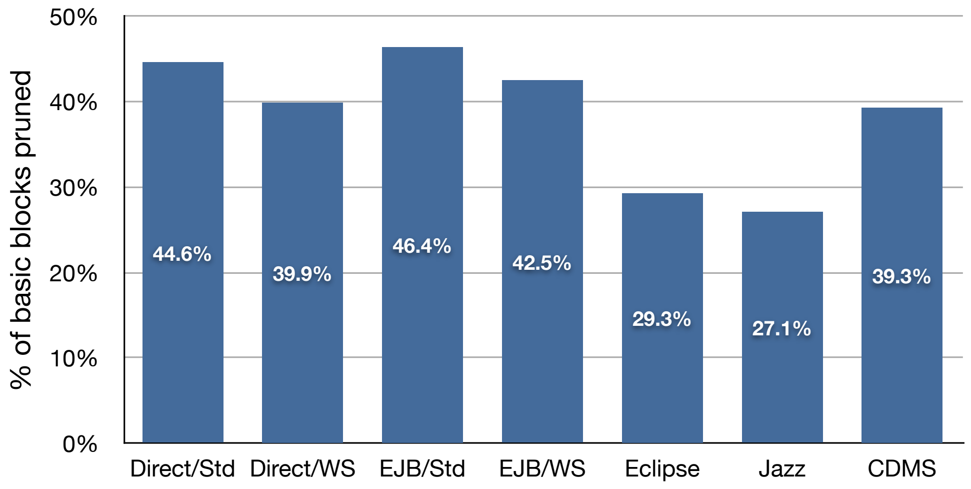


Figure 6.1: Pruned basic blocks

Benchmark	Analysis time (hh:mm:ss)	
	Orig	Pruned
Direct-Std	00:00:18	00:00:17
Direct-WS	01:34:01	00:04:41
EJB-Std	00:04:24	00:01:46
EJB-WS	N/A	29:23:16
Eclipse	24:37:12	06:37:22
Jazz	02:49:55	00:39:06
CDMS	00:04:35	00:02:05

Table 6.2: Analysis times

Pruned basic blocks. Figure 6.1 shows the percentage of basic blocks over all executed methods that have been identified as unexecuted and therefore pruned away. The results show that the pruning technique is very effective in practice: on average, 38.4% of basic blocks were removed. Pruning was most effective for the *EJB-Std* benchmark, in which nearly half of the basic blocks (46.4%) were pruned away. Even for the *Jazz* benchmark, for which the pruning rate was the lowest, over one quarter (27.1%) of the basic blocks were identified as unexecuted.

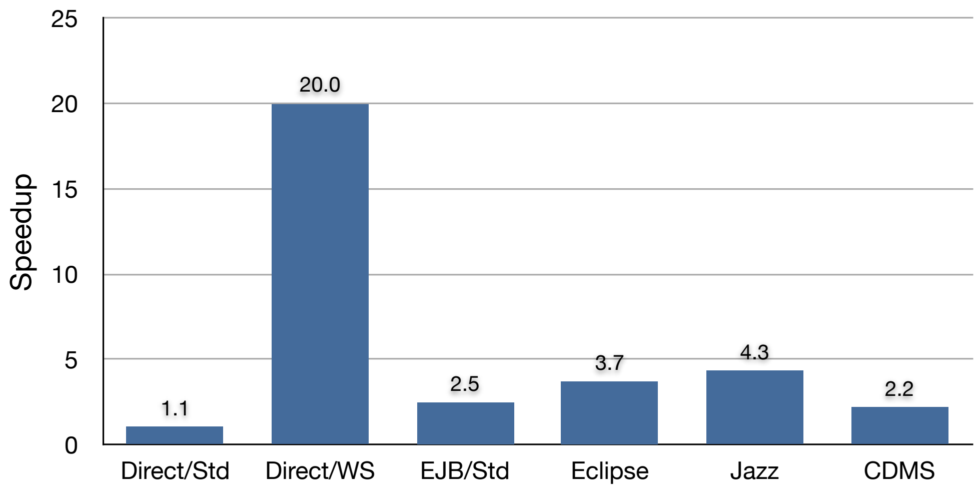


Figure 6.2: Speedup due to basic block pruning

Execution time. Since most flow-sensitive analyses need to associate state with each basic block, pruning executed basic blocks 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 6.2 shows the times for each benchmark. Note that *EJB-WS* exhausted the available 2GB of memory on our test machine when pruning was disabled, and thus no time is provided for it in the table. Nevertheless, CFG pruning made it possible to analyze this benchmark, which constitutes a clear gain in scalability.

Figure 6.2 shows the speedup of the analysis due to basic block pruning. The results show that, as expected, pruning has a clear impact on the analysis time for all benchmarks, with speedups ranging from 1.1 to 20, and with an average speedup of 5.6[5]. For example,

[5]An average is provided for reference despite the high variance in the execution times.

Direct-WS can be analyzed in less than 5 minutes with pruning enabled as opposed to more than 90 minutes without pruning.

It is interesting to note that a large percentage of pruned basic blocks does not necessarily translate into a higher speedup of the analysis. For example, *EJB-Std* has the highest percentage of pruned basic blocks, but the resulting speedup is only 2.5. In contrast, both *Eclipse* and *Direct-WS* have lower pruning rates (29% and 40% respectively), but their speedups are higher (3.7 for *Eclipse*, 20 for *Direct-WS*). This is due to the fact that not all basic blocks contribute equally to the total cost of the blended escape analysis. There are several factors that influence the total runtime cost associated with analyzing a given basic block. First, the instructions that are contained in a basic block determine its base cost. For example, call instructions are the most expensive to process. Instructions that manipulate the heap or reference variables also contribute to the cost associated with a given basic block. Purely scalar instructions, on the other hand, are ignored by the analysis. Pruning a basic block containing a call is therefore more profitable in terms of potential speedup than pruning a block that contains a large number of scalar instructions. Moreover, even for basic blocks that contain call instructions, the cost of analyzing the basic block normally increases with the number of potential targets for the call. Pruning a basic block containing a call with many potential targets is thus more likely to result in a significant speedup of the analysis.

Second, basic blocks that belong to methods that are analyzed repeatedly (i.e., methods that belong to a large strongly-connected component (SCC) in the calling structure) are generally more profitable to prune. For instance, the dynamic call graphs for both *Direct-WS* and *EJB-WS* comprise SCCs that have in excess of 200 methods. It is therefore not

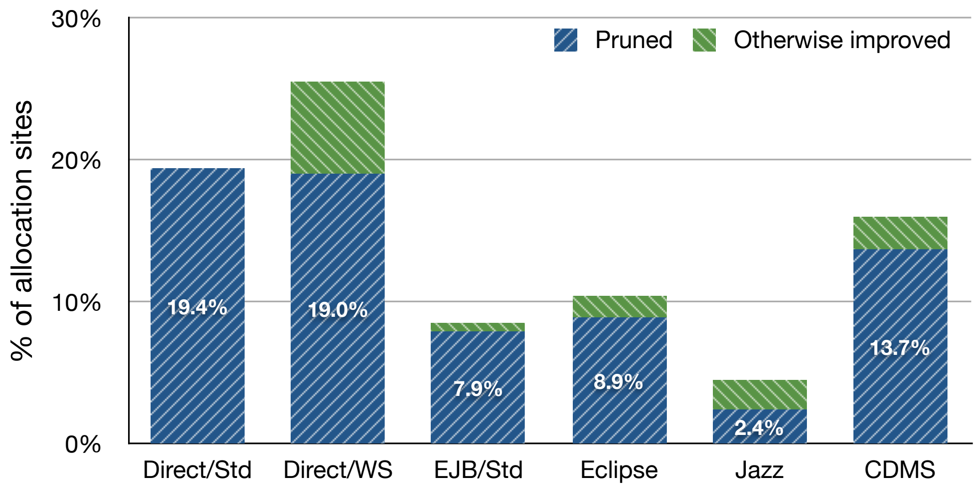


Figure 6.3: Disposition improvement (by allocation sites)

surprising that the pruning technique is particularly effective for these two benchmarks.

Precision improvements. In addition to scalability improvements, pruning away unexecuted basic blocks can also lead to increased precision of the blended analysis results. Specifically, for our blended escape analysis, removing unexecuted code can improve precision in two ways. First, the contributions of the pruned code to the computed connection graphs can obviously be avoided, thereby achieving better focus on the actual execution. Second, pruning can remove some dynamic ambiguity by removing call sites or allocation sites for which dynamic information could not be matched precisely. For example, if a method contains two different allocation sites with the same allocated type `T`, a limitation of our profiler prevents us from precisely associating object instances of type `T` found in the trace to their corresponding sites. Rather, both sites will be identified as potential allocation sites for each instance of `T`. If enough information is available to prune away one of the sites, however, all instances of `T` will be correctly matched to the remaining site.

In order to measure the effect of CFG pruning on the precision of the analysis, we compute the disposition improvement metric. Recall that the disposition of an object is the final escape state that is assigned to it by the escape analysis. Figure 6.3 displays the disposition improvement results tallied by abstract objects (i.e., allocation sites). For each benchmark, the bottom portion of the bar represents objects corresponding to allocation sites that the pruning technique marked as unexecuted and 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 4.5% and 25.5% of the objects benefit from the pruning algorithm. Identification of unexecuted allocation sites is responsible for 53% to 100% of the improvements, and is clearly the most effective aspect of the pruning algorithm. However, Figure 6.3 also shows that a small number of objects are assigned a more precise disposition, up to 6.5% in the case of the *Direct-WS* benchmark.

Note that in the remaining sections, we only report the results of the pruned algorithm in our discussions of the metrics.

6.2.2 Disposition

Recall that in our blended escape analysis objects can receive one of three possible dispositions: *captured*, *escaping*, and *mixed*. For the purpose of identifying temporaries, we are interested in captured objects. In order to assess the relative importance of temporaries in framework-intensive applications, we measure the disposition breakdown, that is the percentage of all object instances that correspond to each disposition. Figure 6.4 shows

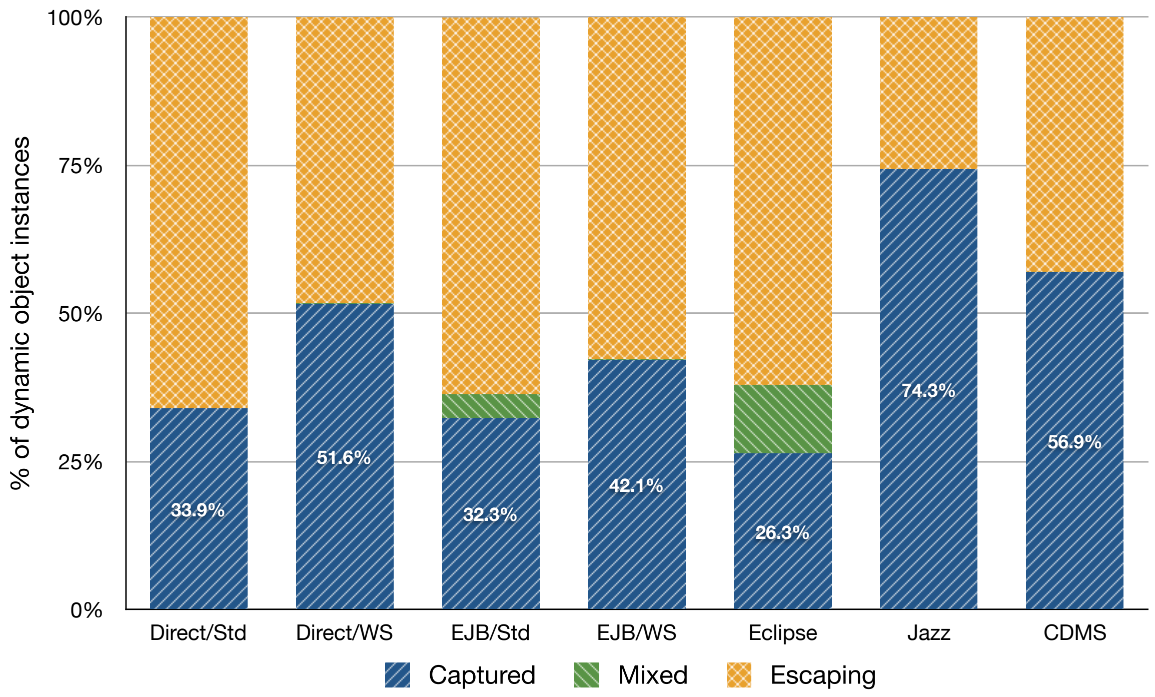


Figure 6.4: Disposition breakdown (by object instances)

disposition breakdown results across all benchmarks. On average, 45.3% of all object instances never escape globally, clearly indicating that temporaries account for a significant portion of allocated objects in framework-intensive applications. Moreover, only 2.3% 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. *Eclipse* has the highest percentage of mixed disposition instances with 11% and the lowest percentage of captured instances.

6.2.3 Capturing depth

Because the empirical results for the disposition breakdown metric suggest that temporaries are common in framework-intensive applications, it is necessary to understand how

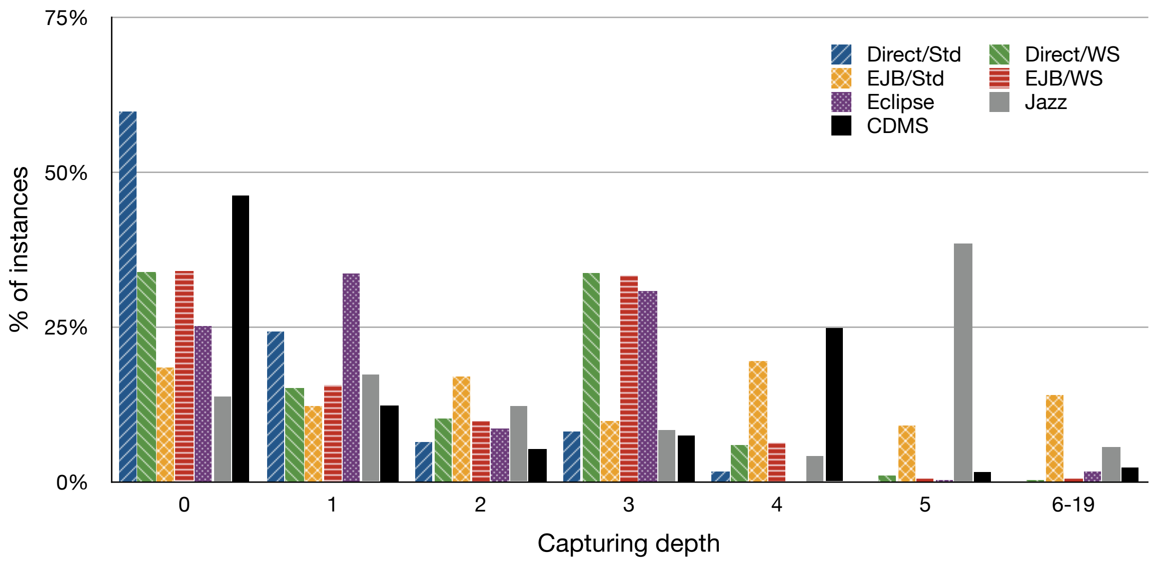


Figure 6.5: Capturing depth (by instances)

temporaries are used in practice. The capturing depth metric attempts to measure how far temporaries are propagated from their allocation sites. Recall that this metric measures the length of the acyclic path in the calling structure from allocation to capture of an object.

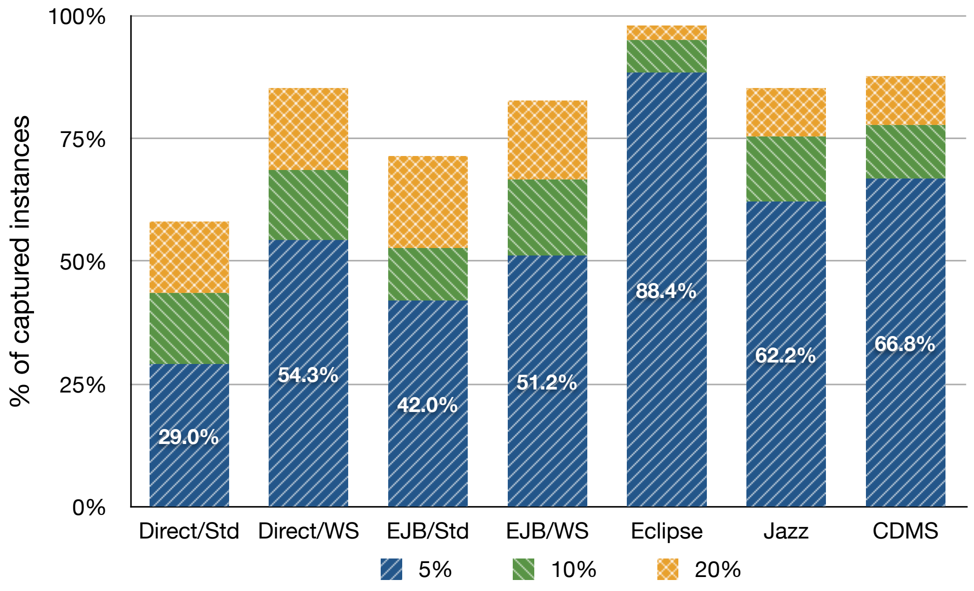
Figure 6.5 shows the distribution of capturing depths tallied by object instances for all benchmarks. On average, 33% of the instances are captured in their allocating methods (depth 0), and 18.6% are captured in a direct caller of their allocating method (depth 1). Modern JIT compilers often employ local escape analyses (i.e., within a single method) that are likely to be effective at identifying and optimizing these temporaries. However, nearly half of the instances (48.4%) are captured more than one call away from their allocating methods, requiring an interprocedural escape analysis.

The capturing depth metric also shows that for some benchmarks temporary usage is very complex. For example, in the *EJB-Std* benchmark, more than 11% of instances are captured 6 or more calls away from their allocating method. Also, capturing depths often exceed 10 calls (and can reach up to 19 calls). 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. Clearly, attempting to manually explore the behavior of temporaries using either source code inspection or low-level dynamic traces is a complex and difficult task. Tool support is therefore required to assist developers in understanding how temporaries flow through framework-intensive applications.

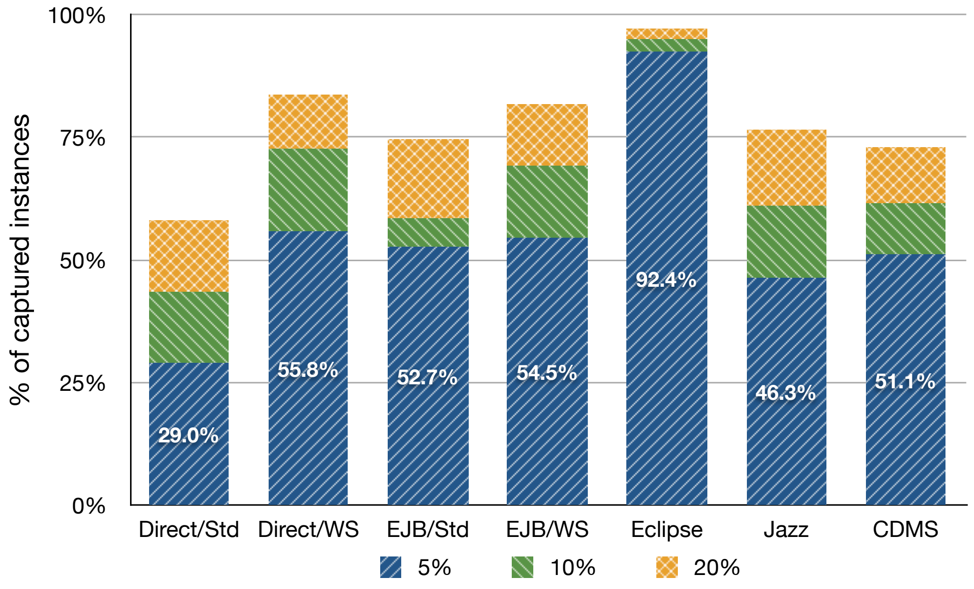
6.2.4 Concentration

The concentration metric aims to determine if temporaries are concentrated in a few specific regions of the program, or if their use is spread across multiple program regions. We measure the percentage of temporaries captured by the top 5%, 10% and 20% of the capturing methods (or alternatively capturing CCT contexts). Figure 6.6a shows the concentration results by capturing methods. The results indicate that about half of the temporaries, on average, are explained by the top 5% capturing methods. To a user trying to identify object sources of object churn, this means that our analysis can reduce the set of methods to be examined to 5% of all capturing methods. Note that because the concentration metric only considers *capturing* methods, the set of methods to be considered most often represents even less than 5% of all *executed* methods.

In the case of *Eclipse*, the top four capturing methods account for over 75% of the temporaries, while the top 5% methods explain 88.4% of all captured instances. These unusual results are largely due to the fact that this benchmark first populates a cache by



(a) By capturing methods



(b) By capturing contexts

Figure 6.6: Concentration results

loading and parsing classes from the disk (e.g., creating temporaries to read compressed archives). Other benchmarks show a different concentration of temporaries. For example, in the case of *EJB-Std*, 42% of the instances are explained by the top 5% of the capturing methods, or 2 out of the 31 capturing methods. Note that our blended escape analysis already focuses attention from 1979 observed methods down to just 31 capturing methods. In all benchmarks, the top 20% of methods explain the majority of captured instances (between 58.1% and 98.0% for *Direct-Std* and *Eclipse*, respectively). In *EJB-Std*, the top 20% of capturing methods explain 71.4% of the instances.

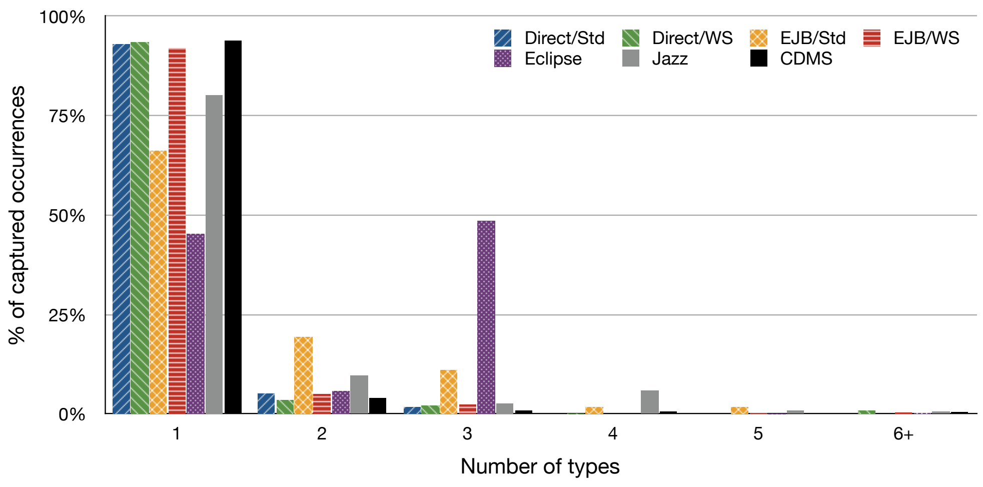
Figure 6.6b shows the concentration results by capturing CCT contexts rather than by methods. Clearly, there is a strong similarity between both sets of results. For example, *Direct-WS* and *Eclipse* show very similar concentrations when measured by methods or contexts. For some benchmarks, however, the difference is more significant. In *CDMS*, for instance, the top 5% of capturing methods explain 15% more instances than the top 5% of capturing contexts. This discrepancy is due to the fact that some methods are called from many distinct paths, with varied escape behavior. This is typical of applications that make heavy use of frameworks. A top capturing method therefore explains all of the temporaries that are captured by each of its corresponding CCT contexts, whether they are top capturing or not.

6.2.5 Complexity of data structures

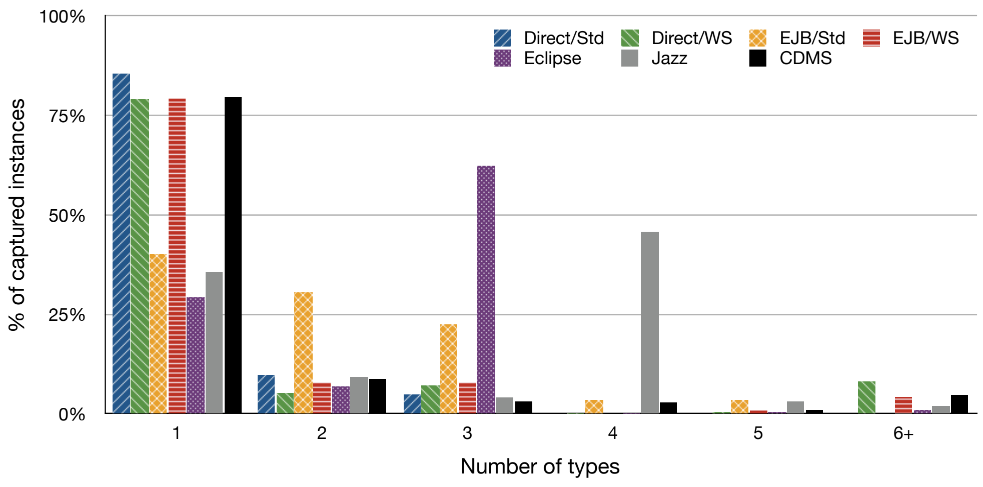
Recall that in order to study temporaries as part of larger data structures, we defined five metrics that quantify various characteristics of each temporary data structure: number of types, number of allocating methods, number of merged abstract objects, height of data structure, and maximum capturing distance. Each metric is computed *by occurrences* (i.e., by only counting instances of the root of a data structure) as well as *by instances* (i.e., by counting all object instances that comprise a data structure). Observe that the distinction between data structure occurrences and instances effectively changes the object population being considered (and thus affects percentages computed with respect to the set of all objects).

Figure 6.7 shows the number of types in captured data structures. While the results tallied by occurrences (a) might at first suggest that most data structures are relatively simple in structure, the instance-weighted metrics (b) reveal a more nuanced picture. For *Jazz*, half of the captured instances occur in data structures containing 4 or more types. Even in the simpler *EJB-Std* benchmark, 29.3% of the instances are from temporary structures with at least 3 types. *Direct-WS* has the most complex temporaries, with 8.1% of its instances originating from data structures containing 6 or more distinct types, with some structures reaching up to 10 types. Despite the relatively low percentage of temporaries in *CDMS* that fall into the 6 types or more category, the benchmark features highly complex data structures. For example, 1480 instances in total are part of data structures with at least 6 distinct types, including 532 instances in data structures with 10 to 12 types. A more thorough discussion of temporaries in *CDMS* is presented is Section 6.3.

Figure 6.8 presents the results for the number of allocating methods in data structures across all benchmarks, and Figure 6.9 shows the results for the number of merged abstract objects. Recall from Section 4.3.1 that merged abstract objects are a dynamic object abstraction that combines all abstract objects (i.e., allocation sites) of a single type within the



(a) By occurrences



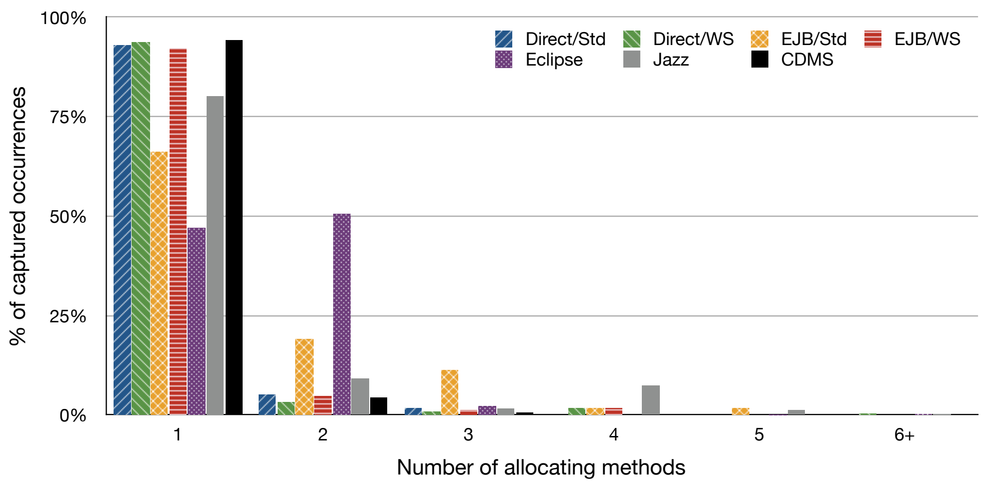
(b) By instances

Figure 6.7: Number of types in data structures

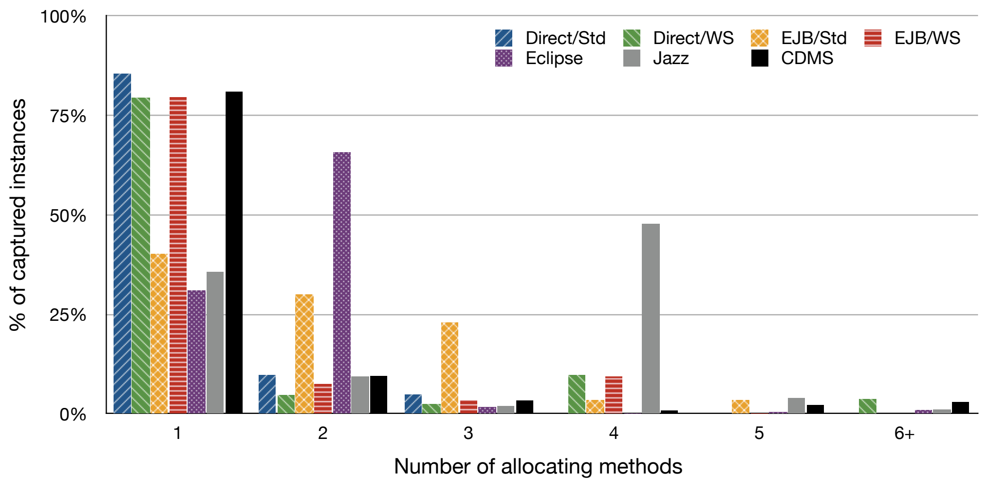
same allocating method. Clearly, there is a strong similarity between the number of types (Figure 6.7), the number of allocating methods (Figure 6.8) and the number of merged allocation sites (Figure 6.9). This correlation suggests that most methods creating part of a data structure allocate instances of a single type during execution.

Figure 6.10 shows the height of data structures for each benchmark.

Figure 6.11 shows the maximum capturing distance metric results for all benchmarks. Recall that this metric measures the length of the longest acyclic part from allocation to



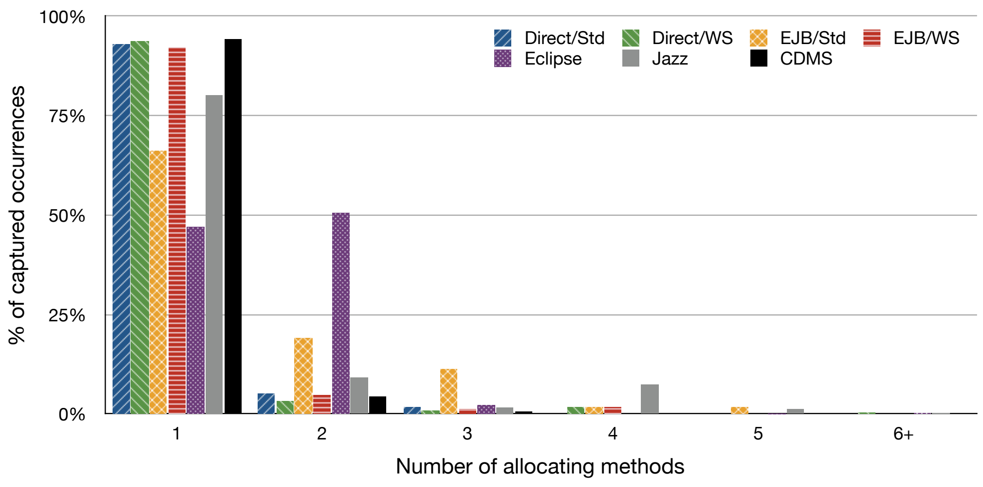
(a) By occurrences



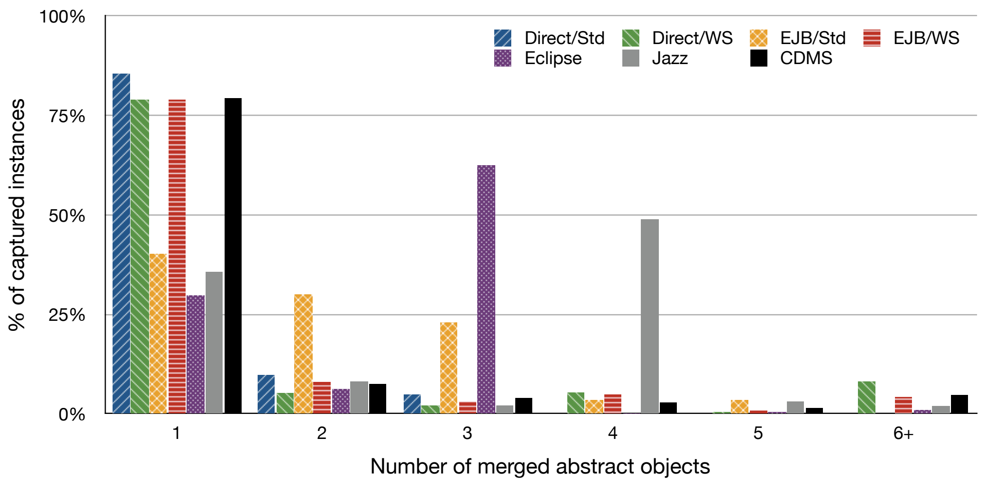
(b) By instances

Figure 6.8: Number of allocating methods per data structure

capture over all objects contained in a data structure. There is again a clear similarity between the maximum capturing distance in data structures and the capturing depth results from Figure 6.5. This suggests that most non-trivial data structures (i.e., those with at least two objects) are composed of objects that were created at similar depths in the capturing subtrees of the CCT. There are however some notable differences between the capturing depth results from Figure 6.5 and the maximum depths from Figure 6.11. For instance, *Eclipse* has 33.6% and 30.8% of instances that are capturing 1 and 3 levels away from their



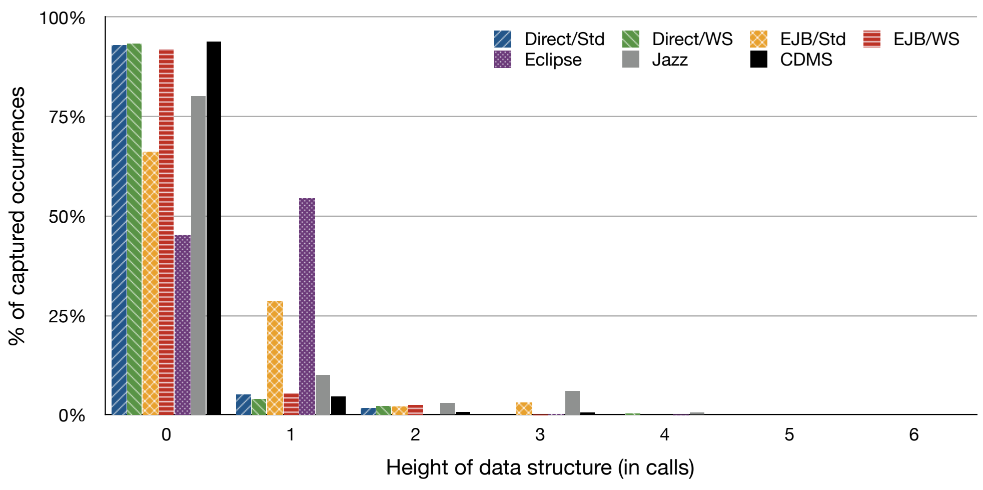
(a) By occurrences



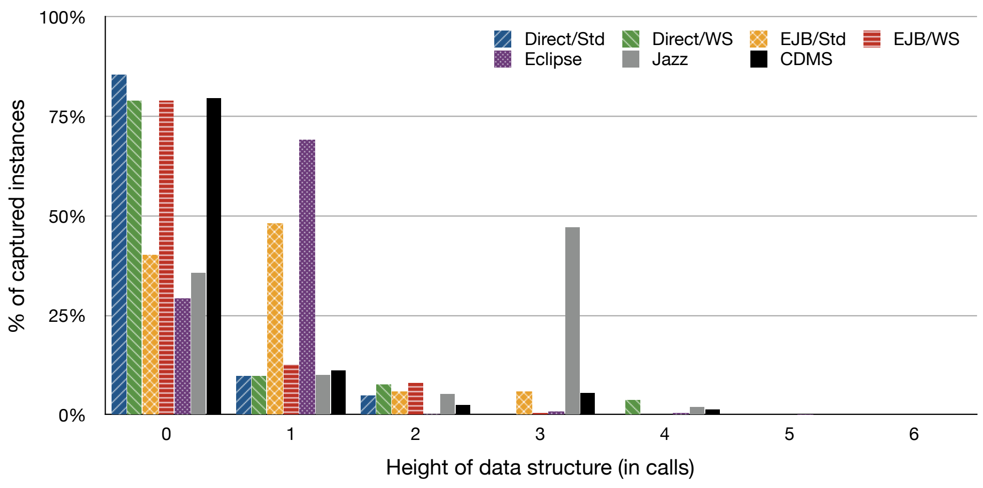
(b) By instances

Figure 6.9: Number of merged abstract objects in data structures

allocation, respectively (as seen in Figure 6.5). However, Figure 6.11b shows that only 3.1% of data structure instances are captured at depth 1, but that 61.1% of them are captured at depth 3 instead. This suggests that *Eclipse* builds a large number of data structures that are allocated at depth 3 and then extended with other objects further up in the CCT. Also, 37% of data structure occurrences in *EJB-Std* are captured at least 6 calls away from their furthest allocation, as compared to 12% of the total instances for the same category. This means that a developer inspecting the source code to try to identify problematic data



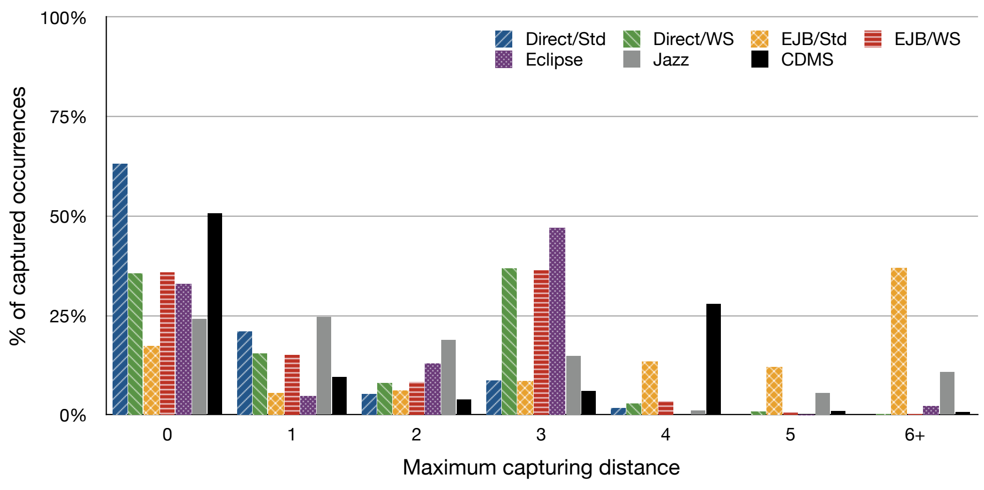
(a) By occurrences



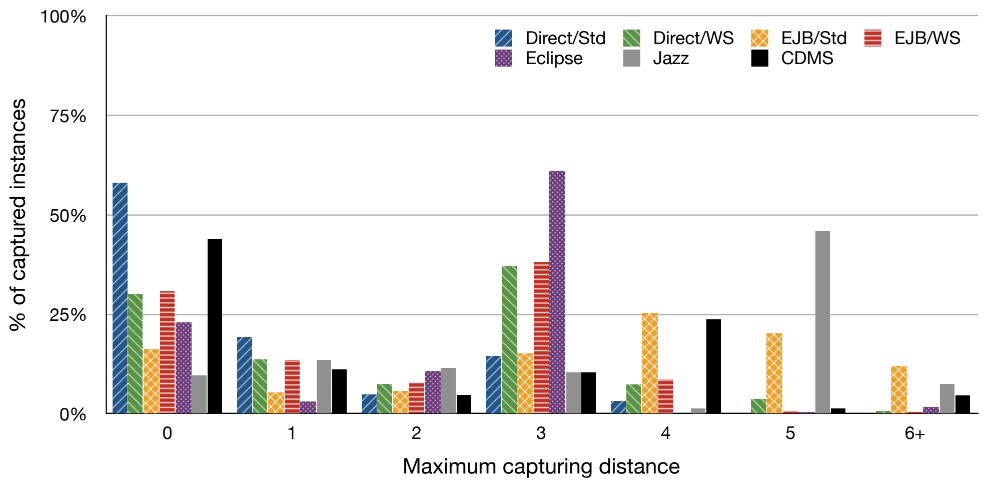
(b) By instances

Figure 6.10: Height of data structures

structure usage in this benchmark would frequently encounter situations where temporaries are assembled and propagated through a large number of methods. Such situations are difficult to discern and handle without proper tool support.



(a) By occurrences



(b) By instances

Figure 6.11: Maximum capturing distance per data structure

6.3 Performance understanding in *CDMS*

In this section we demonstrate how blended escape analysis can be used 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 for object churn. Table 6.3 shows the typical information that is obtained from such profilers[6], namely a list

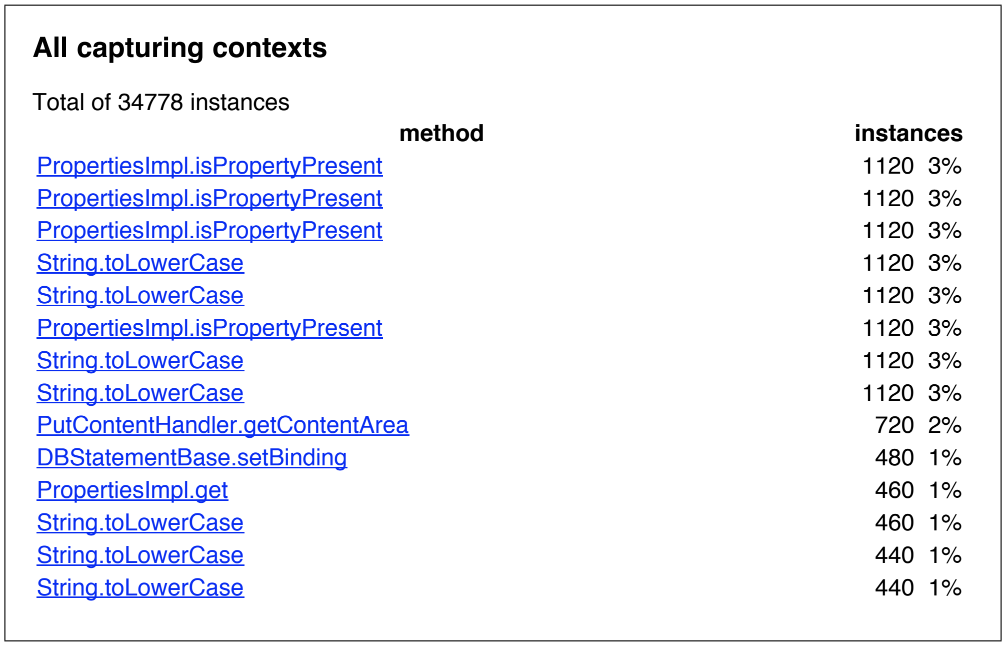
[6]We used *Jinsight* to compute the information.

Type	Instances	% of total
`char[]`	7732	24.92%
`java.lang.String`	6841	22.04%
`java.lang.StringBuilder`	5198	16.75%
`byte[]`	1489	4.80%
`java.lang.StringBuffer`	957	3.08%
`java.util.HashMap$Entry`	908	2.93%
`java.lang.Integer`	859	2.77%
`java.lang.Object[]`	678	2.18%
`java.util.AbstractList$Itr`	457	1.47%
`cdms.Id`	420	1.35%
(+ 244 more classes...)		

Table 6.3: Top allocated types with instance counts in *CDMS*

of types that were most frequently instantiated. Even though this ranked list accounts for more than half of the allocations recorded in the trace, it yields very little useful information. The top allocated types are clearly string-related, but more information is needed in order to determine *how* these objects are created and used throughout the program.

We use the results of the blended escape analysis to provide additional insight into the profiled program region. Recall from Section 4.3.1 that because our blended escape analysis is based on a static object abstraction, we 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.



All capturing contexts

Total of 34778 instances

method	instances	
PropertiesImpl.isPropertyPresent	1120	3%
PropertiesImpl.isPropertyPresent	1120	3%
PropertiesImpl.isPropertyPresent	1120	3%
String.toLowerCase	1120	3%
String.toLowerCase	1120	3%
PropertiesImpl.isPropertyPresent	1120	3%
String.toLowerCase	1120	3%
String.toLowerCase	1120	3%
PutContentHandler.getContentArea	720	2%
DBStatementBase.setBinding	480	1%
PropertiesImpl.get	460	1%
String.toLowerCase	460	1%
String.toLowerCase	440	1%
String.toLowerCase	440	1%

Figure 6.12: Top capturing contexts, as identified by our analysis. Note that method names that appear more than once correspond to different call paths leading to that method (i.e., different calling contexts).

Our technique allows users to browse individual details to understand the disposition of particular objects at each calling context.

Top capturing contexts. Figure 6.12 shows a fragment of the information provided to the user by our blended escape analysis tool. It consists of a list of the top capturing CCT contexts and the number of instances they explain. The method names associated with these contexts already provide some insights into the behavior of the program. Clearly, checking for the existence of some properties and converting strings to a lowercase result in a very large number of temporaries being created. Note that the `isPropertyPresent` method is called 5370 times in the short trace under investigation. This method appears 4 times in the top capturing contexts from Figure 6.12 (corresponding to each context in which is it called), and each context explains 1120 temporaries, for a total of 4480 temporaries

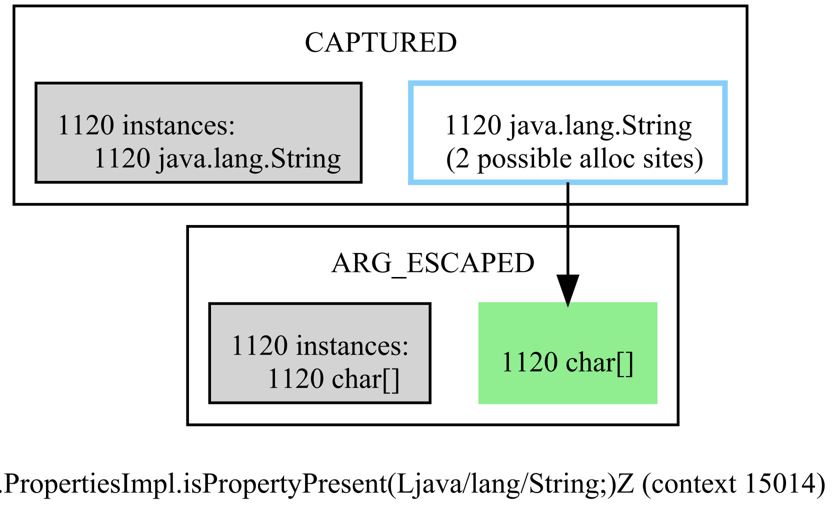

CAPTURED
1120 instances: 1120 java.lang.String
1120 java.lang.String (2 possible alloc sites)
ARG_ESCAPED
1120 instances: 1120 char[]
1120 char[]
cdms.PropertiesImpl.isPropertyPresent(Ljava/lang/String;)Z (context 15014)

Figure 6.13: Reduced connection graph for `PropertiesImpl.isPropertyPresent`

(65% of all allocated `String` instances). A quick investigation of the reduced connection graphs for each of the four contexts reveals that they are all identical to the one shown in Figure 6.13.

Figure 6.13 shows that all of the objects captured by the `isPropertyPresent` method are strings.[7] In order to understand where the strings originate, our tool computes a pruned version of the CCT where only *interesting* contexts are shown. A context is considered interesting if it allocates or captures objects; all other contexts are elided from the CCT. Figure 6.14 shows this reduced CCT representation for one context representing the `isPropertyPresent` method.

`PropertiesImpl.isPropertyPresent` (indirectly) calls `String.toLower-`

[7]Note that the `char[]` objects are not captured because of an optimization in the Java libraries that allows multiple `String` objects to share their underlying character arrays.

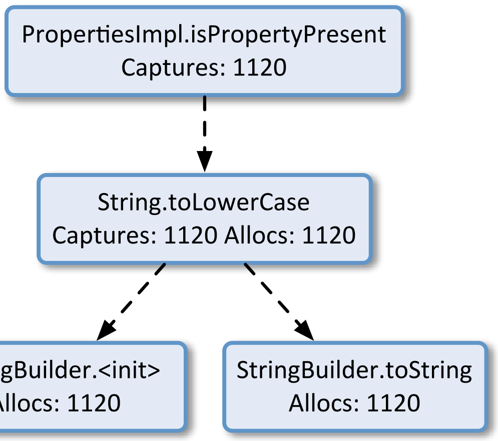


Figure 6.14: Interesting contexts rooted at `PropertiesImpl.isPropertyPresent`

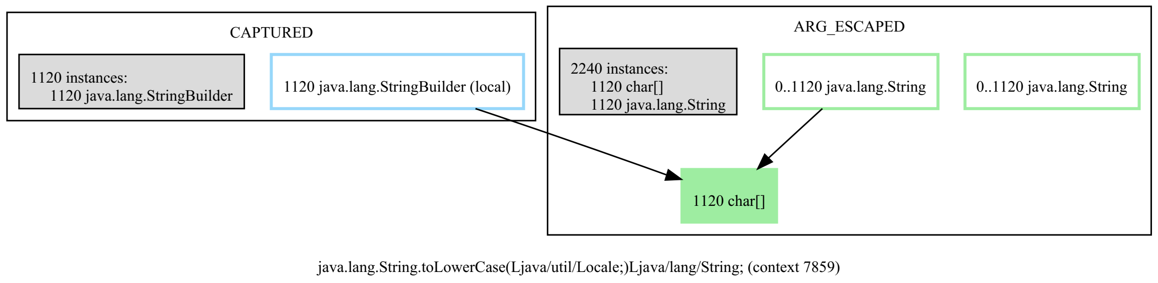

CAPTURED
1120 instances:
1120 java.lang.StringBuilder
1120 java.lang.StringBuilder (local)
ARG_ESCAPED
2240 instances:
1120 char[]
1120 java.lang.String
0..1120 java.lang.String
0..1120 java.lang.String
1120 char[]
java.lang.String.toLowerCase(Ljava/util/Locale;)Ljava/lang/String; (context 7859)

Figure 6.15: Reduced connection graph for `String.toLowerCase`

`Case`, which is responsible for creating the `String` temporaries. Recall from Figure 6.12 that `String.toLowerCase` is itself a top capturing context. Its reduced connection graph appears in Figure 6.15. As expected, it allocates a temporary `StringBuilder` object which is then discarded, and a freshly created `String` is returned to the caller, as shown in Figure 6.15.

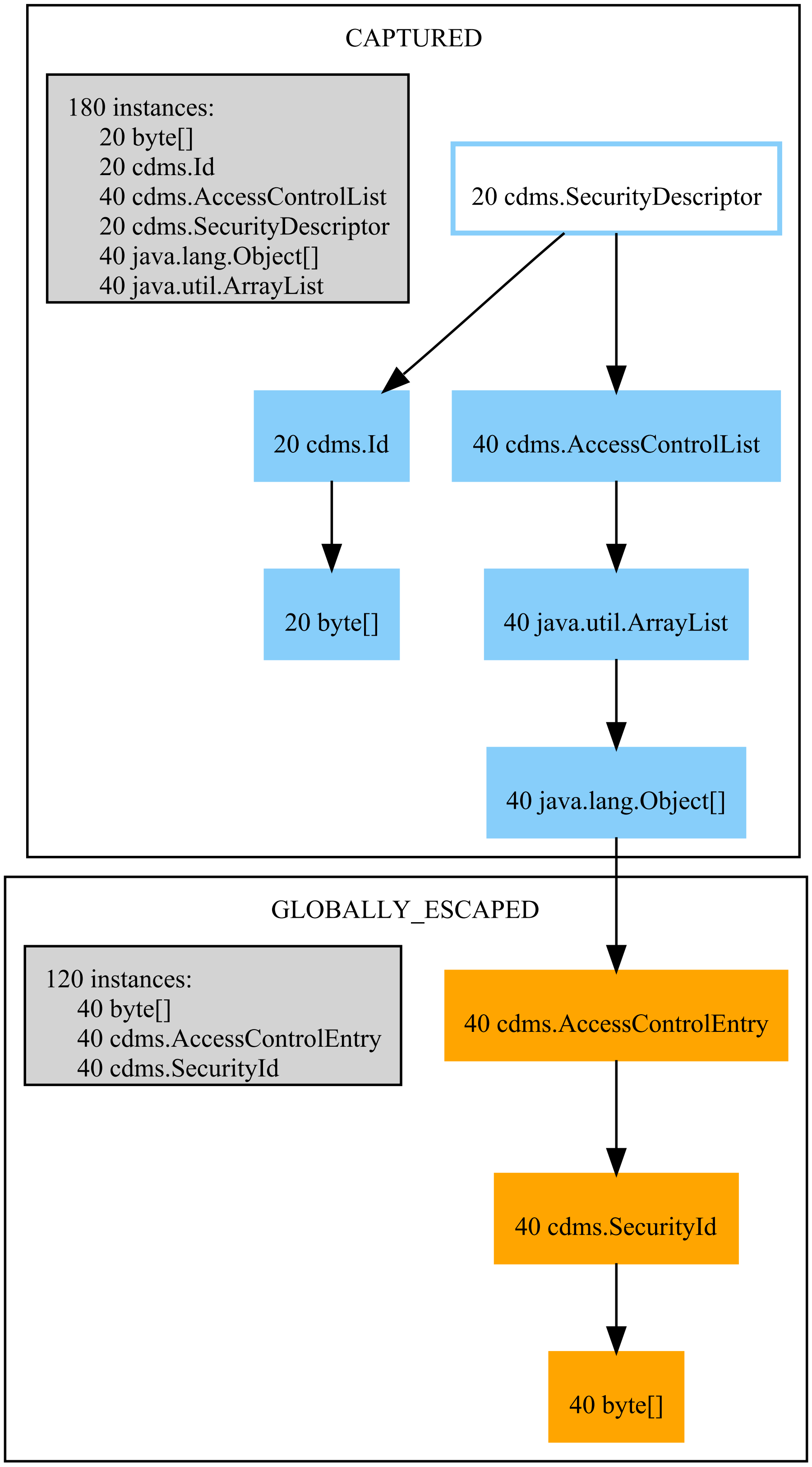
The problem illustrated here, while simple in nature, is widespread throughout the application. Thousands of temporaries are created simply to ensure that property descriptors do not contain uppercase letters. It would be much more efficient to enforce this invariant by sanitizing all inputs only once before being used in the entire system.

Checking for access Continuing the investigation, our analysis points out another intesting method from the point of view of object churn: `SecurityServer.hasConnectAccess`, which explains 180 captured instances. The reduced connection graph for this method appears in Figure 6.16.

The `hasConnectAccess` method was invoked 20 times in the trace, and as shown in Figure 6.16, it created 20 `SecurityDescriptor` structures, one per invocation. While this looks like a significant overhead, it is only part of the full story. The invocation of this method triggers a series of calls that result in 1000 objects being created, 880 of which never escape the `hasConnectAccess` method. The pruned CCT rooted at this method appears in Figure 6.17[8].

Figure 6.17 shows that every time `hasConnectAccess` is called, a fresh security descriptor is created and initialized by deserializing a stream of bytes. A quick search through the results indicates that this behavior is present in two additional parts of the program. Because `SecurityDescriptor` instances are created for specific `Id` instances, caching `SecurityDescriptor`s within their associated `Id` objects would prevent most of these temporaries from being created repeatedly. It is also worth noting that the behavior described here is guarded by a global flag that determines whether security checks are enabled. During performance testing, it is conceivable that such checks could be disabled, thus leading to very different performance characteristics for the application. This kind of situation often happens in practice, and illustrates the importance of comprehensive performance testing as well as the necessity to develop tools and techniques to analyze applications after deployment.

[8]Note that due to space limitations, certain allocating contexts have been left out of the figure.



cdms.SecurityServer.hasConnectAccess(Lcdms/Id;)Z (context 16430)

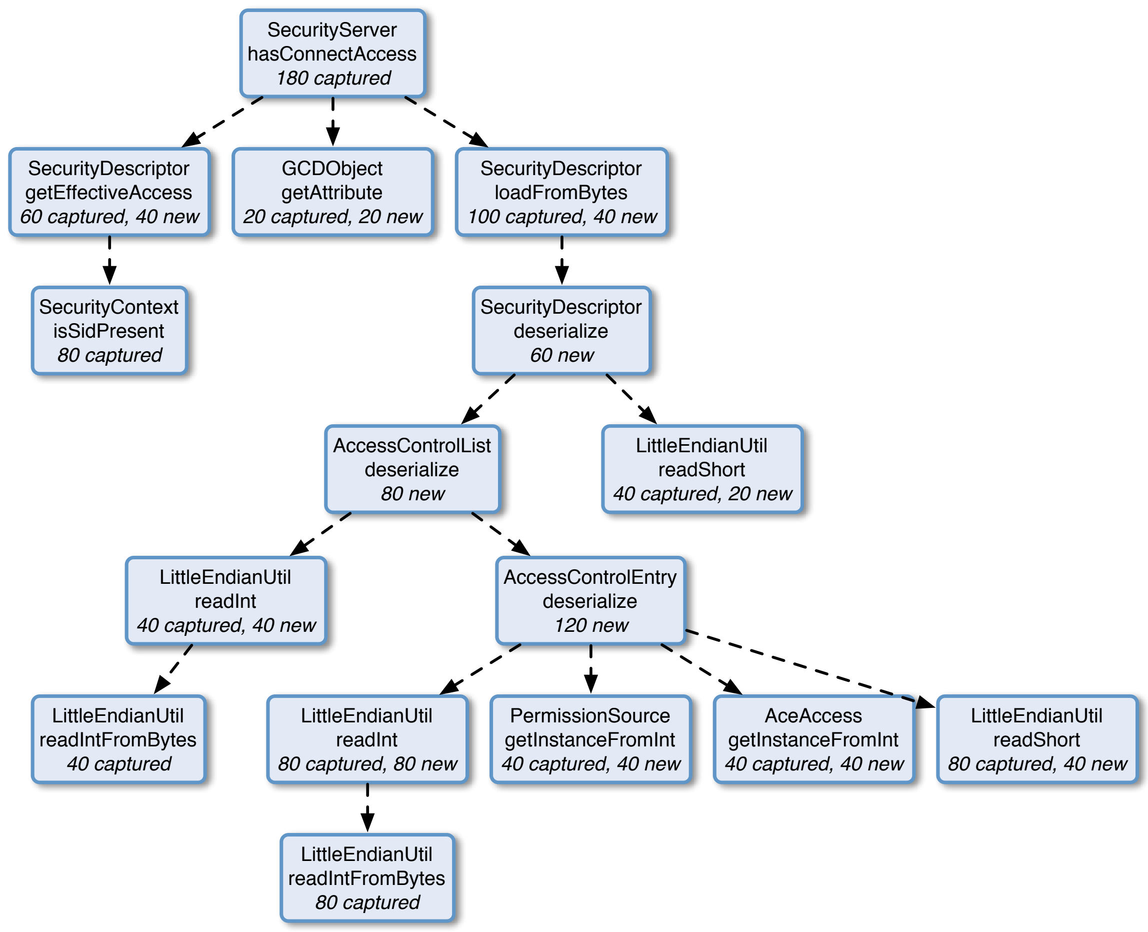


Figure 6.17: Interesting contexts rooted at `SecurityServer.hasConnectAccess` allocate 1000 instances and captures 880 of them.

Discussion By focusing on where objects are used, rather than on where they are allocated, our technique is able to aggregate disparate events into a single summary for the user to focus on. By postprocessing the blended escape analysis using the CCT, and by retaining distinct escape information along different paths in the blended analysis, we are able to provide results relevant to the region we are studying. The postprocessing step also allows us to remove extraneous information.

In the future, we would like to automate many of the steps described in this chapter. For example, automatically identifying subtrees in the CCT that are responsible for significant object churn (e.g., Figure 6.17), while greatly simplified by our technique, still heavily relies on manual exploration.

Chapter 7

Conclusions and future work

The growing availability of reusable software components has led to faster development of complex systems at the cost of an increase in runtime complexity. Framework-intensive applications typically exhibit different runtime characteristics from traditional applications. Understanding the behavior of framework-intensive applications is often difficult or even impossible with the current static and/or dynamic analysis tools. We have shown that blended analysis makes it possible to analyze these applications with great precision and at practical cost.

7.1 Blended analysis

We have presented the blended analysis paradigm, a new analysis technique that narrows the focus of a static analysis to a set of executions of interest. This is accomplished by recording a lightweight dynamic profile from which a calling structure is derived and using it to limit the scope of a static analysis. Blended analysis preserves many of the advantages of a full dynamic analysis, such as the ability to handle dynamic class loading and reflection, the increased scalability and precision, while maintaining the amount of runtime overhead to a minimal level.

We have also developed an optimization technique for blended analyses that further reduces the amount of code that is examined by pruning away unexecuted basic blocks intraprocedurally. The optimization technique relies on observed calls and allocations as evidence of the non-execution of basic blocks. The additional dynamic information required for this optimization is inexpensive to record, and results in significant analysis speedups in practice.

Finally, we have designed and implemented *Elude*, a general framework for blended analysis of Java applications. *Elude* allows new blended analyses to be implemented quickly and easily. By leveraging the popular WALA analysis framework, *Elude* ensures that a rich infrastructure is avaible to analysis implementors.

7.2 Empirical evaluation

We have used *Elude* to instantiate the blended analysis paradigm by developing a blended escape analysis focused on object churn, a common performance problem in framework-intensive applications caused by the excessive creation of temporary objects. Blended escape analysis provides an approximation of object lifetimes during execution, thereby allowing us to pinpoint excessive temporary usage in large, framework-intensive applications.

We have defined a set of new metrics that characterize both the usage and complexity of temporary data structures. We have applied them to a set of 7 open-source and commercial framework-intensive applications, and performed a detailed analysis of the results. Furthermore, we have demonstrated the effectiveness of our approach by performing a detailed investigation of a commercial framework-intensive application, revealing problematic scenarios.

7.3 Future work

The blended analysis framework that we have developed and presented in Section 3.6 provides an ideal vehicle for further exploration of the blended analysis paradigm. In the future, we would like to continue to design new tools and techniques to gain a better understanding of framework-intensive applications. We intend to start our exploration by extending the blended analysis work in three ways: (i) designing automated solutions to the object churn problem, (ii) designing new blended analyses to study and solve other important problems in framework-intensive applications (e.g. security), and (iii) investigating other possible combinations of static and dynamic analysis techniques, as well as their benefits and costs.

7.3.1 Automatic optimizations for the removal of object churn

Chapter 6 has demonstrated that blended escape analysis is an effective technique to identify program locations responsible for significant churn. We plan to extend this work to provide tools and techniques to automate the discovery and removal of object churn. To this end, two main problems need to be addressed. First, it is necessary to investigate ways to identify groups of methods that are collectively responsible for object churn. Often, the compound impact of temporary usage in a given program region can be much more significant than any of the individual method results indicate. Therefore, is it important to investigate ways to identify such program regions automatically, and to find effective ways to present this information to the user.

Also, techniques need to be designed to automatically ameliorate object churn. Currently, the burden of fixing the problem is still left to the developer. Possible optimization techniques include code transformations within a software component as well as code specialization of one or more components to fine tune their interaction. We plan to study and classify causes of object churn that occur in practice. From this classification, it will be possible to establish a compendium of best practices for developers, and more importantly to devise specific solutions for each situation that can be automatically applied. These techniques will need to be evaluated on larger set of representative framework-intensive applications.

7.3.2 Applying blended analysis to other problems in framework-intensive applications

Blended analysis has many potential applications outside of object churn and performance understanding. For example, it could be applied to the security domain, in particular to taint analysis. Taint analysis is used to identify a wide range of security vulnerabilities such as SQL injection and cross-site scripting (XSS), but it is known to be very costly and thus difficult to scale with reasonable precision to common web applications, either dynamically [NS05] or statically [TPF+09]. A blended taint analysis would benefit from a much improved scalability and provide better information than can be obtained using current techniques.

Another mostly unexplored use for blended analysis consists of studying the flow of data in framework-intensive applications. Because the vast majority of web applications are data-centric in nature, most program understanding techniques that rely on traditional control flow are inappropriate in this context. It is therefore important to devise new tools that can accurately represent object usage through the entire program. There are two main challenges in this work. First, new value-flow analyses that achieve the right balance between precision and scalability have to be designed. Second, the overwhelming amount of information computed by these analyses have to be distilled in order to be presented to developers. We intend to study recent work in large-scale software visualization techniques in order to find ways to present the information that allows developers to quickly and efficiently gain a better understanding of their applications. Integration with integrated developments environments (IDEs) is an interesting direction for building research prototypes for the visualization tools.

7.3.3 Combinations of static and dynamic analyses

We intend to expand the notion of blended analysis and explore the design space of combined static and dynamic analyses. The cost and precision of a blended analysis are influenced by three main factors: (i) the profiling methodology, (ii) the call representation used to trigger the static analysis, and (iii) the static analysis algorithm itself. Note that these factors are not independent from each other: the profiling methodology limits the possible call representations that can be derived from the execution trace for instance.

We want to explore alternative call representations using the existing dynamic profiles by designing alternative aggregation strategies. Our current aggregation schemes can produce call graphs and CCTs from a full call tree. Call graphs are small and easy to analyze, at the cost of heavily conflated behavior. CCTs are more precise, but are often an order of magnitude larger than call graphs with respect to the number of nodes they contain. We therefore intend to devise new, non-uniform aggregation strategies that would provide precision where needed without the typical increase in the size of the resulting calling structure. For example, containers in Java are known to cause imprecision in static analyses. By representing calls from containers in a more precise manner (e.g., using an object-sensitive representation), this problem could be greatly reduced. While a larger call representation generally translates into a more costly analysis, the precise impact of each aggregation strategy needs to be evaluated. It is often the case that increasing the precision of an analysis decreases its running time, because the adverse effects of overapproximations are avoided. Also, a more precise call representation would contain many copies of the same method. It is possible to design analyses that exploit this fact (e.g. by reusing previously computed results) in order to limit the runtime increase due to a larger call representation.

Additionally, the profiling methodology used to collect the execution trace deserves further exploration. Profiling framework-intensive applications would ideally be almost invisible to the end users, thereby allowing the collection of large amounts of data from long-running processes. Intuitively, sampling techniques could greatly reduce the profiling overhead, but their effect on the precision of the static analysis needs to be evaluated. On the other hand, it would be interesting to investigate the possibility of collecting more information in the profile, such as call sites and allocation sites. The current profiling technology used in the research prototype does not allow this information to be recorded. This missing information in turn causes greater imprecision for the static analysis, and can negatively affect both the cost and the usefulness of the results.

Bibliography

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, Nevada, USA, 1997, pages 85–96. ACM Press.

[ABLU00] 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*, 39(1):118–134, 2000.

[ACGS04] Glenn Ammons, Jong-Doek Choi, Manish Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[AEGK07] Shay Artzi, Michael D. Ernst, David Glasser, and Adam Kiezun. Combined static and dynamic mutability analysis. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007, pages 104–113.

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999, pages 35–46. ACM Press.

[BHT08] D. Beyer, T. A. Henzinger, and G. Theoduloz. Program analysis with dynamic precision adjustment. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008, pages 29–38. IEEE Computer Society Press, Washington, DC, USA.

[BSF04] Matthew Q. Beers, Christian H. Stork, and Michael Franz. Efficiently verifiable escape analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2004, pages 96–122. Springer.

[CGS$^+$99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999, pages 1–19. ACM Press.

[CGS$^+$03] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):876–910, 2003.

[CodePro] Codepro profiler. <http://www.instantiations.com/codepro/profiler/> .

[CS05] C. Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005.

[CS06] C. Csallner and Yannis Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analalysis (ISSTA)*, 2006, pages 245–254.

[DGC01] David David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):685–746, 2001.

[DJM$^+$02] Wim DePauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeana Yang. Visualizing the execution of Java programs. In *Software Visualization: State of the Art Survey, LNCS 2269*, 2002.

[Ern03] Michael Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings of the International Workshop on Dynamic Analysis (WODA)*, 2003.

[FLL$^+$02] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002, pages 234–245.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Atlanta, Georgia, United States, 1997, pages 108–124. ACM Press, New York, New York, USA.

[GJ06] Alex Groce and Rajeev Joshi. Exploiting traces in program analysis. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the International Conference on Compiler Construction (CC)*, 2000, pages 82–93. Springer-Verlag.

[GSH97] Rajiv Gupta, Mary Lou Soffa, and John Howard. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4), October 1997.

[HBM$^+$06] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanovic. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(3):476–516, 2006.

[HPROF] Hprof: A heap/cpu profiling tool. <http://java.sun.com/developer/technicalarticles/programming/hprof.html> .

[IKN09] Hiroshi Inoue, Hideaki Komatsu, and Toshio Nakatani. A study of memory management for web-based applications on multicore processors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the International Conference on Compiler Construction (CC)*, April 2003, volume 2622 of *LNCS*, pages 153–169.

[LH06] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In A. Mycroft and A. Zeller, editors, *Proceedings of the International Conference on Compiler Construction (CC)*, March 2006, volume 3923 of *LNCS*, pages 47–64. Springer, Vienna.

[MACE02] Markus Mock, Darren Atkinson, Craig Chambers, and Susan Eggars. Improving program slicing with dynamic points-to data. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2002.

[MHM98] Sungdo Moon, Mary Hall, and Brian Murphy. Predicated array data-flow analysis for run-time parallelization. In *Proceedings of the International Conference on Supercomputing (ICS)*, 1998.

[Mit06] Nick Mitchell. The runtime structure of object ownership. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. Properties of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[MS03] Nick Mitchell and Gary Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[MSS06] Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. Modeling runtime behavior in framework-based applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[NS05] James Newsome and Dawn Song. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[OAH03] Alessandro Orso, Taweewup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2003.

[RRH02] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2002.

[Ryd03] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the International Conference on Compiler Construction (CC)*, April 2003, pages 126–137.

[SAB08] Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2008. ACM Press.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[SPMS09] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analalysis (ISSTA)*, Chicago, IL, USA, 2009, pages 225–236. ACM Press, New York, NY, USA.

[SS05] Kavitha Srinivas and Harini Srinivasan. Summarizing application performance from a components perspective. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, September 2005, pages 136–145.

[SSG$^+$09] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for Java runtime exceptions. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analalysis (ISSTA)*, Chicago, IL, USA, 2009, pages 153–164. ACM Press, New York, New York, USA.

[TBV07] Aaron Tomb, Guillaume Brat, and William Visser. Variably interprocedural program analysis for runtime error detection. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analalysis (ISSTA)*, July 2007, pages 97–107.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Minneapolis, Minnesota, United States, 2000, pages 281–293. ACM Press, New York, New York, USA.

[TPF$^+$09] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, 2009, pages 87–97. ACM Press, New York, New York, USA.

[vPG02] Christoph von Praun and Thomas Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002, pages 115–128.

[WALA] T.J. Watson libraries for analysis (WALA).
<http://wala.sourceforge.net> .

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999, pages 187–206. ACM Press.

[XAM$^+$09] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[YourKit] Yourkit profiler.
<http://www.yourkit.com/> .

[ZSCC06] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa Ontario, Canada, 2006, pages 263–271.

[ZSZ$^+$09] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. Allocation wall: a limiting factor of Java applications on emerging multi-core platforms. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Orlando, Florida, USA, 2009, pages 361–376. ACM Press, New York, NY, USA.

Curriculum Vita

Bruno Dufour

1999–2002	B.Sc. in Computer Science, McGill University, Canada.
2002–2004	M.Sc. in Computer Science, McGill University, Canada.
5/2005-8/2005	Summer intern, IBM TJ Watson Research Center, Hawthorne, NY, USA.
5/2006–8/2006	Summer intern, IBM TJ Watson Research Center, Hawthorne, NY, USA.
9/2004–5/2005	Teaching assistant, Rutgers, The State University of New Jersey, New Brunswick, USA.
2003	Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren and Clark Verbrugge. EVolve: An Open Extensible Software Visualization Framework. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis)*, pages 37–46, June 2003, San Diego, CA, USA.
2003	Bruno Dufour, Karel Driesen, Laurie Hendren and Clark Verbrugge. Dynamic Metrics for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 149–168, October 2003, Anaheim, CA, USA.
2004	Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the Dynamic Behaviour of AspectJ Programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 150–169, October 2004, Vancouver, Canada.
2007	Bruno Dufour, Barbara G. Ryder and Gary Sevitsky. Blended Analysis for Performance Understanding of Framework-based Applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, July 2007, London, England.
2008	Bruno Dufour. Blended Analysis for Improving the Quality of Framework-intensive Applications. In *Foundations of Software Engineering Doctoral Symposium*, November 2008, Atlanta, GA, USA.
2008	Bruno Dufour, Barbara G. Ryder and Gary Sevitsky. A Scalable Technique for Characterizing the Usage of Temporaries in Framework-intensive Java Applications. In *Foundations of Software Engineering (FSE)*, November 2008, Atlanta, GA, USA.