University of New Mexico
UNM Digital Repository

5-1-2009

Programmer feedback and dynamic analysis to enable optimization in Java applications: the D.U.P.O. framework

David Mohr

Follow this and additional works at: https://digitalrepository.unm.edu/cs_etds

David Mohr

Candidate

Computer Science

Department

This thesis is approved, and it is acceptable in quality
and form for publication:

Approved by the Thesis Committee:

Darko Stefanovic, Chairperson

Amer Diwan

Patrick G. Bridges

Programmer Feedback and Dynamic Analysis to Enable Optimization in Java Applications: The D.U.P.O. Framework

by

David Mohr

BSCS, The University of Texas-Pan American, 2006

THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Computer Science

The University of New Mexico

Albuquerque, New Mexico

May 2010

Dedication

To my grandparents: Oma und Opa, ohne euch wäre ich nie so weit gekommen. Ich bedanke mich sehr für eure Unterstützung über diese langen Jahre hinweg, und für alles was ihr mir schon vorher mit auf den Weg gegeben habt.

"Es ist nicht genug zu wissen - man muss auch anwenden. Es ist nicht genug zu wollen - man muss auch tun." – Johann Wolfgang von Goethe

Acknowledgments

I would like to thank my advisor Darko Stefanovic for being patient and for supporting me throughout my journey. Without the close collaboration with Amer Diwan this work would have never been completed. Thanks go also to Patrick G. Bridges for being a very responsive committee member when time was short. I'm very grateful that my colleague Devin Coughlin helped by taking the transformation work out of my hands.

Many thanks go out to my parents, who have always been there for me. And a big thank you to Nina for being patient most of the time.

I also really appreciate the help of the reviewers, Peter Kolloch, ThanhVu Nguyen, and Nina. And finally, thanks to Mark Marron, who helped to get the stone rolling at the beginning of this work.

Programmer Feedback and Dynamic Analysis to Enable Optimization in Java Applications: The D.U.P.O. Framework

by

David Mohr

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Computer Science

The University of New Mexico

Albuquerque, New Mexico

May 2010

Programmer Feedback and Dynamic Analysis to Enable Optimization in Java Applications: The D.U.P.O. Framework

by

David Mohr

BSCS, The University of Texas-Pan American, 2006

M.S., Computer Science, University of New Mexico, 2010

Abstract

It is inherently difficult for static analyses to make precise decisions about dynamic features of modern object-oriented languages. This makes it more difficult to apply optimizations aggressively. This thesis introduces the D.U.P.O. framework to facilitate the use of dynamic analyses to enable performance optimizations. Since dynamic analyses cannot guarantee complete code coverage, a two part strategy is employed: unit tests are used as a de facto specification, and the programmer provides final verification. The interaction can be kept at a minimum by using the rich information provided by a dynamic analysis. Object inlining is discussed as an example instance of the framework.

Contents

Contents

Contents

Contents

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Traditionally program optimizations use static safety analysis, and are implemented in the compiler. In recent years, modern programming languages have gained features which are inherently difficult to analyze with static analysis. Taking Java as an example, these features are reflection, dynamic class loading, and native methods. While analyzing programs which make use of these features using static analysis is still possible, it results in either imprecise or unsound results. This prohibits the aggressive optimizations which are needed today.

The alternative to static analysis is dynamic analysis. It fares much better with the majority of these features, but in turn has its own inherent problem: how to ensure good coverage. We propose a solution for this prominent question: to use unit tests as a de facto specification of the program. Good unit tests exercise all code, and hence provide a reliable way to obtain good coverage for dynamic analysis. As even this does not guarantee complete coverage, we chose to ask the programmer for confirmation. This interaction gives us the final assurance that the particular optimization is safe to perform, conveniently solving the remaining problems that dynamic analysis has with the language features mentioned above.

We introduce a framework to facilitate using this novel approach: D.U.P.O., or **D**ynamic Analysis, **U**nit Tests, **P**rogrammer Interaction for **O**ptimizations. It is independent of particular optimizations and very general. It provides best practices, guidelines, and an implementation of common functionality for all steps involved: data collection, data storage and management, analysis, programmer interaction and performing the transformation.

As an example instance of the D.U.P.O. framework we present an implementation of object inlining. Object inlining is essentially an optimization which replaces references with the referenced object's members. The version implemented for this thesis is not pushing the state of the art in object inlining research itself, allowing a close examination of the way it works. This allows the reader to focus on the details of object inlining, that it is in fact safe, and how it fits into the framework.

Chapter 2

Motivation

2.1   Need for Speed

Since its formulation in 1965, "Moore's Law" has characterized the development of integrated circuits as doubling their transistor counts every two years [22, 23]. For many years this has resulted in an enormous improvement in computing power with every generation of chips.

This was accompanied by a steady increase in both primary [22] and secondary [32] storage sizes enabling users to handle extremely large input sizes. This trend is not likely to stop in the near future. As an extreme example, the Large Hadron Collider will produce about 15 petabytes per year [8].

While "Moore's Law" (which is more accurately described as a prediction) currently still holds true [33], the increase in transistor counts are no longer proportional to improved computing speed [27]. Limiting factors, such as the maximum propagation speed of signals, lead to a leveling-off of computing speed increases [27].

For easily parallelizable algorithms this does not pose a challenge, since it is possible

3

to scale the number of processing units with the input size. Yet, there will always be instances where this is not the case, and the algorithm does not scale well when parallelized. Since it is infeasible for the processing time to scale proportionally to the input when the input is continuously getting larger, there is an increased need for aggressive optimization strategies.

2.2   Analysis Difficulties

Aggressive optimization strategies require accurate program analysis. This is not a simple undertaking: Modern object-oriented programming languages are making increased use of features such as reflection, dynamic class loading, and native methods. While, for instance, reflection was considered too slow for general programming in the recent past, it has become feasible with today's processing power. All these language features are often used in modern programs [16]. Some examples are: the class loaders defined by Maven project [1] that allows the retrieval of dependencies over the network, plugins in most projects are loaded through reflection, e.g. in the Eclipse IDE [13], and middleware like the persistence solution Hibernate [17].

It is these very dynamic features that make it very difficult for static analysis to remain accurate:

1. "Reflection" allows the inspection and manipulation of objects at run-time.

   Essentially reflection is a form of meta-programming. While it would be possible to analyze it using a static approach, the analysis complexity increases greatly [16].

2. "Dynamic class loading" allows the addition of executable code at run-time.

   Once executable code can be added at run-time, a static analysis can never inspect the whole program at once. It is possible of course to re-analyze after a class is

loaded, but this requires a complicated change management. If a re-analysis determined that a previously applied optimization no longer applied, the running code and/or data needs to be migrated as the optimization is reverted.

3. "Native methods" allow code to be executed outside of the Java run-time environment.

    This different run-time environment means that the same analysis methods deployed on the Java parts of the program cannot be applied. Usually native methods allow execution of arbitrary code, so in general analysis is not possible[1].

Optimizations in compilers have to preserve the semantics of the program because otherwise correct execution cannot be guaranteed. Colloquially a program transformation, e.g., an "optimization", is termed "safe" if it does not change the program semantics. Traditional compilers have to ensure that the optimization is safe in all possible executions since the compiler has no knowledge of how the program is going to be used. As it is not feasible to analyze all possible executions, abstract interpretation [11] was invented, which is used by most, if not all, static analyses.

Abstract interpretation maps from the actual program states to a lattice of abstract states. It uses rules for each program operation to transition in the abstract lattice. This makes it possible for an analysis to be completed in a reasonable amount of time. However, this mapping comes at the expense of analytic accuracy. Some program states compress well and mapping into the abstract lattice preserves most properties. Regardless this is not true of most operations; often information is lost when mapping into and transitioning in the abstract lattice. Abstract interpretation has worked well in the past, but it is inherently difficult to handle features like reflection. Essentially reflection uses the value of variables to decide further actions, but the abstracted states do not provide exact variable values.

In the end, the level of accuracy reached through static analysis is not acceptable. Some

---

[1]In principal machine code could be analyzed, but this is not a practical solution.

research ignores these language features, resulting in either an unsafe analysis on programs using them, or greatly reducing the applicability of the transformation [16]. Since it is likely that usage of these language features will only increase over time as programming becomes higher-level, it is unlikely that static analysis will be able to give sufficient accuracy for deciding when to apply program transformations. A consequence of these limitations is that we need an alternative to static analysis that enables a more accurate analysis, allowing us to apply optimizations more aggressively.

Some compilers use profiling data to decide when to apply transformations. Most notably just-in-time (JIT) compilers rely heavily on sampling data to decide when to apply transformations. This dynamically collected information is only used in conjunction with a static analysis result: the latter is used to show that the optimization is safe to apply, and the former is only used for profitability analysis.

2.3   A Dynamic Approach and Unit Tests

The alternative to static analysis is dynamic analysis, which collects its data from concrete runs. In contrast to its static cousin, it is not limited by the number of possible program states, but only by the number of encountered states. Dynamic analysis has access to all details of the program's behavior and can produce very detailed reports limited only by its overhead cost. The flip side of observing concrete runs is that there is no guarantee that all states which the program can be in are actually observed. This encourages dynamic analysis to be primarily used for gathering performance data. We believe that this limitation can be mitigated and dynamic analysis can be used successfully for safety decisions as well.

If somehow the dynamic analysis was able to observe all possible usage patterns of the program, then the collected data would allow a complete, and thus safe, analysis. It would yield a superset of the information we can gather from a static analysis and be more

accurate. Unfortunately, there is no method to ensure that a number of program runs cover all possible program states. In fact, such a proof would require static analysis, which we are trying to avoid for the reasons mentioned above.

After thorough examination of the different approaches, we are still left with the question: what other ways are there to ensure a program transformation is correct? If there was a formal specification of the program, then the transformed program could be tested against the specification to ensure that it is still valid. Most modern object-oriented programming languages are not easily verified by formal methods since, again, these methods are essentially a form of static analysis. In practice verification tools, e.g., ESC/Java2 [10] used in conjunction with JML [26], perform static verification, but only consider one method at a time, and not the correctness of the entire program. This restriction allows a static analysis to be used without having to analyze the complicated control flow made possible by the aforementioned dynamic language features.

What comes closest, though not a formal method, is the use of unit tests to verify correct program behavior. The term "Unit test" itself is not formally defined. Commonly it is used to refer to a set of tests which verify correct behavior of the program's units. The term "unit" is also not formally defined, but in the context of object-oriented programming languages a class can generally be considered a unit. The idea of unit testing came from methodologies like "Test-Driven Development" [14, 34] or "Extreme Programming" [2, 18]. The goal is always to provide a set of tests which exercises the complete program. Part of unit testing, in difference to other testing strategies, is to test every unit in isolation, e.g., through the use of mock objects. Note that due to the lack of exact definition a number of programmers do not at all, or do not always, adhere to this principle of isolation.

Assuming that unit tests are complete, they can be used as a de facto specification of the program's behavior since the tests cover all the functionality of the program. Therefore, the execution of the unit tests can be used by the dynamic analysis to capture the complete behavior of the program. We propose this as a solution to finding the representative run

for dynamic analyses.

Other tests aside from unit tests can also be incorporated, as can any other runs which are deemed representative. These additional inputs to the analysis become important when the unit tests are known to be incomplete. When unit test incompleteness is known, these other inputs must be representative – otherwise incorrect conclusions will be drawn.

Even with complete coverage, one modern object-oriented features remains problematic: dynamic class loading. As it is not possible to determine in advance if, when, and how classes are loaded [16], it is difficult for analyses to take dynamic class loading into account. Essentially it cannot be guaranteed that the whole program is available for analysis, but whole program analysis is essential for a number of more aggressive optimizations.

One solution is to perform the dynamic analysis at run-time and execute the optimization on-line. In practice, this is only possible in rare cases. Dynamic analysis has a high overhead cost when compared to the regular run-time of programs, prohibiting on-line operation as the slowdown is not acceptable in production. The better solution is to use off-line analysis and find a way to provide the missing information.

2.4   Programmer Involvement

Part of the reason for the missing information is in the limitations of programming language semantics. Let us consider reference and value semantics of variables as an example[2]. The C++ language has a construct which allows the programmer to select between reference and value allocation, but Java does not have this feature. Instead, Java uniformly uses reference semantics with objects[3]. Therefore, it would be an optimization to use value semantics when it is beneficial. Any analysis for this optimization has to handle

---

[2]This is the basis for our example instance of the framework presented in Chapter 5.

[3]Note that Java uses reference semantics for object allocation, while always passing parameters by value.

these problematic language features we have discussed. For this case it would be easy to add a *use-value-semantics* annotation to the source code, rendering the analysis obsolete, since that is all the transformation needs to know to be applied.

Using annotations does have a number of drawbacks. A value annotation changes the semantics of using the object, e.g., an assignment now copies the values of the object instead of creating another reference. Note that this is *not* apparent by reading the assignment statement, instead one has to be aware of the declaration of the object to know what effect the statement has. One could speculate that this is the reason that value semantics are not a language feature of Java. An alternative, using different syntax for accessing a reference object and a value one, is used by the C language. While it is immediately apparent what semantics the operations have[4], changing between the two different modes now becomes cumbersome: not only the declaration, but most operators on the variable in question must be changed[5].

We must pose the question: is it really desirable to keep this information at the source level? Changing between value and reference semantics can make drastic performance differences depending on the usage patterns. Thus there are advantages to not specifying it explicitly but instead having it be automatically determined by the optimizer. We must also keep in mind that the question of reference vs. value semantics is just an example. It would be too demanding of the programmer to specify extra information in the source code for a number of optimizations.

The use of annotations does show that the information required often is available: just not in the source code, but instead in the programmer's head. We want to make use of this additional information source while retaining as much automation as possible. In this thesis we aim to make use of profiling information to minimize the interaction with the

---

[4]C does not have different operators for assignment, just for accesses.

[5]We do realize that automated refactorings could make changing from reference to value semantics easy, however, value semantics still are not available in Java.

programmer. This allows the programmer to focus his attention on the specific places where an optimization is likely to be safe and profitable.

Chapter 3

The D.U.P.O. Framework

The methodology to use off-line dynamic analysis, unit tests as the input for the analysis, and getting final verification from the programmer is a new one. This is why we created our framework. The name, D.U.P.O., is a simply acronym for its main components: **D**ynamic Analysis, **U**nit Tests, and **P**rogrammer Interaction for **O**ptimizations. Its purpose is to guide the programmer implementing a specific optimization using this methodology with recommended practices, and also code for common functionality. Hence D.U.P.O. does not handle anything specific to particular optimizations.

What we have outlined in Chapter 2 is a direct template for our framework. D.U.P.O. is very general, and the specific needs for each step will vary depending on the particular optimization; nonetheless there is some general structure which will always be the same. That structure is discussed in this chapter.
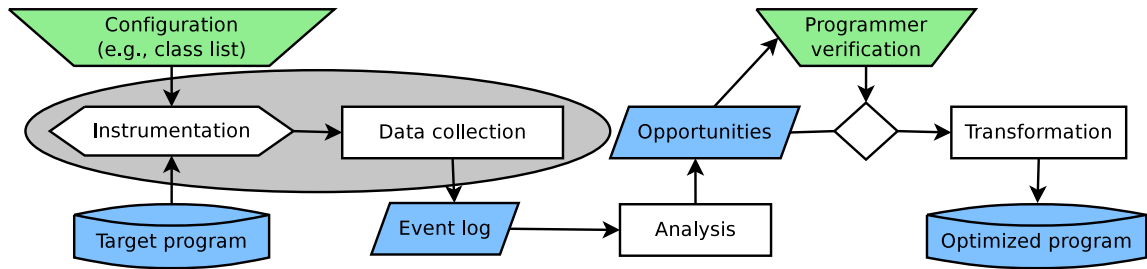
Figure 3.1: Brief D.U.P.O. framework overview.

3.1 Introduction

First we give an overview of the whole system, and introduce some of the terminology. The normal workflow for applying an optimization with the D.U.P.O. framework to some program is shown in Figure 3.1.

The dynamic analysis consists of two parts: data collection and analysis. First we collect the data by observing a program run. The run is observed by strategically instrumenting the program beforehand. During execution the instrumentation monitors the behavior of the program. Once it decides that the program performed a noteworthy action, we say that an event was generated. Throughout the program execution many events will be generated. For flexibility the D.U.P.O. has the notion of a logger, which decides what to do with each event. After all the data have been collected, they are analyzed. When favorable opportunities are found, they are presented to the programmer. Together with the opportunity, additional information should be displayed which help the programmer in making a decision about its applicability (this is called the *support information*). If the programmer confirms that the opportunity is safe, then it is automatically applied.

More precisely we define:

Definition 1 (run)**.** *A "program run", or just run, refers to the execution of the instrumented program.*

Definition 2 (event)**.** *An event refers to the data generated by the instrumentation during*

a run for some particular program action.

Definition 3 (logger). *A logger refers to an object to which the instrumentation passes the events. It decides how the event should be handled.*

Definition 4 (log). *An* event log*, or just* log*, contains the data produced by a* logger*; it is a stream of* events*.*

The ideas presented in this chapter are implemented together with the example instance of Chapter 5. They are not separated into a separate project, but are logically divided into packages. While not completely prepared for use with other optimizations, it is designed with reuse in mind.

3.2   Data Collection

In difference to static analysis, dynamic analysis requires run-time support. The instrumentation code will usually be added through off-line or load-time bytecode rewriting. Both work essentially the same way, and there is very little difference between the methods for the needs presented here[1]. Source code modification was not chosen for the same reasons people usually avoid it: it is much more complex. Ultimately this choice is up to the individual optimization, but we believe that in the vast majority of cases there is no need for information that is available at the source code level, but not at the bytecode level. This is particularly true since an inspection of the instrumented program should not be necessary.

We believe that JVM modifications to collect the data are not the best option for implementations of D.U.P.O. Working within the JVM provides performance benefits, but also

---

[1]The difference between off-line and load-time binary rewriting is more an implementation question: Load-time instrumentation works better when dynamic code loading is involved, although it is possible to simply rewrite the code before it is dynamically loaded as well.

is significantly more difficult to implement. Java optimizations are often implemented directly in the JVM, because they are *on-line* optimizations, and can be seamlessly integrated into the compilation process[2]. Note that in our framework applying the optimization is necessarily an *off-line* process: The programmer involvement, explained in Section 2.4, dictates it. Therefore bytecode rewriting off-line or at load time is the best choice. The implementation in this thesis uses off-line bytecode rewriting.

What data must be collected by the instrumentation is entirely dependent on the individual optimization. We do believe that sometimes the collected data can be used for more than one optimization (See Section 6). While it is certainly possible to record *all* operations, and then simply extract those necessary to enable a specific optimization on demand, the data collection time would be prohibitive. It follows that generally only the necessary data should be collected, unless it is known that some superset would in fact enable it to be utilized for more than one optimization.

Even though different optimizations use different data, each stage in the software development life-cycle has similar requirements on how to handle the data. While testing the instrumentation, for example, it can be useful to simply ignore any data which would normally be collected. When debugging what data are actually collected, it can be useful to work with small examples, and directly display the collected data on the console. Essentially a mechanism is needed to switch between storage back-ends. The back-end classes, in contrast to their implementation, being independent of the optimization.

This back-end class must be initialized, so some management code is required to be executed before the actual program is started. The programmer has to specify this *main* method so that the instrumentation can insert the call to the framework initialization as the first statement in the main method. Similarly there might be a need to perform some cleanup actions before exiting the application, e.g., closing a database connection. Java

---

[2]Here compilation refers to the translation from bytecode into native code by the optimizer within the JVM.

gives us a nice mechanism for this: shutdown hooks[3]. The hook is inserted by the initialization code, and instrumentation is only required in one place. This is preferable to adding instrumentation directly to call the cleanup code, since there are several ways to terminate an application[4]. Note that several shutdown hooks can be registered, and are executed in parallel when the JVM shuts down. This is undesirable, since the framework could be shutting down in parallel to application shutdown procedures which still generate events. This is easy to work around, application shutdown hooks could be redirected to the D.U.P.O., and called from its shutdown hook. This establishes the correct ordering[5].

Bytecode rewriting has the advantage that for several runs the instrumentation only has to be added once. But not every run has the same data collection requirements, e.g., once the instrumentation has been tested with no output, one may want to see the output on the console. So a mechanism is needed to switch between these different output methods at run-time. Usually parameters are passed to programs on the command line. But this method might already be used by the application itself; we want to pass parameters to the framework only. It would be possible to use arguments for both, and have D.U.P.O. clean up the arguments before the application sees them. But this method has the downside of conflicting parameter names between the application and the framework. We chose to use environment variables instead[6]. While not being immune to conflicts, they are not traditionally used by Java programs, which makes a conflict unlikely. It also provides a clear separation between the parameters passed to the application and the ones passed to the framework. This mechanism is used to select the logger at run-time.

At instrumentation time a default logger is selected. This can be overwritten at program start. Then it is up to the optimization to install the appropriate type of logger through a Factory method. The framework provides a mechanism to handle some common data

---

[3]See the Java shutdown hook documentation.

[4]One way to exit an application is to call System. exit (), which could be called anywhere in the whole program.

[5]The redirection of application shutdown hooks has not yet been implemented by us.

[6]Java properties would be equally suited for this purpose.

which is independent of the optimization, e.g., Java types. We will elaborate on this in the next section.

3.2.1 Working with Types

We believe that types are often going to be of interest for the analysis. Types in Java are represented internally in a ".class" file as part of the constant pool, but everywhere else as strings. This makes them easily human readable, but slow to process. Since the instrumentation code in hot places will be executed extremely often, we optimized the look up by using integers to represent types. When an event includes type information, we now only have to log an integer, and not a string.

This translation is most efficient at instrumentation time, so that an integer is directly used in the added code. The instrumentation uses a database as a back-end for the translation from a type string. This allows us to switch to a dynamic look-up, where the database call is inserted instead of the integer constant when necessary. The additional advantage of using a central database for the type information is that it is consistent between runs. Therefore there is no need to normalize the integer type when processing more than one run at once.

3.3   Data Storage

Regular runs need to store the event data. The kind of data is of course dependent on each optimization. Yet it is the nature of most dynamic analyses to produce many events, with a small number of event categories, and little data for each individual event. The combined stream of events often yields a large amount of data, even several gigabytes (as is the case for the example instance of object inlining). Initially we thought that it would be a good idea to store the data in a database. But it turns out that regular databases are

not well adapted to handling the kind of data streams which the instrumentation produces (see Section C.1).

While there might be exceptions, we believe that most analyses will aggregate while passing over the stream of events. So when random access is not required, a flat file layout will yield a significant performance improvement, while still being simple to implement. It is possible to use Java serialization to write the data, but this would again introduce some overhead. We believe the most efficient way is to use a custom format. Since events usually carry little data, they are easy to write out by hand, and also easy to parse. For details see Section B.3. Note that if random access is in fact required, it is likely to require a different storage format.

3.4   Data Management

The off-line nature of our framework makes data management important. Depending on what information an optimization needs to collect, every run potentially generates a large amount of data. The time required to collect the data is usually proportional to the number of events. This makes it important to retain the results, as collecting them again would be expensive. With the easy availability of large storage capacities, we do not think there are any reasons to quickly discard the collected data.

Thus throughout the development time of every optimization, a large number of runs will be stored as it is executed in varying stages and on different programs[7]. This makes keeping an inventory of the run metadata essential, as otherwise the details of a run will too easily get lost or mistaken. The framework stores the parameters of every run in a database, and there is a front-end available to manage the metadata, independent of the individual optimization (See Section A.2).

---

[7]We worked on 1050 runs, and still has 374 stored on disk at the time of writing.

3.4.1 Run Metadata

There are a number of interesting general parameters to record for each run. They are of organizational nature, and are independent of what is recorded for each optimization. The goal of the common metadata collection is to associate the environment of each run with the analysis results. Not only does this improve reproducibility, but also helps tremendously while the optimization is still in development.

For each run we record the following metadata:

- **Application.**
  A string describing the application, and a string describing the version. The values are initially set during instrumentation. The application name can be overwritten at program startup. This is useful mainly for application bundles, e.g., the DaCapo benchmark suite [3], where the whole bundle was instrumented at the same time, but runs of each application need to be recorded individually. There is no need to overwrite the version number in this case, since the applications version can be derived from the bundle's version.

- **Start and finish time.**
  The start and end time of the run. This is used to order runs. In particular in the development phase it is useful to know how long a run took, and with which version of the instrumentation it was compiled. While version of the instrumentation is not explicitly recorded, with good programming practice, i.e. always checking code into a version control system (VCS) before its use, it can be exactly determined.

- **Main class.**
  This is used for disambiguation in case there are several entry points for an application. An example is different front-ends, or unit tests.

- **Current working directory.**

The directory in which the run was started in. This is important to allow easy location of data files which the run's logger might have saved.

- **User.**

  The username of the user who started the run. It allows to distinguish between similar data collections by different people. If there are several users working on the same project, they can share the same main database for consistency.

- **Classpath.**

  The classpath used for the run. This is recorded because for Java programs this is an essential part of the run-time environment.

- **Console output.**

  We recommend to save the console output for reference. Since this is done easily using external scripts, the framework only provides a method to save the location where the output was redirected to.

- **Data log.**

  If a logger was used which saves data to a file, then the location of the file can be saved.

- **Note.**

  It can often be important to make a note about a run, for instance, "This run aborted with an exception".

- **Group.**

  At times runs belong together; they form a group. Grouping becomes essential when unit tests are observed in isolation, and hundreds of runs are created by one execution of the complete unit test suite. Currently a run can only belong to one group[8].

---

[8]The one group limitation could easily be removed with some implementation effort.

We believe that this set of metadata captures the most important common attributes of a run.

3.5   Analysis

The analysis part of the framework does not come with many notes or guidelines. The framework implementation makes it easy to allow the analysis to run either in-process with the data collection, or as a separate pass after all the data have been collected. It is recommended to run the analysis separately, mainly for reduced interference, and to allow re-analyzing the results. An example of interference is the changed memory behavior, which could result in too little stack or heap space being available.

As was already mentioned in Section 3.3, we expect most analyses to require linear scans of the log, but there are likely to be exceptions to this.

It is of note that since the analysis has to determine whether it is safe to perform an optimization, it cannot rely on sampling data. Sampling data is only suitable for performance predictions, and not to determine exact behavior as is required for most aggressive optimizations. Optimizations which *do not* require exact behavior are not the target for this framework. They are likely better implemented within the JVM.

Let us introduce some common terminology:

Definition 5 (opportunity). *An opportunity is one particular place where an optimization can be applied.*

The analysis needs to keep in mind that the programmer has to verify the opportunities it finds. Since usually it is not known in advance how many opportunities will be found, the analysis should also gather some heuristic to sort the opportunities. A heuristic might need additional data to be collected, which should be considered from the beginning on.

3.6   Programmer Interaction

Once a number of opportunities have been found by the analysis, they need to be verified by the user. Confirming a large number of opportunities is not an option since it would require too much time. Therefore the opportunities need to be sorted by expected pay-off value, and the programmer can then confirm only the most valuable ones[9].

Aside from limiting the number of opportunities, it is also required to keep the interaction with the programmer on each one at a minimum. Ideally the programmer is only asked a simple *yes/no* question, although slightly more complex interactions are suitable as well if they do not require too much specialized knowledge.

Aside from asking only simple questions, we can cache the decisions of the programmer, and reuse them if possible. The system knows what information was presented to the programmer when the decision was made. It can monitor if the conditions change, and once they do, the programmer has to reconfirm his choices. It is up to each individual optimization, however, whether to really cache the programmer's input or not: If it is determined that the assumptions for the optimizations are very volatile, then no cache should be used. We argue that one should strive to design optimizations in this framework so that they are not particularly volatile, and a cache can be used effectively to ease the burden on the programmer.

3.7   Transformation

If the programmer confirms any opportunities, the optimization can be applied by transforming the program in the places the opportunities specify. The options of how the optimization is applied are the same ones as for the data collection, which was discussed in

---

[9]The exact threshold of opportunities to display can be user configurable.

Section 3.2. The choice between the options, however, is a slightly different one. Implementing the optimization in the JVM is not beneficial for the same reasons as for the data collection; since this framework is an off-line process. This time there is an advantage to transforming the source code over transforming bytecode: the source code can readily be presented to the programmer, while bytecode can only be inspected by experts[10]. We do not expect that manual inspection of the transformation result will be a required feature for many optimizations. In fact, this aspect should not be used as part of the regular workflow, although it could conveniently be offered as an optional setting, e.g., for debugging purposes. Rather we believe that knowledge about the exact state of, and changes to the source code allows the system to present clearer questions to the programmer in the verification step described in the previous section. If a source rewriting is chosen, the result must of course still be compiled. Then the optimized program is ready.

Implementing the transformation outside of the JVM offers another advantage: it automatically becomes portable across JVMs. This creates exciting new opportunities for cross-validation, i.e. checking that any speed-up is not just an artifact of interaction between existing JIT optimizations. It should be noted that while the framework does not need to be adjusted to do this, the details, and the infrastructure requirements are out of scope of this thesis.

Through the experiences we have gained from implementing the example optimization, we can now recommend to nonetheless use bytecode rewriting over source code rewriting. The complexities of the latter outweigh the advantages for interacting with the user: it is more complicated to cover the whole Java language when dealing with source code compared to dealing with bytecode.

---

[10]We claim that anyone knowing Java bytecode well enough to understand what it does can be considered an expert.

3.8   Conclusion

Certainly not all optimizations fit the framework presented here. In particular when the analysis is inexpensive so that it can be performed online, then an implementation inside the JIT compiler might be more appropriate. But we believe D.U.P.O. offers good support for a class of optimizations which are not easily integrated into the JVM's adaptive compilation system. Additionally some optimizations which could be implemented online might benefit from better analysis and more comprehensive profiling to allow more aggressive application than before.

In the range from completely automated to manual optimization, this framework stands somewhere in the middle: while requiring some manual input, it is mostly automated. We believe this is a novel balance, which has not been explored before in this form.

Chapter 4

Comparison with other Approaches

If program transformations are lined up on a scale from fully automated to completely manual, this framework can be placed somewhere in the middle. Ideally its implementations are situated more on the automated side, but it cannot be guaranteed. As far as we are aware, this level of automation is not well explored.

The contrast with manual work is obvious; the transformations used here are not permanent, and can at any time be omitted again. But the integration of data collection and analysis provides a contrast to code refactoring tools, which usually solely rely on the programmer's decision. The difference to fully automated optimizations is obvious: this framework uses programmer interaction. This introduces another difference: being an off-line optimization. The off-line nature has the advantage that the implementation of the optimization does not have to include any undo code, or monitoring. This key difference makes it difficult to compare optimizations implemented in this framework to, for example, those implemented within the JVM.

The most similar existing work is probably Daniel von Dincklage's work on specifying intended semantics to enable optimization [31]. He focuses on bridging the gap between the actual Java semantics and the programmer intended semantics using an iterative ap-

proach of specifying intentions, and finding reasons for failing to apply an optimization. His approach is more interactive than what is proposed in this thesis: his tool asks the programmer repeatedly to specify the intended semantics at places his analysis has identified to block optimizations. Another difference is the granularity of the questions; in von Dincklage's work the programmer input is given for individual semantics, e.g., "No exceptions will occur at this point". Our framework asks higher level questions specifically designed for an optimization opportunity, such as "Does field f of class C have value semantics?"

Chapter 5

Object Inlining

As an example of the D.U.P.O. framework we implemented the *object inlining* optimization. We define object inlining as replacing an object reference with the referenced object's members (fields and methods). It is primarily a storage transformation; what was a reference field before is transformed into one or more value fields. Accesses to the transformed fields must also be transformed to refer to the new location(s), as well as redirecting method calls where the inlined field is the receiver.

5.1 Motivation

Most people consider it good software engineering to modularize and separate concerns into components. In object-oriented programming languages, like Java, this practice results in programs using many, often small, classes.

Allocation of memory, garbage collection, and dereferences can make the use of many small objects very costly at run-time. In the Java programming language there is no language support for avoiding these costs while still maintaining a good programming style:

Java only has heap based allocation and no native support exists for allocating objects on the stack. Object inlining is an optimization which allows the programmer to use fine-grained modularization while minimizing the aforementioned run-time costs. Object inlining combines objects so that they are allocated and initialized together, and in most cases saves some dereferences to access the inlined object.

There were a number of reasons why we chose object inlining as the example instance of our framework:

- It is easy to follow what effect the transformation has on the source code.

- It is easy to understand how the transformation affects interaction of existing with dynamically loaded code.

- While being a well explored optimization, it benefits from the features this framework offers.

The inlining algorithm which is described here is not very sophisticated. The focus is on highlighting how the concept works and fits into the framework.

5.2   Introduction

The idea behind object inlining is easy to understand, but grasping the details necessary to understand the safety analysis initially can be difficult. Therefore we will illustrate the concept with an example, and incrementally introduce more concepts and terminology. We hope that this progressive build-up help the reader to become easily acquainted our implementation. The terms introduced here will be more thoroughly defined in Section 5.3.

The usual workflow is instrumentation[1], analysis, and transformation. In this chapter we describe the process in reverse order. Taking this approach makes it easier to follow,

---

[1]Instrumentation is followed by data collection, which is simply running the instrumented code

```
1  class  Point {                          1  class  Circle {
2      int x;                              2      int radius ;
3      int y;                              3      Point  center ;
4  }                                       4  }
        (a)                                        (b)
```

Figure 5.1: The classes used in the introductory example.

```
1  class  Circle  {
2      int  radius ;
3      int  center_x ; /* inlined from Point */
4      int  center_y ; /* inlined from Point */
5  }
```

Figure 5.2: The result of inlining `Point` into `Circle.center`.

since the requirements for each stage are derived from the one following it. After all, the goal of the whole process is to enable the transformation.

5.2.1 Data Structure Changes

Consider the class definitions shown in Figure 5.1. In (a), class `Point` defines a simple point in 2D space. This `Point` is used in (b), where class `Circle` defines a circle in 2D space. The field `center` of the circle, is a `Point`.

In this example it is possible to replace the field `center` of class `Circle` with the members of `Point`, fields `x` and `y`. We say that class `Point` was *inlined* into the field `center` of class `Circle`. This results in the layout shown in Figure 5.2. Note that the reference `center` itself was eliminated.

---

with the appropriate inputs. Since the run-time support is provided by the framework, there is no additional description necessary.
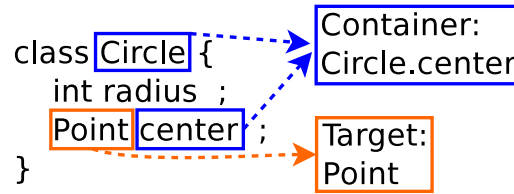
Figure 5.3: Illustration of the two central terms to object inlining.

```
1  class  Point {                          1  class  Circle {
2      int  x;                             2      int  radius ;
3      int  y;                             3      Point  center ;
4                                          4
5      Point(int  x,  int  y) {            5      Circle  (int  r) {
6          this .x = x;                    6          this . radius  = r ;
7          this .y = y;                    7      }
8      }                                   8  }
9  }                                       9
```

(a)   (b)

Figure 5.4: The extended base classes for the introductory example.

The class `Point` is called the target, and we say we *inline* `Point`. The field `center` of class `Circle`, or `Circle.center` is the container. We say we *inlined into* `Circle.center`. Putting these two terms together, we say we *inlined* the target *into* the container. The terms target and container are also illustrated in Figure 5.3.

5.2.2 Code Changes

Now let us consider behavior; how the data structures that `Point` and `Circle` provide are used. For this we extend the definitions in Figure 5.1 with a constructor so that it can be conveniently initialized, as shown in Figure 5.4.

With the extended base classes in mind, look at the simple program presented in Fig-

```
1   Circle  c1,c2;
2
3   c1 = new Circle(1);
4   c1. center  = new Point(1,1);
5
6   c2 = new Circle(2);
7   c2. center  = new Point(2,2);
8
9   System.out. println  ("C1:␣" + c1. center .x + "," + c1. center .y);
10  System.out. println  ("C2:␣" + c2. center .x + "," + c2. center .y);
```

Figure 5.5: Short program which uses the extended base classes.

ure 5.5. It creates two circles, each having a distinct `Point` instance in its `center` field. It then prints out the coordinates of the center of each circle.

Now consider the same inlining as before, `Point` getting inlined into `Circle.center`, shown in Figure 5.6. For the simple program to remain valid, it has to get adjusted accordingly, as shown in Figure 5.7. Note that the call to `new Point(...)` that initialized the `center` field was eliminated, as this field is now part of the class `Circle`. It is replaced by a call on the circle instance, `new_center(...)` which takes over the initialization functionality of the constructor. The other change is the access to the coordinates `x` and `y`. Before the inlining a reference to a `Point` instance was obtained through the field `center`, and then the field `x` or `y` was accessed, Now, the `x` or `y` coordinate is accessed directly through `Circle`'s field `center_x` or `center_y` respectively.

The main difference is that the reference `center` does not exist any more. As such, it cannot, and need not, be dereferenced[2]. The reference played a role in both the constructor call (where it is implicit, as `new Point(...)` both initializes the memory, and calls

---

[2]The term dereference is not usually used in the Java language. But since Java references are technically just pointers, we felt that the term is established well enough to use it in the Java context without additional explanation.

```
1  class Circle {
2      int radius ;
3      int center_x ; /* inlined from Point */
4      int center_y ; /* inlined from Point */
5
6      /* the Circle constructor was unaffected by the inlining */
7      Circle (int r) {
8          this . radius = r;
9      }
10
11     /* the transformed Point constructor */
12     void new_center(int x, int y) {
13         this . center_x = x;
14         this . center_y = y;
15     }
16 }
```

Figure 5.6: The result of inlining the extended `Point` into the extended `Circle.center`.

the constructor on the new memory chunk), and in the access of the x and y coordinates. Therefore a dereference was removed when both the constructor call, and the coordinate

```
1  Circle c1,c2;
2
3  c1 = new Circle (1);
4  c1.new_center (1,1) ;
5
6  c2 = new Circle (2);
7  c2.new_center (2,2) ;
8
9  System.out. println ("C1:_" + c1. center_x + "," + c1. center_y );
10 System.out. println ("C2:_" + c2. center_x + "," + c2. center_y );
```

Figure 5.7: Short program which uses the inlined extended base classes.
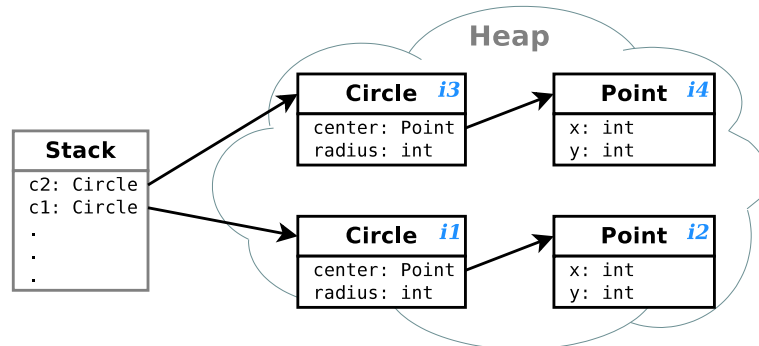
Figure 5.8: Heap snapshot at the end of running example 1.

accesses inside the print statement, were transformed. It should be noted that the trans-
formed constructor itself did *not* eliminate any dereferences since the transformation result
dereferences `this` in the same way as the original constructor.

5.2.3   Analysis Principles

After the transformation has been introduced, we can consider the analysis that determines
if it is safe to apply in the first place. The objective is to check if the transformation
preserves the behavior of the original program. As mentioned earlier, object inlining is a
storage transformation, enabling its effects to be studied by looking at heap snapshots.

In order to discuss the object instances shown in the heap snapshot, we must identify
and name them. An id is assigned to each instance, the instance id. It is displayed as *iX* in
the snapshot, where *X* is a running number, next to the instance's class name (cursive, and
in light blue, if color is available).

First, consider the original program in Figure 5.5. The state of the heap at the end of
the program execution is shown in Figure 5.8. Here, every instance of `Circle` points to
a unique instance of `Point` (*i*1 → *i*2, *i*3 → *i*4). Additionally, there exist no other object
references to those instances of `Point`. As a result, what was an independent storage
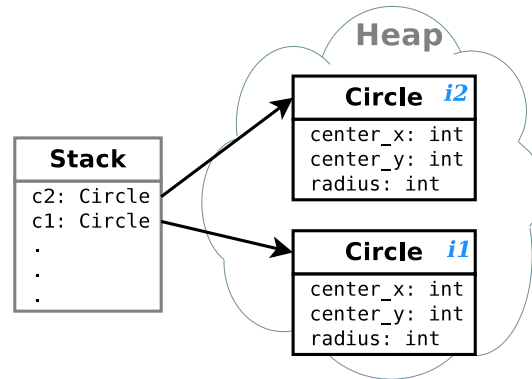
Figure 5.9: Heap snapshot after inlining at the end of example 1.

location before the inlining remains an independent storage location after, and the inlining is safe to perform.

The inlined program in Figure 5.7 has a different heap snapshot at the end of its execution, as shown in Figure 5.9. Since the `Point` object instances are no longer allocated, the size of the heap has been reduced leaving no object reference `center` anymore.

In contrast consider the code in Figure 5.10. It is very similar to the previous one shown in Figure 5.5, but only creates one `Point`, and assigns it to the `center` field of both `Circle` instances.

In this second example is *not* safe to inline. The reason is easily observable in the heap snapshot at the end of program execution, as shown in Figure 5.11. Notice how both instances of `Circle` hold a reference to the same instance of `Point` ($i2 \rightarrow i1$, and $i3 \rightarrow i1$). If `Point` was inlined into `Circle.center`, then this single storage location in the original program would become two storage locations (the heap snapshot at the end of program execution would still look like the one shown in Figure 5.9). Hence the update of `c1.center.x` on line 11 would not be reflected in `c2.center.x`, and behavior would not be preserved. The general case is that any `target` instance that has more than one alias will change program behavior if it is inlined.

```
1   Circle  c1,c2;
2   Point  p;
3
4   p = new Point(1,1); /* Only one Point for both circles */
5
6   c1 = new Circle(1);
7   c1.center = p;
8   c2 = new Circle(2);
9   c2.center = p;
10
11  c1.center.x = 3;
12
13  System.out.println ("C1:␣" + c1.center.x + "," + c1.center.y);
14  System.out.println ("C2:␣" + c2.center.x + "," + c2.center.y);
```

Figure 5.10: Example 2. A variant of the code shown in example 1 that is not safe to inline.

Heap snapshots are an excellent visualization tool and well suited to introduce the concept. For the simple programs presented here, it sufficed to inspect a single heap snapshot since each reference was only assigned to once. In more complex programs references are frequently assigned to, and a heap snapshot would have to be examined after every change. Considering that heap snapshots are relatively expensive to collect, in
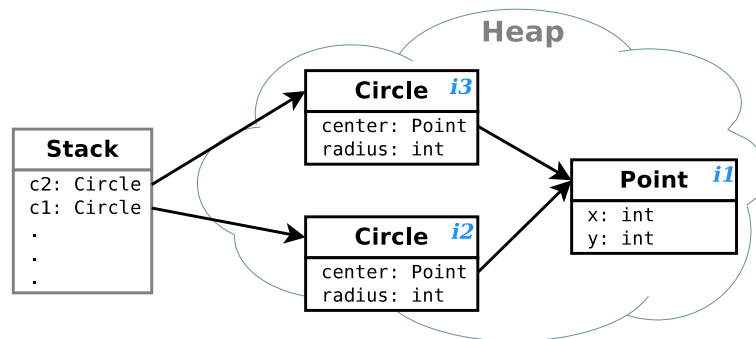
Figure 5.11: Heap snapshot at the end of running example 2.

practise we collect information to come to the same conclusion in a different manner.

5.2.4 Actual Analysis

In the previous section the heap snapshots were examined to essentially construct points-to relations. The actual analysis is implemented so that it implicitly keeps track of the state of the heap. It does so by recording an event whenever an object instance is dereferenced (or otherwise used, which will be introduced later). The event needs to include information on *what* instance was dereferenced, and *where* the dereferenced object instance was obtained from. This allows the analysis to reconstruct points-to relationships.

The dereferences of the target `Point` that can be observed in Figure 5.5 are on the last two lines, where the coordinates are printed out, namely the subexpressions `c1.center.x`, `c1.center.y`, `c2.center.x`, and `c2.center.y`. We obtained these subexpressions from looking at the source code[3], but at run-time things look slightly different.

Take for example only the expression `c1.center.x`. At run-time when the data is collected, the instance of `Point` referenced in the expression is $i2$, as we see in the heap snapshot in Figure 5.8. This reference was retrieved from the container "field `center` of `Circle` instance $i1$". Notice that the particular field that was accessed here, `x`, is not important, we only care that *something* of this `Point` instance was accessed. In the expression `c1.center.x` there actually is another dereference, the one of `c1`. This is a different and distinct event, which we will discuss later.

The fact that instance $i1$ is of type `Circle`, and $i2$ is of type `Point` must also be recorded. This is done in a *creation* event, which logs the instance id together the instance's type.

---

[3]The uses of the coordinates `x` and `y` in the constructor are left out w.l.o.g. to simplify the analysis demonstration. They are in fact irrelevant, as we will point out later.

The analysis then considers all events which involve targets of type `Point`: the four dereferences mentioned earlier. We refer to the source statements which generate these events, but also remind the reader that this is only meaningful because no reference is reassigned or statements executed more than once in the example program. The stream of events the analysis processes is:

- *Creation* event, instance $i1$ has type `Circle` (for easier understanding this information is repeated in the other events, instead of just mentioning the `instance` id.

- *Creation* event, instance $i2$ has type `Point`.

- *Creation* event, instance $i3$ has type `Circle`.

- *Creation* event, instance $i4$ has type `Point`.

- For `c1.center.x`, the target is the `Point` instance $i2$, the container is `Circle` instance $i1$'s field `center`.

- For `c1.center.y` an event with the exact same data is generated. As mentioned before, this is caused by the event not recording which field of the `Point` instance was accessed, as we are only interested in reconstructing the points-to relationships.

- For `c2.center.x`, the target is the `Point` instance $i4$, the container is `Circle` instance $i3$'s field `center`.

- For `c2.center.y` the same applies; it is exactly the same event as `c2.center.x`.

At this point we can come to the same conclusion as in the previous section. Each `Point` instance was accessed *only* from the same container, and each of these containers *only* had that `Point` instance as a target. Thus, we arrive at the same conclusion, that the inlining of `Point` into `Circle.center` is safe.

Collecting events from every use might appear strange at first, since we could also just monitor any assignment which changes a reference. The reasoning is twofold: the approach described here is safe in regards to native methods. Native methods can modify essentially arbitrary instances on the heap. Any reference which is reassigned in a native method would not be detected by an instrumentation which only monitors assignments. Since we monitor actual uses, the effects of the reassignment are visible to our instrumentation. The second reason is that this information provides valuable information for performance impact estimates, which we will describe and use later in Section 5.7.2.

5.2.5   Beyond the Example

The terms target and container are central to the safety analysis. The term container was previously used in different situations, and it is crucial to differentiate between them: First, we said that `Circle.center` is a container and then later that "field `center` of instance $i1$" is a container. This is both true as they are simply different kinds of containers. Notice how they are related: instance $i1$ is of type `Circle`. There is another container, "instance $i3$'s field `center`", which is related to `Circle.center` in the same way, since $i3$ is also of type `Circle`. This relationship led us to chose the name abstract container for containers in the form `Circle.center`. In contrast, the other containers mentioned above contain information from the dynamic data collection and hence are called dynamic containers.

Another distinction is necessary. Imagine the instructions "`Point p = c1.center;` `System.out.println (p.x);`" were added to the end of the example in Figure 5.5. In this case the `Point` instance $i2$ now has two references, one from `c1.center` and one from `p`. As mentioned earlier an event is created for every dereference, and the subexpression `p.x` of the print statement is a dereference. We know that at run-time p references instance $i2$, since p is an alias of `c1.center`. So $i2$ is the target part of the event. The reference originated from the expression "p", which is not a field, but something different: a local

variable. Hence different *kinds* of containers are required: a field container and a local container.

Now we can go back to the source expression `c1.center.x` that was discussed earlier. At that time we focused only on the dereference of the `Point` instances. In the expression the `Circle` instance $i1$ was dereferenced as well, and in this case the reference was obtained from the local variable `c1`. Hence the event has the target instance $i1$, and the local container "c1". But this is imprecise, as we do not know where the local variable "c1" is located at. The example only has one scope, but there could be many local variables by the name of "c1". When you consider loops, then even the *same* local variable in the source code can refer to a different storage locations at run-time.

We see that in reality being exact requires the use of both container attributes: its kind, and whether it is a dynamic or an abstract one. This makes `Circle.center` an abstract field container, and "instance $i1$'s field `center`" a dynamic field container. These two terms are used frequently, and will be abbreviated later as af-container, and df-container respectively. Similarly `c1` could be the abstract local container[4]. At run-time each local variable is in a particular stack frame so the dynamic local container additionally makes a note of the stack frame id. There are more container kinds which will be introduced later.

Now that we have seen that there are multiple containers in even simple programs, we give a name to the abstract container into which we want to inline a target. If we perform an inlining, then that container is called the inlining destination, or just destination.

So far target was used ambiguously, similar to the ambiguous use of container. Once target referred to a class, and then later to an instance of a class. Thus to be precise target instance is used when referring to an object instance. The abstraction works similar to the way an abstract container is derived from a dynamic container: the target can be derived from the target instance by retrieving the object instance's class.

---

[4]We don't actually use abstract local containers and as such they are not completely defined in this thesis.

Until now dereferences were introduced to create events, but there are additional operations the analysis needs to know about. The inlining process removes the reference to the inlined object, so any Java operation which works on a reference needs to be monitored. One of these operations is the equality operator, "==". In the example we saw that our instance of object inlining only works on 1-to-1 relationships between containers and targets. Since the reference to the target disappears, comparing references is no longer directly possible. It would be possible to compare targets by their containers, but a comparison could also involve a non-inlined target. Then, however, the comparison would involve a target instance and a container instance, which is not allowed by the type system. Although it would be possible to handle this, we chose to simply forbid comparisons of references for simplicity's sake.

It follows that for each comparison of an object instance an event has to get generated. The convenient encoding for such an event is to use the same layout that the dereference introduced: the target and container combination. For a comparison we simply use a special container: the "pseudo" container. Whenever the analysis sees such a container, it knows that something unsafe happens and can eliminate the target from getting inlined.

Now that an event is no longer generated just for dereferences, we adopt the more general term "use". Thus an event is generated for every use of an object reference. There are other reasons that a container and target combination can be disqualified from being used in an inlining. The complete list of reasons is as follows:

- An af-container has more than one target,

- a target instance has several df-containers,

- a target instance is used in a reference comparison, and

- a target instance escapes the analyzed code.

It is important to understand that for object inlining in principle all *unsafe* situations can be *made safe* by using appropriate compensation code. However, using compensation code is often not desirable, because in general inlining in too many places can cause memory bloat[5], and the compensation code can slow the program down. A prominent example of compensation code is "shadow updating". This occurs when one instance is mutated, and the changes are propagated to other instances. For this thesis we focus on situations where compensation code is not required.

5.3 Definitions

There are a number of terms which were introduced by example. Here follow more thorough definitions:

Definition 6 (abstract container). *An abstract container is a Java source expression which returns a value. Although we are only interested in expressions which return an object reference, in principle a container could return any value. It is fully defined in Section 5.4.1.*

Definition 7 (af-container). *An af-container is a combination of a class, C, and one of its fields, f: "C.f". It is an abbreviation for abstract field container, which is one possible kind of abstract containers.*

Definition 8 (df-container). *A df-container is the abbreviation for dynamic field container. It consists of an object instance, i, and one of its fields, f. Let i be of type C. Then where i.f is the df-container, C.f is the corresponding af-container.*

Definition 9 (target). *A target is a class which could be inlined into an abstract container. Its type usually is the same as the container's field's type, but at a minimum has to be compatible with its subtype.*

---

[5]For the implementation presented here memory bloat is not an issue, since only 1-to-1 mappings are considered.

Definition 10 (target instance)**.** *A target instance is an object instance, i, of type T, where T is a target.*

Definition 11 (destination)**.** *A destination is an abstract container into which a target will get inlined. Each abstract container is a possible inlining destination; those which are selected are subsequently called destinations.*

Definition 12 (instance id)**.** *Every class gets a new instance id field, which is initialized by the constructor to be an integer which is unique per run. It is used to define object instance identity. It is used since the official Java API does not include a method to extract object identity[6].*

Definition 13 (stack frame id)**.** *A stack frame id is added to every stack frame as a local variable. This id is initialized by the first instruction with a unique integer for every stack frame. This enables the stack frame to be distinguishable.*

Definition 14 (use)**.** *An object instance is used when a reference to it is read from memory. Arbitrary access to reference is not allowed in Java, so the only cases are: 1) It is dereferenced, 2) it is used in a comparison, or 3) it is passed as a parameter. All three cases are fully defined in Section 5.5.*

Definition 15 (dereference)**.** *In Java, a dereference is denoted by the (infix) dot (".") operator at the source level. In bytecode it is an implicit action in a number of operations, e.g.* invokevirtual *and* getfield*, and requires an object reference to be on the stack. It is fully defined in Section 5.5.*

Definition 16 (event data)**.** *For object inlining the data each event produces are* what was used*, the target instance, and* where *was the reference obtained from, the dynamic container.*

---

[6]The hash code provided by the HotSpot JVM Object implementation returns the object identity. But this is not guaranteed, especially not across JVMs.

The definitions of containers and dereferences are more complex, and have their own sections dedicated to them. The reason for the complexity is that their definition needs to encompass a whole range of Java behavior. The recommended way to add data collection instrumentation is bytecode rewriting, as discussed in Section 3.2. But the large number of bytecode instructions makes a compact definition impossible.

Instead of basing our definitions on the bytecode, or the source code[7], we chose an intermediate representation provided by the toolkit we chose for the bytecode rewriting, Soot [28]. The representation we will use for a thorough definition is Jimple, a minimalistic intermediate representation used by Soot, which covers all of Java's bytecode instructions. Understanding Sections 5.4 and 5.5 requires some knowledge of Jimple. We provide an overview of Jimple, sufficient for the present purposes, in Appendix D.1.1. For more detail the reader is referred to [28].

5.4 Containers

A container is a general term for different kinds of locations where object instances can be loaded from. There are dynamic and abstract containers. The former are present at run-time only, and are logged by the instrumentation. Abstract containers are what is used by the transformation. There exists a complete many-to-one relationship from dynamic containers to abstract containers.

5.4.1 Definition

In Jimple, any dereference will target a *local*, so the dynamic container is the location where the *local* received its value from.

---

[7]The structure of Java source code makes it similarly complex as the bytecode instructions.

Definition 17 (dynamic container)**.** *A dynamic container is defined by its*

1. Kind*; which can be one of* local*,* static*,* Instance*,* argument*,* return*, and* foreign*.*

2. Name*; the name used to refer to the container.*

3. Id*; a unique identifier for this* kind *of container with the given* name*.*

4. Attribute*; the default is no attribute, or it can take on* array *or* this*.*

A abstract container exists for every kind of dynamic container. In this thesis, only af-containers are used, which were already defined in Definition 6. Further abstract containers could be defined similarly, as the static version of their corresponding dynamic container.

5.4.2   Container Details

A brief description of each dynamic container kind, and how it affects the other properties, follows.

The dynamic **local** container is a local variable; its *name* is the name given to the local variable at the source level. Since the variable name is only valid in the current stack frame, the *id* is the stack frame id (see Definition 13).

There are two subtypes of the dynamic local container: An dynamic argument container is a parameter to the current method. The implicit $0th$ parameter, the receiver object, is represented by this container as well. An *argument* is exactly like a *local variable*, except that it has a predefined value: the value passed by the function call. The *name* and *id* are collected identically. For the receiver argument, the attribute *this* is set.

A dynamic **return** container is a return value from a function call. It also behaves like a local variable, but it does not have a name. Since every return value is a unique dynamic

container , a unique name is assigned to it[8]. Its *id* is again the stack frame id.

A dynamic **static** container is a static (class) field. Its *name* is the name given at the source level . Its *id* is the class it is defined in.

A dynamic **field** container is a field of an object instance[9]. Its *name* is the name given at the source level[8]. Its *id* is the instance id of the instance which contains the field.

A dynamic **foreign** container is a "pseudo" container which marks arbitrary accesses that can occur in unmonitored code. See Section 5.5.3 for more details. Its *name* is set to the field the target was assigned to, or the method which was called. Its *id* is set to the class the field or method is a part of.

The *attribute* is universal to all dynamic container kinds. Currently it only marks a container as being an *array*, or as being the special *this* reference.

For the remainder of this document, entries in the log will be written as tuples of the form (*target id*, *container kind*, *container id*, *container name*, *container attribute*) .

5.4.3   Completeness

Since a dynamic container can be the source of any object instance on the stack, we need to make sure that our definition of dynamic container covers all possible sources of references on the stack. Defining completeness in terms of the source language is tedious because there are many types of expression to consider. At the bytecode level the situation is similar; there are a large number of bytecode instructions which load values onto the stack and which therefore would need to be examined to determine the completeness of the definition in Section 5.4.1. An alternative is offered by the Jimple IR introduced by the

---

[8]Soot generates names already, so we reuse them.

[9]The implementation calls a *field* container still an *instance* container. We have renamed it in this text for added clarity, but not yet updated the source code.

Soot framework.

Referring to the Jimple grammar in Section D.1.2, a dynamic container has to be a *local* (see `invokeExpr`). To know what kind of dynamic container a local is, we track what value was assigned to the local. Looking at the grammar, a value is assigned to a local either through an `identityStmt`, or through the `assignStmt` → `local = rvalue` production.

- `identityStmt`: Both `@this` and `@parametern` are considered dynamic argument containers. We do not care about `@exception` parameters, because these are not inlining candidates anyway. For `@this` statements, the *this* attribute is set.

- `rvalue` → `concreteRef` → `field`: A **static** dynamic container[10] is used.

- `rvalue` → `concreteRef` → `local.field`: An **instance** dynamic container is used.

- `rvalue` → `concreteRef` → `local.field`: A **copy** of *local* is created, add the **array** attribute set.

- `imm` → `local`: Dependent on the name of *local*:

  1. If *local* just a named stack location (Soot uses $ to mark these names), then [**copy** of *local*].

  2. Else *local* is an actual local variable, and **local** dynamic container is used.

- `expr` → `invokeExpr` → …: A **return** dynamic container is used.

- `expr` → `new` : A **return** dynamic container is used.

- `expr` → `(type) imm` → `(type) local`: A **copy** of *local* is created[11].

---

[10]The reference `field` is typed; and thus refers to both the class and a specific field of the class.

[11]The production `imm` → `constant` is of no interest, because a constant cannot be an object instance

Note that for `expr` all derivations which are not listed here cannot yield an object type, and are ignored. All ways to assign an object instance to a local are described above. This way we can be assured that all possible dynamic `containers` are covered.

5.5 Event Generation

Now that a `container` is assigned to every Jimple local variable, the `containers` are used to generate the logging code[12]. This code is inserted *before* every use. The reason for adding the logging statement before the use is simple: since method calls themselves might generate log entries, logging before the call is a simple way to keep a natural ordering of log entries.

The most common use is a dereference. At the level of Jimple, as well as the byte-code level, a dereference is not an explicit operator (in contrast to the "." at the source code level), but rather a dereference is performed at the same time the referenced object instance is used for a field reference, or a method invocation, etc. The Jimple IR is again used to specify when logging is required. Not only does this provide us with a compact description, but it is also used verbatim as a specification for the implementation.

5.5.1 Method Calls

In the grammar in Section D.1.2, method calls are grouped under a second-level `invokeStmt` label. The kind of method call is determined by the `invokeExpr`, which can either be a `specialinvoke` (constructor call), `interfaceinvoke`, `virtualinvoke` (both "regular" method calls), or a `staticinvoke` (static method call). A static method call is not connected to an object instance, while the remaining ones are, so Soot internally

---

[12]They are not used to generate log entries directly; every log entry is generated dynamically during the program's run.

groups `specialinvoke`, `interfaceinvoke`, and `virtualinvoke` as an instance invoke expression (see the `InstanceInvokeExpr` interface in Soot's API documentation).

Constructor calls are very restricted by the Java Language Specification. They have to be executed right after allocating the object, or as the first call if it is a `super` constructor call. So in case of a `super` constructor call, it is not possible to create logging code *before* the call[13]. Since constructor calls would only create a dereference of the receiver object, they are irrelevant for the safety analysis (the irrelevance is explicit in the algorithm in Section 5.6). We therefore choose not to create logging code for constructor calls.

5.5.2 Field Accesses

Field accesses are either a write access, deriving `assignStmt` $\rightarrow$ `local.field = imm;` or a read access, deriving `assignStmt` $\rightarrow$ `local = rvalue; rvalue` $\rightarrow$ `concreteRef; concreteRef` $\rightarrow$ `local.field`. Both will generate logging code. Note that because of the semantics of Jimple, it is not possible to have a direct field copy in the form

```
local.field = local'.field
```

Such an operation would in practice be represented in Jimple by two statements:

```
tmp = local'.field
local.field = tmp
```

---

[13]It *is* possible to generate code before the constructor call, but it would not pass the JVM verifier.

5.5.3 Foreign Method Calls

Since dereferences are only seen in instrumented code, the possibility exists that an object instance, which the instrumentation monitors, is passed on to some external library. We have no way of knowing how the instance is accessed in that case, so it should not be inlined. Therefore when we see a method invocation where the base class is not under our control, we scan all the parameters, and for any object instances that are under our control, we generate logging code. The container in this case is *foreign*, which marks that arbitrary accesses to the object in question could have happened. As mentioned before, the object instance will not be inlined if it was a target instance of a *foreign* use.

5.5.4 Pointer Comparisons

Java does not use the term pointer. Instead within the source language it is called a *reference*. Java does not allow pointer arithmetic, but *references* can still get compared. To our knowledge, this is the only operation available on *references* aside from passing them to functions, and dereferencing them.

Since object inlining removes the reference, such a comparison is not easily possible any more. As mentioned before, if two inlined target instances are compared, then their container can be used for the comparison instead. But comparisons between inlined and non-inlined instances are not possible this way. The analysis could take this into account, but since there would still be comparisons which are not possible, we chose to not allow them completely. Therefore we have to log any comparison operation, so that the analysis can disqualify any instances which were involved in the comparison operation. The container used for comparisons is also *foreign*.

5.5.5 Examples

The introduction to this section already had some basic examples showing when log entries are generated, but some more basic examples will be provided here. We discuss them using the proper terminology introduced above. Longer examples, on actual Java programs, as well as on Jimple code, can be found in Appendix B.1.

Let x be some object instance and $id(x)$ be the function that maps object references x to their unique id. For a local variable it retrieves the stack frame id and for an instance the instance id.

In the tuples container is abbreviated as c to make the records more readable.

1. **Local Variables.**

   For a local variable a, the statement

   > a.b

   creates

   (*target:* id(a), *c-kind:* local, *c-id:* id(stack frame), *c-name:* "a", *c-attrib:* none)

   The target here is a, because it is on the left of the dot dereference operator. The call $id(a)$ retrieves the instance id of the instance which a references. The call $id(\text{stack frame})$ refers to the id which is added to each stack frame (see Section 5.4.2).

   Now we are able to uniquely identify the container "local variable a": We know what stack frame it is located in, its name, and what instance it references.

2. **Static Fields.**

   For a static field *bar*, the statement

   > Foo.bar

   creates **no event** because it is **not a dereference** in Java. Rather, it is a special instruction which retrieves the static field.

On the other hand, the statement

Foo.bar.baz

creates

(*target:* id(bar), *c-kind:* static, *c-id:* Foo, *c-name:* "bar", *c-attrib:* none)

The call to *id* for the target is always the same. Since there can only be one class named *Foo*, and classes have unique field names, the container is uniquely identified[14].

3. **Instance Variables.**

   Instance variables are more complicated, because one source statement often contains several dereferences. Additionally, implicit dereferences take place since there is no requirement to use the special variable *this* to access fields of the current object instance. Let *f* be a field of an object instance.

   This implicit dereferencing at the source level is explicit in the byte code. In these examples we will show the dereference explicitly, so at the source level the following translation is done:

   f$\quad\Rightarrow\quad$**this**.f

   This statement creates

   (*target:* id(this), *c-kind:* arg, *c-id:* id(stack frame), *c-name:* "this", *c-attrib:* this)

   Since *this* is the implicit *0th* parameter to the method, it is of argument kind and not the dereference of an instance variable. As previously mentioned, arguments are treated exactly like local variables otherwise.

   Now consider compound statements:

   this.f.x$\qquad$(same as: f.x on the source level)

---

[14]The presence of multiple class loaders does allow two distinct classes to be in memory at the same time. This is not currently supported, but would be possibly to add by remembering the class' loader.

This statement creates two events:

(*target:* id(this), *c-kind:* arg, *c-id:* id(stack frame), *c-name:* "this", *c-attrib:* this)

(*target:* id(f), *c-kind:* instance, *c-id:* id(this), *c-name:* "f", *c-attrib:* none)

The second event is the dereference of an instance variable. $id(f)$ is retrieved as mentioned before. The container of f is the current receiver object, *this*, and its identifier is recorded. The name of the instance field is f and appears verbatim in the record. This record uniquely identifies a field container: we know which instance, and what field name contained the reference to the target.

Another compound statement:

this . f . g . y   (same as: f . g . y)

Creates three events:

(*target:* id(this), *c-kind:* arg, *c-id:* id(stack frame), *c-name:* "this", *c-attrib:* this)

(*target:* id(f), *c-kind:* instance, *c-id:* id(this), *c-name:* "f", *c-attrib:* none)

(*target:* id(g), *c-kind:* instance, *c-id:* id(f), *c-name:* "g", *c-attrib:* none)

The way the dereference events are recorded is exactly as described above, but a pattern becomes apparent: in a chain of dereferences, the current container is the target of the previous dereference. This chain has to start with a local variable, argument variable, or static field, while the remaining dereferences have to be of instance kind.

Note that the special reference *this* can be reconstructed from the log as any dynamic argument container with attribute *this*.

5.6 Analysis

The analysis works mainly with field containers. In Java, most objects are allocated on the heap, making field containers the case where the optimization is most likely to be profitable. Additionally the transformation code needs to be written for every container type, so a restriction on just one container kind was necessary due to time constraints. The abstract container was defined earlier, which is important for the analysis. Take for example the following df-container:

container(*kind:* instance, *id:* 5, *name:* "f", *attrib:* none)

Assume that type(5)=Foo. Then the df-container is abstracted as Foo.f. The abstract container consists of static and not dynamic data (the specific container instance). Similar abstract containers can be defined for other dynamic containers, which is a topic of more research.

The analysis algorithm's goal is to check all types (targets) and abstract container combinations simultaneously for inlining safety. It is easy to decide if one instance can be inlined into a specific target: it has to have only one container. It follows that an instance is unsafe to inline when it has more than one container. Safe instances of type T contribute their abstract containers as candidates to the type T. These abstract containers are possible inlining targets for T. Unsafe instances eliminate abstract containers as targets for a T.

Take Figure 5.12 as an example: instance id 4 is dereferenced exactly once, with the log entry being displayed on line 10. It therefore is safe to inline it into its container. Its container, instance 1, is of type A. Instance id 5, however, is accessed twice: The record on line 16 has a container with id 2, and the record on line 20 has a container with id 3. Instance id 5 is unsafe to inline. Aggregating the data for type B, we see that it has an instance, id 4, with an abstract container $A.b$. The unsafe instance, id 5, also has the

```
1   class  A { B b; }
2   class  B {}
3
4   A a1 = new A(); // instance  id 1
5   A a2 = new A(); // instance  id 2
6   A a3 = new A(); // instance  id 4
7
8   a1.b = new B(); // instance  id 4
9   a1.b.foo();
10  // ( target =4, kind=instance,  containerid =1, name=b)
11
12  B b = new B(); // instance  id 5
13
14  a2.b = b;
15  a2.b.foo();
16  // ( target =5, kind=instance,  containerid =2, name=b)
17
18  a3.b = b;
19  a3.b.foo();
20  // ( target =5, kind=instance,  containerid =3, name=b)
```

Figure 5.12: Example of the dynamic container of an unsafe instance eliminating the abstract container gathered from a safe instance.

abstract container *A.b*, which eliminates the candidate introduced by instance id 4. We conclude that it is not safe to inline *B*.

Containers themselves have to be checked as well[15], so we maintain a list of safe types per container. If a container has a list of more than one type which is safe, then it is not considered an inlining target.

What state does the analysis need to keep for every instance? When only one container has been encountered, then the instance is safe to inline, but can later become unsafe if

---

[15]Mainly because of sub-typing. We do not want to inline a type into a container if the container houses different subtypes.

another container is encountered. Thus for safe containers, we need to save information on exactly one container. Unsafe instances on the other hand have multiple containers associated with them. As described above, the containers of unsafe instances are only used in abstracted format, which is a little more compact to represent.

Formally, let A_i be the analysis result of the target instance i. It can be in one of three states:

1. *None*: no container has been seen yet. This is the initial state.

2. *One*: one container has been encountered, its details are saved. The instance is safe to inline into the given container.

3. *Many*: the instance has been accessed from more than one container. The instance has become unsafe. The abstracted containers are added to the type of i.

Let L be the log containing records in the format (t,k,c,n,a), where t is the target, k is the container kind, c is the container id, n is the container variable name, and a is the container attribute. The tuple (k,c,n,a) refers just to the container.

The main algorithm is Algorithm 5.1. The first pass is over the entire log, and calls *mergeInstance* (Algorithm 5.2), which uses a little helper function, *invalidateContainer* (Algorithm 5.3). The second pass loops through all instances discovered, and calls *validateContainer* (Algorithm 5.4), on each instance. The final result can be found in M, which maps from types to a set of inlining targets.

Let n be the number of log records generated at run time. The first pass looks at all of them, which makes it $O(n)$. Let m be the number of object instances created during the run. The second pass looks at all instances, which makes it $O(m)$. The total run-time then is $O(n+m)$, and since the number of instances is bound by the number of log records, we can simplify this to $O(n)$ over all.

Note that since *n* can be rather large, the analysis can in practice still seem slow. Partially this is because it is I/O bound.

Algorithm 5.1 Find the set of target containers for every type.

Require: *A*, initialized to *Empty* for every instance

Ensure: *M*, a map from type to a set of abstract containers; *C* a map from abstract container to types

{Check safety for every instance}

for (t, k, c, n, a) in *L* **do**

   type ← lookupType (t)

   A_t ← mergeInstance(A_t, type, (k, c, n, a))

end for

{Abstract from instances to types}

for A_t in *A* **do**

   type ← lookupType (t)

   k ← validateContainer (type, A_t) {Might return return nothing if not safe}

   M(type) ← M(type) ∪k

   $C(k)$ ← $C(k)$∪ type

end for

{Check that every container only has one inlining candidate}

for k in *C* **do**

   T ← $C(k)$

   if $|T| > 1$ **then**

     for t in T **do**

       M(type) ← M(type) $-k$

     end for

   end if

end for

Algorithm 5.2 Function `mergeInstance`

Require: *state*, the current instance analysis state; type, the type of the instance, $C = (k,c,n,a)$ the container to merge with

Ensure: the new state for the instance

  if $k = Argument$ **and** $a = this$ **then**

    return *state* {We ignore accesses through *this*}

  else if *state* is *None* **then**

    return *One(c)*

  else if *state* is *One(C′)* **then**

    $(k,c,n,a) \leftarrow C$

    $(k',c',n',a') \leftarrow C'$

    if $k = k'$**and**$c = c'$**and**$n = n'$**and**$a = a'$ **then**

      return *One(c)* {The same container remains.}

    else

      invalidateContainer (type, C)

      invalidateContainer (type, C')

      return *Many* {The container C' did not match the container C}

    end if

  else

    invalidateContainer (type, C)

    return *state* {*state* is *Many* already.}

  end if

Algorithm 5.3 Function `invalidateContainer`

Require: type, a type; $C = (k,c,n,a)$ a container

Ensure: I_{type}, a map from type to a set of invalid abstract containers.

  ctype $\leftarrow$ lookupType (c)

  $I_{\text{type}} = I_{\text{type}} \cup (ctype,n)$ {Add the abstract container to the set for the given type}

Algorithm 5.4 Function `validateContainer`

Require: type, a type; *a*, the instance analysis result

Ensure: Zero or one abstract containers which are safe for the given type

  if *a* is *None* **then**

    return *One*(*c*)

  else if *a* is *One*(*C*) **then**

    $(k, c, n, a) \leftarrow C$

    ctype $\leftarrow$ lookupType(c)

    if $(ctype, n) \in I_{\text{type}}$ **then**

      return *None*

    else

      return (ctype, *n*) {A valid container was found}

    end if

  else

    return *None* {*a* was *Many*, which results in no action at this stage.}

  end if

5.7 Framework Instantiation

Now that we have introduced object inlining and in detail what data needs to be collected, we will highlight how the optimization was implemented in the framework. If the previous sections seemed slightly disconnected, then this section is supposed to bring it all together. Note that not all areas of the framework need to be described in relation to object inlining since some are simply used as the framework provides them.

Data management, as described in Section 3.4, is an example of an area which is almost completely managed by the framework. The user only has to clean up the data files by hand. This feature could be provided by the framework, but was not implemented as

a fail-safe mechanism: Run data usually takes a long time to collect, and we do not want to accidentally delete it. Since the available and valid run information is managed by the framework, forgotten data files are not cluttering the list of runs.

5.7.1   Data Collection & Storage

Soot, which is often known as a static analysis toolkit, can also be used as a bytecode rewriter. It is used to add the necessary instrumentation, because some static analysis is useful to add the correct dynamic data collection code. The prime example is the identification of local variables. Soot disambiguates local variables from stack locations, and allows easy access to locals through its Jimple IR. Another advantage of Soot is that it allows modularization of instrumentation components (see Appendix B.2 for some more details).

We settled on using the flat file approach described in Section 3.3, which was easy to implement because only two types of events exist:

1. A dereference, where the target and the dynamic container are stored. The container fields were described in Section 5.4.1. Note that the meanings of the container fields are dependent on the container kind, while the field types are uniform.

2. An object creation, which is needed to look up types of instances based on their instance id. It stores (instance id, "type in string form") tuples.

Those data are required for the safety analysis, and can also be used for a ranking heuristic of the opportunities.

The workflow for the data collection is:

- Apply the instrumentation by binary rewriting using Soot.
  Some setup is required to ensure that all the dependencies are available. Since Soot

analyzes all classes, it has to have access to all dependencies. Essentially dead code, or advanced usage scenarios, result in these additional dependencies over the default run-time ones.

- Run the instrumented program on the unit tests to generate the log.
  Expect this step to take a while. We have observed run-times varying from a couple of seconds to several hours.

5.7.2 Analysis & Programmer Interaction

For the analysis, the log file is parsed and the stream of events is analyzed by the code described in Section 5.6. Afterwards we have a list of opportunities that analysis determined to be safe.

At this point the ranking heuristic comes into play. As already discussed in Section 2.4, only a limited number of them should be presented to the programmer. The goal of the heuristic is to rank the opportunities in such a way that the programmer can consider the most profitable ones.

There are two aspects of object inlining to consider for the heuristic: speed-up and memory usage improvements. Due to the 1-to-1 constraint of the inlining algorithm, the memory usage aspect is the less interesting, and easy to determine. Remember that an inlining removes a object reference, and an object header. The exact size of these depends on the JVM and system in use, but let us just assume that references are 64 bits, and object headers 128 bits, for a sum of 172 bits. So for every target that is inlined into a container, we save 172 bits per container instance. Therefore one factor in the heuristic is how many instances of the container were created.

The potential to speed up program execution is much more difficult to gauge. First let us consider allocation costs. Since the inlined target is now allocated together with

the container, this action is saved. Similarly a collection action is also saved. These two factors also depend on the number of container instances, but are difficult to quantify. In particular on modern JVMs, e.g., Sun's HotSpot JVM, concurrent garbage collection is performed, and the impact of this change alone is not easily measured. As a conservative measure we can assume that the impact on the GC run-time is negligible unless the number of instances was extremely large. The exact threshold for this can again be configurable by the user, but as a default setting we propose 25% of all allocated objects during the program run. When this threshold is exceeded, the opportunity will be shown to the programmer. The benefit of using a percentage threshold is that we automatically limit the number of opportunities that enter the final list through this criterion.

The number of eliminated dereferences plays an important role. Since modern computers are often limited by the memory wall [36], removing memory access has a high potential to speed up execution. Consider the uses introduced earlier. By far dereferences during method calls, and field access, are the most common cases. This dereference is eliminated, saving the program a memory access. The exact impact of this is also difficult to measure, since caching effects are highly unpredictable: if the reference that the inlining eliminated happened to be in cache before, then the improvement in run-time will be small. If the reference happened to be in a register, then the improvement will be virtually non-existent.

A heuristic could try to be more context sensitive, and for example consider how often this reference was used in a block. The added precision of such an analysis is questionable though, since it is not possible to factor in any JIT optimizations. Overall it seems like the simple heuristic, to simply count the uses of the target provides the best cost/benefit ratio.

The exact weight between the scores of memory usage reduction, allocator and GC run-time improvements, and eliminated dereferences is still an open question.

Before the programmer gets involved, the support information needs to be gathered.

For object inlining a number of facts are interesting:

- What other classes have fields with the target type?

- What methods accept a target as parameter?

- How often do locals of the target type appear in the sources?

This information helps the programmer to get a quick overview of how the target is used in the program.

Once the ranking is determined, and the support information gathered, the question for the programmer is very simple: "Should the target be inlined into abstract container?" Since inlining is essentially a conversion from reference to value semantics, we believe most programmers will be able to understand and quickly answer the question.

Caching these answers is simple: the optimization just asks the programmer if the previous answers still hold, and possibly presents the list of inlining opportunities which the programmer has approved. Presenting the list becomes a hindrance if it grows too long. But it would remind the programmer about his previous choices and we think that a change in assumptions would be noticed quickly once the list of existing inlinings is visible.

Note that not all of the ideas presented in this section are implemented. We envision a simple GUI, which displays the ranked opportunities. The programmer then can request further information by clicking on an opportunity, after which he is presented with an overview of the available support information. A further click on these should display the corresponding details. The support information should be gathered beforehand, and not on demand, as otherwise interactive delays might discourage the programmer from selecting an opportunity simply because of the wait involved.

5.7.3 Transformation

We chose to use source code rewriting for applying the transformation for the reasons outlined in Section 3.7. There are not many libraries available for transforming Java source code, possibly because it is a relatively complicated process. Yet refactoring, a feature which has been more automated in IDEs in recent years, requires automatic transformation of source code. Code refactoring is implemented in most major Java IDEs, for instance, Eclipse's refactoring code is available, but it is not well suited for use outside Eclipse. The main problem is that documentation is almost non-existent. We found a library called Recoder[16] [15]. It offers an API that is intuitive to use, some documentation[17], and is well maintained at the time of writing. Some more details on Recoder, as well as general Recoder comments, can be found in Appendix D.2.

The Java programming language is, as are most higher level programming languages, very feature rich[18]. This means that any undertaking in source code rewriting is not an easy task. So while the author wrote the initial Recoder code, it was later handed off to his colleague Devin Coughlin, who then extended it to handle many of the intricacies which occur in real world source code.

The main idea for the transformation is to take all members of the target, and replace the field of the destination with them (Some basic examples of this can be found in Section 5.2). Name conflicts are avoided by combining the destination field name with the target member's name. While conflicts are still possible, they are unlikely and could be resolved by using additional counters. This scheme is chosen to make the output of the transformation easy to understand; in principle, any name can be chosen as long as the references are fixed up accordingly.

---

[16]While the library seems to be officially called "RECODER", its website also refers to it as Recoder.

[17]Some documentation is in contrast to virtually none that we found for other libraries.

[18]Java is a complex language despite simplifying several aspects like stack/heap allocation, and pointers compared to other languages in its league.

After the members are migrated, all references through what was the destination before now have to be rewritten to use inlined members instead. Special care has to be taken with constructors, as they need to be converted to a regular function. Whenever the constructor was called before, this regular function has to be called instead. It takes over initialization, since the allocation step does not exist anymore.

5.8   Related Work

Object inlining has been the subject of research for some time. Here we present some related work. A direct comparison is not really possible, as discussed in Chapter 4, but this list is provided for reference. Also note that we believe object inlining as presented in this thesis could be pushed much further, as outlined in Chapter 6.

Dolby and Chien's work [12] has a similar safety condition. They only allow "one-to-one" fields to be inlined. Their definition of object inlining seems to be expressed in terms of dynamic analysis, while the implementation in their Concert compiler, to our knowledge, relies on a (static) adaptive iterative data flow analysis.

Laud's object inlining [20] improves upon Dolby's work by additional analysis which avoids some dynamic dispatching that would previously have been necessary. The cases which can be inlined are similar to Dolby's.

The object combining work of Veldema et al. [30] is more general as it allows the combination of functionally unrelated objects. Their focus is more on combined (de)allocation than on removing dereferences, which they nonetheless perform in a separate pass of their optimization.

Lhoták and Hendren's evaluation [21] is quite similar to our implementation. They used dynamic analysis to produce a trace, which is then analyzed to identify inlining opportunities. However, their solution does not seem as automated as our approach.

Budimlić et al. identified object inlining as one way to improve the compilation of scientific applications [6]. They are using "copy folding" [5] to eliminate aliases to be able to inline more objects with only one alias. The remaining safety conditions, reference comparisons, and passing to unmonitored methods [7], are similar to the ones we currently employ.

Wimmer and Mössenböck's automatic object inlining [35] uses dynamic analysis and online transformations. It is implemented within the JVM, and they use profiling information to get inlining candidates, which then have to fulfill two safety preconditions: The container and the target must be allocated together, and the target must not get overwritten. Run-time monitoring is used to detect safety violations, which reverts the transformation. The price of their fully automated approach is the overhead of the run-time monitoring, and of the on-line transformations.

5.9 Results

We applied our object inlining to most of the programs of the DaCapo suite, version 2006-10-MR2 [3]. As Table 5.1 shows, the object inlining analysis was always able to find some opportunities, although for some programs none seemed profitable enough to pursue. Table 5.2 lists the results of applying a selection of the opportunities we discovered[19]. It shows that sometimes when the heuristic said that the opportunity should be worthwhile, we did not observe a noticeable improvement in run-time. None of these results show any significant speed-up.

---

[19]The complexities of source code rewriting prevented us from applying more of the opportunities we found.

Program	# of Ops.	Best Op. %	Sum of %s	Sum of top 5 %s
antlr[1]	61	33%	77%	71%
bloat[1]	22	0%	0%	0%
chart[1]	15	2%	12%	8%
fop[1]	62	0%	1%	1%
pmd	14	7%	11%	11%
luindex[1]	20	12%	19%	16%
lusearch[1]	22	3%	10%	9%
jython[1]	35	14%	35%	35%

Table 5.1: List of inlining opportunities found.
Column (1) lists the program name; column (2) lists the number of opportunities, (3) lists the percentage of dereferences of the *best* opportunity, (4) lists the percentage of dereferences of all opportunities combined, and (5) lists the percentage of dereferences of the ten best opportunities. All percentages are based on the total number of monitored dereferences performed in the program.

[1] The data for the analysis was obtained from regular representative runs, not unit tests.

5.10   Discussion

The results are consistent with the lack of sophistication of the inlining analysis. The important fact to take away from these results is that there never was a case when an opportunity, which the analysis found to be safe, was in practice unsafe.

Object inlining is an optimization whose effectiveness on any given program will

Program	# of Ops. used	Combined %	Error	Speed-up
antlr	2	55%	N	2.7 %
luindex	1	2%	N	1.4 %

Table 5.2: List of achieved inlining speed-ups.
Column (1) lists the program name; column (2) lists the number of opportunities which were applied, (3) lists the percentage of dereferences of all applied opportunities, (4) if an error occurred while running the transformed program, (5) lists the speed-up of the transformed program. See Appendix B.4 for the details.

strongly depend on the way that program is written. It is common knowledge that Java does not handle many small objects particularly well. This makes it likely that most of the programs we benchmarked were written in such a way to avoid such situations. In this case object inlining never has the chance to recover overhead cost because by design not too many small objects were created. It would be interesting how Java best practices would change *if* the object inlining optimization was more widely employed.

What is surprising is the inaccuracy of the selected performance heuristic, the number of dereferences of the container. The results for e.g. the antlr program show that there was no significant speed-up despite *very* promising heuristics. One possible explanation for this is the interaction of the transformation with other optimizations performed within the JVM. A more detailed investigation would have been difficult to realize without detailed knowledge of the JIT system and its internals, something which we do not posses and can be difficult to acquire[20].

It has also been shown that even small variations in an environment, particularly the memory layout, can have unexpected effects on performance[24]. We did observe rather large variations in benchmark performance, despite DaCapo being a standard benchmark suite. Further work is needed to determine the exact cause.

---

[20]Since Sun's HotSpot JVM was closed source for a long time, its internals were not known in detail. This might have recently changed with the OpenJVM effort, however.

Chapter 6

Future Work

Object inlining is only one example instance of this framework. There are a number of other optimizations and ideas which fit well into the outline presented in Chapter 3.

Improved Object Inlining. As mentioned before, the safety analysis and the transformation which were implemented for this thesis do not push the state of the art in object inlining itself. We believe that the approach presented in Chapter 5 has the potential to be an improvement over the state of the art. The next step which we propose is to make the analysis sensitive to time. The idea is to allow several containers for a target, but to avoid the need for write-propagation. This can be achieved by restricting multiple containers to only be allowed when the target is read from, but not modified any more.

An outline of the new safety condition would be: For every write event, ensure that the target only has one container. For every read event, if there exists a write event to the target at a later point in time, ensure that it only has one container. If no further write events exist, then multiple containers are allowed. Note that this analysis does require some compensation code: when multiple containers appear, there needs to be code inserted which copies the content from one to the other on assignment.

Additionally, as mentioned in Section 5.10, we were not fully able to explain our results. Further work explaining the performance variations, or the failure of the heuristic would be interesting to answer. This is likely to be a time consuming endeavor, however.

Include Libraries in Whole Program Optimizations. Our implementation of object inlining is limited to work on the application classes. This restriction is largely imposed by the reliance on source code rewriting. While it is possible to also find the sources of the libraries, and transform these in unison, the time required for such a setup is prohibitive. Java programs traditionally bundle libraries instead of using system wide copies as is common in languages like C. If binary rewriting is used, then a transformation involving not only application classes, but also library classes, could be easily implemented. The way the application is distributed would remain the same.

Standard library classes are more difficult to handle, since the JVM relies on some of them for internal use. With an appropriate black list, standard library classes could be included as well, which would lead to a tightly optimized software stack.

Note that including library classes might not always be an option because programmer verification is required. The programmer does not know the internals of the libraries, so making qualified statements is not possible. Further research would need to be done if the optimizations could only be applied to the boundary of application code and library classes, or also to the library internals.

Dead Code. Finding "dead code" is a small optimization which reduces memory usage, and can provide a speed-up through better cache utilization. It is an example how the event logs can be reused. Consider again the events logged by the object inlining optimization (see Section 5.7.1): Since dereferences and other uses are logged, some checks are available through a very simply analysis: Does some class get instantiated, but its instances are never used? Similar checks could be done for every field of every class. Unused fields, classes or object instances could be removed from the program. With some

enhanced instrumentation it would also be possibly to check if all basic blocks in the code are reached.

Just like object inlining, this transformation is also best performed off-line, since on-line it is difficult to ensure that said piece of code will in the future never be reached. Similarly, programmer confirmation is important, since for example dynamically loaded code could cause the conditions to change. And again there are benefits for dead code removal to be an optimization and not a refactoring, since the dead code is often in place because it might, or will, be used again, and should not be permanently removed.

Unit Test Evaluation. A venue which our research group is currently exploring is to compare the semantics of the code executed by unit tests to the code executed by real runs. The idea is that when different behavior is detected, to warn the programmer that the unit tests are not adequate.

Although possible, there does not seem to be much benefit from running such a comparison on-line, since code changes only need to verified once, and not every time the program is run. And the programmer should be able to easily confirm if a miss-match is really a shortcoming of the unit test, or simply behavior which is not tested for some reason.

The kinds of semantics which should be compared are subject of ongoing research.

Memoization. Memoization is the automatic caching of function return values based on their parameters, so that previous computations are reused, instead of rerun. We explored this optimization in the past, see Section C.2, and were at the time not successful at exploiting it. The question of side effects is something which is always difficult to define in non-functional programming languages.

In this framework, however, we can use the programmer's knowledge to provide this fact, which would otherwise be very difficult to determine automatically. The data collection can be used to profile the program, and use some heuristics to find *functional looking*

functions, which then the programmer can confirm. Again it is useful to keep the implementation of memoization separate from the program logic, since the function's behavior can easily change in a way that makes it be unsuitable for memoization. If this happens, then the heuristic is also useful if the programmer forgets to invalidate his previous selection.

Trace harvesting. Another idea would be to look at event logs, and try to use machine learning to gather *facts* about its behavior. These *facts* could then be used to compare the unit tests with the real runs. This is in some way the more general version from what the unit test evaluation idea presented before. Whether this is possible, or even useful, is an open question.

Chapter 7

Conclusion

Using dynamic analysis for off-line optimizations is a novel approach. The framework presented here is a set of code and guidelines on how to implement such optimizations. The idea to use unit tests as a de facto specification of behavior is also a new idea, which mitigates one of the main questions which always gets asked when dynamic programming is involved: how a representative run is chosen. Since even unit tests do not guarantee that the dynamic analysis is able to observe all program states, and dynamic analysis does not resolve all analysis difficulties introduced by dynamic features of modern programming languages, additional information needs to be provided by the programmer. Human involvement in this framework is kept at a minimum to ensure an effortless workflow. This is achieved by heuristics to sort opportunities, and by designing the interaction in a way that its input can be often reused.

As an example instance of the framework object inlining was implemented. The inlining algorithm itself is not particularly advanced, but its workings are described in detail so that the reader can understand that it is correct, and how it fits into the framework. As such, the optimization as implemented does not provide much of a speed-up, but is very illustrative of the process.

Chapter 7. Conclusion

Only time will tell if this framework is accepted. We believe it offers exciting new possibilities for implementing optimizations which were either not possible before, or were difficult to implement because they had to work around information which was essentially unavailable.

Appendix A

Framework Implementation Notes

This chapter documents the details of how the framework is used in practice.

A.1 Instrumentation and Data Collection

The first step for the instrumentation is to identify all classes that are considered part of the application. Often this can be done simply by looking at all the class files which are built by the application, since dependency libraries are usually present as jar files. Among this list of classes the *main* class has to be identified. The main class is the one passed to Java when the program is started, or the one entered in the jar MANIFEST. Soot simply interprets the first class handed to it as the main class, so the list of classes just must be reordered.

The next step is to compile the list of dependencies. Usually Java programs include dependencies in a directory as jar files. These will be required for Soot to be able to resolve all references. Additionally the program might have optional dependencies, which also need to be resolved for Soot to work correctly. The list of all dependencies, combined

with the location of the application classes, makes up the classpath that is needed for running the instrumentation.

The remainder of the settings for the instrumentation are set as environment variables. The framework needs to know what kind of instrumentation it should add, the so-called `COLLECTION`. The kind of logger which is used by default is set using the `EVENTLOG` variable. The rest of the metadata are the application name (`APP_NAME`) and the application version (`APP_VERSION`). Now the instrumentation can be run, using the *soot-eventlog* script, passing the list of classes as arguments.

The output of the instrumentation is a number of rewritten class files. These should be substituted in the program's environment (support files, etc) and then run to collect the data. Runtime requirements are mainly additional packages on the class path; those required by the instrumentation code itself. In practice the framework provides a wrapper script for the *java* command, called *javasimplelog*. The wrapper is useful because it allows changing the run-time requirements of the framework without having to adjust the already prepared project. E.g., activating the instrumentation through a "javaagent" is something which we experimented with, but did not end up using in practice.

Once the program run is finished, the metadata is stored by the framework's initialization code and the data is stored depending on the logger which was involved.

For unit tests an additional step is required after the program's classes have been instrumented. The unit tests themselves have to be instrumented so that the framework initialization can be run. This is done with the exact same procedure as with the regular classes, just that the list of unit test classes is used, and the *soot-unittests*[1] script is used.

---

[1]The *soot-unittests* script currently assumes the use of JUnit. Other unit test frameworks would be easy to adopt.

A.2   The "AnalysisUtility" Program

The utility program *AnalysisUtility* was written to ease the metadata management of runs. The following commands are implemented:

list. By default lists the last 10 runs. It accepts an optional argument *num*, which lists the last *num* runs. The *num* 0 is a special case, which lists all stored runs. Note that groups are abbreviated: The group id will be displayed prefixed by "g", and the most recent run info is shown.

info. One or more *id(s)* as arguments are required. Info will list the available metadata for the specified run id(s).

delete. One or more *id(s)* as arguments are required. Delete will remove the specified run(s) from the database. This command will first list a part of the metadata, and then waits 3 seconds before proceeding with the request. This hopefully minimizes accidental deletions. See also the next command.

force-delete. One or more *id(s)* as arguments are required. The behavior is just as for the *delete* command, but the safety pause is omitted.

note. A run id, and a note as arguments is required. The note is attached to the run id, and will be displayed by the *info* command. Note that with the command line interface the note has to be quoted, as it usually contains spaces.

stdout. A run id, and a file name as arguments is required. The file name will be stored in the database, and displayed by *info* to contain the log of the standard output of the command.

addgroup. A description as an argument is required. It will add the group, set the description, and then display the group id.

lsgroup. By default will list all groups. It accepts a group id as an optional argument,

which will display all runs of the group instead (with the same output format as *list*).

setgroup. A group id or the string "none" as parameter is required. It sets the group id for subsequently created runs. This persistent storage of the current group id ensures that all runs from unit tests are added to the same group. The parameter "none" returns to regular behavior, where new runs are not added to any group.

cleangroup. A group id as parameter is required. It will delete all runs in the group, with one 3 second safety pause for *all* runs together. This is useful to clean up after failed unit test runs.

updgroup. A run id and a group id as parameters is required. It will change the group of the run.

A brief overview is displayed when the utility is called without a parameter. The overview will list some outdated commands as well, which are put into context in Section C.1.

Appendix B

Object Inlining Notes

Note: The implementation calls a *field* container still an *instance* container. We changed these terms since using *instance* in abstract instance container is overloaded and slightly confusing. At the time of writing the source code has *not yet* been updated to reflect this change.

B.1   Event Log Examples

The following programs are annotated with the events that their statements create. These event annotations need to be read in the order the program would execute them, not necessarily from top to bottom. In the discussion following the code, the events are extracted in program execution order to allow easier analysis.

B.1.1   Simple Safe Java Program

A simple example where a safe inlining opportunity is found.

```
1   class  Number {
3     int  i ;
4
5     public  Number (int  x)  {
6        // assign  stack  frame  id  3,4
7        // assign  object  id  2  to  x ;  id  3  to  y
8        this . i  = x;
9        // ( target =2,3,  kind=arg,  containerid =3,4,  name=this)
10    }
11  }
12
13  class  Line  {
14    Number x,y;
15
16    Line  (int  a,  int  b)  {
17       // assign  stack  frame  id  2
18       // assign  object  id  1
19
20       this . x  = new Number(a);
21       // ( target =1,  kind=arg,  containerid =2,  name=this)
22
23       this . y  = new Number(b);
24       // ( target =1,  kind=arg,  containerid =2,  name=this)
25    }
26  }
27
28  class  Length {
29    public  Length (Line  l)  {
30       // assign  stack  frame  id  5
31       // assign  object  id  4
32
33       System.out. println  ("Length of " + l + " = " + (l.y.i−l.x.i));
34       // ( target =1,  kind=arg,  containerid =5,  name=r)
35       // ( target =3,  kind=instance,  containerid =1,  name=y)
36       // ( target =1,  kind=arg,  containerid =5,  name=r)
37       // ( target =2,  kind=instance,  containerid =1,  name=x)
38    }
39  }
40
41  class  Square {
```

```
42    public Square (Line l) {
44      // assign stack frame id 6
45      // assign object id 5
46
47      System.out. println ("A␣square␣with␣sides␣" + l + "␣has␣circumference␣" +
           4∗(l.y.i − l.x.i));
48      // ( target =1, kind=arg,  containerid =5, name=r)
49      // ( target =3, kind=instance,  containerid =1, name=y)
50      // ( target =1, kind=arg,  containerid =5, name=r)
51      // ( target =2, kind=instance,  containerid =1, name=x)
52    }
53  }
54
55  public class Main
56  {
57    public static void main( String [] args ) {
58      // assign stack frame id 1
59      Line l = new Line(1,4);
60      new Length(l);
61      new Square(l);
62    }
63  }
```

The log then contains the following events:

```
1     // ( target =2, kind=arg,  containerid =3, name=this)
2     // ( target =1, kind=arg,  containerid =2, name=this)
3     // ( target =3, kind=arg,  containerid =4, name=this)
4     // ( target =1, kind=arg,  containerid =2, name=this)
5     // ( target =1, kind=arg,  containerid =5, name=r)
6     // ( target =3, kind=instance,  containerid =1, name=y)
7     // ( target =1, kind=arg,  containerid =5, name=r)
8     // ( target =2, kind=instance,  containerid =1, name=x)
9     // ( target =1, kind=arg,  containerid =5, name=r)
10    // ( target =3, kind=instance,  containerid =1, name=y)
11    // ( target =1, kind=arg,  containerid =5, name=r)
12    // ( target =2, kind=instance,  containerid =1, name=x)
```

Here it is safe to inline *x* and *y* into *l* (instance id 1), because:

- *x* (instance id 2) is only dereferenced through container 1 with the name *x* (lines 8,12), or it is dereferenced through the special case with argument kind and the name *this* (line 1), which is always safe.

- *y* (instance id 3) is only dereferenced through container 1 with the name 6 (lines 6,10), or it is dereferenced through the special case with argument kind and the name *this* (line 3), which is always safe.

B.1.2 Simple Unsafe Java Program

This example shows a program where it is not safe to inline.

```
1   class  Cell  {
2           public  String  _value ;
3           public  Cell ()  {
4                   //  assign  object  id  2
5           }
6   }
7
8   class  Target  {
9           Cell   _inlineSource ;
10
11          public  Targcet ()  {
12                  //  assign  object  id  1
13          }
14
15          public  void  setCell (Cell  cell )  {
16                  //  assign  stack  frame  id  3
17                  this . _inlineSource  =  cell ;
18                  //  ( target =1, kind=arg,  containerid =3, name=this)
19          }
20
21          public  void  setCellValue ( String  value)  {
22                  //  assign  stack  frame  id  2
23
24                  Cell  newCell =  new Cell () ;
25
26                  this . setCell (newCell);
```

```
27          // ( target =1, kind=arg,  containerid =2, name=this)
29          // Note here that the kind of the container is argument, which
30          // is like a local variable. The containerid then refers to the
31          // frame id
32
33          this . _inlineSource . _value = "foo";
34          // ( target =1, kind=arg,  containerid =2, name=this)
35          // ( target =2, kind=field ,  containerid =1, name=_inlineSource)
36          newCell._value = value ;
37          // ( target =2, kind=local,  containerid =2, name=newCell)
38      }
39
40      public  static  void main( String [] args ) {
41          // assign stack frame id 1
42          Target  t = new Target () ;
43          t . setCellValue ("bar") ;
44          // ( target =1, kind=local,  containerid =1, name=t)
45          // A record here is not created for the method call , but rather
46          // for the dereference of t
47      }
48 }
```

The log then contains:

```
1          // ( target =1, kind=local,  containerid =1, name=t)
2          // ( target =1, kind=arg,  containerid =2, name=this)
3          // ( target =1, kind=arg,  containerid =3, name=this)
4          // ( target =1, kind=arg,  containerid =2, name=this)
5          // ( target =2, kind=instance,  containerid =1, name=_inlineSource)
6          // ( target =2, kind=local,  containerid =2, name=newCell)
```

In this example the inlining is determined to be not safe because the Cell (instance id 2) is not only accessed through *Target._inlineSource* (line 5) but also through the local *newCell* (line 6).

B.1.3   Simple Unsafe Jimple Program

Since the implementation is going to work with the Jimple version of the program, here is an example of how the data would be gathered in practice. Each stack location receives a name in Jimple, so when a dereference occurs, the container is always a "local" variable. Containers are tracked by tagging every local variable with the container information, which then is translated into a log entry when a dereference takes place.

When a local gets assigned a value, the relevant container information is saved. In the example below, the container information is everything but the target from the tuples explained in Section 5.4.2:

(*container kind*, *container id*, *container variable name*, *container attribute*)

Since the tuple only contains information about the container, and to easily differentiate the tuples from the five-tuples of the log entries, they are presented as:

container(*kind*, *id*, *variable name*, *attribute*)

Each tuple is again shown below the statement which generates it. The container tuples are only kept in memory at instrumentation time, and used to create the appropriate log entry code when the dereference is found in the code.

Soot names all stack locations explicitly in Jimple. This has the side effect that local variables at the bytecode level look very similar. They can be differentiated on their names, since stack locations names are prefixed with "temp$". This creates situations where targets are accessed through containers with different names, one name being the true container, the other a temporary name for a stack location. Consider

```
1   A a = new A();
2   a.foo();
```

which generates the following Jimple statements

```
1        A a,  temp$0;
3
4       temp$0 = new A;
5        specialinvoke   temp$0.<A: void <init>()>();
6       a = temp$0;
7        virtualinvoke   a.<A: void  foo()>();
```

Assuming "new A()" is assigned instance id 1, and the current stack frame id is 5, the log record is:

(*target:* 1, *c-kind:* local, *c-id:* 5, *c-name:* "a", *c-attrib:* none)

Note that the call to the constructor does not generate a log entry, as described above in Section 5.5.1.

This is the same example as the one in Section B.1.2.

```
1   class  Cell  extends  java . lang . Object
2   {
3       public  java . lang . String  _value ;
4
5       void  <init>()
6       {
7           Cell  this ;
8
9           //  stack  frame  id  = 4
10          //  object  id  = 2
11
12          this  :=  @this: Cell ;
13          //  this  :=  container ( kind=arg,  id=4,  name=this)
14
15           specialinvoke   this .<java . lang . Object:  void  <init>()>();
16
17          return;
18      }
19  }
20
21  class  Target  extends  java . lang . Object
```

```
22  {
24      Cell  _inlineSource ;
25
26      void  <init>()
27      {
28          Target  this ;
29
30          // stack frame id = 2
31          // object id = 1
32
33          this  := @this: Target ;
34          // this := container(kind=arg, id=2, name=this)
35
36          specialinvoke  this.<java.lang.Object: void <init>()>();
37
38          return;
39      }
40
41      public void  setCell (Cell)
42      {
43          Target  this ;
44          Cell  cell ;
45
46          this  := @this: Target ;
47          cell  := @parameter0: Cell;
48
49          /*      this._inlineSource = cell;      */
50          this.<Target: Cell  _inlineSource > = cell ;
51
52          return;
53      }
54
55      public void  setCellValue (java.lang.String)
56      {
57          Target  this ;
58          java.lang.String  value, temp$2;
59          Cell newCell, temp$0, temp$1;
60
61          // stack frame id = 3
62
63          this  := @this: Target ;
```

```
64          value := @parameter0: java.lang.String;

66
67          /*      Cell newCell = new Cell();          */
68          temp$0 = new Cell;
69          // temp$0 := container(kind=local, id=3, name=temp$0)
70          specialinvoke temp$0.<Cell: void <init>()>();
71          newCell = temp$0;
72          // newCell := container(kind=temp, id=3, name=newCell)

73
74          /*      this.setCell(newCell);          */
75          virtualinvoke this.<Target: void setCell(Cell)>(newCell);
76          // (target=2, container=arg, id=3, name=this)

77
78          /*      _inlineSource._value = "foo";      */
79          temp$1 = this.<Target: Cell _inlineSource>;
80          // temp$1 := container(kind=field, id=1, name=_inlineSource)
81          // Note here that the id refers to the object id when it is an field
                container
82          temp$2 = "foo";
83          // Note that temp$2 is not marked as a container, because its value is
                not a class under our control
84          temp$1.<Cell: java.lang.String _value> = temp$2;
85          // (target=2, container=field, id=1, name=_inlineSource)

86
87          /*      newCell._value = value;          */
88          newCell.<Cell: java.lang.String _value> = value;
89          // (target=2, container=local, id=3, name=newCell)

90
91          return;
92      }

93
94      public static void main(java.lang.String [])
95      {
96          java.lang.String [] args;
97          Target t, temp$0;

98
99          // stack frame id = 1

100
101         args := @parameter0: java.lang.String [];

102
103         /*      Target t = new Target();          */
```

```
104          temp$0 = new Target;
106          //  temp$0 := container(kind=local,  id=1, name=temp$0)
107          specialinvoke  temp$0.<Target: void <init>()>();
108          t = temp$0;
109          //  t := container(kind=local,  id=1, name=t)
110
111          /*       t. setCellValue("bar");            */
112          virtualinvoke  t.<Target: void setCellValue (java.lang. String )>("bar");
113          //  ( target =1,  container=local,  id=1, name=t)
114
115          return;
116      }
117  }
```

The only difference in the log entries generated are that we previously ignored stack frame ids when the method body had no explicit statements.

B.2   Instrumenting with Soot

Since our object inlining implementation is based on a source code transformation, the framework needed to include a capability to only monitor those classes for which the source code is present. The framework marks those classes which can be transformed, which also solves the issue that some Java library classes could not be optimized either since they are used internally by the JVM. We chose Java interfaces as a simple, non-invasive marker which will also be present in the output files.

So the framework ensures that all monitored classes implement the edu.unm.cs.oal. eventlog .core .Instrumented interface. These markers have to be added before the rest of the instrumentation is applied. As discussed in Section D.1, an execution of Soot will process a number of packs, each with different phases. Any of the *whole-Jimple* packs automatically enable whole-program analysis, which triggers a number of static analyses to be executed. Since these analyses are not required for the framework, we edited the Soot sources, and added a new pack: the **jctp**, or Jimple complete transformation pack. It allows accessing all Jimple bodies through the Scene class, without having any side

effects.

The framework adds the ClassTransformer phase to the jctp pack. The ClassTransformer adds the marker interface to all application classes, as well as the instance id field and accessor function. Now that these are in place, the remainder of the instrumentation can easily determine what needs instrumentation and what does not by doing a type check. The rest of the instrumentation can then be added to the **jtp** pack, since every function body is transformed at a time.

The first phase in the jtp pack is to mark the first actual statement of each body using the `MarkerTransformer` class. This is necessary for a number of reasons: In Jimple, some *identity* statements are introduced. These are essentially meta-information which Soot encodes in pseudo-instructions. These *identity* statements have to appear as the first statements, we cannot add instrumentation before them. Another reason are so called *traps*, essentially ranges where exceptions are caught. Since the instrumentation has to be added *before* the statement which triggers it, the code is naturally added with the insertBefore function. This has the side effect that these traps become misaligned. So while searching for the first real statement, the existence of traps is also recorded so that they can later be moved back to the right place.

Next is the `GetterTransformer` phase. It identifies pure getter functions: those who do nothing but return a member. This is important because access through a getter causes a different event pattern than direct field access. And if there are no side effects[1], then the getter function could be replaced by a direct field access (possibly after changing the field's visibility).

The `instance id` is initialized in the next phase by the `InstanceidTransformer`. While the field was already added in by the `ClassTransformer`, it was not yet initialized. This phase adds the initialization code to all constructors. The initialization is done by retrieving the next value from the `edu.unm.cs.oal.eventlog.core.Counters` class, either through direct access, or by calling an function, depending on flags passed to the instrumentation code[2]. Lastly a call to the logger is inserted to record any creation of in-

---

[1]While side effects in general are difficult to determine, a check for one exact statement excludes any possible side effects.

[2]Direct access is faster, but adds slightly more code to the class. Retrieving the value through a

stances of this class. The initialization is only performed if the instance does not yet have a valid id. This guarding is necessary so that constructors can call each other in arbitrary ways.

The stack frame id gets added in the `StackframeidTransformer` phase as a local variable. This variable is immediately initialized from the `Counters` class, with the same options as mentioned for the instance id initialization. Note that at the Jimple level nested blocks are collapsed into one body.

The `MainTransformer` phase adds the framework initialization code to the main method[3]. It adds one call to the `Eventlog` class to add a logger to the system. Note that because of static initializer, other code can actually run *before* the main method is executed. The capture of all events is ensured by using a static initializer in the `EventLog`, which adds a `BufferLogger`. This logger simply buffers events in memory, and when the real logger is set, passes them on.

This concludes the instrumentation of the initialization functionality. Now the second `MarkerTransformer` phase is run, which moves the traps as mentioned earlier.

Now the instrumentation of the main body takes places in the `InstrumentationTransformer` phase. Here the instrumentation is added which creates the log entries as specified in Section 5.5.

B.2.1 Instrumenting Unit Tests

Unit tests are essentially mini-programs; every `junit.framework.TestCase` is the equivalent of a *main* function. Therefore we cannot use the `MainTransformer` phase, since it only looks for *main* functions, and Soot only sees one class as the *main* class, while with unit tests we have several. The `TestCaseTransformer` allows adding the initialization to several classes. It shares most code with the `MainTransformer`, but works on JUnit test cases instead of the main function.

Similarly, the rest of the usual instrumentation should not be applied, since the behavior

---

function call can be easily synchronized, if this is required by the target program.

[3]The class contains the main method is user specified, it cannot be automatically determined.

of the unit tests itself does not have to be observed. The goal is to observer the behavior of the actual program which the unit tests only set up. So the rest of the program has to be instrumented using the method described in the previous section, and the unit tests themselves only need the instrumentation described here.

One exception is created by the usage of *mock objects*. Mock objects can i.e. implement an interface of the program. In this case the interface is instrumented, and the rest of the program expects any implementation of it to have properties like the instance id. A simple solution is to add the instance id to all classes which are part of the unit testing code, since its presence does not cause any side effects when not used. Therefore the `InstanceidTransformer` is used when instrumenting unit tests as well.

B.3   Flat File Storage

The flat file storage used for object inlining follows a very simple format. This simplicity ensures that writing introduces as little overhead as possible. The reason to remove all possible overhead is that the large number of events generated result in a lot of time being spent on writing them out to disk. Note that buffering does not seem necessary, as the underlying OS will do this automatically.

The header is needs to be written and read using Java's `DataOutputStream.writeChars` and `DataInputStream.readChars` functions, respectively. It is a string in the format

```
runid=${runid};${type}:\${version}\n
```

where

- *runid* is the id of the run,

- *type* is the type of the data in the log, and

- *version* is the version the reader needs to support to be able to read it.

After the header follows an arbitrary number of events. An event is encoded in a mixed format, where strings are properly encoded, but numbers are just written out in a binary format to save the encoding time. The record starts with two characters: "\", followed by a single upper case character. Object inlining uses "C" for a creation event, and "E" for a dereference.

A creation event consists of the instance id, a Long, followed by the type id, an Int. A dereference event consists of the target id (Long), the container kind (Int), the container name (UTF), the container id (Long), and the attribute (Int).

Note that the number of records is not stored, and a linear scan of the file is required to determine it. This information is not directly stored because knowing the exact number of records is not important, unless the analysis is run anyway. The file size serves as a heuristic for the number of records. Note that the file size is not an exact predictor due to the records being of variable length since they contain strings.

B.4   Object Inlining Result Details

Here we name the opportunities which were used for the results in Tabel 5.2.

1. **antlr**
   `antlr.CharScanner.inputState` (its declared type) `antlr.CharBuffer.queue` (its declared type)

2. **luindex**
   `org.apache.lucene.analysis.standard.\`
   `StandardTokenizerTokenManager.input_stream` (its declared type).

Appendix C

Previous Topics of Study

There are a number of topics the author studied previously, which did not result in a publication. These topics were abandoned for some reason, and here we strive to explain some of the causes. In some way this is a list of negative results.

C.1 Notes on Databases as Large Storage Repositories

Initially we used the database to store the event log. The requirement was to store a lot of data, which seemed like an ideal job for a database. The schema for the two types of object inlining events was easy to set up. The class name could be easily looked up using a simple join. As an optimization we omitted to tell the database the foreign keys, so that it would not spend the time at every insert to verify integrity of the entry. Nonetheless, initial performance was very slow. We consulted some guides for optimizing the database configuration, since none of us was an expert in this area. However, it seemed that since all involved queries are rather simple, this tuning did not yield much of a performance improvement.

After doing some research, we found out that today's databases automatically start transactions if the user does not start them himself. Automatic transactions mean that after every statement the database does its best to ensure the data is written out to disk

before it returns control to the client[1]. This is very harmful for performance, because every statement during data collection was a write. On top of this it circumvented the OS caching mechanism. The solution was relatively simple: we manually control transactions, and commit a number of insertions together. A simple counter in the logger controlled when a transaction would be committed, and a new one started up. As a result performance was improved, but the system worked still slower than what we expected.

For example, the schema did not store the number of events a given run had, since this is derived information. Yet a simple `COUNT()` query on all event entries for some run id took a very long time to complete, despite an index being available for this column. It turns out that somehow the database is not able to use the index well for COUNT() queries, and the database did a linear scan to figure out the number of rows. While there might be ways to speed this operation up without caching the result, we did not discover them. As a workaround we changed the "AnalysisUtility" to display the rest of the metadata without counting the number of entries[2]. Since the number of rows was roughly proportional to the run time of the data collection, the exact count was not usually necessary.

In the beginning a different safety algorithm was used. Its simplicity allowed us to express it in SQL. We felt that this was a very nice feature, as the algorithm was easy to understand. Little computation was necessary in the client, it was mainly the SQL query that computed the instances that were safe. One of the queries joined the event log table onto itself, including a GROUP BY. Overall it was not a very simple query, but not a complex one either since it only worked with one table. Yet the database was extremely slow executing the query. Some research revealed that a certain kind of index would likely speed things up significantly, since it would be used by the database to perform the join – previously the join required a whole table traversal, which actually turned it into a n^2 algorithm. Adding the index did speed the analysis up significantly, from several hours to minutes, but adding the index itself took more than a day. Additionally it made the size of the database explode: together with the other indices, like the primary key index, and the run id index, the database had several times the size of its payload.

With the indices taking up so much space, we soon realized that our machine was not

---

[1]We used synchronous database access.
[2]The now obsolete *sinfo* command shows the run information without counting the rows.

set up properly: the database simply did not have enough disk space allocated. Since we were still in an experimental stage, we started to delete runs whose data had become obsolete. This turned out to be far from easy: A DELETE query would run for days. When it was finished, the OS reported no freed space at all. Some research found the reason: The database, PostgreSQL, only marked rows as deleted, but does not release the storage it had acquired so far. At the time we were trying to do a modification to the database, and PostgreSQL was not able to use the deleted rows for the operation. This resulted in PostgreSQL aborting the modification. A full release of the space is difficult, probably because the data has to be compacted, or defragmented, so that the space can be released. In PostgreSQL this is done through a so-called vacuum. The vacuum is usually performed automatically. It turned out that after the delete operation it had sprung into action – it was just nowhere from being done. We let the so called autovacuum run for a couple of days. Since there was no easy way to gage its progress, we cancelled it.

Additional research revealed that this slow behavior might have been caused by the lack of disk space that we were trying to solve in the first place. At this point we reconfigured the server and allocated more disk space for the database. Afterwards we manually restarted the vacuum. The vacuum remained extremely slow. After a couple of days it was done, and it had freed up disk space. The database was still bigger than we had expected, but that was most likely due our limited understanding of the space requirements of the indices.

Out of concern that these extremely slow operations would soon be required another time, we switched databases to MySQL. The reasoning was that MySQL has the reputation of being fast, possibly at the cost of sub-par separation of conflicting queries. Since our data collection seemed to exhaust the database with just one client, this potential pitfall did not concern us[3]. After the switch, performance did increase compared to PostgreSQL. There was no direct comparison possible because during the switch the database schema was updated.

However, the indices were managed by MySQL similarly to PostgreSQL: they take up a lot of space. Soon thereafter it became necessary to clean up the database as well. Again

---

[3]MySQL has come a long way, and it is likely these restrictions did not apply to its current version any more.

the delete operation was extremely slow, and did not complete after running for more than a day. At this point it became clear that databases were simply not well suited for our purpose. It is possible that there are tricks that could be used to obtain good performance out of a database for our usage pattern. But we had exhausted our knowledge, and still did not reached an acceptable level while operating the database.

Since the data is essentially a stream of information, a flat file storage system turned out to serve us well. This has the advantage of almost instantaneous deletion operations, since all the events of one run are always grouped together in one file. It turns out this primitive approach works well when only linear scans of the log are required, as is the case with object inlining.

C.2   Memoization

Memoization is a technique to avoid executing functions several times for the same inputs. A memoized function's return value is stored, and can be looked up based on the function's parameters. This allows returning the cached value when the same parameters are encountered again.

As can be deduced from this description, memoization is only valid for functions without side effects. What constitutes a side effect in a non-functional language like Java is not easy to determine. For example, information can be stored temporarily in fields, instead of passing it along as parameters for a chain of function calls on some receiver object. Technically this is a side effect, while in practice such behavior would not prevent memoization.

Caching the return value is also difficult because Java objects are not generally immutable. If a reference to the return value is stored, how can we ensure that the object is not mutated through the returned reference? How could an analysis determine if this is happening? Object equality is not easily determined, since the `equals` method is not generally required to be implemented. So any analysis of mutation would run into the same pitfalls as already mentioned for the side effect analysis.

An alternative to storing a reference to the return value is storing a copy of the return

value. But how can we copy a return value? Again it is not required for objects to implement the `clone` method. If it is not implemented, is a deep or a shallow copy more appropriate? This is not generally decidable.

Although we did implement a prototype for adding memoization to functions, these open questions lead us to abandon this topic. It seemed like either memoization would only be applicable in very restricted scenarios, or would have the potential to generate incorrect results. Note that memoization could again be considered for implementation in the framework presented in this thesis, as we outline in Section 6.

C.2.1   Implementation

We implemented memoization in AspectJ [19]. It does not include analysis for when memoization is valid, but some limited profitability heuristic is included. A pointcut defines to which functions the memoization is applied. The implementation works as follows:

- AspectJ presents the function arguments as an `Object[]` array to the aspect. This is wrapped in an `ArgsKey` class to attain better control over the hashing.

- The cache is essentially a hash table. The `ArgsKey` instance is used to look up if the result has been previously computed.

- If the arguments had not been encountered before, the return value is computed, stored, and the statistics are updated with a miss.

- If the look-up is successful, the memoized value is returned, and a hit is recorded in the statistics.

Notice how this implementation uses the hash value for the equality check. This is not a good idea in general, since the hash value can collide, but does often work in practice.

The statistics can be used to determine if there is a high enough hit-miss ratio that memoization is a good idea. It does not take memory usage into account. Despite these limitations, the aspect is a useful implementation of memoization, and does work in practice. It is just not automated, but needs to be controlled manually. Notice how the actual

implementation of memoization is exactly one of these cross-class concerns which AspectJ was created for.

C.3   Automatic Data Structure Selection

Another area we explored was how the standard Java collection library is used in practice. The goal was to find a way to automatically select or switch the implementation of an interface based on usage patterns. The documentation states clearly that some operations on some collection classes are relatively expensive, but the assumption was that this is not always considered by the programmer. E.g., `ArrayList.remove()` takes $O(n)$ time, while a naïve user might expect it to only take $O(1)$ time. We monitored the usage of collection classes in real world programs to understand how these classes are used so that we could implement an effective automatic selection or transformation.

C.3.1   Usage Analysis

We analyzed the usage patterns by adding logging code to the collection classes. Once the instrumented classes are on the boot class path, any usage is logged, which is much easier than looking for every use of these classes. We used *Javassist* [9], an easy to use bytecode rewriting toolkit, to add the instrumentation.

The logging code outputs a time stamp as well as which class and method was called. We then graphed time vs. number of method calls to see how often, and in what situations the expensive methods were used. We collected data from most of the DaCapo 2006 benchmark suite, and some of SPEC jvm98. Take the *jess* benchmark as an example. Figure C.1 shows the behavior of an instance of the `Vector` class. Here we can see `indexOf` being executed relatively frequently, which is not efficient to execute for the `Vector` class. In contrast, Figure C.2 shows an instance where almost exclusively the most commonly used, and efficient, functions were called. Comparing the total number of calls shows that Figure C.2 was used around 100 times more often than Figure C.1.

The graphs for the rest of the benchmarks were similar. The more complex usage
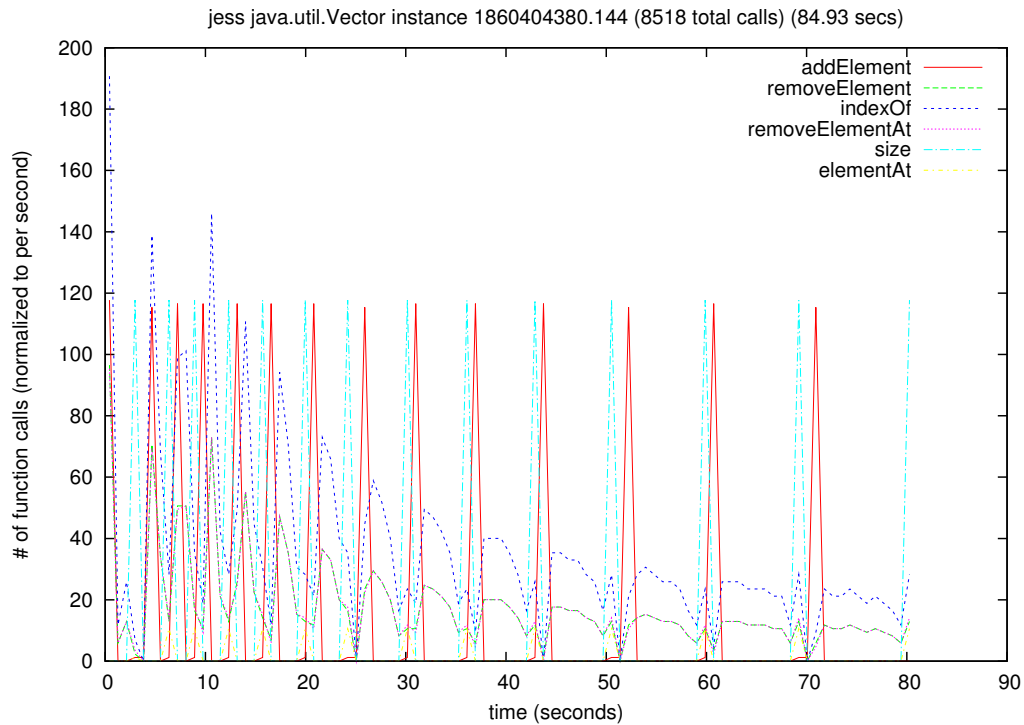
Figure C.1: Graph of the frequency of method calls over time for the jess benchmark. A variety of functions is used, including `indexOf`, which is not very efficient.

patters would mainly appear for the relatively little used instance, while the instances that were used the most had simple access patterns, using almost exclusively the efficient functions.

C.3.2 Conclusion

Contrary to our assumption, the analysis showed that the benchmarks programs we considered did use the collection classes in reasonable ways. There certainly were places where an optimization was possible, but these places were in timing insensitive areas.

An explanation could easily be the selection of the benchmarks. Only selected pro-
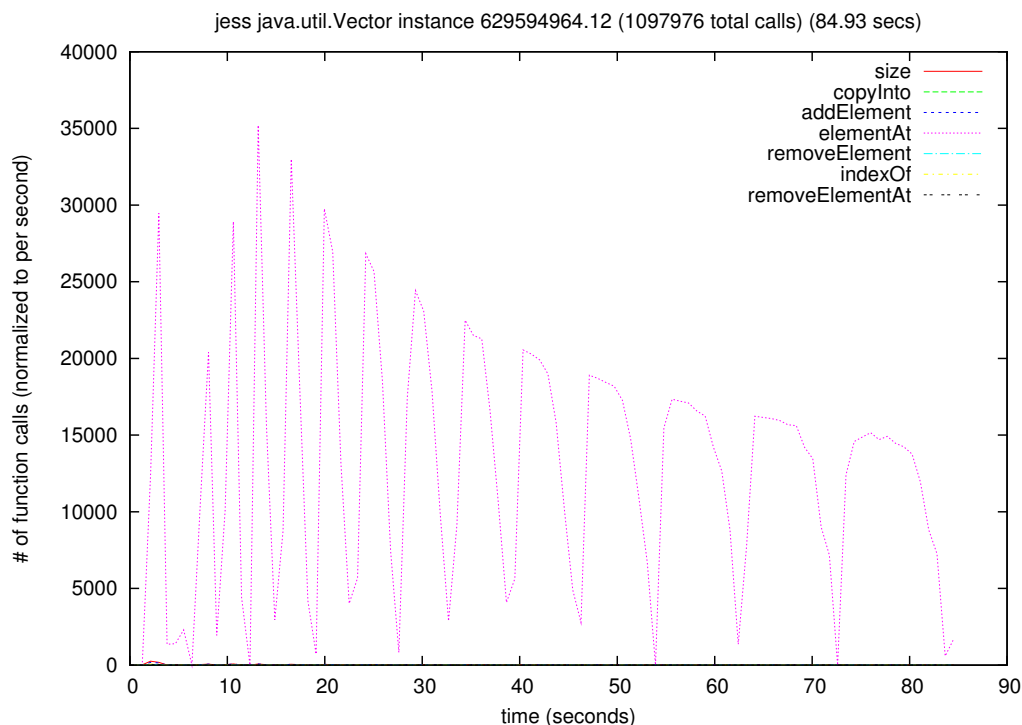
Figure C.2: Graph of the frequency of method calls over time for the jess benchmark. This instance is used extensively, with most calls going to the `elementAt` function, which is efficient in the `Vector` class.

grams are added to benchmark suites, and generally the code quality is relatively high[4]. It seems that data structure selection is usually handled well, as long as the standard collection library is used. We suspect that in performance critical places, or when unusual usage patterns appear in practice, custom containers are used. We believe that there is a large number of programs in use in the real world, where the basic data structure selection is not handled as well as in these benchmark suites. These programs, however, are often proprietary, and are not easily available to researchers. Also for publications it is expected that an optimization is evaluated on the standard set of benchmarks, and not just some proprietary program. These expectations would make any result difficult to publish, even

---

[4]While benchmark suites strive to only include high quality programs, it is well known that individual benchmarks often contain some deficiency. Unless very old benchmarks are considered, these are often rather subtle, and not of principal nature.

Appendix C. Previous Topics of Study

if it was beneficial to certain classes of programs.

Appendix D

Introduction to the Essential Libraries

D.1   Soot Overview

Soot is a Java optimization framework under development at McGill University since 1999. Its main feature are three different intermediate representations: **Baf**, "a streamlined representation of bytecode which is simple to manipulate"; **Jimple**, "a typed 3-address intermediate representation suitable for optimization"; and **Grimp**, "an aggregated version of Jimple suitable for decompilation" [28].

Soot was designed to take Java bytecode as input, but has since gained the feature to read in Java source code as well. The bytecode is first translated to Baf, possibly optimized in this format, translated to Jimple and again possibly optimized. From Jimple it is translated to Grimp, and for a last time possibly optimized. Lastly it is translated back to Baf and from there back into bytecode. Soot is able to also output any of its intermediate formats in a human readable format, as well as output as Java source code [25].

Baf is a stack based representation, like Java bytecode. The difference is that the constant pool is abstracted away and all instructions become fully typed[1]. It also makes local variables explicit, which are marked through *identity* statements. **Jimple** eliminates the stack to make all operands explicit. This allows all operands to be typed, so that in

---

[1]Most, but not all, Java bytecode instructions are typed. Exceptions are i.e. `dup` and `swap`.

turn operations can become untyped. It also eliminates the "essentially interprocedural" `jsr` instruction. **Grimp** looks similar to Java source code. It allows expression trees in contrast to the essential flat representation of Jimple. It is easier to read, and to construct stack code from.

The three different intermediate representation allow a transformation implemented in Soot to work at the abstraction level that is most appropriate, since transformations are implemented between all IRs.

Soot's operations are grouped in *packs*, each of which contains one or more *phases* [4]. Each *pack*, and *phase*, can be enabled or disabled. First Soot creates Jimple code for all method bodies (the **jb** pack). Then a series of whole program packs are applied, the call graph pack (**cg**), the whole-Jimple transformation pack (**wjtp**), the whole-Jimple optimization pack (**wjop**), and then the whole-Jimple annotation pack (**wjap**). All these packs can be changed by the user and operate on what Soot calls the **Scene**; the set of all classes which Soot was able to load. The "whole" packs are not enabled by default, since they require the analysis of all code. In regular mode only application classes are analyzed and library classes are only considered for typing information.

Afterwards the following packs are executed: The Jimple transformation pack (**jtp**), which is enabled but empty. This is where most user intra-procedural analysis will be added to. The Jimple optimization pack (**jop**) is disabled and contains optimizations shipped with Soot. It is followed by the Jimple annotation pack (**jap**), which contains phases that add annotations. These are also disabled by default[2].

The **bb** pack converts each body to Baf. It is followed by the **tag** pack which optimizes the tags which were added in the jap pack. Also available is the Dava [25] body pack (**db**), which is used when decompiling bytecode with Soot.

Over the years, the Soot framework has evolved more and more capabilities. It is used as an optimization framework, and a number of static analysis are implemented in it. As mentioned before it can also decompile Java class files. It can be used as a compiler, e.g., to create Java classes from scratch. This flexibility allowed us to use Soot as a bytecode rewriting tool: first the existing classes are decompiled into the Jimple IR, then they are

---

[2]More accurately, the jap pack is enabled, but all its phases are disabled.

modified using the compiler capabilities of Soot.

D.1.1   Jimple

Jimple is one of the intermediate representations of Soot. It was designed to be compact, and easy to work with. A three-address format is used, so the general layout of a Jimple instruction is $x = y \circ z$, where $\circ$ is some operator. Instead of operating on a stack, Jimple uses explicit names for the stack locations used in the bytecode. Its operators are untyped, but all variables are typed. An example of Jimple code is shown in Section B.1.3.

Jimple is very compact and only has 19 different operations. This low number makes it much easier to reason about than either Java source or Java bytecode.

D.1.2   Jimple Grammar

Raja Vallée-Rai explains in his Master's thesis [29] the Jimple grammar. The production rules are reproduced in Figures D.1 and D.2.

The productions start with the *stmt* rule. Note that Jimple describes only the body of Java methods. The remainder, class and member definitions, are identical to those in Java source code.

D.2   Recoder Overview

The Recoder[3] library [15] was created at the University of Karlsruhe. It is a framework for "source code meta-programming"; it allows to read in Java source code, transform it, and output the results again as Java source code.

The transformation is enabled by the meta model which Recoder constructs of the source code. This model is not unlike the abstract syntax tree, but was extended to an

---

[3]The library is at times spelled RECODER and at other times Recoder. We adopted the latter for easier readability.

attributed syntax graph. Still each element has links to its parents and children. Interfaces are used to describe the properties each element can have, e.g., a `MethodDeclaration` is a `MemeberDeclaration`, as well as a `StatementContainer`, and a `ParameterContainer`, etc. Additional information, such as types and cross-references are also added by Recoder.

A program transformations manipulates this "syntax tree". All changes are reported to a `ChangeHistory` service, which allows to reverse changes and also tries to minimize updates by not updating the complete model unless it is necessary. Recoder does some model error checking, but unfortunately it does not catch all errors.

Recoder ships with an API documentation, which is as usual produced by Javadoc. However, it is only very sparsely annotated with additional information. It certainly is a sign of a well design program that it is *still* possible to use Recoder with relative ease. Nonetheless sometimes trial and error has to be used to understand how their internal model represents Java source code.

Another requirement imposed by Recoder makes writing transformations more challenging: it is not possible to detach an element of the model, and attach it somewhere else. Instead, the element has to be cloned. If references to the element or its children are kept, they become invalid after its parent has been detached. The nesting relationship of elements which were discovered in separate passes is not easily determined, which often leads to bugs that are difficult to find. If this restriction is kept in mind, then it is possible, and at times cumbersome, to avoid these issues.

stmt →	assignStmt	identityStmt	gotoStmt	ifStmt	invokeStmt	switchStmt	monitorStmt	returnStmt	throwStmt	breakpointStmt	nopStmt;
assignStmt →	local = *rvalue*;	field = *imm*;	local.field = *imm*;	local [*imm*] = *imm*;
identityStmt →	local := @this: type;	local := @parameter*n*: type;	local := @exception;
gotoStmt →	`goto` label;
ifStmt →	`if` *conditionExpr* `goto` label;
invokeStmt →	`invoke` *invokeExpr*;
switchStmt →	`lookupswitch` *imm* { `case` value$_1$: `goto` label$_1$; ... `case` value$_1$: `goto` label$_1$; `default`: `goto` defaultLabel; };	`tableswitch` *imm* { `case` low: `goto` lowLabel; ... `case` high: `goto` highLabel; `default`: `goto` defaultLabel; }
monitorStmt →	`entermonitor` *imm*;	`exitmonitor` *imm*;
returnStmt →	`return` *imm*;	`return`;
throwStmt →	`throw` *imm*;
breakpointStmt →	`breakpoint`;
nopStmt →	`nop`;

Figure D.1: Jimple grammar (statements).
This is an exact reproduction of the table in [29] on page 24.

$imm \rightarrow$	local $\mid$ constant
$conditionExpr \rightarrow$	imm_1 $condop$ imm_2
$condop \rightarrow$	$> \mid < \mid = \mid \neq \mid \leq \mid \geq$
$rvalue \rightarrow$	$concreteRef \mid imm \mid expr$
$concreteRef \rightarrow$	field $\mid$ local.field $\mid$ local$[imm]$
$invokeExpr \rightarrow$	specialinvoke local.m(imm_1, ..., imm_n;) $\mid$ interfaceinvoke local.m(imm_1, ..., imm_n;) $\mid$ virtualinvoke local.m(imm_1, ..., imm_n;) $\mid$ staticinvoke local.m(imm_1, ..., imm_n;)
$expr \rightarrow$	imm_1 $binop$ imm_2 $\mid$ (type) imm $\mid$ imm instanceof type $\mid$ $invokeExpr$ $\mid$ new refType $\mid$ newarray (type) $[imm]$ $\mid$ newmultiarray (type) imm_1, ..., imm_n $[]*$ $\mid$ length imm $\mid$ neg imm
$binop \rightarrow$	$+ \mid - \mid > \mid < \mid = \mid \neq \mid \leq \mid \geq \mid * \mid / \mid$ $<< \mid >> \mid <<< \mid \% \mid$ rem $\mid \& \mid \mid \mid$ cmp $\mid$ cmpg $\mid$ cmpl

Figure D.2: Jimple grammar (support productions).
This is an exact reproduction of the table in [29] on page 25.

Appendix E

Object Inlining In Practice

This chapter provides instructions on how the object inlining implementation is used in practice. It is written as a brief tutorial to help anyone attempting to use our software.

E.1   Installation

The first step is to check out the subversion repository at `https://digamma.cs.unm.edu/svn/alg-opt/trunk/david/sharing/object-inlining/`[1]. Let us assume that the check out is stored at `${OI}`. The instrumentation and analysis code is managed in an Eclipse project. We recommend checking out the `soot-eventlog` sub-folder of the URL above from within Eclipse to get the easiest setup, and the ability to push changes to SVN from Eclipse. Although all major components are written in Java, a number of shell scripts provide a more easily usable interface. It is recommended to add `${OI}/bin` to `$PATH` so that these scripts can be easily invoked. We assume this has been done for the remainder of this chapter.

The file `${OI}/settings` contains a number of settings that the scripts in `bin/` are using. If any setting needs to be adjusted for the local system or user, it should be placed in `${OI}/setings.local`, which takes precedence over the global settings. For ex-

---

[1]The repository does not allow public access. Please contact the author to receive a copy of the source code.

ample SOOT_EVENTLOG controls the classpath entry where both the instrumentation and the analysis code will be loaded from. Instead of constantly updating the jar file at ${OI}/lib/sootsimple.jar, the variable should be set to the folder where the Eclipse project stores its class files.

E.2 Instrumentation

With the initial setup out of the way we show how to use the analysis on the *pmd* program, version 3.7. After extracting the zip file we create the sootsimple subdirectory to group all the files which we add for the instrumentation.

For the configuration step we need to identify all the class that pmd uses. Since the class files are usually bundled in a jar file, we build pmd using "cd bin && ant". Then all class files end up in the build directory. Since later we want to run the instrumented classes, we immediately move this directory out of the way: "mv build simplelog/". The list of classes can now be easily formatted using the "findclasses" script. From within the sootsimple directory we run "findclasses build > classes" to collect the class list. This file can be edited to exclude classes from the instrumentation. Inspecting the list reveals that the unit test classes are also included. As the lest of test classes should be stored separately, we instead run the following command: "findclasses build | egrep '^test.' -v > sootsimple/classes && findclasses build | egrep '^test.' > sootsimple/unittests".

Sometimes the build directory contains more than just the classes, for example input data could also be present. The instrumentation will only transform the class files, so we populate the instrumentation output directory simplelog/sootOutput with the contents of the build directory: "cp -a build sootOutput". The instrumentation will later overwrite all class files, so copying them over as well does no harm.

Before the classes can get instrumented, we need to collect all the dependencies of pmd. Depending on the program these are already included in a lib subdirectory, but sometimes they need to be downloaded manually. So that the dependencies can be easily

discovered, they should be placed in the `sootsimple/support` directory (a symlink is sufficient). Then the little script template shown in Figure E.1 can be run, which places the instrumented class files in `sootsimple/sootOutput`.

The environment variables set on lines 19-23 configure the behavior of the instrumentation. `COLLECTION` selects what instrumentation should be added (it selects the package inside of `edu.unm.cs.oal.eventlog.collection`) and `EVENTLOG` selects which class inside that package should be used as the logger. The following variables are self explanatory: the application name, version, and its class path. Note that on line 24 the "`soot-eventlog`" script is run, which is another one of the helpers in `${OI}/bin`.

After instrumenting the regular classes the unit tests need to receive a special instrumentation, as mentioned before in Section A.1. We use another script template for this purpose, Figure E.2. It is very similar to the previous template. It differs by operating on the unit test list, instead of the class list and that the unit test classes are cleaned up beforehand.

E.3   Data Collection

After everything is instrumented the program needs to be run so that the data collection can start. When running the program the regular way, as opposed to the unit tests, a simple substitution of "`javasimplelog`" for the "`java`" command should be sufficient. A proper classpath setup is still required, only the instrumented classes, and not the original ones, should be available on the class path.

The more complicated example is collecting data from running unit tests. Since pmd provides an ant setup to run the unit tests, we chose to modify ant's `build.xml` to our needs. It is modified similar to the way that the "`javasimplelog`" script operates: we have to add the dependency classes on the classpath. Initially we have to create a symlink so that this script can find the instrumented classes: "`ln -s sootsimple/sootOutput build`". Then we edit the `build.xml` file, and add to the "`<path id ="dependencies.path">`" section some *pathelement* entries with the framework and its dependencies. An example of this is shown in Figure E.3.

Some slight modifications to the way tests are run are necessary as well. To make it easier to separate the collected data, we instruct JUnit to run every test in its own JVM instance by changing "`forkmode="perTest"`". The original build script used a separate directory for the unit test classes, but since for us they are co-located with the regular classes, we have to change ${dir.regress} to ${dir.build} within the "`todir`" attribute. For us it was also necessary to add a formatter, as shown on line 3 in Figure E.4. For pmd, some tests failed to run with the instrumentation, so we set the "`haltonerror="yes"`" attribute for the "`batchtest`" command, and incrementally added exclusion patterns for the failing tests, as shown on line 11.

After making all these modifications, we can collect the data from the unit tests. Automatic grouping needs to be enabled first by running "`AnalysisUtility addgroup "pmd␣3.7␣unit␣tests"`". The output will show the newly created group id. Let 3 be the group id displayed. Now we need to set this as the default group id to associate new runs with by running "`AnalysisUtility setgroup 3`". Now we can execute "`ant tests`" from the `bin` directory to start the data collection.

E.4   Analysis

The collected data is stored in the current directory, which is `bin/`. The scripts automatically determine where the data is, so the location does not need to be specified. Running "`AnalysisUtility list`" will show us the last couple of runs. The most recent one should be the group of unit tests. It can get analyzed by running "`AnalyzeSimple ganalyze 3`", where 3 is the group id. The analysis output will appear on the terminal, so it is advisable to redirect it to a permanent storage location, for example by running "`AnalyzeSimple ganalyze 3 2>&1 | tee group3 .analysis`". This command saves both regular output as well as errors, while displaying the output in the terminal as well as saving it in the file `group3.analysis`.

The analysis output is human readable, and very verbose. The most important information is listed first, e.g. the list of safe opportunities.

```sh
1  #!/bin/sh
2
3  BASE=`dirname $0`
4  BASE=`readlink -f $BASE`
5
6  mkdir -p "$BASE/log"
7
8  LOG="$BASE/log/instrument"
9  CLASSES="$BASE/classes"
10
11 echo "Logging␣to␣$LOG..."
12
13 for f in $BASE/support/*jar; do
14   SUPPORT=$SUPPORT:$f
15 done
16
17 cd build
18
19 COLLECTION=simplelog \
20 EVENTLOG=hybrid \
21 APP_NAME=pmd \
22 APP_VERSION=3.7 \
23 CLASSPATH=$CLASSPATH:$SUPPORT \
24 time soot-eventlog -d ../sootOutput `cat $CLASSES` >&
     $LOG
25
26 R=$?
27
28 exit $R
```

Figure E.1: Script template to instrument a program.

```
1  #!/bin/sh
2
3  BASE=`dirname $0`
4  BASE=`readlink -f $BASE`
5
6  LOG="$BASE/log/unittests-instrument"
7  CLASSES="$BASE/unittests"
8
9  echo "Logging␣to␣$LOG..."
10
11 for f in $BASE/support/*jar; do
12    SUPPORT=$SUPPORT:$f
13 done
14
15 # Clean up previous test files, they'll conflict on the
      classpath otherwise
16 cd $BASE/sootOutput
17 rm -f `cat $CLASSES | class2path`
18
19 cd $BASE/build
20
21 COLLECTION=simplelog \
22 EVENTLOG=hybrid \
23 APP_NAME=pmd \
24 APP_VERSION=3.7 \
25 CLASSPATH=$BASE/sootOutput:$CLASSPATH:$SUPPORT \
26 time soot-unittests -d $BASE/sootOutput `cat $CLASSES` >&
      $LOG
27
28 R=$?
29
30 if [ "$R" != "0" ]; then
31    tail -n 50 $LOG
32 fi
33
34 exit $R
```

Figure E.2: Script template to instrument unit tests.

```
1   <path id="dependencies.path">
2     <pathelement location="${dir.build}" />
3     <fileset dir="${dir.lib}">
4       <include name="jaxen-1.1-beta-7.jar" />
5       <include name="jakarta-oro-2.0.8.jar" />
6       <include name="xercesImpl-2.6.2.jar" />
7       <include name="xmlParserAPIs-2.6.2.jar" />
8     </fileset>
9                   <pathelement location="/nfs/home/home1/
                       dmohr/chi/eclipse/sootinstrument/bin" /
                       >
10                  <pathelement location="/nfs/home/home1/
                       dmohr/chi/eclipse/sootinstrument/boot-
                       bin" />
11                  <pathelement location="/nfs/home/home1/
                       dmohr/java/sharing/object-inlining/lib/
                       jdbc-postgresql.jar" />
12                  <pathelement location="/nfs/home/home1/
                       dmohr/java/sharing/object-inlining/lib/
                       jdbc-mysql.jar" />
13  </path>
14
```

Figure E.3: Example of classpath changes to the pmd build script.

```
1    <target name="test" depends="requires-junit,compile,copy
        " description="Runs␣the␣unit␣tests">
2      <junit printsummary="yes" haltonfailure="no" forkmode=
          "perTest">
3        <formatter type="xml"/>
4        <classpath>
5          <path refid="dependencies.path" />
6        </classpath>
7        <batchtest fork="yes" todir="${dir.build}"
            haltonerror="yes">
8          <fileset dir="${dir.build}">
9            <include name="test/**/*Test.class" />
10           <!-- exclude tests which make the
                 instrumentation fail -->
11           <exclude name="test/net/sourceforge/pmd/ast/
                ASTAnnotationTest.class" />
12         </fileset>
13       </batchtest>
14     </junit>
15 ...
16   </target>
17
```

Figure E.4: Example of changes to the unit test run in the pmd build script.

References

[1] Apache Software Foundation. Apache Maven Project. URL http://maven.apache.org/. [Online; accessed 2010-04-01].

[2] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN 0321278658.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

[4] Eric Bodden. Packs and phases in Soot, 2008. URL http://www.bodden.de/2008/11/26/soot-packs/. [Online; accessed 2010-02-19].

[5] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. *SIGPLAN Not.*, 37(5):25–32, 2002. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/543552.512534.

[6] Zoran Budimlic, Mackale Joyner, and Ken Kennedy. Improving Compilation of Java Scientific Applications. *International Journal of High Performance Computing Applications*, 21(3):251–265, 2007. doi: 10.1177/1094342007078437. URL http://hpc.sagepub.com/cgi/content/abstract/21/3/251.

REFERENCES

[7] Z. Budimlić. *Compiling Java for High Performance and the Internet*. PhD thesis, Rice University, 2001.

[8] CERN. Worldwide LHC computing grid, 2008. URL http://public.web.cern.ch/public/en/LHC/Computing-en.html. Retrieved on 2010-03-23.

[9] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *In 2nd International coference on Generative Programming and Component Engineering (GPCE '03), volume 2830 of Springer Lecture Notes in Computer Science*, pages 364–376. Springer-Verlag, 2003.

[10] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362/2005 of *LNCS*, pages 108–128. Springer Berlin / Heidelberg, 2005.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[12] Julian Dolby and Andrew A. Chien. An automatic object inlining optimization and its evaluation. In *PLDI 2000*, pages 345–357. ACM Press, 2000.

[13] Eclipse Foundation. Eclipse IDE. URL http://www.eclipse.org/. [Online; accessed 2010-04-01].

[14] Boby George. An initial investigation of test driven development in industry. In *ACM Sympoium on Applied Computing*, pages 1135–1139, 2003.

[15] Dirk Heuzeroth, Uwe Aßmann, Mircea Trifu, and Volker Kuttruff. The COMPOST, COMPASS, Inject/J and RECODER tool suite for invasive software composition: Invasive composition with COMPASS aspect-oriented connectors. In *Generative and Transformational Techniques in Software Engineering*, volume 4143/2006 of *Lecture Notes in Computer Science*, pages 357–377. Springer Berlin / Heidelberg, 2006. doi: 10.1007/11877028_14.

REFERENCES

[16] Martin Hirzel, Daniel von Dincklage, Amer Diwan, and Michael Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2), April 2007. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1216374.1216379. Article 11.

[17] JBoss, by Red Hat. Hibernate. URL http://hibernate.org/. [Online; accessed 2010-04-01].

[18] Ron E. Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201708426.

[19] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP 2001*, pages 327–353. Springer-Verlag, 2001.

[20] Peeter Laud. Analysis for object inlining in Java. In *In Proceedings of the Joses Workshop*, pages 1–8, 2001.

[21] Ondřej Lhoták and Laurie Hendren. Run-time evaluation of opportunities for object inlining in Java. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, pages 175–184. ACM Press, 2002. ISBN 1-58113-599-8.

[22] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. URL ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.

[23] Gordon E. Moore. Excerpts from a conversation with Gordon Moore: Moore's Law, 2005. URL ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf.

[24] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, 2009. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1508284.1508275.

[25] Nomair A. Naeem and Laurie Hendren. Programmer-friendly decompiled Java. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 327–336. IEEE Computer Society, 2006. ISBN 0-7695-2601-2. doi: http://dx.doi.org/10.1109/ICPC.2006.20.

[26] Erik Poll, Patrice Chalin, David Cok, Joe Kiniry, and Gary T. Leavens. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *In Formal Methods for Components and Objects, 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 342–363. Springer, 2006.

[27] Herb Sutter. The free lunch is over - a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005. URL http://www.gotw.ca/publications/concurrency-ddj.htm.

[28] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999. URL www.sable.mcgill.ca/publications.

[29] Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montreal, October 2000.

[30] Ronald Veldema, J. H. Ceriel, F. H. Rutger, and E. Henri. Object combining: A new aggressive optimization for object intensive programs. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 165–174, New York, NY, USA, 2002. ACM. ISBN 1-58113-599-8. doi: http://doi.acm.org/10.1145/583810.583829.

[31] Daniel von Dincklage. *Algorithmic Optimizations*. PhD thesis, University of Colorado, 2007.

[32] Chip Walter. Kryder's law. *Scientific American*, August 2005. URL http://www.scientificamerican.com/article.cfm?id=kryders-law.

[33] Wikipedia. Moore's law — Wikipedia, the free encyclopedia, November 2008. URL http://en.wikipedia.org/wiki/Moore%27s_law. [Online; accessed 2009-08-04].

[34] Laurie Williams, E. Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *In Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering*, pages 34–45. IEEE Computer Society, 2003.

REFERENCES

[35] Christian Wimmer and Hanspeter Mössenböck. Automatic feedback-directed object inlining in the Java HotSpot(tm) virtual machine. In *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 12–21, New York, NY, USA, June 2007. ACM. doi: http://doi.acm.org/10.1145/1254810. 1254813.

[36] Wm. A. Wulf and Sally A. Mckee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–24, 1995.