A Closer Look at Inter-library Dependencies in Java-software

Michael Thies
Universität-GH Paderborn, Fachbereich 17, Fürstenallee 11
D-33102 Paderborn, Germany
mthies@uni-paderborn.de

Abstract Optimizing Just-in-Time compilation of Java programs depends on information gained from state-of-the-art program analysis techniques. To avoid extensive analysis at program execution time, analysis results for libraries can be precomputed statically and then combined dynamically at runtime. This paper provides evidence that such a hybrid approach would in fact work well for larger programs. We have implemented a compositional variant of method side-effects analysis, which has been applied to three real world Java libraries. The dependency structure of the results and its meaning for compositional analysis is discussed. In addition the library interfaces exploited by specific client programs are examined.

1 Introduction

Platform-independence and flexibility are key benefits of the Java programming language[4] and its runtime environment, while lack of performance probably constitutes their greatest weakness. Separate compilation of Java source texts to machine independent byte-code ensures platform-independence. Flexibility stems from the fact that the Java virtual machine (JVM) loads, resolves and links classes dynamically only at program execution time[7].

The JVM conceptually contains an interpreter for the Java byte-code instruction set, but current implementations compile byte-code to native machine code just-in-time (JIT compiler) in order to improve performance. JIT-compilation works very well for coarsely structured code dealing with primitive datatypes. However, speed gains for inherently object-oriented code, which contains frequent calls to often very short methods, are disappointing. Because a JIT-compiler has to do its translation at runtime, it is limited to simple, local optimizations based on cheap, local program analysis.

Extensive optimizations of object-oriented code depend on information about larger parts of the class hierarchy to be optimized, e.g. a class with all its ancestors and subclasses. As that kind of global analysis is not feasible at runtime, we have suggested to perform global program analysis statically ahead of runtime and to augment the Java byte-code files with the results gained[12, 13].

In order to preserve the flexibility offered by the Java runtime environment, analysis will not consider the whole program as a monolithic, unchangeable entity. Instead analysis is restricted to the largest possible parts of software that will realistically always be reused and updated together: software components or libraries. A library consists of

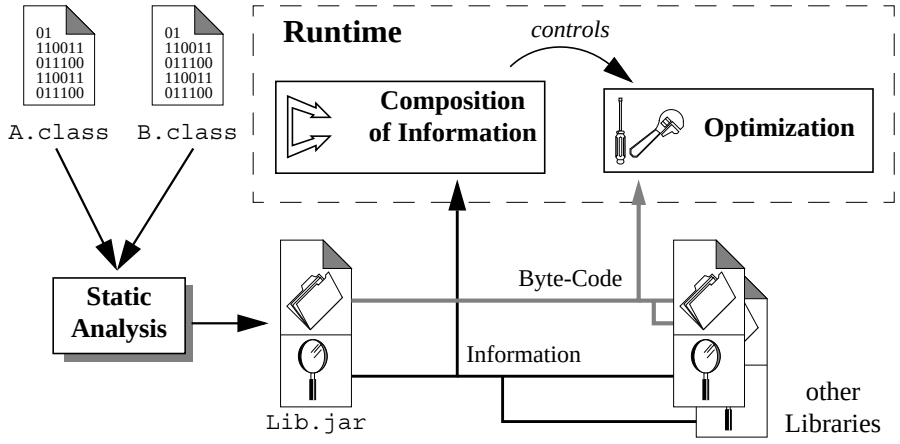

Figure 1: Static analysis of libraries and dynamic composition at runtime

multiple closely-related, complete Java packages, i.e. all classes and interfaces belonging to those packages must already be present in the library. All library members are analyzed together, augmented with the results and grouped into a Java archive file for distribution and deployment. Bundling all annotated library members into a single file prevents uncontrolled changes to the contained classes that could invalidate the computed analysis information. Furthermore use of archive files for the distribution of Java libraries is already in active use.

Naturally separate analysis of a single library cannot compute complete self-contained information. Interfaces between the library and external software components cause open spots to appear in the results, where missing information on other components has to be filled in at runtime. This adds slightly to the link stage performed by the JVM, which must also compose analysis information, when references across library boundaries are resolved. Figure 1 gives an impression of the resulting system architecture. However, restricting analysis to software components offers the following advantages — compared to whole-program analysis:

- Analysis results for a software component can be reused whenever the component itself is reused.
- Updating a software component requires just a single re-analysis of the component. All client programs utilizing the component will pick up any changed information automatically.
- Optimization is still possible, even if analysis results are not available for all parts of a program, e.g. small special purpose applications built on top of a large, common library.

The remainder of this paper is structured as follows: Section 2 introduces the specific analysis problem that we have studied. We focus on the treatment of incomplete software programs. Section 3 gives quantitative results gained for several Java libraries and their client applications, most notably the Swing library[11] and some real world appli-

cations utilizing Swing. Our primary interest is in the structure of the results, which influences the composition of the information at runtime. After discussing related work in section 4 we present our conclusions and some ideas for future work.

2 Side-effects analysis for method parameters

Our goal was to get an impression of, how well static analysis based on libraries and composition of results at runtime would perform for realistic libraries and application programs. Therefore we have implemented a relatively simple, albeit non-trivial example analysis, which records side-effects of method invocations on actual parameters, especially the invoking object (the receiver). For each method *m* analysis determines which actual parameters passed to *m* are potentially modified by an invocation of *m*. Modifications mainly occur as assignments to instance variables of parameter objects, either directly in the body of *m* or indirectly in other methods invoked from *m*. For the time being we ignore native methods which may also modify objects and are invisible to any analysis based on Java byte-code.

This specialized form of method side-effects analysis delivers information that can be used to control at runtime several classic optimizations. For example recognition of common subexpressions across method invocations and invariant code motion from loop bodies containing calls require this kind of information. Furthermore this analysis is closely tied to the call structure of the software to be analyzed and yields compact results. Thus it is ideal for our purpose, as it reveals the dependency structure inherent in the subject software and indeed could work well with our mixed static analysis/dynamic composition approach.

Our current implementation of this analysis proceeds in three major steps:

1. Information is computed locally for each method body.
2. Additional results for related families of methods are derived, i.e. a method combined with all its overriding implementations in subclasses.
3. Information on callees is utilized transitively to gain precomputed global information, as far as possible inside the context of an isolated library.

So initially our analysis computes for each method *m* and for each formal parameter p_i of *m* the predicate $\mathrm{Mod}(m, i)$ such that $\mathrm{Mod}(m, i) = \mathrm{true}$ iff an invocation of *m* may modify p_i. The algorithm can detect only possible modifications of p_i, because p_i might be changed or left unchanged depending on control flow inside *m*. An implicit parameter p_0 of all non-static methods is used to represent the receiver. There are three possible outcomes for the computation of each $\mathrm{Mod}(m, i)$. If an assignment to an instance variable of p_i (not to p_i itself) occurs in the body of *m*, $\mathrm{Mod}(m, i)$ will be true. Otherwise the result will be a disjunction of the form

$$\mathrm{Mod}(m, i) = \mathrm{Mod}(c_1, i_1) \vee \ldots \vee \mathrm{Mod}(c_k, i_k) \vee \mathrm{Mod}_{\mathrm{F}}(c_{k+1}, i_{k+1}) \vee \ldots \vee \mathrm{Mod}_{\mathrm{F}}(c_n, i_n)$$

where $c_1, \ldots, c_n$ are all the methods called by *m* which have access to p_i. Each callee c_j gets passed a reference to p_i of *m* as its own parameter p_{c_j, i_j} or can access such a reference via that parameter. There may be multiple disjunctive terms referring to the same callee, e.g. if p_i is passed in different parameter positions on different calls. Calls

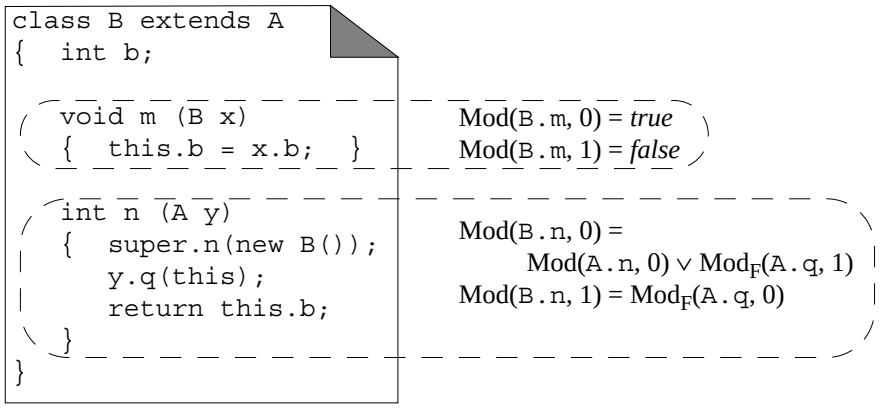

Figure 2: Examples of Mod computations for method parameters

to methods $c_1, \ldots, c_k$ are statically bound and the remaining callees use the predicate Mod_F (defined below) to model dynamic binding. An empty disjunction evaluates to false, which means that *m* neither modifies p_i by direct assignment nor makes p_i accessible to its callees (if any). Figure 2 shows some example methods with their Mod computations.

In order to supply information about call sites which make use of dynamic method binding, an extended predicate Mod_F is computed next. This predicate describes the effects of a whole family of related methods, which consists of a single method *m* defined in a certain class *C* and all overriding definitions of *m* in subclasses of *C*. It can be computed as follows:

$$\mathrm{Mod}_F(m, i) = \mathrm{Mod}(m, i) \vee \mathrm{Mod}(m_1, i) \vee \ldots \vee \mathrm{Mod}(m_t, i)$$

where $m_1, \ldots, m_t$ are all methods overriding *m* defined in subclasses of *C*. Note that Mod is only meaningful for concrete method implementations, while Mod_F also extends to abstract methods and methods defined in interfaces. In the latter case the family of related methods consists of all methods that implement a certain interface method.

Separate family results are computed for all methods, even if they belong to the same family, because analysis can then select information that describes just the set of callees possible under dynamic binding. To achieve this, the available static type information about the receiver of the method call determines the family member to consult for side-effects information. The results for single methods are still preserved independently, as they convey precise information on method invocations referring to `super` and for calls to constructor bodies. In addition to the aforementioned situations Java also uses static method binding for calls to static, private or final methods, but these yield identical Mod and Mod_F results anyway.

The last phase of our analysis precomputes as much information as possible inside the library. This means that Mod or Mod_F occurrences in disjunctions are substituted by their definitions, if they refer to library methods. Recursive method calls may lead

to cyclic term structures that can be ignored, because they do not contribute any information. Repeated substitution finally delivers results in which all non-constant equations consist solely of predicates applied to methods external to the library. These remaining predicate applications form the spots where information on other libraries (or the client application) has to be inserted at runtime so that the boolean equations can be solved.

Apart from methods implemented outside of the library there is one further area that requires composition of results at runtime: classes derived from library classes. Only method implementations visible beyond library boundaries are subject to this problem. Because our definition of a library requires the contained packages to be complete, just public or protected instance methods from public classes and methods defined in public interfaces need special treatment. These methods collectively form the inheritance boundary of the library. We introduce one virtual (non-existent) placeholder method per boundary method that is used to model the side-effects of overriding method implementations in derived classes outside the library. Thus the refined definition of the predicate Mod_F looks like this:

$$\mathrm{Mod}_F(m, i) = \mathrm{Mod}(m, i) \vee \mathrm{Mod}(m_1, i) \vee \ldots \vee \mathrm{Mod}(m_t, i) \vee \mathrm{Mod}_F(\hat{m}_1, i) \vee \ldots \vee \mathrm{Mod}_F(\hat{m}_s, i)$$

where $\hat{m}_1, \ldots, \hat{m}_s$ with $0 \le s \le t + 1$ are the placeholder methods mentioned above. Information on placeholder methods can also be precomputed at static analysis time, at least partially, from method family results for classes derived from the same immediate superclass outside the library.

Thus each library provides auxiliary information that supports propagating results upwards along the inheritance chain, and in exchange depends on other software components to synthesize similar information for classes further down the chain. With this addition the theoretical foundations to analyze method side-effects on their parameters for each software component separately are in place, but performance in practice will significantly depend on the structure of the analysis results obtained.

3 Structure of analysis results

We have implemented the special form of method side-effects analysis described in section 2 inside a prototypical software analysis environment. Our primary goal in developing this environment is easy exploration of different analysis algorithms in the context of real world software and problem specific visualization of results. The framework itself is written in Java and uses the JavaClass library[3] to decompose binary Java byte-code files.

Method side-effects were analyzed for three quite diverse Java libraries and some of their typical client applications. We chose version 1.1 of the well known Swing library, a library from Javasoft for building applications with graphical user interfaces[11]. This is currently one of the most popular libraries that does not use any native methods. At 1444 classes and interfaces containing 12074 method implementations, constructors and abstract method declarations it is also one of the largest libraries. The sheer size of

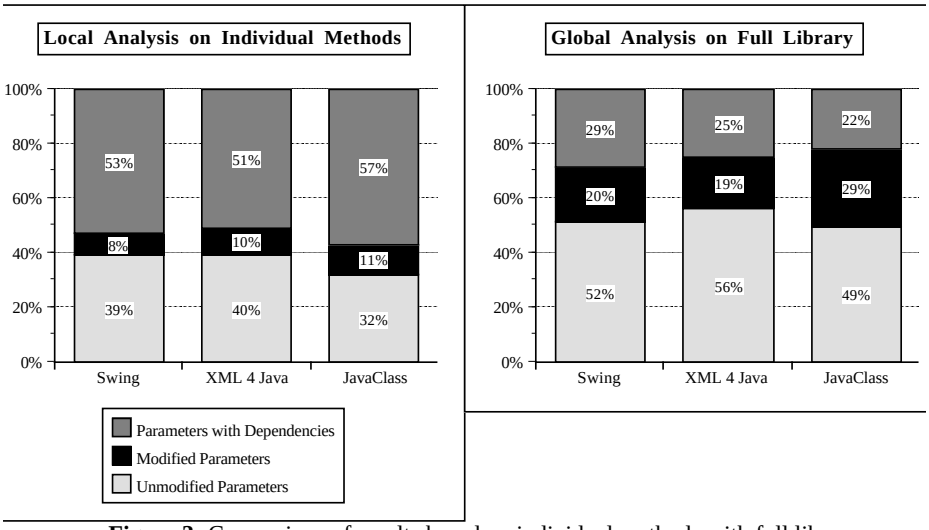

Figure 3: Comparison of results based on individual methods with full library

the complete library (nearly 2.1 MB for `swingall.jar`) means that local installation of Swing is required to make deployment practical. Of course this could as well be a statically analyzed and annotated version of the library.

In contrast the other two libraries are more specialized and thus substantially smaller. Version 1 of the IBM alphaWorks XML Parser for Java (`xml4j.jar`) is a validating parser for structured documents in the e**X**tended **M**eta **L**anguage[14]. It consists of 207 classes with 1902 methods and comes as a 0.55 MB archive file. Finally version 3.0 of the JavaClass library[3] —used in implementing the analysis environment itself— is made up of 275 classes containing 1399 methods in a 0.24 MB archive.

Figure 3 shows analysis results for the three libraries. It compares results based on separate local analysis of each method with equations already precomputed in the context of the library. Unmodified and modified parameters represent equations that evaluate to the constant values false and true respectively, both require no composition at runtime. In addition unmodified parameters increase the chance of optimization opportunities being discovered from this information. So parameter side-effects analysis based on an isolated library seems in fact to be worthwhile, as about 50% of the parameters immediately yield positive results allowing for latter optimizations. Of course this number will even increase after information has been composed across library boundaries so that information on parameters with dependencies is available.

Parameters with dependencies correspond to disjunctions referring to results for methods outside the analysis context. Between 45% and 61% of those can be resolved inside the library alone, reducing the work to be done at runtime correspondingly. A closer look at the structure of those equations helps to estimate the complexity of the composition process. Figure 4 relates the external method references in the three libraries to the methods referenced most frequently. It becomes obvious that typically information on a very small number of different external methods suffices to fill in a large

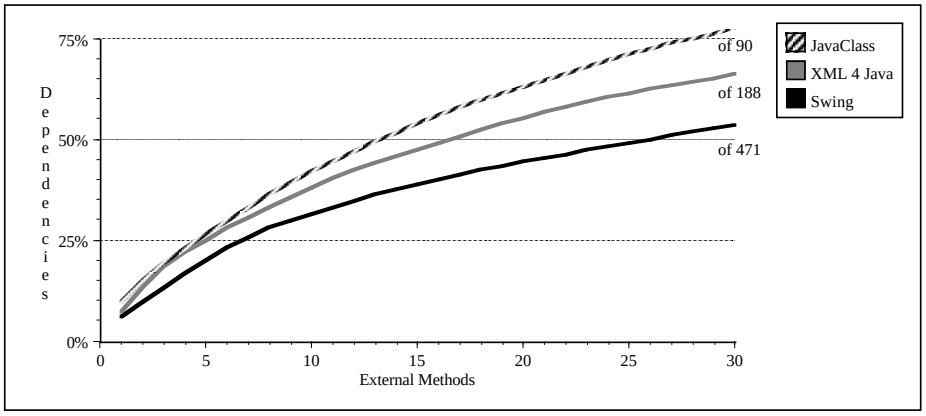

Figure 4: Frequency of method occurrences in parameter dependency equations

fraction of the missing terms in the equations. For example, in the results for the Swing library more than 50% of the remaining open spots refer to just 26 different external methods. This are just 6% of the 471 distinct external methods the Swing library results depend on.

So cheap (simultaneous) substitution of all identical term occurences throughout all equations would be desirable. One should design the data structures for storing analysis results as annotations to the library archive file with this in mind. However, even a straightforward design that stores parameter equations in additional classfile attributes near the byte-code of the methods would integrate quite well with the current resolution and linking stage of the JVM. Whenever a method name external to the library needs to be resolved, the appropriate method needs to be looked up and turned into a method reference, which indirectly refers to the method's byte-code. At the same time any analysis information stored for the method could be retrieved and substituted in the equations for the calling method.

Another important aspect is the amount of space required to represent the analysis information. Figure 5 shows the distribution for the length of the parameter disjunctions in terms. Only parameters with non-constant equations have been considered. About half of the disjunctions are made up of just a single term and only very few contain more

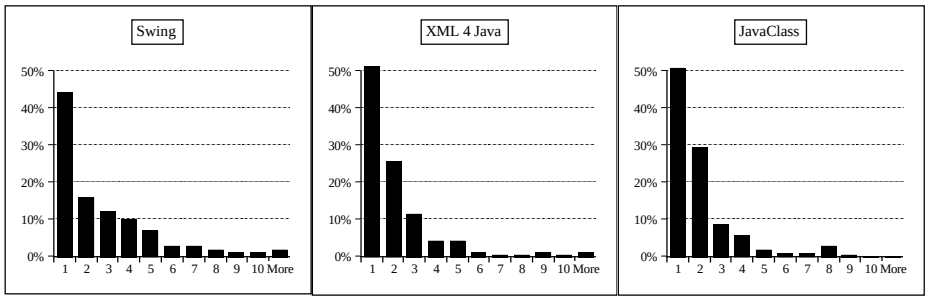

Figure 5: Length of parameter disjunctions in terms

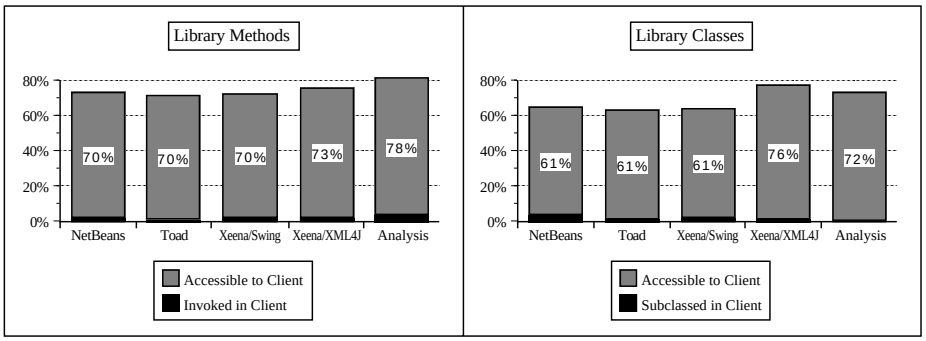

Figure 6: Width of interface between client applications and library

than 5 terms. A single term can be compactly represented inside a classfile, as the constant pool entries required for method resolution already define a mapping from fully qualified method names to small integers. These numbers could be used to encode applications of the Mod or Mod_F predicate.

For further space savings it would be valid to simply replace disjunctions longer than a certain threshold with the constant true, e.g. disjunctions of more than 5 terms. Then information would still represent a conservative approximation of reality. Beyond the size reduction in term representations, time is also saved for composition of those terms. The quality of the results is lowered only very slightly, because rather few equations would be subject to the modification. Moreover longer disjunctions have a significantly lower probability to finally evaluate to false, as required for optimizations.

Our examination of the three isolated libraries has shown that a significant amount of information (more than 70%) can be precomputed at static analysis time. Composition across library boundaries at runtime can be done quickly, as the remaining equations are short and depend only on relatively few different external methods.

Furthermore composition at runtime is of course not done eagerly, but only on demand, triggered by the library calls contained in the currently executing client application. So we examined the width of the interface between the three libraries and some of their typical, non-trivial client applications. Naturally there is a large selection of applications using Swing. We chose the development environment NetBeans[9], the IBM alphaWorks Toad environment for dynamic and static analysis of Java programs[8] and the visual XML document editor Xeena[6], again from IBM alphaWorks. Xeena is also a client of the XML parser library, while our analysis software itself makes use of the JavaClass library. The size of these applications ranges from 98 to 2496 classes, containing from 15,000 up to more than 300,000 byte-code instructions. All of them contain one method call every 5 to 6 byte-code instructions on average.

Figure 6 relates the interface actually exploited by an application to the full interface provided by a library. Two aspects were measured: First, the number of library methods possibly invoked from the client, because these calls trigger computation of complete results for the callees; second, the number of library classes/interfaces that are extended/implemented by application classes or interfaces. This represents the inheritance boundary between the two software components, where information for placeholder

methods has to be derived and utilized. In both diagrams 100% corresponds to the full code of the library, including private and package-scoped classes and methods. It turns out that only a small fraction (less than 5%) of the already acceptable composition work would be triggered by these client applications directly. Of course information on parts of the library that are reachable from there would have to be completed as well.

So the combination of static analysis that precomputes information internal to the library, and composition driven by the interface actually exploited by the client application seems to be a promising route to fast, accurate program analysis information.

4 Related work

Of course side-effects analysis in itself is nothing new. Clausen describes in [2] a traditional optimizer which performs full side-effects of method invocations. But his work is solely concerned with whole program analysis and does not need to support compositional analysis.

Precomputing analysis information and storing it as byte-code annotations is suggested by Azevedo, Hummel, Kolson, and Nicolau in [1]. However, they consider only single method bodies for analysis, which avoids any external information dependencies, but restricts the quality of the achievable results for many problems. The high call frequency present in Java programs precludes this approach for general analysis.

Hölzle and Agesen compare static analysis with dynamic program observation for type based optimizations in SELF programs. While both techniques produce results of comparable quality, when applied separately, they conclude that a combination of both approaches would be beneficial[5]. Again the static analysis part is based on the full program and separate analysis of software components is not considered.

The Java analyzer JAN — a software tool from the IBM alphaWorks Toad environment — performs class hierarchy analysis statically and allows analysis to build upon a subset of already pre-analyzed classes[8, 10]. This is used to avoid repeated processing of different versions of the Java standard libraries. JAN then constructs a call graph for the whole program as an aid for program understanding. As the standard library has no further dependencies on other libraries, JAN implements a restricted special case of the ideas evaluated in this paper. Furthermore fast deployment of precomputed information is a less prominent criterion in the context of software analysis compared to optimizations performed at runtime.

5 Conclusions and future work

We have implemented a compositional analysis for method side-effects on parameters in Java and applied it to three real world Java libraries. Our results show that separate static analysis of libraries allows for large amounts (≥70%) of information to be precomputed. The structure of the results for the libraries permits fast composition of information at runtime and compact storage of intermediate results. We have found in-

ter-library dependencies between libraries and non-trivial client applications that further reduce expected composition effort significantly. In conclusion, static analysis before runtime based on a single library generates accurate, reusable information that can quickly be adapted to the context in which the library is utilized.

Future plans include the implementation of additional compositional program analysis algorithms and the development of a Java runtime environment that performs composition of analysis results and uses them to control dynamic optimizations. Another area with room for improvements is the quality of the analysis results. More advanced techniques like type inference could deliver more precise information on dynamically bound method calls than the static type information from the byte-code that we currently use. Different levels of precision for library internal versus inter-library information would be another possibility to improve overall results without slowing down composition.

6 References

1. A. Azevedo, J. Hummel, D. Kolson, and A. Nicolau. Annotating the Java Bytecodes in Support of Optimization. ACM 1997 Workshop on Java for Science and Engineering Computation, Las Vegas, Nevada, June 1997.
2. Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. In *Concurrency: Practice and Experience*, 9(11):1031-1045, November 1997.
3. Markus Dahm. Byte Code Engineering with the JavaClass API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, May 1999. `http://www.inf.fu-berlin.de/~dahm/JavaClass`
4. James Gosling, Bill Joy, and Guy Steele. The Java Language Specification. In *The Java Series*. Addison-Wesley, 1996.
5. Urs Hölzle, Ole Agesen. Dynamic vs. Static Optimization Techniques for Object-Oriented Languages. In *Theory and Practice of Object Systems*, 1(3), 1996.
6. Shlomit S. Ifergan, Yoelle S. Maarek, and Sigalit Ur. Xeena: another alphaWorks technology. `http://www.alphaworks.ibm.com/tech/xeena`
7. Tim Lindholm, Frank Yellin. The Java Virtual Machine Specification. In *The Java Series*. Addison-Wesley, 1996.
8. Bilha Mendelson, Sara Porat, Matt Greenwood, and Doron Cohen. TOAD: another alphaWorks technology. `http://www.alphaworks.ibm.com/tech/toad`
9. Netbeans, Inc. Welcome to NetBeans. `http://www.netbeans.com`
10. Sara Porat, Bilha Mendelsoh, and Irina Shapira. Sharpening Global Static Analysis to Cope with Java. CASCON'98, Ontario, December 1998.
11. Sun Microsystems, Inc. Java(TM) Foundation Classes. `http://www.javasoft.com/products/jfc`
12. Michael Thies, Uwe Kastens. Statische Analyse von Bibliotheken als Grundlage dynamischer Optimierung. In *JIT'98 Java-Informations-Tage 1998*, Springer, November 1998.
13. Michael Thies. Static compositional analysis of libraries in support of dynamic optimization. Technischer Bericht, Reihe Informatik, University of Paderborn, August 1999.
14. XML Technology Group. XML Parser for Java: another alphaWorks technology. `http://www.alphaworks.ibm.com/tech/xml`