



Pós-Graduação em Ciência da Computação

## **“Otimização Bytecode Java na Plataforma J2ME”**

**Por**

***Tarcisio Pinto Camara***

**Dissertação de Mestrado**



Universidade Federal de Pernambuco  
posgraduacao@cin.ufpe.br  
[www.cin.ufpe.br/~posgraduacao](http://www.cin.ufpe.br/~posgraduacao)

RECIFE, AGOSTO/2004



UNIVERSIDADE FEDERAL DE PERNAMBUCO  
CENTRO DE INFORMÁTICA  
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TARCISIO PINTO CAMARA

“Otimização Bytecode Java na Plataforma J2ME ”

*ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM  
CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA  
DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO  
REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE  
EM CIÊNCIA DA COMPUTAÇÃO.*

*ORIENTADOR(A): Prof. Dr. André L. M. Santos  
CO-ORIENTADOR(A): Prof. Dr. Geber Lisboa Ramalho*

RECIFE, AGOSTO/2004

## ACKNOWLEDGMENTS

I thank God and my parents for my life, my education and my principles.

I thank a lot my teachers, André Santos and Geber Ramanho, for all dedication, attention and motivation that made this work possible. I also thank teachers of the evaluation commission, Paulo Borba and Ricardo Massa, for attention and contributions for this work.

I could not miss to thank Cleber Zanchettin and Joel da Silva, two new brothers that life gave to me, for being my family during this journey. Thanks guys.

Finally, I would like to thank Eric Lafortune, for developing and publishing ProGuard, as well as C.E.S.A.R/Meantime for opening the source code of the J2ME applications we have used in our tests and allowing to us to publish results about them; and I also thank CNPq (Brazilian Research Council) for supporting this research project.

## RESUMO

Aplicações para os dispositivos móveis, como telefones celulares e *paggers*, implementadas em J2ME (*Java 2 Micro Edition*) são desenvolvidas sob severas restrições de tamanho e desempenho do código. A indústria tem adotado ferramentas de otimização, como *obfuscators* e *shrinkers*, que aplicam otimizações de programa inteiro (*Whole Program Optimizations*) considerando que o código gerado não será estendido ou usado por outras aplicações. Infelizmente, os desenvolvedores freqüentemente não conhecem suficientemente nestas ferramentas e continuam sacrificando a qualidade do código na tentativa de otimizar suas aplicações. Este trabalho apresenta um estudo original identificando a efetividade das otimizações mais comuns nos *obfuscators*. Este estudo mostra também que a otimização de *Method Inlining*, conhecida pelos benefícios de desempenho, tem sido negligenciada por estas ferramentas por normalmente esperar-se que ela tenha efeito negativo sobre o tamanho de código. Assim, este trabalho contribui com uma implementação de *method inlining* entre classes e fundada no princípio de otimização de programa inteiro, capaz de melhorar tanto o tamanho do código como o desempenho da aplicação, ao remover cerca de 50% dos métodos alcançáveis. Finalmente, na tentativa de ajudar os desenvolvedores a tirar o melhor proveito destas ferramentas, o estudo inclui também um guia de boas práticas de programação considerando as otimizações implementadas pelos *obfuscators*.

**Palavras-chave:** 1. Compressão de Dados. 2. Ofuscadores de Bytecode J2ME. 3. Method Inlining. 4. Boas Práticas de Programação

# ABSTRACT

Applications for mobile devices, like cell phones and pagers, implemented in the *J2ME Platform* (Java 2 Micro Edition) are developed under strong performance and code size constraints. Industry has been adopting optimization tools, such as obfuscators and shrinkers, which apply *Whole Program Optimizations* considering that the generated code will not be extended or used by other applications. Unfortunately, developers often do not know these tools enough and keep sacrificing code quality in order to optimize their applications. This work presents an original study identifying the effectiveness of the most common optimizations in the obfuscators. This study has shown us that *Method Inlining*, an important optimization with known performance benefits, has been disregarded by these tools, since it often has negative effects on code size. Thus, this work contributes with a cross-module whole-program method inlining implementation that improves both performance and application code size, while removing around 50% of the reachable methods. Finally, in order to help developers to take the best advantage of these tools, we have also included a best programming practices guide considering the optimizations implemented by obfuscators.

**Keywords:** 1. Data Compression. 2. J2ME Bytecode Obfuscators. 3. Method Inlining. 4. Best Programming Practices

# TABLE OF CONTENTS

<b>1</b>	<b>INTRODUCTION.....</b>	<b>1</b>
1.1	WORK DESCRIPTION .....	2
1.2	DOCUMENT STRUCTURE.....	3
<b>2</b>	<b>JAVA 2 MICRO EDITION.....</b>	<b>4</b>
2.1	J2ME HISTORY .....	4
2.2	J2ME ARCHITECTURE .....	6
2.3	KVM/CLDC CONSTRAINTS .....	8
<b>3</b>	<b>WHOLE PROGRAM OPTIMIZATION .....</b>	<b>11</b>
3.1	WHOLE PROGRAM OPTIMIZATION IN J2ME .....	11
3.2	WHOLE PROGRAM OPTIMIZATION IN OBFUSCATORS .....	13
3.3	METHOD INLINING IN OBFUSCATORS.....	17
3.4	FINAL REMARKS .....	18
<b>4</b>	<b>METHOD INLINING AND THE PROBLEM OF CODE SIZE INCREASE .....</b>	<b>20</b>
4.1	INTERNAL STRUCTURE OF THE JAVA VIRTUAL MACHINE .....	21
4.2	BYTECODE EXPANSION WHEN COPYING .....	23
4.2.1	<i>Creation of temporary variables .....</i>	<i>24</i>
4.2.2	<i>Re-indexing variable instructions.....</i>	<i>24</i>
4.2.3	<i>Jump instructions.....</i>	<i>25</i>
4.2.4	<i>Replacing return by goto .....</i>	<i>25</i>
4.2.5	<i>Switches .....</i>	<i>25</i>
4.2.6	<i>Accesses to the constant pool .....</i>	<i>26</i>
4.3	THE DECISION ALGORITHM.....	26
4.3.1	<i>Call graph.....</i>	<i>26</i>
4.3.2	<i>Restrictions imposed by the Java Virtual Machine.....</i>	<i>28</i>
<b>5</b>	<b>PROPOSED METHOD INLINING TECHNIQUE .....</b>	<b>30</b>
5.1	IMPLEMENTATION DECISIONS.....	31
5.2	COPY AND MODIFICATION OF THE BYTECODE .....	32
5.3	DECISION ALGORITHM .....	36
<b>6</b>	<b>EXPERIMENTAL RESULTS.....</b>	<b>42</b>
6.1	EXPERIMENTS SETUP .....	42
6.2	PARAMETERIZATION .....	43
6.3	OPTIMIZATION OCCURRENCE .....	45
6.4	EXECUTION MEMORY AND PERFORMANCE .....	48

6.5	CODE SIZE REDUCTION BY OBFUSCATORS .....	51
<b>7</b>	<b>BEST PROGRAMMING PRACTICES .....</b>	<b>53</b>
7.1	SITUATIONS ALREADY RESOLVED BY OBFUSCATORS .....	54
7.1.1	<i>No identifier needs to be shorted.....</i>	<i>54</i>
7.1.2	<i>Unused features in frameworks do not need to be suppressed .....</i>	<i>54</i>
7.1.3	<i>Primitive type constant values do not need to be replaced by hand .....</i>	<i>55</i>
7.1.4	<i>Fields do not need to be made public to avoid field access methods (get and set).....</i>	<i>56</i>
7.1.5	<i>Long methods can be divided into small context methods called once.....</i>	<i>56</i>
7.2	SITUATIONS NOT RESOLVED BY OBFUSCATORS .....	57
7.2.1	<i>Constant propagation still unavailable .....</i>	<i>57</i>
7.2.2	<i>Dead code elimination still unavailable .....</i>	<i>57</i>
7.2.3	<i>Control flow analysis still unavailable .....</i>	<i>57</i>
7.2.4	<i>Devirtualization still unavailable .....</i>	<i>58</i>
7.2.5	<i>Merging of classes still unavailable .....</i>	<i>58</i>
7.2.6	<i>Call graph considers objects instantiated anywhere, not only locally .....</i>	<i>58</i>
7.3	SITUATIONS THAT JEOPARDIZE OBFUSCATORS .....	59
7.3.1	<i>Reflection API usage .....</i>	<i>59</i>
7.3.2	<i>Relative resource addressing.....</i>	<i>59</i>
7.3.3	<i>Unnecessary code.....</i>	<i>60</i>
7.3.4	<i>Throwing and catching exceptions .....</i>	<i>60</i>
7.3.5	<i>Synchronization .....</i>	<i>60</i>
7.3.6	<i>Switches .....</i>	<i>60</i>
7.4	OTHER RECOMMENDATIONS .....	61
7.4.1	<i>Do not initialize big arrays in line.....</i>	<i>61</i>
7.4.2	<i>Types byte, short, char and boolean are usually converted to int .....</i>	<i>61</i>
7.4.3	<i>Avoid nested and anonymous classes .....</i>	<i>61</i>
7.4.4	<i>Reuse objects .....</i>	<i>62</i>
<b>8</b>	<b>RELATED WORKS .....</b>	<b>63</b>
8.1	LANGUAGE-INDEPENDENT METHOD INLINING RESEARCH .....	63
8.2	METHOD INLINING IN COMPILERS AND TOOLS .....	64
8.3	RELATED WORKS OF BEST PROGRAMMING PRACTICES .....	66
<b>9</b>	<b>CONCLUSION .....</b>	<b>67</b>
9.1	CONTRIBUTIONS.....	67
9.2	FUTURE WORK .....	68
	<b>REFERENCES.....</b>	<b>71</b>

# FIGURES

FIGURE 2-1: JAVA 2 EDITIONS AND THEIR TARGET MARKETS [KVMWP].....	6
FIGURE 2-3: J2ME SOFTWARE LAYER STACK [KVMWP] .....	7
FIGURE 2-5: VARIANTS OF THE JAVA PLATFORM FOR SMALL DEVICES [ORTIZ, 2002] .....	8
FIGURE 3-1: BUILD PROCESS FOR J2ME APPLICATIONS. ....	13
FIGURE 4-1: JVM RUNTIME DATA AREAS. ....	22
FIGURE 4-3: EXAMPLE OF METHOD INLINING WITH TEMPORARY VARIABLES. ....	23
FIGURE 4-4: RTA EXAMPLE. ....	27
FIGURE 5-1: MAKE ABSTRACT EXAMPLE.....	32
FIGURE 5-2: EXAMPLE OF METHOD INLINING WITHOUT TEMPORARY VARIABLES.....	33
FIGURE 5-3: EXAMPLE OF VARIABLE RE-INDEXING. ....	34
FIGURE 5-4: EXAMPLE OF CODE PREPARATION. ....	35
FIGURE 5-5: EXAMPLE OF CALL GRAPH STRUCTURE FOR A METHOD. ....	37
FIGURE 5-6: DECISION ALGORITHM ACTIVITIES. ....	37
FIGURE 5-7: FORMULAE OF CODE SIZE INCREASE ESTIMATION.....	38
FIGURE 6-1: CODE SIZE REDUCTION BY OBFUSCATORS. ....	51



# TABLES

TABLE 3-1: OPTIMIZATIONS IMPLEMENTED BY ANALYZED OBFUSCATORS.....	15
TABLE 3-2: OPTIMIZATIONS OCCURRENCE IN OBFUSCATORS. ....	16
TABLE 3-3: OPTIMIZATION MECHANISMS USED BY OBFUSCATORS.....	17
TABLE 6-1: BYTECODE SIZE REDUCTION BY ALGORITHM PARAMETERIZATION. ....	44
TABLE 6-2: NUMBER OF INLINED METHODS. ....	46
TABLE 6-3: NUMBER OF INLINED CALL SITES. ....	47
TABLE 6-4: REDUCTION ON TOTAL MEMORY ALLOCATED.....	49
TABLE 6-5: APPLICATION PERFORMANCE IMPROVEMENT. ....	50



# INTRODUCTION

This new century is witnessing a new trend in computer science research: ubiquitous or pervasive computing. According to it, computation will be increasingly embedded in mobile devices (such as cell phones and pagers), providing to users relevant information and services anytime and anywhere. A myriad of applications from which users can daily benefit are being developed, ranging from simple e-mail systems to complex applications, such as intelligent Personal Digital Assistants, interactive multiplayer games, e-commerce location-sensitive transactions systems, and so on [Hansmann, 2003].

The main platform used by the industry for programming mobile devices is Java 2 Micro Edition (J2ME), a smaller version of the Java Standard Edition (J2SE) in order to fit it into the strong execution memory and processing constraints of these devices<sup>1</sup>. These constraints often force J2ME developers to sacrifice object-oriented principles and many software quality recommendations, such as code legibility and ease of maintenance, in order to reduce the number of classes, methods and fields of the application and consequently its processing requirements.

A solution to face these constraints is to use *Whole Program Optimizations* [Dean, 1996], where the application is globally transformed, considering that the optimizing tool knows the entire application code. Industry has been increasingly adopting this approach, using tools like obfuscators or shrinkers.

Obfuscators were originally implemented to make reverse engineering difficult, while applying some automatic transformations like reducing names of classes, packages and class members (methods and fields). As a side effect, these optimizations also make the programs smaller. Nowadays, some of these tools, sometimes called shrinkers, have included other optimization techniques, specifically aiming at reducing application size, like: removal of unused classes and members; and flattening the class hierarchy.

---

<sup>1</sup> The J2ME applications must be very small (often up to 50 kilobytes) [Knudsen, 2002b] and they have to consider processing capacity as low as that of 25 MHz processors [KVMds].

In spite of obfuscators being very popular in the J2ME community, it is easy to find developers that do not know these tools enough and keep sacrificing code quality in order to optimize their applications.

## ***WORK DESCRIPTION***

This work presents an original study identifying optimizations most common in the most popular obfuscators and where their implementations differ. It also identifies new optimizations being implemented and gives guidelines about what else could be taken into account to select a tool.

This study highlights that method inlining optimization has been neglected by these tools for frequently having as a side effect the increasing of the code size. Method inlining, a well-known optimization, consists basically of choosing a certain set of call sites and replacing them by the code of the called method. This optimization presents a proved performance gain [Dean & Chambers, 1994], by avoiding the call overhead and exposing other optimizations. It is a good solution for automatically resolving many situations the programmers try to avoid by hand.

This work introduces a novel cross-module and whole-program method inlining optimization technique for the Java Virtual Machine. This technique both *improves the performance of the application and performs some code size reduction, while removing around 50% of the reachable methods*. This was possible by exploring, on one side, the low level characteristics of the Java Virtual Machine, in particular the way a method body is copied, and, on the other hand, considering the possibility of removing methods when all their call sites have been replaced.

For several years, our research team has been providing consulting services on J2ME for industrial scale applications [CESAR/Meantime]. This experience, combined with the knowledge acquired in this work concerning the internal structure of obfuscators, shown us that using the best optimizations is not enough to assure J2ME applications as smaller and faster as possible. *It is essential for programmers to know the capabilities of the adopted tool in order to avoid unnecessary design sacrifices and to improve optimization results*. A lot of effort is wasted avoiding situations already dealt automatically, while other programming practices may confuse the optimization algorithms.

In order to reduce this problem, this work also introduces best practices for programming J2ME applications considering the optimizations implemented by

obfuscators. To our knowledge, the technical literature has not covered these issues so far. The practices are organized so that we highlight situations not resolved by obfuscators, as well as situations handled by them and practices that jeopardize their usefulness. These best practices have been used by an industrial software development team in CESAR/Meantime [CESAR/Meantime], our partner.

## *DOCUMENT STRUCTURE*

The next chapters lead the discussions in the rest of the work.

Chapter 0 presents the state of the art of the J2ME Technology, including its importance as a Java Technology, its architecture and constraints. This is important to justify extreme developers care about application size and performance.

Chapter 0 discusses the whole program optimizations and introduces our empirical study identifying which ones are most common in obfuscators and where their implementations differ. The chapter also comments why the J2ME platform benefit this kind of optimization.

Chapter 0 details the method inlining optimization and the problem of the code size expansion. This chapter contributes with a full analysis of low level factors of the Java Virtual Machine involved in the increase of code size during the optimization.

Chapter 0 introduces our proposed technique for implementing method inlining optimization, while detailing how we have combined a low level approach and the whole-program assumption to face each factor of code expansion.

Chapter 0 discusses our experiments to evaluate the impact of the proposed technique on the application code size, execution performance and memory, as well as the surprising reduction in the number of methods and calls after optimization.

Chapter 0 presents a best programming practices guide, developed considering our study about the optimizations available by obfuscators and our experience while extending one of them.

Chapter 0 lists the most important related work, including method inlining language-independent researches, and method inlining implementations in compilers and tools. The chapter also presents some works on best programming practices guides.

Finally, Chapter 0 discusses some conclusions of the work, presenting the most important contributions of this research and some future works.

## JAVA 2 MICRO EDITION

The main platform used by the industry for programming mobile devices is Java 2 Micro Edition (J2ME). Some of the main benefits of using Java in these devices are:

- *Hardware-independence*: standardizing resources provided by the virtual machine allows applications to be executed in several different devices, since they share the same virtual machine.
- *Dynamic content*: new applications and new application versions can be installed and configured in the devices, instead of being restrict just to preloaded ones. It allows better adequacy to user needs.
- *Developer community*: application development is not limited to device manufacturers. Any Java developer can, with some effort, implement applications to devices previously unavailable.
- *Security*: applications security and validation can also be found in J2ME. Language constraints forbid illegal access to critical device resources, avoiding wrong or malicious instructions.

This chapter presents the state of the art of the J2ME Technology, including its importance as a Java Technology, its current architecture and possible trends.

### *J2ME HISTORY*

The first version of the Java language was developed as part of the Sun's Green Project [Green] in order to create products to small computers and devices. Due to its strong portability, its potential for more complex computer environments was evident, and each new version added more and more features (Applets, AWT, RMI, JDBC, Serialization, Reflection, etc).

Soon, the number of required classes and resources made it unavailable to be implemented by simpler devices, like cell phones, *paggers*, PDAs (Personal Digital Assistance), radios, televisions, etc. The software development to this emergent business domain was kept limited to manufacturer laboratories, often using hardware-dependent low-level programming.

In the first four years of the Java platform, the language became the main choice for internet applications. Sun has introduced other technologies addressed to small devices, but some of them did not have the expected acceptance:

- The *JavaCard* platform (1996) [JavaCard] defines a very small Java environment for *smart cards* e *Java rings* [Cameron & Day, 1998]. This platform is still well used by interested community, but it is too small to cover the needs of applications for larger devices, like cell phones.
- The *PersonalJava* platform (1997) [PersonalJava] implements an environment just a little smaller than the traditional one used in personal computers (PC's). It addresses devices like portable computers and advanced PDA's. Unfortunately, the environment was too large to be embedded in cell phones, pagers and some simple PDA's.
- The *EmbeddedJava* platform (1998) [EmbeddedJava] was addressed towards embedded systems manufacturers, so that they could define which resources would be provided by the device's virtual machine. This flexibility made applications hardware-dependent and its development limited to the manufacturers themselves.

Recognizing that the Java architecture needed some radical reorganization [CLDC 1.0; Appendix1], Sun regrouped Java technologies in three editions, each one addressed to a specific business range:

- *Java 2 Platform Enterprise Edition (J2EE)* [J2EE]: For enterprises needing to provide Internet business server solutions.
- *Java 2 Platform Standard Edition (J2SE)* [J2SE]: For the desktop market, where applications do not need advanced features.
- *Java 2 Platform Micro Edition (J2ME)* [J2ME]: For consumer and embedded device manufacturers, as well as service providers who wish to deliver content to these devices.

Each edition defines a set of development tools, like libraries and APIs (Application Programming Interfaces), together with a Java virtual machine properly scaled for the execution environment. Figure 0-1 presents the three Java editions and their relations with available virtual machines.

Since its initial versions, the J2ME Platform was well accepted by the Java community. Actually, each new feature or API is identified as a Java Specification Request (JSR) and produced by a consortium of huge industrial partners, including device manufacturers, service and content providers and software companies. In the

end, the J2ME Platform has been a successful return to the initial goal of the Green Project, while making Java available for pervasive computing.

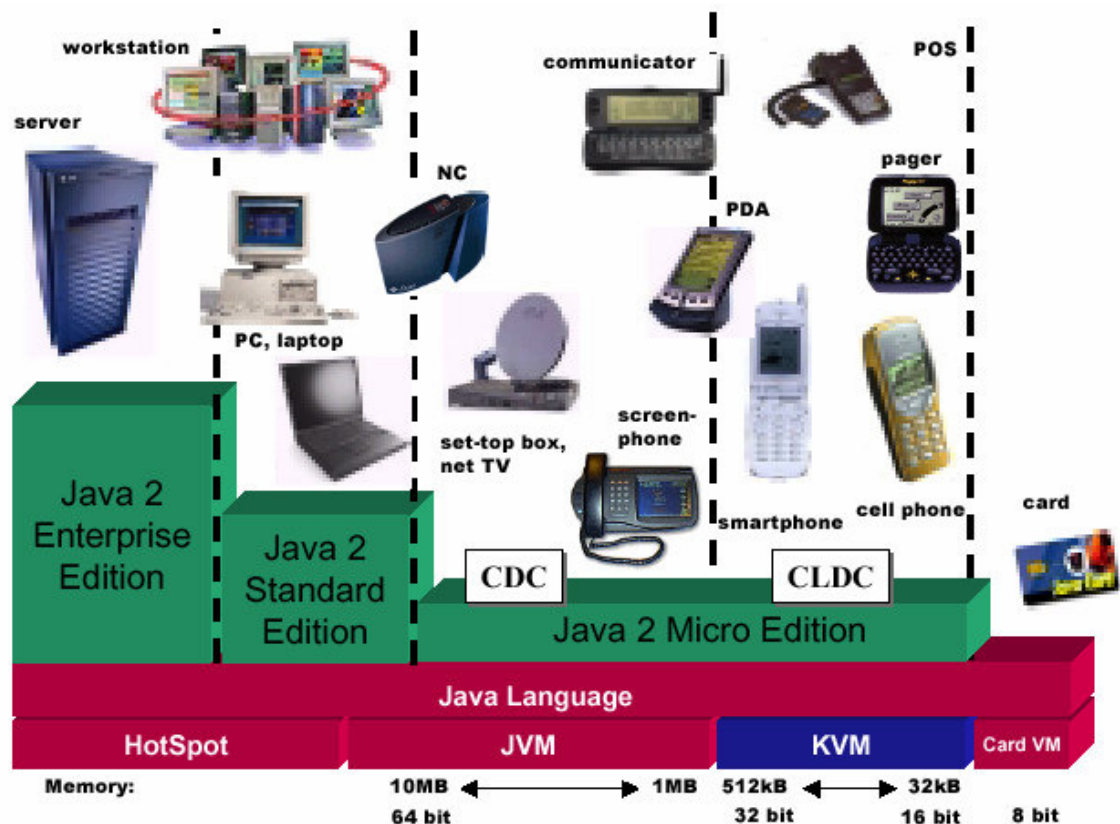


Figure 0-1: Java 2 editions and their target markets [KVMwp]

## J2ME ARCHITECTURE

Due to the wide range of hardware and execution environments that J2ME targets, its architecture was designed to be modular and scalable, allowing flexibility while defining which features must be available for each device class. These requirements were modeled in three software layers that must be built on the Host Operating System of the device:

- *Java Virtual Machine Layer:* This layer is an implementation of a Java virtual machine that is customized for a particular device's host operating system and supports a particular J2ME configuration.
- *Configuration Layer:* The configuration layer defines the minimum set of Java virtual machine features and Java class libraries available on a category of devices and market segment. A device can support only one configuration.
- *Profile Layer:* The profile layer defines the minimum set of APIs available on a particular "family" of devices. Profiles are implemented upon a



particular configuration. Applications are written for a particular profile and are thus portable to any device that supports that profile. A device can support multiple profiles.

Figure 0-2 presents a graphical representation of the J2ME architecture layers:

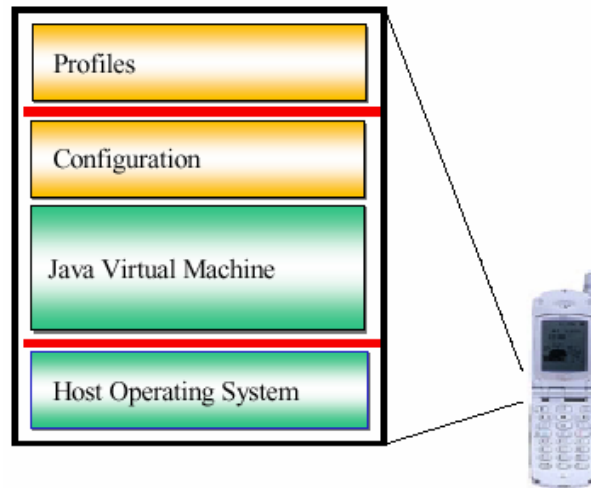


Figure 0-2: J2ME software layer stack [KVMwp]

The implementations of the configurations and virtual machines are always very closely aligned. Together they are designed to capture just the essential capabilities of each category of devices. Any differentiation into devices families must be specified in the profile layer. Nowadays, there are only two available configurations:

- *Connected Device Configuration (CDC)* [CDC 1.0]: The CDC uses a compact version of the JVM (*Java Virtual Machine*) with all classical resources and features, but with some constraints about memory usage. This configuration targets devices that provide at least a few megabytes of memory to the Java environment.
- *Connected Limited Device Configuration (CLDC)* [CLDC 1.0] [CLDC 1.1]: The CLDC uses a limited virtual machine named KVM (*Kilobyte Virtual Machine* or *K Virtual Machine*) and targets devices with several processing and memory constraints, making available only a few kilobytes of memory to Java.

Figure 0-1 also presents these two configurations and their relations with each virtual machine.

As the profiles address market segments, the number of profiles currently available and being developed are much wider than the number of configurations and they are continuously being reorganized. Figure 0-3 presents the current relationship among all Java technologies for small devices, including profiles and configurations.

*Personal Profile* stack, composed by CDC [CDC 1.0], Foundation Profile [FP 1.0] Personal Basis Profile [PBP 1.0] and Personal Profile [PP 1.0], has been developed upon CDC in order to provide an environment similar to the PersonalJava Technology [PersonalJava].

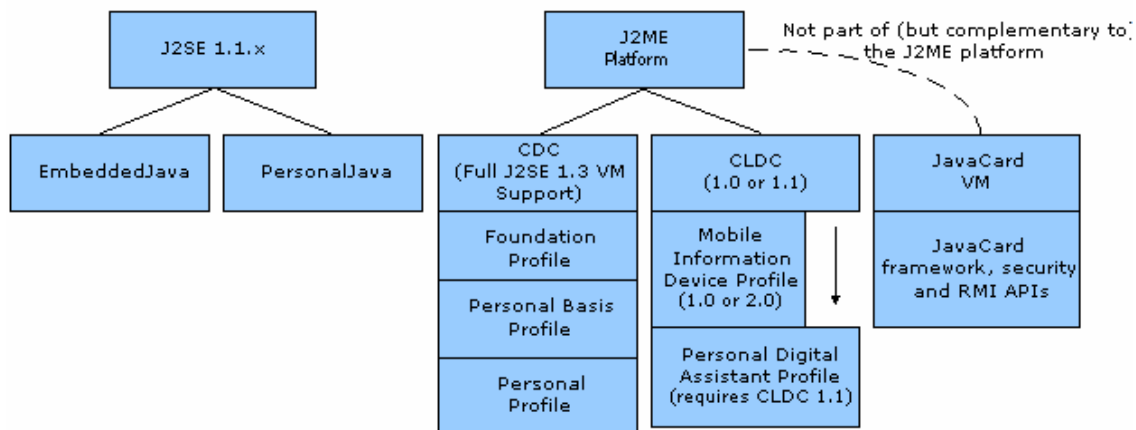


Figure 0-3: Variants of the Java Platform for small devices [Ortiz, 2002]

*Mobile Information Device Profile* (MIDP), the first and most popular profile, was developed upon CLDC addressed to *Mobile Information Devices* (MIDs), like cell phones, pagers and some simple PDAs. It is already in the second version [MIDP 2.0] and there are a number of compatible devices being sold in the world. Besides, MIDP has been used as basis for other profiles, like *Personal Digital Assistant Profile* (PDAP) that only provides some optional packages specifically for PDA applications [PDAP].

Thus, CLDC and MIDP standards are assumedly a key part of the J2ME Technology [PDAP] and they can be considered the most basic, popular and well-established J2ME stack. However, they are also one of the most limited Java execution environments, imposing several critical constraints. These constraints, which justify and require aggressive optimizations, will be discussed in the next section.

## KVM/CLDC CONSTRAINTS

In order to allow J2ME developers to create portable applications, the profile, configuration and virtual machine specifications require minimal resources and libraries that devices must make available to be compatible.

The high-level design goal for the KVM was to create the smallest possible “complete” Java virtual machine that would maintain all the central aspects of the Java programming language [KVMwp]. However, KVM was designed for small devices that

typically contain 16-bit or 32-bit processors, clocked as low as 25 MHz, and a minimum total memory footprint of approximately 128 kilobytes [KVMds]. Regarding these requirements, KVM implementation uses to have only 50-80 kilobytes of object code and needs only a few tens of kilobytes of dynamic memory to run [KVMwp]. In spite of its reduced size in memory, not much memory is left for the applications. It is possible to find devices that reject applications larger than 50 kilobytes, including bytecode and resources [Knudsen, 2002b]. Such small execution environment justifies extreme developer care about application size and performance.

As presented before, CLDC specification has currently two versions [CLDC 1.0] [CLDC 1.1]. Each one defines the subset of the Java programming language and virtual machine features that the device must provide. CLDC Specification version 1.0 defines that the supported KVM must be fully compatible with the standard *Java Virtual Machine Specification* [Lindholm & Yellin, 1999], except for the following differences:

- No floating point support
- No user-defined class loaders
- No thread groups and daemon threads
- No finalization of class instances (method `Object.finalize()`)
- Many exception and error classes are not available
- No native methods (Java Native Interface - JNI)
- No reflection (package `java.lang.reflect`)
- No weak references (class `java.lang.ref.WeakReference`)

The CLDC Specification version 1.1 consists in an incremental release that is intended to be fully backwards compatible with CLDC Specification version 1.0 but also to address slightly bigger devices. It does not include any new major changes, just adding requirements for some features, like floating-point support and some minor library changes to make it more compatible with J2SE.

Except for these constraints, the KVM supporting each CLDC specification must be fully compatible with the standard *Java Virtual Machine Specification* [Lindholm & Yellin, 1999], including the standard *classfile* format. In fact, there is not a specific compiler for J2ME. The application is compiled on the same way, however, before

being installed in the device, it is submitted to a *pre-verification* process which checks if the constraints imposed by the configuration were satisfied. That procedure still inserts some marks in the *classfile* to ease the class loading and validation tasks of the device's operating system.

Finally, both versions of the MIDP Specification [MIDP 1.0] [MIDP 2.0] were designed assuming only CLDC 1.0 features, so that they will also work on top of CLDC 1.1, and presumably any newer versions. This means that, even considering the probable MIDP stack evolution, J2ME developers must expect a severely constrained execution environment, when compared with standard Java platform. In such an environment, aggressive optimizations are unavoidable, not only to improve the application performance, but also to reduce the application size.

## WHOLE PROGRAM OPTIMIZATION

*Whole Program Optimizations* consider that the optimizing tool knows the entire application code and transform it based on this assumption. They can generate a great part of the code assuming that it will not be extended or imported by other applications. For example, the optimizing tool can safely resolve many polymorphic calls, remove unused code, or rename and rearrange many classes, methods and fields.

Of course the external interface of the application cannot be fully optimized, since it can be called by libraries or by the Virtual Machine itself. For example, initial methods called by KVM engine, such as `startApp`, cannot be renamed or removed and they can be called an undetermined number of times.

As not all public methods are really external interface of the application, the optimizing tools usually offer some way (like scripts) to define which classes, methods and fields cannot be optimized. In general, these entries are also considered as possible start points used to trace reachable code of the application.

Whole program optimizations could be perfectly included as part of the compiler. However the industry, especially J2ME community, has opted to implement these optimizations in obfuscators and shrinkers, keeping the compiler simpler and supporting all Java Platforms (e.g. J2ME and J2SE). Nowadays, some obfuscators have included more and more optimizations for several purposes.

This chapter discusses the state of the art of *whole program optimizations* focusing on demand of J2ME for optimization. Section 0 discusses why we can safely apply this kind of optimizations on most J2ME applications. Sections 0 present our empirical study, identifying what optimizations are most common in the most popular obfuscators and where their implementations differ. Section 0 details the existing method inlining implementations, exploring experimentally which kind of opportunity each tool can deal with. Section 0 discusses some final remarks about this chapter.

### *WHOLE PROGRAM OPTIMIZATION IN J2ME*

*Whole program optimizations* can be safely applied on the most important J2ME environment: the CLDC and MIDP stack. The security model defined in the CLDC

Specification [CLDC 1.0] [CLDC 1.1] and the application model defined in the MIDP Specification [MIDP 1.0] [MIDP 2.0] forbid that applications interact with each other after downloaded and installed in the device. Unfortunately, the total amount of code devoted to security in Java 2 Standard Edition far exceeds the memory budget available for a Java virtual machine supporting CLDC. Therefore, some compromises and simplifications were necessary.

CLDC application-level security model uses a metaphor of a closed “sandbox” that ensures the system libraries are closed and predefined by CLDC, profiles (such as MIDP) and manufacturer-specific classes specifications. This security model also specifies that a Java application can load application classes only from its own Java Archive (JAR) file. These restrictions (about system and application classes loading) mean the application programmer can consider that no class will be dynamically loaded in execution time other than those ones already considered in design time. It seems improbable these restrictions change even in early future versions of the specification. As described in CLDC 1.1 Specification [CLDC 1.1], the expert group members were generally satisfied with the previous versions of the specification, and did not see any need for radical changes in the new ones. The MIDP application model even allows multiple applications to be delivered in one JAR file, called *MIDlets Suite*. In these cases, each application, called *MIDlet*, can interact with each other sharing data and code, however the set of system and classes of all applications of the suite are still predefined in the context of the JAR file, on the same way as described by CLDC security model.

Note that dynamic class loading is still available in J2ME, through the method `Class.forName(String className)`. This method allows that programmers access a class by its name (for example, to instantiate it). Even in this case, the loaded class must be inside the application JAR file or it must be one of the system classes. However, the optimizing tool is not able to automatically identify these classes as part of the application, because they are not directly referenced and the class name can be programmatically built in the execution time. In this case, optimizing tools use to make available some way to programmers indicate which classes are accessed using this mechanism. This will be better discussed in Section 0.

## WHOLE PROGRAM OPTIMIZATION IN OBFUSCATORS

*Obfuscators* were originally developed to make reverse engineering difficult, replacing human-readable identifiers inside the Java *classfile* with meaningless short strings, making the resulting applications more difficult to understand through *decompilation*<sup>1</sup>. As a side effect, these obfuscators also made the programs smaller [RetroGuard]. Soon, industry noticed it was possible to employ other optimizations in order to reduce even more the application size and to improve the execution performance.

In this work, we use the term *obfuscator* for any tool that automatically employs *Whole Program Optimizations*, so that the submitted program is globally transformed, considering other programs will not use the generated code.

Obfuscators are often included in the build process between the compilation and pre-verification steps, acting directly over the already compiled bytecode. This approach allows obfuscators to optimize applications even when no source code is available. Of course, optimizations previously performed by the compiler are automatically kept in the final version of the application. Figure 0-1 introduces a graphic representation of the applications build cycle in J2ME.

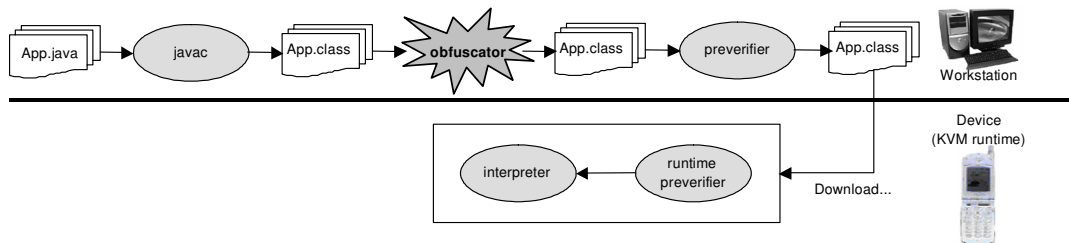


Figure 0-1: Build process for J2ME applications.

Unfortunately, optimizing directly the bytecode has also some implementation disadvantages. Sometimes, optimizations implemented by the obfuscator can open opportunities to optimizations previously performed by compilers. In this case, they cannot be reused and must be reimplemented on bytecode level.

Depending on the optimizations, obfuscator's developers have also to include additional steps to extract high-level information from bytecode, such as call graph and flow graph. These structures are often already available in compilers.

<sup>1</sup> *Decompilation* is a reverse engineering process that generates the source code while interpreting the application bytecode.

There are many available tools that can be classified as obfuscators. We studied some of the most popular tools, such as RetroGuard [RetroGuard], DashO [DashO], Jshrink [Jshrink], Jax [Jax] and ProGuard [Lafortune], identifying the optimizations they announced as implemented or as future work.

RetroGuard [RetroGuard] was developed by Retrologic as an open source project with the main goal of making applications harder to reverse engineering. It became popular because it is pre-installed in the Sun's J2ME Wireless Toolkit [J2MEwtk 1.0.4]. We used version 1.1.9.

Jshrink [Jshrink] is a commercial tool developed by Eastridge Technology and includes a good graphical user interface for configuration and reverse engineering. It is listed by ProGuard as one of the commercial alternatives tools. We used version 2.19 and an evaluation license for the experiments.

ProGuard [Lafortune] was developed by Eric Lafortune as an open source extension of RetroGuard with the main goal of making applications smaller. It became popular because it is also pre-installed in the Sun's J2ME Wireless Toolkit [J2MEwtk 1.0.4]. We used version 1.7.2 for the experiments but ProGuard has new versions published frequently.

Jax [Jax] is a research project developed by IBM [Tip et al., 1999] [Tip & Palsberg, 2000], implementing additional complex optimizations, like *merging adjacent superclasses*. Unfortunately, its source code is not public. Nowadays the Jax project is being discontinued and integrated to the IBM's development environment. We have tested Jax version 7.3a then available for download and free for use.

DashO [DashO] is a commercial tool developed by preEmptive Solutions that also has similar tools for .NET architecture. We used a copy of DashO Embedded Edition version 1.0 with an evaluation license for the experiments.

Table 0-1 presents the name and a short description of the optimizations we found in obfuscators, grouped by its main goal.

Optimizations	Description
<b>Optimizations against reverse engineering</b>	
Classfile recreation	It removes unused constant pool entries and attributes used to store compiler information, such as line number of the source code and local variables names.
Class and member names compression	It replaces the class, field and method names with short names (often one letter). Overloading (methods with the same name) is used whenever possible.
Package name compression	It replaces all or part of the package name with short names (often one letter), so that the grouping of classes is kept.



Class package relocation	It moves all optimized classes to the default package, which has no name. If the moved class accesses some <code>package</code> or <code>protected</code> member in an unmoved class, these members must be made <code>public</code> to prevent access violation. This optimization acquires better results than <i>Package name compression</i> .
<b>Optimizations for program size reduction</b>	
Removal of unused elements	It traces and removes classes, fields and methods not referenced directly or indirectly from some start method, such as <code>startApp()</code> .
Removal of write-only fields	It traces and removes fields that are only written but never read. The instructions that wrote to the field are removed too, but the instructions that evaluated to the assigned value are often kept, because they can include some side effect.
Removal of unused method body	It makes a method abstract if its body is never executed but the method cannot be removed for any reason. For example, if the method implements some interface and it is called virtually, but it does not belong to an instantiated class.
Merging adjacent superclass	It removes intermediate classes in the class hierarchy, moving all methods and fields of a class to its superclass. In order to keep the size of the resulting objects, many conditions must be satisfied, for example: either the superclass is not instantiated or the subclass has no fields.
Merging static classes	It detects classes where all class members (fields and methods) are static and moves all them to another class (static or not). All references to previous members should be moved properly. Then, source static class can be removed from application. This optimization is simpler than <i>Merging adjacent superclass</i> one and it can virtually be applied more times, since it requires less conditions.
<b>Optimizations for execution performance improvement</b>	
Devirtualization	It replaces slower virtual call instructions with faster static call instructions. In order to do that, methods are made static and private when possible.
Method inlining	It replaces some method calls with the code of the called method. It is only possible for calls that reach only one method (non-polymorphic).
Intra-procedural optimizations	Optimizations inside the method body, such as <i>constant folding</i> (evaluation of constant expressions in compile time) or <i>dead code elimination</i> (removal of write-only variables and unreachable branches) [Nullstone, 2002]. Some other optimizations, like method inlining, can create opportunities for these optimizations.

Table 0-1: Optimizations implemented by analyzed obfuscators.

Some other interesting and promising optimizations, such as *merging classes instantiated only once*, was not mentioned by any obfuscator at all.

For each of the found optimizations, we prepared an example application that was submitted to all evaluated tools, even if the optimization was not mentioned in the tools' documentation. Then, we decompiled the resulting applications in order to validate the effect over the bytecode. Table 0-2 presents the list of the actual optimizations implemented in the obfuscators. Looking at the table, we noted that there is a lake of optimizations for execution performance improvement.

Our analysis of the generated bytecode showed that almost all optimizations have insignificant differences when implemented by different obfuscators (marked with “X”). The only two interesting exceptions are *package name compression* and *method inlining*, which presented different results (marked with “?”). The different implementations of these two optimizations are discussed in the following paragraphs.

Optimizations	RetroGuard	JShrink	ProGuard 1.7.2	Jax 7.3	DashO EE
Classfile recreation	X	X	X	X	X
Class and member names compression	X	X	X	X	X
Package name compression	?	?	?		
Class package relocation			X	X	X
Removal of unused elements		X	X	X	X
Removal of write-only fields				X	X
Removal of unused method body				X	
Merging adjacent superclass				X	
Merging static classes				X	
Devirtualization				X	
Method inlining				?	?
Intra-procedural optimizations					

Table 0-2: Optimizations occurrence in obfuscators.

*Package name compression* was implemented in different ways by the tools. RetroGuard implemented it so that only each word of the package path is compressed, keeping the number of levels of the package tree. Jshrink opted for replacing the whole package path with one letter, but keeping the groupings of classes of each package. ProGuard optionally allows all optimized classes to be moved to one user-specified package, similar to class *package relocation optimization*, but if so the groupings of classes are lost.

*Method inlining* results were even more distinct. Actually, the DashO documentation does not identify *method inlining* as having been implemented, however we found some indications of inlined method in reports generated by the tool, but only for trivial instance field access methods (non static get and set). Jax *method inlining* was mentioned briefly in some articles as being only a secondary goal and applied for

methods whose only function is to set or retrieve a field's value [Tip et al, 1999]. Understanding that method inlining is a very important optimization, we refined its analysis, as presented in the next section.

### *METHOD INLINING IN OBFUSCATORS*

In order to identify the scope of the existing implementations, we opted for exploring experimentally the results of the tools (in this case DashO and Jax) when applied to carefully controlled situations, representing opportunities we consider promising to method inlining optimization. These opportunities are very simple programming situations, however sometimes they are hard to be isolated in real applications. As we are just interested in studying how far tools deal with such simple situation, we opted to prepare some simple programs isolating each situation. We submitted programs to the tools, and investigated resulting code.

For the experiments performed here, we used version Jax 7.3 and an evaluation copy of the DashO Embedded Edition. Both of them were configured so that all optimizations over method names were disabled, allowing the reverse engineering process of the resulting programs. Table 0-3 presents sample methods, number of calls to them and what tools dealt with them in any way, analyzing the generated bytecode.

Oportunities/Obfuscators		Number of calls	Examples	Jax 7.3	DahsO EE
Instance	Trivial field access methods	3	<code>public int getIndex () { return this.index; }</code>		X
	Array field access methods	3	<code>public int getItem (int i) { return this.items[i]; }</code>		
	Empty methods	1	<code>public void doNothing () { }</code>		
	Methods called once	1	<code>public void doSomething () { ... }</code>		
Static	Trivial field access methods	3	<code>public <b>static</b> int getIndex () { return this.index; }</code>	X	
	Array field access methods	3	<code>public <b>static</b> int getItem (int i) { return this.items[i]; }</code>	X	
	Empty methods	1	<code>public <b>static</b> void doNothing () { }</code>	X	
	Methods called once	1	<code>public <b>static</b> void doSomething () { ... }</code>		

Table 0-3: Optimization mechanisms used by obfuscators.

While elaborating the examples, we noticed that, apparently, the tools handle **static** and **instance** methods differently. We then created additional versions of the examples, also exploring this factor.

**Trivial field access methods** is an example that recovers the value of a field or sets a value to it, being a good example of optimization opportunity without creating temporary variables, even if they have been called several times. Notice that a simple additional comparison can make the code non-trivial, demanding the creation of temporary variables. This will be explained in section 0.

**Array field access methods**, either one-dimensional or multidimensional, are good examples of small non-trivial methods where generation of temporary variables is needed. However, even if the methods are called several times, the removal of its header can compensate for the code expansion during the copying of the bytecode.

**Empty methods**, since they are not part of interfaces implementation or a polymorphic call, they can also be removed. In this case, the optimization simply removes the method and all its call sites.

**Methods called only once** usually can be optimized and removed, regardless of their size.

In our experiments, Jax dealt with many examples including non-trivial methods but we were unable to obtain any effect over instance methods, only static ones. DashO dealt with only trivial field access methods and only its instance version. Anyway the obfuscators seem to have neglected method inlining while tried to apply it only where it surely does not increase the application size.

## ***FINAL REMARKS***

Our results should not be used to classify the obfuscators or identify the best one. We are only interested in (i) identifying what optimizations are more common and (ii) identifying what is the trend of new implementations. There are many other factors that must be taken into account to choose a tool, as presented above:

- *Configuration flexibility*: all obfuscator must provide some way for the user to identify which pieces of the code (classes and members) must not be optimized and the start point of the application, used to construct the call graph. Usually, this is done by configuration script for each application being optimized. However, a flexible configuration script language can allow the user to reuse the scripts in several applications with the same architecture.

- *Graphical user interface*: many obfuscators provide a graphical user interface at least to help the user in the construction of the configuration scripts. However, this interface can be very powerful, including even reverse engineering tasks to allow the user to choose the unchanged code elements graphically.
- *Development environment integration*: in a software production environment, it is usual to integrate the obfuscator with the IDE (Interactive Development Environment) or some build tool, like ANT [Ant Project]. Therefore, it can be automatically executed through the development lifecycle. Some obfuscators have plug-ins to the most popular IDEs or they provide command line interfaces to easy integration.
- *Project continuity*: see if the project is really alive and its delivery of new versions and bugs fixes, as well as if there is user support available. It is always possible to choose another tool afterwards, but this can impose some work to retrain the developers and to update configuration scripts and the integration with the development environment.
- *Documentation*: check if the obfuscator has good documentation about how to use it, how to write configuration scripts and how to integrate it with the development environment. We propose the documentation should include best programming practices too, as described in Chapter 0.
- *Price/License*: of course, pay attention to the license agreement and what is needed to get new versions of the tool. Many obfuscators are GNU projects that grant free rights for use and modification. For commercial tools, they use to publish trial version free for use but often with expiration deadlines.

## METHOD INLINING AND THE PROBLEM OF CODE SIZE INCREASE

Method inlining essentially consists of two parts: (i) a decision algorithm that selects a set of method calls (call sites) to be optimized and (ii) a copy mechanism that replaces the selected call sites with the code of the method being called [Serrano, 1997].

This optimization usually brings a direct improvement on the performance of the application, since it removes the overhead for method call and return. In other words, it potentially removes all the activities related to managing the call context stack (frames). Another important method inlining benefit is to open opportunities for other intra-procedural optimizations, like *constant folding* and *dead code elimination*, since they usually can only work on continuous code blocks between calls.

Unfortunately, this technique also has a direct impact over application code size, since inlining a method replicates its code in all the selected call sites, causing code expansion. The increase of code size can also degrade application performance sometimes, because that can cause “*thrashing*” on demand-paged virtual-memory systems. In other words, if the executable size is too big, the system can spend most of its time going out to disk to fetch the next piece of code<sup>1</sup> [Cline, 2003].

As presented in Section 0, most results presented in the literature explore a generic decision algorithm with the objective of maximizing the performance of the application while trying to control code expansion. Many solutions are often too general, disregarding the languages’ implementation specificities. We believe that in order to overcome the problem of the increase in the application code size properly, it is necessary to take also into account the features of the underlying implementation of the programming language. In our case, we explored the runtime and bytecode features of the Java Virtual Machine to trace and to control the exact impact of the changes performed by the optimization over the application size.

---

<sup>1</sup> It is not clear if *thrashing* is really a problem for the J2ME platform, because the memory management policy is implementation-dependent and it can be specially designed for small devices [CLDC 1.0, Section 5.4.5] [Knudsen, 2002b].

Therefore, next sections of this chapter discuss the low-level Java features related somehow to method inlining optimization and to the problem of code size increase. In Section 0 we explain some of the internal structures used by the Java Virtual Machine (JVM) to keep the stack of method calls. Section 0 presents an extensive list of low level details of the JVM related to the size of the bytecode to be copied during inlining. We then present in Section 0 the factors that must be considered in any decision algorithm, discussing the influence of the call graph and the restrictions imposed by the virtual machine.

### *INTERNAL STRUCTURE OF THE JAVA VIRTUAL MACHINE*

The specification of the Java Virtual Machine [Lindholm & Yellin, 1999] defines many data structures to manage the execution of applications. Among these structures we are particularly interested in those related to controlling method calls, notably frames, operand stack, local variables array and invoke instructions. Figure 0-1 shows a graphical representation of these JVM runtime data areas, as defined in JVM Specification [Lindholm & Yellin, 1999].

The frames are managed by each thread. They are used to store the context of the method call. Each frame contains its own array of local variables and its own operand stack. The maximum size of these two structures is defined in the method code.

The array of local variables stores the value of the parameters and local variables of the method. If it is an instance method (non static), the zero index position stores a reference to the object `this`. The positions next to it store the values of the parameters followed by the values of the local variables, in a way that variables of type `double` or `long` use two positions in the array, while the other basic types use a single position.

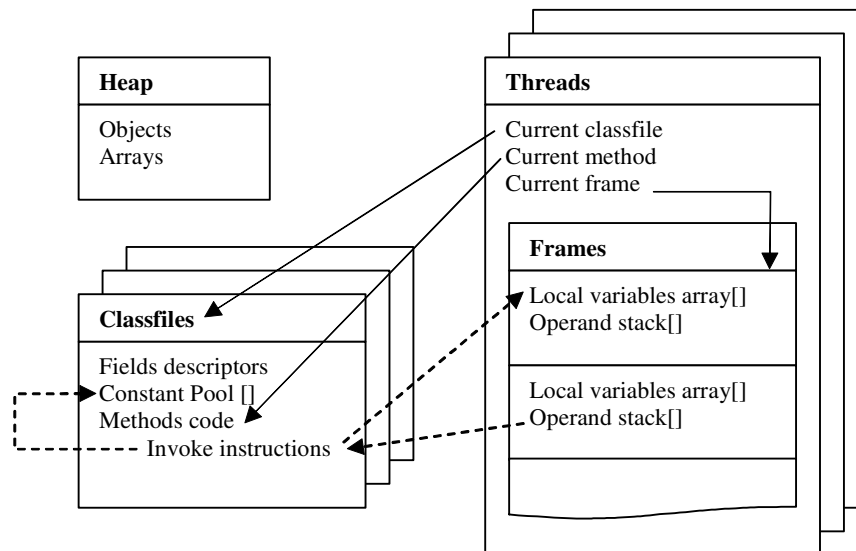


Figure 0-1: JVM Runtime data areas.

The operand stack stores the partial values resulting from the execution of the instructions. Each instruction takes its parameter(s) from the top of the stack, uses them according to the instruction semantics and eventually pushes back on the top of the stack its result. For instance, the family of `load_n` instructions is responsible for copying the value of variable  $n$  to the top of the stack. Similarly, `store_n` instructions are responsible for assigning to variable  $n$  the value on the top of the operand stack.

`Invoke` instructions are responsible for method calls. The instruction takes the method arguments from the top of the operand stack, including the reference to the called object (for non-static methods), and initializes a new frame and its array of local variables. When returning from the execution of the method, the virtual machine removes the frame and places the result returned by the method on the top of the stack.

The constant pool is also an important characteristic of the Java Virtual Machine for our work, since it is one of the structures that most contributes to application size. The constant pool is a table, present in every Java class file, containing symbolic information of all the elements accessed by the class, such as constant values (e.g. strings) and references to methods and fields from the class itself or from other classes it references. The self-sufficiency of the classfile has historical reasons: it was designed this way to ease class distribution over a network (Java applets). But this creates a significant amount of duplication of entries in the constant pools in the classes of an application. For example, if many different classes access a method from a class, each “client” class will have an entry in the constant pool referencing (naming) the method,



including its name, type descriptor and the name of the class where it is defined. Notice that calls to the same method *in a given classfile* share a single entry in the constant pool. The removal of methods through inlining, therefore, produces a direct benefit over size of this structure, since it also removes entries in the constant pool related to the declaration and calls to the method.

### BYTECODE EXPANSION WHEN COPYING

In general, the copied bytecode needs to be modified before being inserted in the code of the calling method, and many of these changes may also increase the application size.

The following sections describe these and other changes that impact on the size of the copied code, of the modified caller method code and, consequently, of the entire application. To illustrate the discussion, Figure 0-2 presents a simple and trivial example of method inlining.

Caller		Callee		Modified caller	
<b>void caller()</b>		<b>void callee(int i, int value)</b>		<b>void caller()</b>	
<b>Source code</b>	<b>Variables</b>	<b>Source code</b>	<b>Variables</b>	<b>Source code</b>	<b>Variables</b>
if ( 1 > 0 ) { callee(2, 0); }	0 this	this.f[i] = value;	0 this 1 i 2 value	if ( 1 > 0 ) { Class v1 = this; int v2 = i; int v3 = value; v1.f[v2] = v3; }	0 this 1 v1 2 v2 3 v3
<b>Bytecode</b>	<b>Op.Stack</b>	<b>Bytecode</b>	<b>Op.Stack</b>	<b>Bytecode</b>	<b>Op.Stack</b>
0   iconst_1	1	0   aload_0	this	0   iconst_1	1
1   iconst_0	1; 0	1   getfield #3	f	1   iconst_0	1; 0
2   if_icmple 11		4   iload_1	f; i	2   if_icmple 18	
5   aload_0	this	5   iload_2	f; i; value	5   aload_0	this
6   iconst_2	this; 2	6   iastore		6   iconst_2	this; 2
7   iconst_0	this; 2; 0	7   return		7   iconst_0	this; 2; 0
8   invokevirtual #2				8   istore_3	this; 2
11   return				9   istore_2	this
				10   istore_1	
				11   aload_1	this
				12   getfield #3	f
				15   iload_2	f; i
				16   iload_3	f; i; value
				17   iastore	
				18   return	

Figure 0-2: Example of method inlining with temporary variables.

The first column represents the caller method, the second, the callee, and the third, the modified caller after expansion due to inlining. The variables section

indicated in each column include, beside their names, the indexes used by variable instructions. Each column also shows, beside bytecode, the state of operand stack after each instruction.

In this example, it was necessary to create a temporary variable for each parameter. As well, it was also necessary to re-index the instructions that access the variables, and to adjust the offset of the jump instructions in the modified caller method (`if_icmple` instruction). These and other transformations will be detailed in the next sections.

### *Creation of temporary variables*

To create the local temporary variables, it is necessary to insert `store` instructions before the copied code in the array, in order to transfer the method arguments in the operand stack to the array of local variables. The size of each `store` instruction depends on the index of the variable it refers to, and it can use from one to four bytes [Lindholm & Yellin, 1999].

It may be necessary to generate an additional `checkcast` instruction before the `store` instruction that stores the reference to the called object, guaranteeing that the type of the created variable is compatible with the type of the object `this` expected by the callee code. This is necessary when the callee method is reachable through polymorphism. Notice that the call graph must also guarantee that only one method is reachable.

In principle, some subsequent intraprocedural optimizations could optimize the copied code and remove some of the created variables. These variable optimizations are often available in compilers, acting on source code or intermediate code level. These optimizations could be perfectly implemented on bytecode level, however they were not covered by this work. Actually, we believe that is a promising future work as shown in Section 0.

### *Re-indexing variable instructions*

Once the parameters and local variables of the callee method are mapped into temporary variables in the modified caller method, all instructions accessing variables in the copied bytecode must be re-indexed to access the new variables.

Since the size of the instructions that access variables (such as `load` and `store`) depends on the index of the variable, the modified bytecode sequence may

become longer than the original bytecode. In the example in Figure 0-2 this did not take place, but as the frequency of these instructions is very high, the impact may be significant.

### *Jump instructions*

The offsets of the `jump` instructions in the caller code and in the copied code must be adjusted to take into account the new inlined code.

In the example of Figure 1, the instruction `if_icmple` of the caller method had to be adjusted; now referring to a new offset (19). In theory it might be necessary to replace instructions like `goto` (3 bytes), by a wider instruction, like `goto_w` (5 bytes), depending on the new offset. However, the current version of the specification of the Java Virtual Machine imposes a restriction on the maximum size of a method bytecode [Lindholm & Yellin, 1999] that removes the need to use instructions like `goto_w`. Actually, `goto_w` is reserved for future versions of the Java Virtual Machine.

In short, the re-indexation of the jump instructions has no impact in code size.

### *Replacing `return` by `goto`*

If there are `return` instructions (1 byte) in the copied code, they must be replaced by `goto` instructions (3 bytes), redirecting the flow to the instruction following the copied code.

In the example of Figure 0-2 this did not happen. We even applied a small improvement on the copied code, removing the last `return` in the code, since the value returned by the method would be already in the operand stack.

### *Switches*

The `tableswitch` and `lookupswitch` instructions can vary their size depending on the place where they are in the bytecode. Their offset tables must be aligned to an address that is a multiple of 4 [Lindholm & Yellin, 1999].

Thus switches in the copied code need to be modified to fit their new location in the modified caller method. In the same way, switches in the caller method may be moved if some method inlining is done before its location in the code, inserting new code. In both cases these modifications may end up increasing the size of the switches' code.

### *Accesses to the constant pool*

Method inlining is often performed between methods of different classes. In this case, all entries in the constant pool accessed by the copied code must also be copied to the class that declares the caller method, if they do not exist there yet.

Entries in the constant pool are one of the main factors contributing to the size of a classfile. The replication of these entries through the application can generate a significant impact in its global size. This is one of the reasons why the number of classes increases a lot the application size. The precise measure of this impact is difficult because it is possible to share them in the scope of each classfile.

### *THE DECISION ALGORITHM*

The implementation of the decision algorithm, responsible for the choice of which method calls will be inlined, is the most complex part of method inlining optimization [Serrano, 1997]. In general, if many calls are selected to be inlined, the benefits for the performance of the application are bigger, but on the other hand each inlined call has the potential of duplicating the copied code, increasing the size of the application.

Although the algorithm demands a large set of specialized information for its parameterization, the analysis is based on a graph, namely the call graph, representing all the possible method calls.

In Section 4.3.1 we present the construction of the call graph and its relevance for method inlining optimization. Then the next section details the restrictions imposed by the Java Virtual Machine itself that must be taken into account for any decision algorithm, regardless of its objective or of the heuristics used.

### *Call graph*

The major difficulty in the construction of the call graph is the identification of virtual calls that can reach more than one method, through polymorphism. For example, in a call to  $e.m()$  the algorithm must decide which of the possible implementations of  $m$  may be executed from the evaluation of the expression  $e$ .

The call graph is particularly important for the method inlining optimization. Polymorphic calls cannot be directly optimized, since it is not possible to define precisely the implementation of the method that will be executed. There is also a technique, named customization, able to transform polymorphic calls in sequences of

monomorphic calls, by inserting code to test the type of the expression before calling each of the possibly reachable method [Whitlock, 2000]. This technique is not explored in the context of this work since it results in an even longer code sequence.

There are many algorithms for the construction of the call graph available in the literature. Some of the most relevant ones are CHA [Dean et al, 1995], RTA [Bacon, 1997], XTA and k-FCA [Tip & Palsberg, 2000]. All of them start from the entry point of the application and traverse recursively the methods, analyzing the instructions that invoke methods. At each call, the method is tagged as reachable and its body will be analyzed later. Methods that are never called are left tagged as unreachable.

The CHA (Class Hierarchy Analysis) is the simplest algorithm. It takes into account only the class hierarchy to determine the possible executions of a method call. However, most of the whole program tools rely on the RTA (Rapid Type Analysis) algorithm [Bacon, 1997] for this task, since its implementation is relatively straightforward and it also presents an acceptable approximation of the accesses that will occur during execution. RTA extends the CHA algorithm also taking into account, besides the class hierarchy, the class instances (new instructions) to determine reachable methods. Therefore, according to RTA, if a method is reachable and there is a call to the method  $e.m()$  in its body, then only implementations of methods with a signature compatible with  $m()$ , belonging to any instantiated subclass of the static type of the expression  $e$  are tagged as reachable.

Figure 0-3 shows a simple example where RTA can find unreachable methods. In the example, as only the class B is instantiated, RTA marks the method  $C.m()$  as unreachable, even if the method  $A.m()$  has been called. In the same example, the CHA algorithm would fail and would mark  $A.m()$  as reachable.

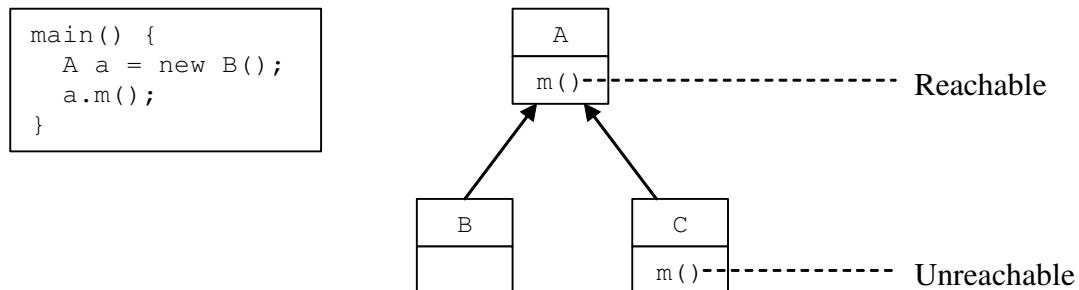


Figure 0-3: RTA example.

Other algorithms, such as XTA and k-FCA, are variations and extensions of RTA. They vary in the precision of the resulting graph, the implementation difficulty, and the amount of memory and processing time they need.

### *Restrictions imposed by the Java Virtual Machine*

Besides calls considered polymorphic by the call graph, the decision algorithm should reject many other situations, due to characteristics of the Java Virtual Machine itself [Lindholm & Yellin, 1999].

**Abstract** methods, by definition, cannot be optimized, since they have no code associated to them. Usually, the call graph not even identifies an abstract method as reachable. Similarly, **native** methods are rejected because it does not have Java bytecode, since their code is external to the virtual machine. **Synchronized** methods also require special attention, since they implement an implicit lock.

**Constructors** are considered in the same way as methods at the bytecode level. They would be excellent candidates for method inlining, since they cannot be polymorphic. But a security mechanism of Java (the preverifier in the case of J2ME) does not allow an object to be initialized directly by the constructor of its superclass. Therefore only constructors implemented with the directive `this` may be optimized.

Methods that **catch exceptions** also require special attention because, when entering the catch block, the operand stack is emptied. Therefore, if the exception is caught only in the modified caller method, its behavior may be affected. On the other side, methods that **throw exceptions** can be optimized with no problem to the application execution; however they will end up changing the exception tracing info while debugging the application, since the real method that throws the exception in execution time becomes different from that written in the source code.

Methods that access special elements, such as **non-public** fields or methods, declared in the same class or inherited from other classes, or even calls to methods of the superclass (using the directive **super**), must be handled specially, since the caller method might not have enough permission to access these elements. In an environment that allows whole program optimization, such as in J2ME applications, some of these elements may be made public, like fields and methods defined in classes of the program. But many situations, like methods that contain calls to superclass methods or that access non-public fields or methods inherited from **libraries** cannot be adequately resolved and

cause the rejection of some or all of the calls to the method from the list of methods that may be inlined.

Methods that have direct or indirect **recursive** calls also need to be handled specially, since they can lead the decision algorithm to a loop.

## PROPOSED METHOD INLINING TECHNIQUE

After this study of the low-level factors that impact method inlining optimization, we formulated our method supported on two activities: (1) we defined some techniques and heuristics to minimize the code expansion when copying the code; and (2) we developed the decision algorithm selecting only the methods and call sites that, when copied, do not increase the total size of the application. Thus, we often manage perform inlining of small methods, such as get and set methods, as well as methods called only once, both common in many applications

In order to minimize the code expansion when copying the code (1), we took advantage of the runtime and bytecode features of the Java Virtual Machine to trace and to control the exact impact of the optimization changes over the application size. The bytecode level optimization of obfuscators is an ideal development environment for this approach.

The other way, in order to select only methods that do not increase application size (2), we also took into account features of the applications to be optimized, like the possibility of relying on whole program analysis, as is the case in J2ME applications. Thus, we can be much more aggressive, performing cross-module inlining and removing methods where all the call sites have been replaced, since they cannot be called from anywhere else. In fact, the key idea was to include the removal method benefit as a parameter of the decision algorithm, so that inlining of many methods will be only worthwhile if method can be removed afterwards.

Taking fully advantage of the removal method benefit as a parameter of the decision algorithm is not a simple task. This was found only in very aggressive compilers, focused in procedural languages [Ayers et al, 1997]. As far as we know, there is no research work about this applied to Java specificities, such as high polymorphism rate and self-container classfile format. As shown in Chapter 0, many of these specificities affect directly the inlining results and bytecode size impact.

Before we detail our technique, Section 0 presents some implementation decisions. Then, Section 0 defines how we handle bytecode copy problems. Section 0 details our proposed decision algorithm.



## IMPLEMENTATION DECISIONS

In order to implement our solution we decided to extend an obfuscator. As said before, J2ME industry has implemented whole program optimizations in obfuscators and shrinkers. Besides, since obfuscators act directly on bytecode level, they are also ideal to our intention of monitoring the real impact of the optimization over application size.

We decided to extend ProGuard [Lafortune], one of the most popular obfuscators in the J2ME community. . As said before, J2ME industry has implemented whole program optimizations in obfuscators and shrinkers. Besides, since obfuscators act directly on bytecode level, they are also ideal to our intention of monitoring the real impact of the optimization over application size.

ProGuard was cited in many technical articles and development environments of Sun Microsystems [Klemm, 1999] [Knudsen, 2002a] [J2MEwtk 1.0.4]. It is also an open source project. Other tools and environments we evaluated either did not provide a minimum support for a complete obfuscator [RetroGuard] [Dahm, 2002] [JikesBT] or did not publish their code [Jax] [DashO] [Jshrink].

Version 1.7.2 of ProGuard already implemented some basic optimizations like *classfile recreation*, *name compression* and *unused members elimination*. For this last optimization, ProGuard implemented a variant of the CHA (Class Hierarchy Analysis) algorithm, to trace the call graph. But it did not build an explicit data structure for that.

The CHA algorithm is efficient enough for unused member detection, the initial goal of ProGuard, but it fails to detect non-polymorphic calls since it does not consider which classes were instantiated, resulting an imprecise call graph [Bacon, 1997]. Therefore, the first change we did was the implementation of an extension of the RTA (Rapid Type Analysis) algorithm. In spite of not being the most sophisticated algorithm, RTA is well known for being fast and very efficient to detect non-polimorphic calls [Tip & Palsberg, 2000], a highly important feature for method inlining optimization.

Our RTA implementation is able to detect virtual calls where, even if there are several methods overriding the referenced method, only one of them belonged to an instantiated class, being marked as uniquely reachable. In this case, the referenced method can be made abstract if its body will never be executed. Figure 0-1 shows a simple example where our RTA implementation makes a method abstract. In the example, as only the class B is instantiated and it has its own implementation of `m()`,

RTA marks the method `C.m()` as unreachable and makes `A.m()` abstract, so that the call `a.m()` becomes monomorphic.

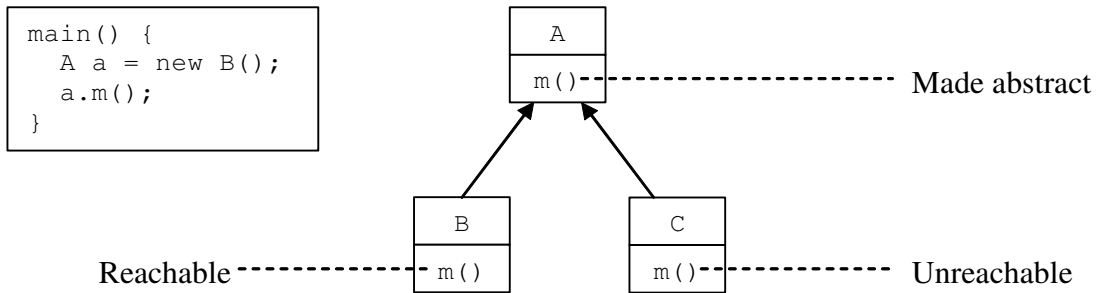


Figure 0-1: Make abstract example.

Besides the call graph construction, another auxiliary necessary implementation was a bytecode handling mechanism able to modify and copy the JVM instructions. We considered to use or to integrate some bytecode toolkits already available [Dahm, 2002] [JikesBT]. However, we noticed that many of their features aimed at bytecode parsing and were already available in ProGuard. Besides, their programming styles are quite different from that used in ProGuard. Therefore, we decided to develop our own mechanism keeping the programming style of ProGuard's original code, and just extending its bytecode parser framework.

After these preliminary changes, we could implement the method inlining optimization itself, presented in the next two sections.

## *COPY AND MODIFICATION OF THE BYTECODE*

In order to minimize code expansion during the copy of the bytecode, initially we identified a special but very frequent situation where the first instructions of the body of the method only loads its parameters, reproducing the state of the operand stack before the call to the method. Figure 0-2 shows an example where the first two instructions load the reference to `this` and the parameter `value` of the callee method.

Caller		Callee		Modified caller	
<b>void caller()</b>		<b>void callee(int value)</b>		<b>void caller()</b>	
<b>Source code</b>	<b>Variables</b>	<b>Source code</b>	<b>Variables</b>	<b>Source code</b>	<b>Variables</b>
if ( 1 > 0 ) { callee(1); }	0 this	this.f = value;	0 this 1 value	if ( 1 > 0 ) { f = 1; }	0 this
<b>Bytecode</b>	<b>Op.Stack</b>	<b>Bytecode</b>	<b>Op.Stack</b>	<b>Bytecode</b>	<b>Op.Stack</b>
0   iconst_1	1	0   aload_0	this	0   iconst_1	1
1   iconst_0	1; 0	1   aload_1	this, value	1   iconst_0	1; 0
2   if_icmple 10		2   putfield #4		2   if_icmple 10	
5   aload_0	this	4   return		5   aload_0	this
6   iconst_1	this; 1			6   iconst_1	this; 1
7   invokevirtual #2				7   putfield #4	
10   return				10   return	

Figure 0-2: Example of method inlining without temporary variables.

In this case, we can copy and modify only part of the code, avoiding the creation of temporary variables and taking advantage of the previous state of the operand stack. To do that, some constraints should be satisfied, in particular (i) initial instructions must be only load instructions indexing all parameters sequentially; (ii) parameters must not be used again in the callee method and (iii) there must be no jumps to the region of the initial load instructions.

Note we do not still have a formal proof of these constraints, however they are very simple and just tries to guarantee the instructions that read the parameters will be executed only once, rebuilding the operand stack before the call.

These constraints may seem too restrictive, but this situation handles most of the field access methods (get and set), simple functions and delegations. We named this mechanism “stack binding”, since it uses the operand stack to connect the copied code and the caller method context, against the “variable binding” mechanism, which uses temporary variables as described in the Section 0. Figure 0-2 highlights the code replaced by method inlining using the stack binding mechanism.

Notice that while using the stack binding, it is often the case that the copied bytecode has the same size of the `invoke` instruction, keeping the same size of the code for each inlining operation (in example of Figure 0-2, instruction `putfield #4` just replaces instruction `invokevirtual #2`). The variable binding is still needed for those cases where it is not possible to use the stack binding, or when it is required to include some `checkcast` instruction to match the type of the called object and the type of the parameter `this`, as described in the Section 0.

The impact of re-indexing variable access instructions was minimized by reusing the indexes of local variables, as if they had been defined inside a block in the modified caller method. Figure 0-3 shows an example where the same caller method refers two different callee methods. Both callee methods had to use variable binding mechanism to be inlined, where temporary variables are created. First result column shows our solution, where variable indexes are reused between pieces of code from each inlined method (1). The second result column shows a supposed solution where variable indexes would not be reused (2). Notice the difference of result code length in the end of the result columns (3). This happens because store and load instructions for variables indexes greater than 3 occupies two bytes instead of one.

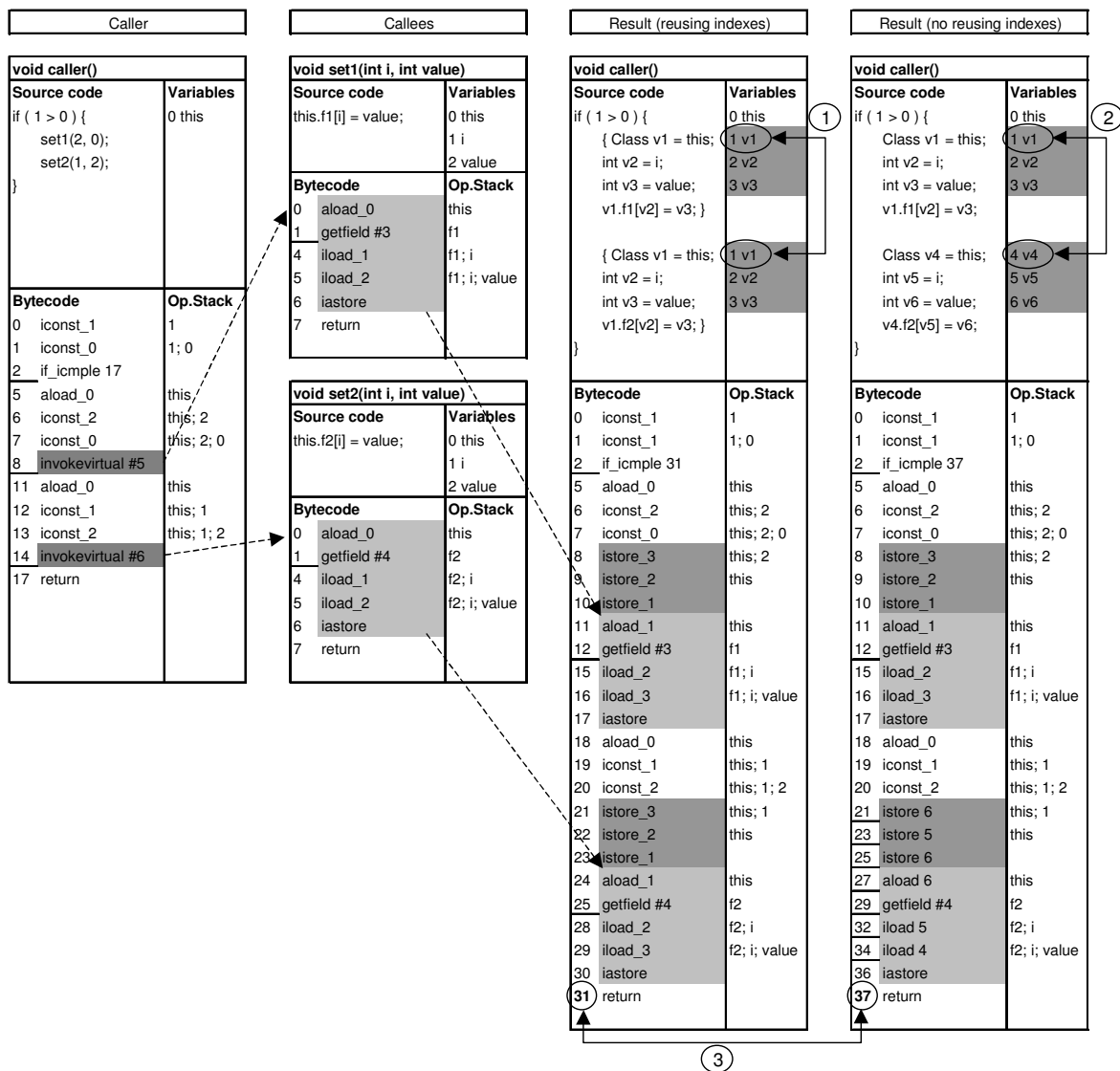


Figure 0-3: Example of variable re-indexing.

Thus, the variables' indexes tend to be kept low even if many call sites are optimized in the same caller method.

In order to reuse variable indexes, we need to prepare and modify callee methods considering only the original variables of the caller method. We opted to late the effective insertion of the modified bytecode in the caller method. Therefore, the preparation of the callee methods are not affected by previous inlining in the same caller.

Figure 0-4 shows an example of this technique. When a callee method is visited (1)(2), its code is modified and prepared for each caller site (notice there can be many caller sites). However, the result bytecode is not inserted in caller sites yet. It is just attached to the respective caller site instead. The effective insertion in the caller method will occur only when caller method is visited (3).

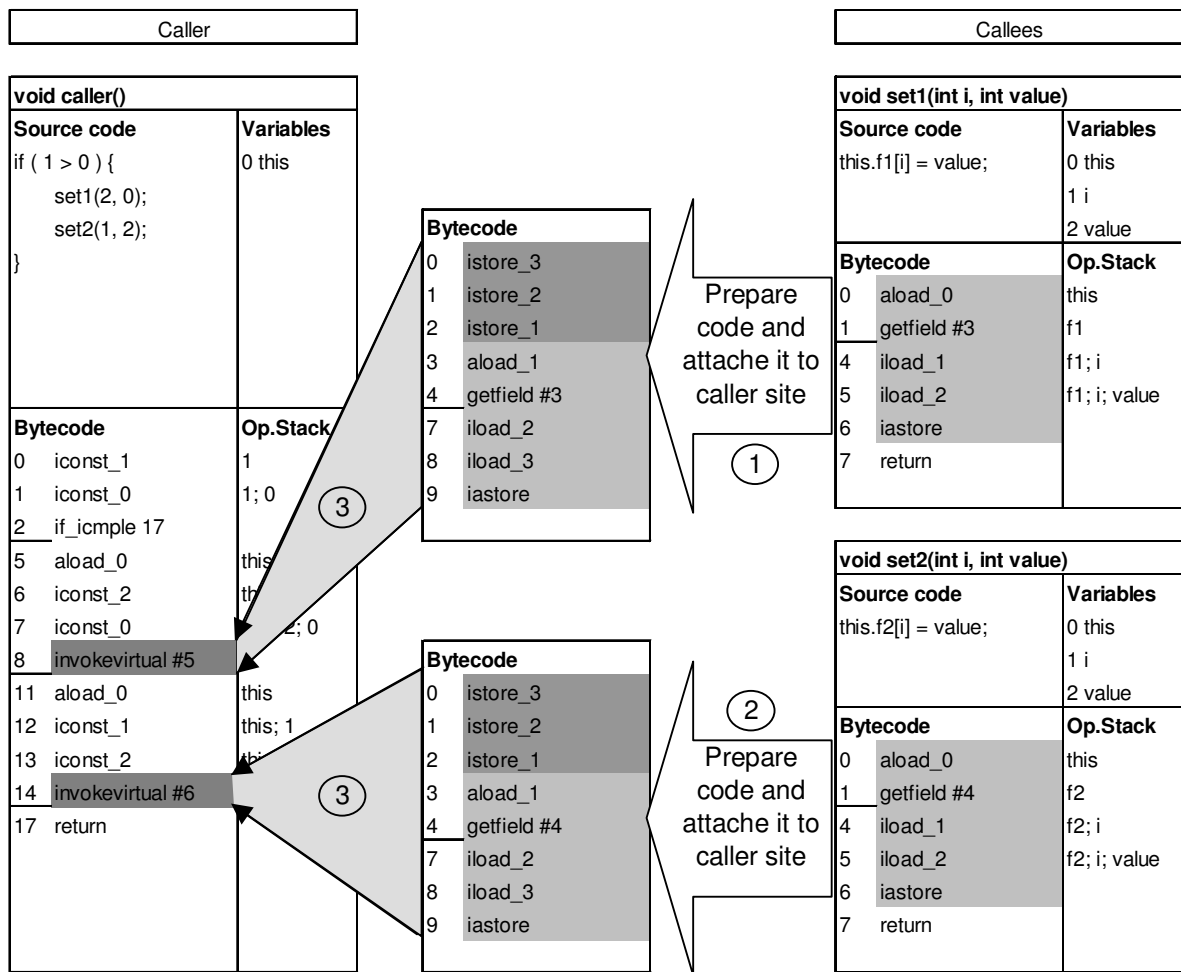


Figure 0-4: Example of code preparation.

Therefore, all selected callee methods are prepared for copying considering only the original variables of the caller method. This assures a safe reuse of variable indexes among several inlinings in the same caller. The relocation of switches in the modified caller method, in cases when a previous call site has been replaced, was handled by

inserting `nop` instructions after the copied code for each of those call sites. Thus the switch instructions that follow them keep their alignment in an address that is a multiple of 4, removing the need for any correction of these instructions.

The presence of switch instructions in the copied code was not handled, causing the rejection of the method as candidate for the optimization. This decision results from our implementation of variable re-indexing has made difficult to modify switches from the callee method. Once we opted to modify the callee bytecode previously without effectively inserting it in the caller method, it prevents the definition of the exact location where the switch is going to be inserted in the caller code. In fact, we preferred to privilege the handling of instructions that access variables, because they are much more frequent than switch instructions.

The replacement of return instructions by `goto` instructions, the removal of the last return instruction and the handling of the constant pool entries were implemented as described in Section 0.

With those techniques, we were able to copy and prepare the proper piece of code from callee and replace it into caller site, handling the most frequent situations that cause code expansion during this process. Next section explains how our decision algorithm selects which caller sites should be replaced.

## *DECISION ALGORITHM*

The decision algorithm was designed in order to reduce the code size considering the possibility of excluding the method when all call sites that refer it have been optimized. In fact, the key idea was to include the positive impact of removing the method as a parameter of the decision algorithm, so that many methods will only be inlined if they can be removed afterwards.

The algorithm visits the reachable methods of the application in an arbitrary order, deciding (a) which calls to the method will be optimized and (b) if it will be possible to remove it after the optimization. After visiting the method, and having decided to optimize part or all call sites to it, this decision will never be reverted, guaranteeing that each method will be **visited only once**. The concern with the performance of the decision algorithm itself is important, resulting in specific research work in this area [Dean & Chambers, 1994].

The effective removal of the method from the code is an additional step that implies in many other activities, such as re-indexing constant pool entries and update

some classfile internal structures. This step was included in an already existing routine of ProGuard that processed some other classfile optimizations.

Our decision algorithm requires a previous prepared call graph structure from our RTA implementation, as described in section 0. Figure 0-5 represents an example of that structure focused on a given method (method #3). We named *caller sites of the method* (1), all those call sites that calls to that method. We named *forward call sites of the method* (2), all those call sites that belongs to the body of the method.

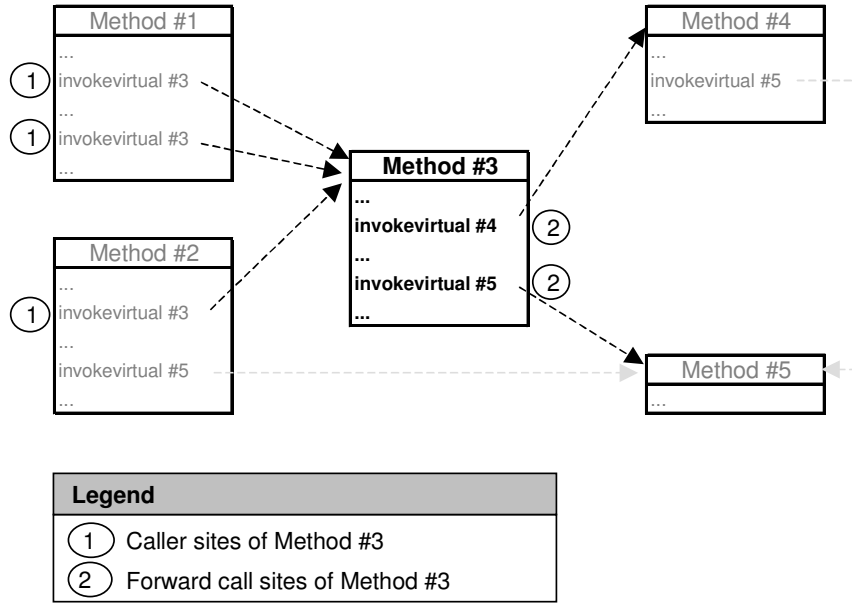


Figure 0-5: Example of call graph structure for a method.

In an overview, when visiting a method (method #3 in the example), our decision algorithm processes the following activities:

1	Check if method is already being <b>visited by another iteration</b>
2	If so, a recursive cycle was detected. Abort inlining for the current method.
3	Check all <b>caller sites</b>
4	Check how many and which caller sites can be inlined (not polymorphic).
5	If all caller sites can be inlined, mark that method can be removed.
6	Check <b>body</b> of the visited method to prepare it to be copied
7	Check if stack binding is possible
8	Evaluate bytecode expansion when coping method instructions
9	Deal with restrictions imposed by Java Virtual Machine
10	Visit all <b>forward call sites</b> to replace their instructions if they are inlined.
11	Visit <b>recursively</b> the method referenced by each forward call site.
12	If forward call site has attached code (it was inlined)
13	Replace call instruction with attached code.
14	If estimated code size <b>expansion is worth</b>
15	Attach prepared code to each caller site selected to be inlined.

Figure 0-6: Decision algorithm activities.

Basically, the algorithm selects caller sites to be inlined, prepares the current method body (as described in section 0), and evaluates the estimated code size expansion according to prepared code. If this expansion estimate is worth, the algorithm attaches prepared code to each respective caller site. Notice that, in order to reuse variable indexes, the attached code effectively replaces caller site instruction only when the method that belongs caller site is visited.

The final expansion estimate concerns to the size of the fully modified method, after all inlining adjusts. Even though inlining of forward call sites have been already inserted into current method body (line 11 in Figure 0-6). This assures the high precision of the expansion estimate.

We suppressed many details of the algorithm in Figure 0-6 in order to make it clearer. For example, performance improvements and many decisions about how to work around restrictions of Java Virtual Machine are not present there. These details will be discussed in the remaining of this section.

During the evaluation of the **estimate of the application size expansion**, the algorithm verifies whether the resulting value is smaller than a certain limiting factor. This factor is the main parameter of the algorithm, named MAX\_EXPANSION. If that limiting factor is zero, it means the algorithm does not tolerate any size increase. Higher values allow more methods to be optimized, improving the application performance, but with a smaller percentage of code reduction or even increasing the code size, as shown in Chapter 0. Figure 0-7 details the parameters considered in the calculation of the expansion.

NCS	: number of call sites to be optimized
CDL	: length of the code array to be modified
CSL	: length of the code array occupied by all invoke instructions of the call sites to be optimized
RMV	: flag indicating if the method will be removed
MDL	: total size of the method, including its header
$CUST = (NCS * CDL) - CSL - (RMV ? MDL : 0)$	

Figure 0-7: Formulae of code size increase estimation.

Of course, the method can only be removed (indicated by flag RMV) if all its caller sites were selected to be inlined. However, it may still be worthwhile to inline



only a subset of the caller sites, depending on the length of the copied code array (CDL) and size occupied by replaced invoke instructions (CSL).

Additionally, notice that in the formulae of Figure 0-7 the value MDL is defined as the total size used by the method, including its header. Since the header is formed by the **constant pool** entries, the exact decision of which entries will be able to be excluded is not a simple task, due to the possibility that they are being shared by other elements of the classfile. Therefore, the estimation of the code size expansion cannot be exact. We tried to make it as accurate as possible, predefining a fixed average size to method headers.

For performance reasons, the algorithm evaluates the size expansion estimate many times during activities of Figure 0-6, in order to reject the inlining of current method as soon as the algorithm detects it is not worth. These partial evaluations were suppressed from Figure 0-6 in order to make algorithm clearer. Initially, the estimate evaluation of each method considers an ideal situation where all calls to the method will be optimized, enabling the exclusion of the method. It also considers that it is possible to apply the stack binding mechanism, reducing the code size to be copied (CDL). **The estimate is then performed successively**, while the evaluation of the method validates each part of this initial hypothesis. For example, if the algorithm detects that not every calls to the method are monomorphic (in line 4 in Figure 0-6), or that it is not really possible to perform the stack binding mechanism (in line 7 in Figure 0-6), the estimate is evaluated again, and it could reject the inlining of the method. Actually, the initial full hypothetic estimation is performed in the beginning of the algorithm, after line 2 of Figure 0-6. Other partial estimations are performed after lines 5, 7, 8 and 9 of Figure 0-6, all of them suppressed from figure. If the inlining of the method is rejected by any of those estimations, all other inlining analysis of the current method are skipped (lines from 3 to 14, except line 11). In this case, only the recursive call of forward call sites (line 11 in Figure 0-6) is performed, in order to keep the recursive aspect of the algorithm.

In the context of the estimative, the size of the copied code is considered the same for all inlined caller sites. However, several factors presented in the Section 0 show that this is not true. Therefore, at the end of the analysis of each method, **a last verification** is still performed considering all the real parameters, including the total size of modified bytecode for each call site (line 14 in Figure 0-6). This is the last situation where the optimization of the method being visited may still be rejected.

Once attached to caller site (line 15 in Figure 0-6), the algorithm assumes that attached code will be necessarily inserted in caller method body. Thus, the decision to inlining that caller site will never be reverted. This is important in order to avoid needing to visit current method again. Unfortunately, this assumption generates a problem: as all attached codes needs to be necessarily inserted in method body, the method bytecode array can increase and virtually it can exceed JVM method body limit specification (64 kbytes). It is really improbable and it has never occurred in our experiments and possibly it will never occur at all. Actually, we preferred to fail in this verification in benefit to the high algorithm performance improvement.

Besides these considerations about the impact on the application size, the constraints imposed by the virtual machine, shown in Section 0, should also be taken into account. They could affect in the number of call sites to be optimized or even reject the whole method.

Thus, **abstract**, **native**, **synchronized** and **constructor** methods are rejected, as well as methods that **catch** exceptions or that contains **direct recursive** calls. For **indirect recursive** calls, the first method where the cycle is detected is rejected, virtually allowing the rest of the methods in the cycle to be optimized, merging them into the first method and generating direct recursive calls.

Methods that **throw exceptions** are inlined normally, with no change to the functional behavior of the application. However, we decide to assume the behavior change on debugging exception tracing. . In fact , once the `throw` instruction is also copied to the caller method, the real method that throws the exception in execution time becomes different from that written in the source code.

When accessed by an inlined method, **non-public** members (fields and methods) declared in the same class or inherited from other classes are made public. This is supported by the whole program assumption. Exceptions to this rule refer to members inherited from **library** classes, which cannot be made public. We opted to let **private methods** unchanged too, because the `invoke` instruction used for private methods (`invokespecial`) is faster than the one used for public methods (`invokevirtual`) [Lindholm & Yellin, 1999].

In all situations where the accessibility problem is not resolved, that is, membersthat cannot be made public and methods with special **super** calls, the algorithm does not reject the method immediately, but it goes ahead considering to select **only**

**calls from the same class**, then it evaluates again the expansion estimative and the possibility of rejecting the inlining of the method.

In spite of these special situations seem restrictive, this decision algorithm managed to handle all example opportunities introduced in Table 0-3. It is still able to remove almost all of the optimized methods, as shown in Section 0.

## EXPERIMENTAL RESULTS

In order to evaluate our proposed method, we compared the original ProGuard with our ProGuard version. For a given set of applications, to which the optimizations should be applied, the evaluation criteria were percentage of the code reduction and the execution performance gain.

Sections 0 and 0 present the experiment setup and the process of adjusting our algorithm's parameters, respectively. Then, Sections 0 shows the results concerning the number of methods and calls excluded during the optimization, whereas Section 0 discusses the optimization impact on performance and memory. Finally, Section 0 compares our method inlining code size reduction with those ones found in other obfuscators.

### *EXPERIMENTS SETUP*

Since there were no standard benchmarks for J2ME optimization, we have chosen some real applications provided by C.E.S.A.R/Meantime [CESAR/Meantime], a well established IT Brazilian company that works in J2ME applications since 2000. We have also included three J2SE (Java Standard Edition) applications in some experiments, to evaluate the proposed method results outside J2ME scope. The selected applications are:

- Eight J2ME games (BreakOut, Ship, Istari, Atlantis, SpaceInvaders, GoldHunter, Pacman and LightTenis) developed by C.E.S.A.R/Meatime, the first two using the wGEM game engine (framework) [Pessoa, 2001];
- Three J2SE applications (Ant, JDepend and the original ProGuard), freely available.

Games were chosen since this is typically a kind of application to which memory and processing power are critical resources. In order to assess the generality of our method, we have selected games with different styles, as well as the three non-J2ME applications. All selected J2ME applications suffered previous strong manual improvements, by a skilled software engineering team, in order to meet the processing

and memory restrictions of cell phones. Any significant improvement in these applications therefore is a particularly good result.

All code size measurements are based on compressed JAR files, containing only the application classes, without resources (e.g., images and sounds). The impact on non-compressed code is not so important because the applications are often distributed compressed; for example, only JAR files can be installed in J2ME devices.

Concerning performance measurements, all J2ME case studies were tested on the emulator DefaultGrayPhone, supplied with the J2ME Wireless Toolkit 1.0.4 [J2MEwtk 1.0.4]. Non-J2ME applications were executed with Java 2 Standard Development Kit 1.4.1. The execution platform was a PC with an AMD Athlon 1.0 GHz processor and 256 Mb RAM memory, running Windows 2000 Professional. The applications were automatically compiled and compressed with the Java 2 Standard Development Kit 1.4.1.

## *PARAMETERIZATION*

Table 0-1 presents the size reduction results varying the MAX\_EXPANSION parameter in our algorithm. Columns 1 and 2 show, respectively, the applications original size and its reduction percentage obtained with original ProGuard v1.7.2. Column 3 exhibits the size reduction achieved with our Extended ProGuard with no method inlining. Notice that Column 3 already improves slightly the size reduction percentage compared with original ProGuard (Column 2), due to our new implementation of the call graph, using RTA.

Columns 4 to 7 show the percentage of size reduction measured when applying method inlining with several MAX\_EXPANSION values, as indicated in parenthesis in the column headers. Column 4, when MAX\_EXPANSION is zero, indicates that the algorithm tries to reject any code size increase when inlining. Columns 5 to 7 show higher values that make the algorithm more tolerant to code size expansion. In the extreme case, Column 7 shows the code size reduction when MAX\_EXPANDED is 65536 (the maximum allowed method size), indicating the algorithm accept a great number of method inlining, regardless to the code size expansion.

The bold values highlight the best code size reduction for each application. Notice that sometimes (applications Istari and GoldHunter) the best value is not acquired by the most conservative parametrization (Column 4). That is because the

estimation of the code size reduction is not straightly exact. However, these best results are always slightly better those ones found in Column 4.

Applications		1	2	3	4	5	6	7
		Original size	ProGuard v1.7.2	Extended ProGuard (without inlining)	Extended ProGuard (MAX_EXPANSION = 0)	Extended ProGuard (MAX_EXPANSION = 100)	Extended ProGuard (MAX_EXPANSION = 200)	Extended ProGuard (MAX_EXPANSION = 65536)
J2ME	Atlantis	26.317	28,09%	28,88%	<b>31,27%</b>	31,02%	31,02%	30,67%
	BreakOut	31.326	48,18%	48,70%	<b>51,40%</b>	51,13%	50,75%	50,75%
	Ship	41.841	41,02%	41,33%	<b>44,07%</b>	43,25%	42,38%	41,07%
	Istari	37.432	38,00%	38,46%	42,31%	<b>42,37%</b>	41,52%	39,39%
	SpaceInvaders	41.723	36,72%	37,29%	<b>39,63%</b>	39,21%	38,97%	38,62%
	GoldHunter	42.243	31,85%	32,60%	34,86%	<b>34,89%</b>	34,46%	33,36%
	Pacman	52.326	55,41%	55,63%	<b>56,83%</b>	56,67%	56,68%	56,09%
	Tenis	52.587	55,41%	55,79%	<b>57,73%</b>	57,72%	57,61%	56,37%
J2SE	Ant 1.5.1	707.376	88,81%	90,74%	<b>90,95%</b>	90,93%	90,89%	90,70%
	JDdepend 2.6	84.535	59,73%	62,34%	<b>64,01%</b>	63,79%	63,64%	62,89%
	ProGuard 1.7	188.547	49,45%	49,57%	<b>50,25%</b>	49,92%	<u>49,44%</u>	<u>47,04%</u>
Average			<b>48,42%</b>	<b>49,21%</b>	<b>51,21%</b>	<b>50,99%</b>	<b>50,67%</b>	<b>49,72%</b>

DEFAULT

AGGRESSIVE

Table 0-1: Bytecode size reduction by algorithm parameterization.

For our surprise, the algorithm often keeps improving of application code size reduction, even when we extrapolate the value of the MAX\_EXPANSION, allowing all the possible methods to be optimized (Column 7). That is because the whole program assumption allows a great number of methods to be removed after inlining in aggressive approaches, as will be shown in Section 0. Besides that, the number of possible methods was intrinsically limited by our conservatory approaches how to deal with occurrence of polymorphic methods (not performing customization optimization); as well as with all restrictions imposed by the Java Virtual Machine. In many of these cases, method inlining is just rejected, instead of trying any other solution that could result in code expansion. This is imperative regarding to our worries about code size expansion in order to cover J2ME constraints.

The underlined values in the bottom right corner of the table (Columns 6 and 7) indicate the only two values when that aggressive approach has generated some

application code bigger than the original ProGuard result, shown in Column 2. Even in these cases, there was no explosion of application size.

In the remaining experiments, we worked with only two versions of our method, namely **default** and **aggressive**, respectively corresponding to columns 4 and 7 in Table 0-1.

### *OPTIMIZATION OCCURRENCE*

Table 0-2 presents the number of optimized methods for each parameterization, classified in three categories: (i) *inlined and kept*, indicating the number and percentage of methods that were inlined but could not be removed; (ii) *inlined and removed*, indicating the number and percentage of methods that could be fully removed after inlining; and (iii) *not inlined*, indicating the number and percentage of methods that was not inlined at all. The average of optimized methods is grouped by application platform (J2ME or J2SE), in order to help the analysis of the optimization impact for each one of them.

			Default			Aggressive		
Application		Number of Methods	Inlined and kept	Inlined and removed	Not inlined	Inlined and kept	Inlined and removed	Not inlined
J2ME	Atlantis	122	0 0,00%	52 42,62%	70 57,38%	2 1,64%	54 44,26%	66 54,10%
	Istari	246	0 0,00%	143 58,13%	103 41,87%	3 1,22%	175 71,14%	68 27,64%
	Ship	220	0 0,00%	100 45,45%	120 54,55%	5 2,27%	113 51,36%	102 46,36%
	BreakOut	164	0 0,00%	79 48,17%	85 51,83%	4 2,44%	85 51,83%	75 45,73%
	GoldHunter	196	0 0,00%	84 42,86%	112 57,14%	1 0,51%	102 52,04%	93 47,45%
	SpaceInvasors	179	0 0,00%	70 39,11%	109 60,89%	5 2,79%	84 46,93%	90 50,28%
	Pacman	145	0 0,00%	60 41,38%	85 58,62%	6 4,14%	72 49,66%	67 46,21%
	Tenis	173	0 0,00%	89 51,45%	84 48,55%	2 1,16%	104 60,12%	67 38,73%
	Average J2ME			0,00 %	46,15 %	53,85%	2,02 %	53,42 %
J2SE	Ant 1.5.1	414	0 0,00%	108 26,09%	306 73,91%	9 2,17%	130 31,40%	275 66,43%
	JDepend 2.6	312	1 0,32%	95 30,45%	216 69,23%	15 4,81%	116 37,18%	181 58,01%
	ProGuard 1.7.2	1.163	1 0,09%	118 10,15%	1.044 89,77%	22 1,89%	211 18,14%	930 79,97%
	Average J2SE			0,14 %	22,23 %	77,64%	2,96 %	28,91 %

Table 0-2: Number of inlined methods.

For our surprise, the optimization was able to remove around 50% of methods in J2ME applications (46,15% in default parameterization and 53,42% in the aggressive one) and around 25% of methods in J2SE workbench (22,23% in default and 28,91% in aggressive). Of course, the application code size reduction was only possible due to this high percentage of removed methods. We believe that J2ME application acquired a greater percentage because, being games, they have some more class fields and consequently field access methods (get and set). These methods can almost always be removed by the optimization.

We also highlight that only a few methods have been kept after inlining, usually only in the aggressive approach. This result indicates that, in order to assure code



reduction, most of the method inlining opportunities seem only to be worth if the method can be removed after inlining.

Table 0-3 shows a similar measurement for calls selected by the algorithm, indicating the percentage of call sites that was inlined or not for each parameterization.

			Default		Aggressive	
Application		Number of Call Sites	Inlined	Not Inlined	Inlined	Not Inlined
J2ME	Atlantis	323	198 61,30%	125 38,70%	226 69,97%	97 30,03%
	Istari	755	379 50,20%	376 49,80%	520 68,87%	235 31,13%
	Ship	620	324 52,26%	296 47,74%	412 66,45%	208 33,55%
	BreakOut	387	231 59,69%	156 40,31%	268 69,25%	119 30,75%
	GoldHunter	691	279 40,38%	412 59,62%	479 69,32%	212 30,68%
	SpaceInvasors	476	232 48,74%	244 51,26%	329 69,12%	147 30,88%
	Pacman	433	179 41,34%	254 58,66%	281 64,90%	152 35,10%
	Tenis	804	481 59,83%	323 40,17%	594 73,88%	210 26,12%
Average J2ME			51,72 %	48,28%	68,97 %	31,03%
J2SE	Ant 1.5.1	884	253 28,62%	631 71,38%	352 39,82%	532 60,18%
	JDepend 2.6	721	289 40,08%	432 59,92%	383 53,12%	338 46,88%
	ProGuard 1.7.2	4.081	191 4,68%	3.890 95,32%	989 24,23%	3.092 75,77%
Average J2SE			24,46 %	75,54%	39,06 %	60,94%

Table 0-3: Number of inlined call sites.

As expected, the aggressive parameterization always optimizes more methods and calls than the default one.

These optimization occurrences were possible due to the generalization power of the algorithm, which was able to remove almost all field access methods (e.g. *get*, *set* and *is*), simple functions and delegations, methods called only once, small methods

called few times, etc. All of these situations are very common in object-oriented applications.

Of course, the results also depend on the programming style and architecture of the application. For example, the ProGuard 1.7.2 (last application in Table 0-2 and Table 0-3) had a bad result since it uses excessively the *visitor design pattern* [Gamma et al, 1995], that produces many virtual and polymorphic calls that are not inlined by our technique.

## ***EXECUTION MEMORY AND PERFORMANCE***

We also evaluated the impact of the optimization on time and memory needed for the execution of the applications. For that, we needed to modify the source code of J2ME games, making them deterministic, i.e., simulating user input and removing random behavior, timers and threads usage. Additionally, we also needed to modify the source code of the applications to show the time and memory used while executing. For some applications of our benchmark, this task is especially hard due to way they were implemented. For instance, for some games, it would be difficult to simulate user input to cover relevant game situations. Besides, changes to make some applications deterministic often require excessive comprehension of their code, especially for J2SE ones. Thus, we decided to modify just a subset of our benchmark, including 4 J2ME games and the Proguard 1.7.2 itself to represent J2SE applications. These modified applications were submitted to the original ProGuard 1.7.2, and to our Extended ProGuard with the default and aggressive parameterization. All optimizations available by each ProGuard versions were enabled, in order to reproduce real usage of the tools where the optimizations can interact each other.

Table 0-4 shows the total memory allocated by each modified J2ME application as shown by the emulator output. For the J2SE application, the ProGuard 1.7.2 itself, the presented value represents the instant memory allocated in the end of the execution.

Each resulting executable, after optimization, presents same memory values for all their executions. Below each memory value, we inform the percentage of reduction compared to the values obtained with the non-optimized application.

The memory results were already expected. The method inlining optimization presented a small influence on the amount of used memory, when compared with the results already obtained by the original ProGuard 1.7.2. The resulting application from default parameterization of our Extended Proguard has used slightly less memory than

those ones resulting from original Proguard. The other side, the resulting application from our aggressive parameterization has used slightly more memory than those ones from original Proguard. This memory impact results from bytecode variations that affect code image loaded in memory. As default parameterization removes many methods without increase code size, the resulting application requires less memory to store its code in memory. Same way, as aggressive parameterization allows replicating some code, the resulting application requires some more memory to store its code in memory because there are more bytecode to be stored.

Total memory allocated (bytes)		No optimization	ProGuard v1.7.2	Extended ProGuard (default)	Extended ProGuard (aggressive)
J2ME	SpaceInvaders	<b>492.412</b>	451.672 8,27%	444.844 <b>9,66%</b>	452.072 <b>8,19%</b>
	Pacman	<b>635.536</b>	626.696 1,39%	626.336 <b>1,45%</b>	634.364 <b>0,18%</b>
	Atlantis	<b>393.432</b>	363.144 7,70%	355.988 <b>9,52%</b>	359.884 <b>8,53%</b>
	Tenis	<b>1.476.828</b>	1.440.028 2,49%	1.431.244 <b>3,09%</b>	1.449.092 <b>1,88%</b>
J2SE	ProGuard 1.7.2	<b>16.415.832</b>	16.231.112 1,13%	15.754.576 <b>4,03%</b>	16.242.776 <b>1,05%</b>

Table 0-4: Reduction on total memory allocated.

Table 0-5 presents the execution time for each modified application, measured as an average of three consecutive executions. Three executions seem enough because time measures vary very slightly among executions, once all tested applications were modified to be deterministic. Below each execution time, we inform the percentage of reduction of the time, compared to the values obtained with the non-optimized application. Therefore, positive percentage values mean improvements on the application performance, and negative percentage values mean the resulting application is slower than the non-optimized one.

Execution time average (ms)		No optimization	ProGuard v1.7.2	Extended ProGuard (default)	Extended ProGuard (aggressive)
J2ME	SpaceInvaders	<b>14.411</b>	14.501 -0,63%	13.059 <b>9,38%</b>	12.912 <b>10,40%</b>
	Pacman	<b>13.703</b>	13.710 -0,05%	13.413 <b>2,12%</b>	13.322 <b>2,78%</b>
	Atlantis	<b>11.403</b>	11.323 0,70%	10.422 <b>8,61%</b>	10.388 <b>8,90%</b>
	Tennis	<b>17.478</b>	17.432 0,27%	16.614 <b>4,95%</b>	16.564 <b>5,23%</b>
J2SE	ProGuard 1.7.2	<b>10.018</b>	10.254 -2,36%	10.208 <b>-1,90%</b>	10.478 <b>-4,60%</b>

Table 0-5: Application performance improvement.

To our surprise, resulting applications from original ProGuard sometimes are slower than the non-optimized one, as shown by negative values in second data column for SpaceInvaders, Pacman and ProGuard 1.7.2 applications. That indicates that some optimization of the original ProGuard has jeopardized performance of some applications. We were not able to detect which previous optimizations were responsible for that; however, those optimizations actually did not target application performance at all.

In our experiments, the default parameterization, shown in the third data column, always improved the application performance compared with the original ProGuard 1.7.2, shown in the second data column. The unique negative value of the default parameterization (still in third data column) refers to when the ProGuard itself is optimized. In that case, the method inlining improvement was not able to compensate the performance degradation caused by previous optimizations of the original ProGuard 1.7.2.

In most of the applications, the execution time for the aggressive optimization is slightly better than for the default one. The exception to this rule was already expected: ProGuard 1.7.2 itself, where the aggressive inlining had increased the application size, degrading the application performance for memory cache reasons. This result means that the aggressive inlining seems to be worth only if it reduces the application code

size. Otherwise, if the code increases, the performance impact is probably worse or equal to the non-inlined version.

Anyway, as the default parameterization never increases the application size, we believe that we can apply default method inlining with no chance to degrade the application performance. In the end, the default parameterization seems to assure both a reasonable performance improvement while reducing some application code size.

## *CODE SIZE REDUCTION BY OBFUSCATORS*

The study presented in Chapter 0 shown to us that method inlining optimization is rarely implemented, having been found only in Jax and DashO. In order to verify the overall improvement of our optimization on code size reduction, we have submitted to these obfuscators the same applications used to evaluate our solution.

For the experiments performed here, we used version 7.3 of Jax and an evaluation copy of the DashO Embedded Edition (the same versions used in study of Section 0). They were configured in order to minimize the size of the applications, including all optimizations provided by each of them. The graph of Figure 0-1 presents a comparison of the code size reduction of these applications after optimized.

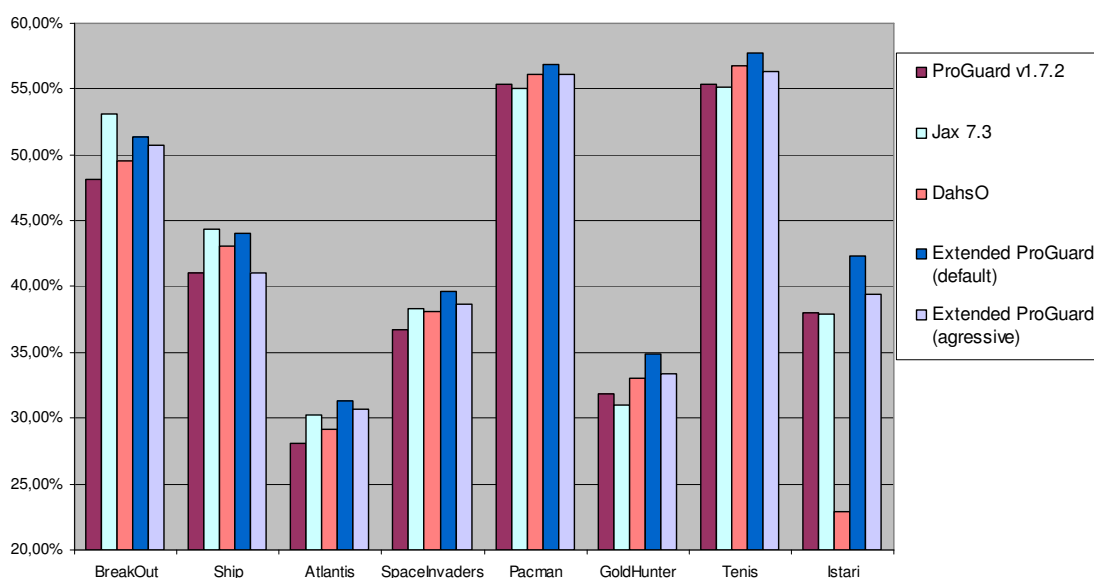


Figure 0-1: Code size reduction by obfuscators.

Despite of the additional size reduction of our optimization being apparently small, that was enough to make ProGuard to stand out among the tools. The only situations in which we lost to Jax refers to applications that use frameworks (wGEM

[Pessoa, 2001]) where Jax managed to apply the class merge optimizations, still not implemented by the other tools, including ProGuard.

In fact, the elimination of unused members is surely the most effective optimization for the application size reduction, represented by similar order of magnitude of results from all tools. However the usage of other optimizations as method inlining performs an additional improvement that can be very important in restricted environments like J2ME.

## BEST PROGRAMMING PRACTICES

This study on the optimizations available by obfuscators combined with our experience while extending one of them gave us valuable know-how about optimizations details. Moreover, for several years, our research team has been providing consulting services on J2ME for industrial scale applications [CESAR/Meantime]. This experience clearly has shown us that *it is essential for programmers to know the capabilities of the adopted tool in order to avoid unneeded design sacrifices and to improve optimization results*. A lot of effort is wasted avoiding situations already resolved automatically, and many other practices can confuse the optimization algorithms. Therefore, we have developed this best practices guide to advise the developers of possible difficulties and facilities when using obfuscators. The issues are organized in four groups:

- *Situations already resolved by the obfuscator*: practices we believe programmers implement only for optimizing the application but that are already resolved by some optimization.
- *Situations not resolved by the obfuscator*: clarifies mistaken practices e.g. practices we believe programmers can do expecting the obfuscators to fix them but that are not really resolved by any available optimization. These situations include limitations and possible future implementations of the tool, as well as especial cases not handled properly.
- *Situations that jeopardize the obfuscator*: highlights undesirable practices that confuse or make some optimization inapplicable or make the obfuscator configuration difficult. These situations must be avoided when possible but can be used if needed.
- *Other recommendations*: includes all other general advices and recommendations not included in the previous cases, possibly not related to the obfuscators job, but still important to keep application size small.

This kind of information depends on the optimizations implemented by each obfuscator. Ideally, obfuscator developers should include this information in the user documentation, since they know the details of the optimizations they implemented. Unfortunately, this is not usual. The next sections present some examples of best

programming practices considering the optimizations implemented by the obfuscator we have extended.

An expert programming staff can extend this document and produce its own best programming practices document considering the optimizations available by the adopted obfuscator and the programmers' skills. This document should be maintained and updated every time a new optimization becomes available (when the obfuscator is changed or a new version is installed) or when a new advice to the programmers seems important.

## ***SITUATIONS ALREADY RESOLVED BY OBFUSCATORS***

This section presents some practices we believe the programmers can try to implement only for optimizing the application but that are already resolved by some optimization. Some practices include a table presenting a source code example with *unneeded* changes; a *recommended* version of that; and the result of this version when *optimized*.

### ***No identifier needs to be shorted***

Short identifiers easily reduce code legibility. Names of classes, fields and method are already replaced with short (often one letter) ones by *class and member names compression*; names of variables are removed by *classfile recreation*; and package names are emptied by *class package relocation*.

Example:

Unneeded	<b>package</b> game; <b>class</b> GameObj { <b>int</b> vx; }
Recommended	<b>package</b> com.company.game; <b>class</b> GameObject { <b>int</b> xspeed; }
Optimized	<b>class</b> A { <b>int</b> a; }

### ***Unused features in frameworks do not need to be suppressed***

Frameworks often implement a lot of methods and fields to be used when needed. Many of these features are not used by one application. However it is not needed to suppress these declarations from the framework for each application, since they are automatically removed by *removal of unused elements*, *removal of unused method body*, *removal write-only fields* and *method inlining* optimizations. Some examples are fields used only by methods never called; or static methods available in utility classes.

Example:



Unneeded	<pre> <b>class</b> A { <b>void</b> m() { /* ... */ } }           // unused body <b>class</b> B <b>extends</b> A { <b>void</b> m() { ... } } <b>class</b> Util {     <b>static</b> A a; /* <b>static</b> B b; */                // write-only     /*<b>static void</b> initA() { a = <b>new</b> A(); } */ // unreachable     <b>static void</b> initB() { a = /* b = */ <b>new</b> B(); } } ... <b>public void</b> startApp() {     Util.initB();           // only class B is instantiated     Util.a.m();             // always B.m() is executed } </pre>
Recommended	<pre> <b>class</b> A { <b>void</b> m() { ... } } <b>class</b> B <b>extends</b> A { <b>void</b> m() { ... } } <b>class</b> Util {     <b>static</b> A a; <b>static</b> B b;     <b>static void</b> initA() { a = <b>new</b> A(); }     <b>static void</b> initB() { a = b = <b>new</b> B(); } } ... <b>public void</b> startApp() {     Util.initB();     Util.a.m(); } </pre>
Optimized	<pre> <b>abstract class</b> A { <b>void abstract</b> m(); } // made abstract <b>class</b> B <b>extends</b> A { <b>void</b> m() { ... } } <b>class</b> Util {     <b>static</b> A a; } ... <b>public void</b> startApp() {     Util.a = <b>new</b> B();           // initB inlined     Util.a.m(); } </pre>

### *Primitive type constant values do not need to be replaced by hand*

Constant declarations make the code easier to understand and to maintain, however constants are implemented as common class fields on bytecode level. For primitive type values, when the constant is used, the compiler usually generates bytecode using directly the constant value, but the field is kept because some dynamic loaded class can access it later. Thus, a whole program analysis does not find any reference to those fields and *removal of unused elements* optimization removes them from the resulting application.

Example:

Unneeded	<code>Image.createImage(10, 15);</code>
Recommended	<pre> <b>public static final int</b> IMAGE_WIDTH  = 10; <b>public static final int</b> IMAGE_HEIGHT = 15; Image.createImage(IMAGE_WIDTH, IMAGE_HEIGHT); </pre>
Optimized	<code>Image.createImage(10, 15);</code>

## *Fields do not need to be made public to avoid field access methods (get and set)*

Declaring public fields is not recommended because it does not protect the user of the class from changes in class implementation. Unfortunately, many programmers do this in order to avoid field access method declarations (get and set). Besides, method call instruction is often slower than direct field access instructions. However, you can create the proper get and set methods because *method inlining* often removes them to you and makes fields `public` if needed. Trivial methods (e.g. that only read or write a field) are always inlined. Non-trivial method inlining depends on the number of times the method is called and the size of the method.

Example:

Unneeded	<b>public int</b> xspeed;
Recommended	<b>private int</b> xspeed; <b>public int</b> getXSpeed() { <b>return</b> xspeed; }
Optimized	<b>public int</b> xspeed;

## *Long methods can be divided into small context methods called once*

Long methods can make the code harder to understand. However, sometimes the programmer does not divide the method in smaller context methods to avoid the new declaration. *Method inlining* optimization often removes methods called once and replaces its unique call with the method code.

Example:

Unneeded	<b>public void</b> update () { // Update map ... // Update objects ... };
Recommended	<b>public void</b> update () { updateMap(); updateObjects(); } /** Update map */ <b>private void</b> updateMap() { ... } /** Update objects */ <b>private void</b> updateObjects() { ... }
Optimized	<b>public void</b> update () { ... ... }

## *SITUATIONS NOT RESOLVED BY OBFUSCATORS*

This section clarifies mistaken practices e.g. practices we believe the programmers can do trusting the obfuscators but that are not really resolved by any available optimization. Some practices include a table presenting a source code example with the *unresolved* situation; and the *expected* result that can be available by some future optimization, not implemented yet<sup>1</sup>.

### *Constant propagation still unavailable*

*Constant propagation* is an intra-procedural optimization where constants values assigned to a variable can be propagated and substituted at the use of the variable [Nullstone, 2002]. Compilers often implement *constant propagation* but some obfuscator optimizations, like *method inlining*, can open new opportunities to this optimization. Besides, some compilers perform constant propagation only in some cases, for example when the variable is explicitly declared as `final`.

Example:

Unresolved	<code>int x = 10; int y = x * 5;</code>
Expected	<code>int x = 10; int y = 50;</code>

### *Dead code elimination still unavailable*

*Dead code elimination* is an intra-procedural optimization where code that does not affect the program (e.g. dead stores) can be eliminated [Nullstone, 2002]. Compilers often implement *dead code elimination* but some obfuscator optimizations, like *method inlining*, can open new opportunities to this optimization.

Example:

Unresolved	<code>int i = 1;           // never used global = 1;         // dead code global = 2; return;</code>
Expected	<code>global = 2; return;</code>

### *Control flow analysis still unavailable*

*Control flow analysis* considers branch instructions, such as `if`, `while` or `switch`, to product an intra-procedural representation, the flow graph. With this graph,

---

<sup>1</sup> In fact, we know by personal communication that some of these optimizations are already included in the official ProGuard's list of future features.

it is possible to detect and remove unreachable branches. Note that, without this analysis, *removal of unused elements* optimization can fail and keep a method that actually will never be executed. Compilers often perform this analysis and some of them are able to remove unreachable branches under some conditions, like evaluation of constant boolean values.

Example:

Unresolved	<pre> <b>boolean</b>    debug_mode = <b>false</b>; <b>if</b> (debug_mode) {          // constant propagation     updateTimeCounter();    // kept but never executed } ... </pre>
Expected	<pre> <b>boolean</b>    debug_mode = <b>false</b>; ... </pre>

### *Devirtualization still unavailable*

*Devirtualization* optimization replaces slower virtual call instructions with faster static linked call instructions. In order to do that, methods are automatically made static, private and final when possible. As this optimization is not still available in the analyzed obfuscator, the programmer must assure that himself.

### *Merging of classes still unavailable*

*Merging of adjacent superclass* and *Merging of static classes* optimizations remove classes from class hierarchy, moving all methods and fields of a class to another class. These optimizations are not still available in the analyzed obfuscator. Besides, even when *Merging of adjacent superclass* optimization is implemented, it often does not manage to be applied due to the restrictions to keep the instantiated object size. So, programmer must always be careful about the number of classes.

### *Call graph considers objects instantiated anywhere, not only locally*

*Call graph* is a data structure that indicates which methods are reachable through each call site. The major difficulty in the call graph construction is the identification of possible executions from a virtual call. The analyzed obfuscator implements an effective enough algorithm [Bacon, 1997] that verifies if the method belongs to an instantiated classes from the class hierarchy. However, once the class is instantiated, its methods are considered reachable anywhere.

Example:

Unresolved	<pre> class A { void m() { ... } } class B extends A { void m() { ... } } ... static A a = new B(); // instantiating class B public void startApp() {     a = new A(); // instantiating class A     a.m(); // A.m() body is always executed } </pre>
Expected	<pre> class A { void m() { ... } } class B extends A { } // method B.m() could be removed ... static A a = new B(); public void startApp() {     a = new A();     a.m(); } </pre>

## *SITUATIONS THAT JEOPARDIZE OBFUSCATORS*

This section highlights undesirable practices that confuse or make some optimization inapplicable or make the obfuscator configuration hard. These situations must be avoided when possible but can be used if needed. Some practices include a table presenting a source code with *undesirable* practice; and a recommended approach as alternative solution.

### *Reflection API usage*

*Reflection* is the capability to refer some class, field or method by string statements, without knowing the exact element being accessed. Note that the name of the referenced element can be replaced by *class and member names compression*, so that the element is not found in execution time. If this feature is really needed, the user must inform the obfuscator to not change the element name.

Example:

Undesirable	<code>Class.forName("MyClass").newInstance();</code>
Recommended	<code>new MyClass();</code>

### *Relative resource addressing*

It is possible to refer to a resource relative to the class location in the package tree. Note that the package tree can be reorganized by *class package relocation*, so that the resource is not found in execution time. If this feature is really needed, it is necessary to relocate the resource too or to inform the obfuscator to not change the package name.

Example:

Undesirable	<code>Class.getResourceAsStream("image.png")</code>
Recommended	<code>Class.getResourceAsStream("\\...\\image.png")</code>

### *Unnecessary code*

Unnecessary code, especially method calls, field reading and class instantiation, must be strongly avoided. For example, *method inlining* considers the number of times the method is called to decide if it will be optimized; or if a field is read only once, it cannot be removed by *removal of write-only fields* optimization; and yet, instantiated classes automatically considers all its declared or inherited methods can be reached by virtual calls.

Example:

Undesirable	<pre> <b>if</b> (x &gt; y){     callMethod(x); } <b>else</b> {     callMethod(y); } </pre>
Recommended	<code>callMethod( (x &gt; y)? x : y );</code>

### *Throwing and catching exceptions*

Throw exceptions only when needed. It is slow and requires additional classfiles attributes. Moreover, methods that catch exceptions cannot be optimized by *method inlining* because it could change the program behavior. Do not use exception as control flow or to frequent user messages.

### *Synchronization*

Synchronized methods are about 10 times slower than normal methods [Hardwick, 2003]. Moreover, they are not optimized by *method inlining* available in analyzed obfuscator, since they implement an implicit lock.

### *Switches*

Switches are very large instructions and they require special treatment to be copied by *method inlining*, since the instruction length depends on its position in the code array. Our implementation rejected *method inlining* of calls to methods that belongs switches.

## OTHER RECOMMENDATIONS

This section includes all other general advices and recommendations not included in the previous cases, possibly not related to the obfuscators.

### *Do not initialize big arrays in line*

When initializing arrays in line, such as in the example, each array position initialization is compiled to an assign instruction. If some big array (e.g. more then 300 positions) needs to be initialized, consider loading it from some binary resource. In order to reduce final application size, the code for binary resource file generation should be coded in another application, outside the optimized source code.

Example:

Declaration	<code>int arr[] = { 0, 0, 1, 0, ... };</code>
Generated bytecode	<code>arr[0] = 0; arr[1] = 0; arr[2] = 1; ...</code>
Alternative solution to initialize array.	<pre>int[] arr = null; try {     InputStream file =         C.class.getResourceAsStream("/"+filename);     DataInputStream in = new DataInputStream(file);     arr = new int[in.readInt()];     for (int i = 0; i &lt; arr.length; i++) {         arr[i] = in.readInt();     } } catch (IOException e) {     e.printStackTrace(); } return arr;</pre>

### *Types byte, short, char and boolean are usually converted to int*

The Java Virtual Machine Specification [Lindholm & Yellin, 1999] almost always operates byte, short, char and boolean data as int, inclusive when loading and storing variables, storing constants, operating math instructions or evaluating branch conditions. *There are special instructions only for arrays of these types.* The effect over class fields is not imposed by specification and it depends on the virtual machine implementation. So, the programmer usually does not need to force the use of these types only for application code size optimization.

### *Avoid nested and anonymous classes*

Nested and anonymous classes are inner classes, declared inside the scope of other classes. However compilers create an entire classfile for each inner class,

including all internal structures. Even more, compilers still have to create some special methods and fields to allow an inner class to access its enclosing class' private information [Lindholm & Yellin, 1999]. Often, these classes can be replaced with some normal implementation.

### *Reuse objects*

It takes a long time only to create an empty object (about 13 times longer than assigning a field, for example) [Hardwick, 2003], so it is often worth updating the fields of an old object and reusing it rather than creating a new one. Moreover, it also reduces the garbage collector task, since fewer objects are removed.



## RELATED WORKS

Method inlining is a well-known optimization and it has been studied and implemented for decades since non-object-oriented programming languages, like Fortran and C [Allen & Johnson, 1988]. This section discusses the main method inlining researches and implementations that are related with our work somehow. Section 0 presents some language-independent method inlining researches and Section 0 discusses some method inlining implementations on compilers and tools. Section 0 discusses some related works on best programming practices.

### *LANGUAGE-INDEPENDENT METHOD INLINING RESEARCH*

Most of the specific research on method inlining found in the literature present the problem in an abstract way, independent from language, platform or application domain. They often explore the decision algorithm in order to maximize application performance, while trying to control the code expansion somehow. Since they are generic approaches, they hardly ever take full advantage of whole program environments..

Manuel Serrano [Serrano, 1997] proposed a method inlining optimization that controls the code size expansion, using a 'factor' initialized experimentally. The factor is reduced by 1 for each nested inlining, stopping when the value becomes zero. Thus, Serrano's work provides an unconventional and interesting approach for dealing with recursive call sites. Unfortunately, it is few compatible with our approach, due to the intrinsic local characteristic of Serrano's decision algorithm, since it analyzes each call site isolated.

Jeffrey Dean and Craig Chambers [Dean & Chambers, 1994] proposed a general decision algorithm where, for each call site, the inlining is performed, its cost and benefit are calculated and, in case it is not worthwhile, the process is reverted. The exact function of the cost and benefit of the inlining is left out, just citing the increase of the size and the performance gain as important factors. For optimization of the algorithm, the work proposes the creation of a database with previously performed analysis (named inlining trials) to be considered in future decisions about similar calls.

Its experiments stressed a compilation time reduction due to inlining trials. Since the cost and benefit are estimated for each call site, it is difficult to take in account a global analysis of the benefit from the removal of the method after optimized.

Vortex [Dean et al., 1996] is a language-independent optimizing compiler that performs object-oriented-focused optimizations on a low-level intermediate language. It was developed in order to unify the effectiveness evaluation of these optimizations over the application performance in several languages. Cross-module inlining is one of the optimizations mentioned as being implemented, however the author does not detail the decision algorithm being used or the parameters taken into account. All benchmarks are substantial in size and there are almost no results about code size increasing.

### *METHOD INLINING IN COMPILERS AND TOOLS*

Cross-module method inlining is effectively implemented only in very aggressive optimizing compilers and tools, which often have as main goal the application performance improvement. Most of these works are related to C/C++ compilers. Java imposes some additional important language specific issues, like the great number of virtual methods, provided they are virtual by default. That makes static analysis harder. Here, we discuss the most related method-inlining implementations.

Rainer Leupers [Leupers & Marwedel, 1999] presents the development of a function inlining approach for C compilers for embedded processors, imposing a global limit over final generated code size. His decision algorithm tries to find the method set that, when inlined, satisfies the limit and obtains the best execution performance. To do this, it requires several input parameters, including information about the real execution flow (profiling). Leupers' work, like ours, is very careful about the impact of method inlining on code size, however C is not an object-oriented language, which introduces many other difficulties, like polymorphism. Besides it does not consider the possibility of whole program optimizations, like including the removal method benefit as a parameter of the decision algorithm.

C++ and most ANSI C compilers allow the programmer to mark functions as a suggestion to be inlined (usually with an explicit "inline" function definition keyword) [Cline, 2003]. Unfortunately, the decision algorithm is completely unclear and the compilers can inline some, all, or none of the calls to a marked function, depending on many factors. Besides, the function can be removed only in some very restricted situations, like declared as static and linked under special directives. Therefore, users

have few guidelines when to mark a function to be inlined and when it will be really inlined or removed, so that there is no guaranteed impact in application size reduction.

Andrew Ayers [Ayers et al, 1997] developed a cross-module and multi-step optimizer that performs aggressive inlining optimization, supported by several low-level factors. Some of them include relying on the target instruction set to reduce the application size impact. Ayers's work, like ours, considers if the method will be removed after inlining as a parameter of the decision algorithm. The optimizer works on an intermediate language and was able to inline FORTRAN, C and C++ languages. Despite C++ is an object-oriented language, the Ayers' work focus mainly on procedural structures, regardless specificities of strongly object-oriented applications, such as Java applications.

Sun's Java HotSpot technology [HotSpot 1.4.1], an evolution of its Just In Time compilers (JIT), in principle can make inlining of frequently called methods in runtime, including replacement of calls with native code. However, one of the main features of this kind of compiler is the capability to undo the optimization if it is not worthwhile anymore, for example due to a new class loaded dynamically or to save execution memory. Therefore, the original bytecode copy of the optimized methods can never be removed, increasing the application size in runtime. Besides, Sun has published a CLDC HotSpot implementation [CLDC HotSpot] tuned to J2ME platform; however it does not implement method inlining at all.

David Whitlock's work [Whitlock, 2000] extends an academic tool, named BLOAT, implementing some inter-procedural optimizations on Java bytecode, among them method inlining. Whitlock's work presents some more details about implementation techniques and difficulties; however it has as main goal the improvement of the execution performance of the applications, not presenting enough concerns or results about the increase in the code size. In fact Nystrom [Nystrom, 1998] originally proposes the BLOAT tool, only with intra-procedural optimizations, like those found in compilers [Nullstone, 2002].

Obfuscators are also tools where method inlining can be found. We consider them the closest related work because they are also addressed to whole program optimizations. As shown in the study presented in Chapter 0, only a few obfuscators implement method inlining (we could find it in Jax and DashO).

## ***RELATED WORKS OF BEST PROGRAMMING PRACTICES***

There are really a myriad of articles, web sites and books that address best Java programming practices [O'Hanley, 2004] [Hardwick, 2003] [Klemm, 1999], however we were not able to find any publication that considers the usage of obfuscators or the impact of automated optimizations over those practices. O'Hanley [O'Hanley, 2004] presents a long list of good programming practices for some Java technologies and constructions, such as *Servlets*, *JSPs* and *Swing*, exceptions, constructors, serialization and so on. Hardwick's work [Hardwick, 2003] is an on-line collection of general recommendations for optimizing Java programs so that they are faster, smaller and more maintainable, however, the impact of optimization tools is not considered, as presented in this work. Klemm [Klemm, 1999] identifies and explains the main Java performance problems sources and it presents a list of source-level guidelines for accelerating Java applications, trying to reduce the object copy and allocation tasks.

Some other documents only recommend the use of obfuscators as a good practice for J2ME [Giguere, 2002] [Larson, 2002] [J2MEwtk 1.0.4], but they do not present any programming guideline to take more advantage of them. Giguere [Giguere, 2002] addresses some interesting guidelines to optimize J2ME application size and it recommends the use of obfuscators to shorten the names of packages, classes, methods and data members. Larson [Larson, 2002] strongly recommends obfuscator as a great way to reduce the application size. Sun's J2ME Wireless Toolkit [J2MEwtk 1.0.4] has already suggests the use of obfuscators since version 1.0.4.

We also could not find any survey about the most common optimizations implemented by obfuscators; instead, we only found some articles that compare compiler optimizations [Nullstone, 2002] [Hardwick, 2003]. Of course, the documentation of each obfuscator indicates what optimizations it implements, but rarely presents programming facilities and difficulties, such as our work.

## CONCLUSION

In this chapter, we present the most important contributions of this research and some future works.

### *CONTRIBUTIONS*

The strong demand for programs based on platforms with high memory and processing constraints, like cell phones, is pushing the implementation and use of optimization tools, such as obfuscators and shrinkers. So far, these tools have neglected the use of method inlining due to its classical problem of increasing code size. This study presents an original implementation of cross-module and whole-program technique for method inlining that improves both performance and application code size. The experimental results show that our technique is able to optimize and exclude around 50% of the reachable methods and calls, reducing the code size more than 3%, in average, and improving the performance up to 10%. These percentages vary according to the application architecture and the algorithm's parameterization. Anyway, our technique is able both to perform a reasonable performance improvement and to reduce application code size in most cases.

The key idea behind this surprising result is to take full advantage of specific features of the target Java Virtual Machine, like carefully handling runtime structures and exploring the safe whole program optimization opportunities of the J2ME platform. In fact, previous efforts fail to overcome the problem of code size increase when using method inlining, because the solutions are too general, disregarding the valuable languages' implementation and target environment specificities. Our results indicate that, in order to guarantee the best results of some optimization methods, such as inlining, the use of these specificities is unavoidable.

Unfortunately, developers often don't know used optimization tools enough and keep sacrificing the code quality in order to optimize their applications. This study shows that these tools, such as obfuscators, are safe but their effectiveness depends on the adopted programming practices. We noticed that to take best advantage of the obfuscators and to avoid unneeded sacrifices, it is essential that programmers (i) choose

a tool that satisfies the project requirements, considering its available optimizations and (ii) know how these optimizations can affect the programming and design decisions.

In order to help the programmer in choosing the obfuscator, this work presented an original study identifying which optimizations are most common in current obfuscators and where their implantations differs. It also identifies trends of new optimizations being implemented and gives some guidelines about what else could be taken into account to choose the tool.

Besides, in order to help developers to program using obfuscators, we introduced a set of best programming practices, organized in situations not resolved by the tools; situations well resolved by the tools and situations that jeopardize the tools usefulness. Despite their obvious importance, these practices have not been enumerated or discussed in the scientific or technical literature so far.

Our Extended version of ProGuard and the presented programming practices have been tested and used by an industrial software development team in C.E.S.A.R/Meantime [CESAR/Meantime]. We submitted our extensions to ProGuard maintainers to contribute the official open-source version.

This work is strongly motivated by the J2ME platform, because of its clear need for space and efficiency optimization and for allowing safe whole program optimization. However, it is important to stress that none of the low level features we have explored in our inlining technique are J2ME-specific. The proposed method inlining can be directly used in any Java platform. Besides, we believe our technique may acquire as good results as whole program analysis is possible, regardless Java platform.

## ***FUTURE WORK***

In spite of the transformations described here have been implemented and tested in many real programs without known problems, we intend to investigate ways of formalizing proofs of the transformations in order to prove they really do not change the semantics of the application at all. To do that, we will probably need a set of rules for the Java instructions, similar to those defined by Borba in “A refinement algebra for object-oriented programming” [Borba et al, 2003].

Another promising work is to combine source code and bytecode level analysis to improve and generalize the transformations, so that we can take advantage of the best from the two approaches. One idea is to reconstruct compiler high-level structures, such

as control flow graph and data flow graph [Muchnick, 1997], from bytecode elements. Thus, we could take advantage of many source code optimization practices, such as reusing many techniques already available in compilers, as well as abstracting many details of the bytecode validation. On the same way, we still keep bytecode optimization advantages, such as the possibility of optimizing and test applications when no source code is available, as well as accurate optimization impact on code size and runtime structures.

Our decision algorithm was designed to benefit and assure control of code expansion, even if this implies to limit the number of methods that can be inlined. In many cases, our method inlining technique just rejects method, instead of trying any other solution that could result in code expansion. This is helpful regarding to our worries about code size expansion in order to cover J2ME constraints, however, some future works could investigate alternative approaches in order to make algorithm even more aggressive. Some promising examples are alternative approaches to deal with occurrence of polymorphic methods or cyclic calls.

Method inlining opens opportunities to other intra-procedural optimizations, such as the identification and removal of unused variables, and the pre-processing of operations on constants, among others [Nullstone, 2002]. These optimizations are frequently implemented in compilers. Unfortunately, as our method inlining technique processes directly the compiled bytecode, we are not able to reuse those optimizations. They must be implemented again at the bytecode level, and performed after method inlining. We believe that the benefit of these intra-procedural optimizations can improve even more the method inlining results, especially if included in the code size estimation of the decision algorithm.

We also intend to study and to evaluate other inter-procedural techniques, such as *devirtualization*, *merging of adjacent superclasses* and *merging of static classes*. These optimizations are not fully explored by most of the existing tools. To take full advantage of these techniques, we intend to keep the approach of fully exploring the implementation specificities, in order to assess whether the cost-benefit ratio is worthwhile, as in the case of method inlining.

Additionally, we intend to study in depth the problem of the insertion and removal of constant pool entries. It seems to be a rich problem, since changes in the constant pool is a determinant factor for a successful code size reduction and still a risk to the aggressive approach of method inlining.

Finally, we also intend to format our best programming practices guide as a study on traditional design patterns [Gamma et al, 1995] in order to indicate how far each design pattern can benefit (or jeopardize) the most common optimizations found in obfuscators.



## REFERENCES

- [Allen & Johnson, 1988] Allen, R. Johnson, S. (1988) *Compiling C for Vectorization, Parallelization, and Inline Expansion*. In Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation. Atlanta, Georgia.
- [Ant Project] *The Apache Ant Project*. The Apache Software Foundation. Last change September, 2004. <http://ant.apache.org/>
- [Ayers et al, 1997] Ayers A., Schooler B., Gottlieb R. (1997) *Aggressive inlining*. In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97). Las Vegas, Nevada.
- [Bacon, 1997] Bacon, D. F. (1997) *Fast and Effective Optimization of Statically Typed Object-Oriented Programs*. PhD thesis, Computer Science Division, University of California, Berkeley. December, 1997. Report No. UCB/CSD-98-1017.
- [Borba et al, 2003] Borba P., Sampaio A., Cornélio M. (2003) *A Refinement Algebra for Object-Oriented Programming*. In Proceedings of 17th European Conference on Object-Oriented Programming ECOOP'2003. Darmstadt, Germany. July, 2003.
- [Cameron & Day, 1998] Cameron, C. Day, B. (1998) *Knuckletop Computing: The Java Ring*. In Sun Microsystems' web site. <http://java.sun.com/features/1998/03/rings.html>
- [CDC 1.0] Sun Microsystems. *CDC - Connected Device Configuration, v1.0a*. JCP Specification, JSR 036. <http://jcp.org/aboutJava/communityprocess/final/jsr036/>
- [CESAR/Meantime] C.E.S.A.R/Meantime. *Centro de Estudos Avançados do Recife. Meantime Mobile Games*. Recife, PE. <http://www.meantime.com.br>
- [CLDC 1.0] Sun Microsystems. *CLDC - Connected, Limited Device Configuration*. JCP Specification, JSR 030. <http://jcp.org/aboutJava/communityprocess/final/jsr030/>

- [CLDC 1.1] Sun Microsystems. *CLDC - Connected, Limited Device Configuration, v1.1*. JCP Specification, JSR 139. <http://jcp.org/aboutJava/communityprocess/final/jsr139/>
- [CLDC HotSpot] Sun Microsystems. (2003) *The CLDC HotSpot Implementation Virtual Machine – White Paper*. May, 2003. [http://java.sun.com/products/cldc/wp/CLDC\\_HotSpot\\_WhitePaper.pdf](http://java.sun.com/products/cldc/wp/CLDC_HotSpot_WhitePaper.pdf)
- [Cline, 2003] Cline M. (2003) *C++ FAQ Lite: Inline functions*. Last change March, 2003. <http://burks.brighton.ac.uk/burks/language/cpp/cppfaq/inline-functions.html>
- [Dahm, 2002] Dahm, M. (2002) *BCEL Byte Code Engineering Library 4.4.1*. The Apache Jakarta Project. Last change December, 2002. <http://jakarta.apache.org/bcel>
- [DashO] PreEmptive Solutions. *DashO Embedded Edition documentation*. <http://www.preemptive.com/>
- [Dean et al, 1995] Dean, J. Grove, D. Chambers, C. (1995) *Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*. In Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95). Aarhus, Denmark.
- [Dean & Chambers, 1994] Dean, J. Chambers, C. (1994) *Towards Better Inlining Decisions Using Inlining Trials*. In Proceedings of the ACM Conference on Lisp and Functional Programming Languages (LFP'94). Orlando, Florida.
- [Dean, 1996] Dean, J. (1996) *Whole-Program Optimization of Object-Oriented Languages*. Technical Report TR-96-06-02, Department of Computer Science and Engineering, University of Washington.
- [Dean et al, 1996] Dean, J., DeFouw, G., Grove, D., Litvinov, V., Chambers, C. (1996) *Vortex: An Optimizing Compiler for Object-Oriented Languages*. In Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'96). San Jose, California.
- [EmbeddedJava] Sun Microsystems. *EmbeddedJava Application Environment*. <http://java.sun.com/products/embeddedjava/>
- [FP 1.0] Sun Microsystems. *Foundation Profile Specification, v1.0a*. JCP Specification, JSR 046. <http://jcp.org/aboutJava/communityprocess/final/jsr04>

- [6/](#)
- [Gamma et al, 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.
- [Giguere, 2002] Giguere, E. (2002) *Optimizing J2ME Application Size*. In the Sun Microsystems' web. Posted in February, 2002. <http://developers.sun.com/techtopics/mobility/midp/ttips/appsize/>
- [Green] Sun Microsystems. *A Brief History of the Green Project*. <http://today.java.net/jag/old/green/>
- [Hansmann, 2003] Hansmann, U. Merk, L. Nicklous, M., Stober, T. (2003) *Pervasive Computing: Second Edition*. Springer Verlag.
- [Hardwick, 2003] Hardwick, J. (2003) *Java Optimization*. Last change April, 2003 – <http://www-2.cs.cmu.edu/~jch/java/optimization.html>
- [HotSpot 1.4.1] Sun Microsystems. (2002) *The Java HotSpot Virtual Machine, v1.4.1 – White Paper*. September, 2003. [http://java.sun.com/products/hotspot/docs/whitepaper/Java\\_Hotspot\\_v1.4.1/JHS\\_141\\_WP\\_d2a.pdf](http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf)
- [J2EE] Sun Microsystems. *Java 2 Platform, Enterprise Edition (J2EE)*. <http://java.sun.com/j2ee/>
- [J2ME] Sun Microsystems. *Java 2 Platform, Micro Edition (J2ME)*. <http://java.sun.com/j2me/>
- [J2MEwtk 1.0.4] Sun Microsystems. *J2ME Wireless Toolkit v1.0.4 documentation*. <http://java.sun.com/j2me/>
- [J2SE] Sun Microsystems. *Java 2 Platform, Standard Edition (J2SE)*. <http://java.sun.com/j2se/>
- [JavaCard] Sun Microsystems. *Java Card Technology*. <http://java.sun.com/products/javacard/>
- [Jax] IBM Research. *Jax Project*. Posted in June, 1998. <http://www.alphaworks.ibm.com/tech/JAX>
- [JikesBT] IBM AlphaWorks. *JikesBT. Jikes Bytecode Toolkit*. Posted in March, 2000. <http://www.alphaworks.ibm.com/tech/jikesbt>
- [Jshrink] Eastridge Technology. *JShrink*. Copyright 1997-2004. <http://www.e-t.com/jshrink.html>
- [Klemm, 1999] Klemm, R. (1999) *Practical Guidelines for Boosting Java Server Performance*. In Proceedings of the ACM

- 1999 on Java Grande Conference. San Francisco, California.
- [Knudsen, 2002a] Knudsen, J. (2002) *Obfuscating MIDlet Suites with ProGuard*, In the Sun Microsystems' web site. Posted in August, 2002. <http://developers.sun.com/techtopics/mobility/midp/ttips/proguard/>
- [Knudsen, 2002b] Knudsen, J. (2002) *Understanding MIDlet Memory*. In the Sun Microsystems' web site. Posted in June, 2002. <http://developers.sun.com/techtopics/mobility/midp/ttips/memory/>
- [KVMds] Sun Microsystems. *The K Virtual Machine - Data Sheet*. <http://java.sun.com/products/cldc/ds/>
- [KVMwp] Sun Microsystems. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices – White Paper*. <http://java.sun.com/products/kvm/wp/KVMwp.pdf>
- [Lafortune] Lafortune, E. *ProGuard documentation*. Source Forge. <http://proguard.sourceforge.net/>
- [Larson, 2002] Larson, E. D. (2002) *J2ME Optimization Tips and Tools*. In Sun Microsystems' web site. Posted in November, 2002. <http://developers.sun.com/techtopics/mobility/midp/ttips/optimize/>
- [Leupers & Marwedel, 1999] Leupers, R., Marwedel, P. (1999) *Function Inlining under Code Size Constraints for Embedded Processors*. In Proceedings of the International Conference on Computer-Aided Design (ICCAD). San Jose, California.
- [Lindholm & Yellin, 1999] Lindholm, T., Yellin, F. (1999) *The Java Virtual Machine Specification – Second Edition*. Sun Microsystems. <http://java.sun.com/docs/books/vmspec/>
- [MIDP 1.0] Sun Microsystems. *MIDP - Mobile Information Device Profile*. JCP Specification, JSR 037. <http://jcp.org/aboutJava/communityprocess/final/jsr037/>
- [MIDP 2.0] Sun Microsystems. *MIDP - Mobile Information Device Profile, v2.0*. JCP Specification, JSR 118. <http://jcp.org/aboutJava/communityprocess/final/jsr118/>
- [Muchnick, 1997] Muchnick, S. S. (1997) *Advanced Compiler Design and Implementation*. Morggan Kaufmann Publishers. 1997. ISBN 1-55860-320-4.

- [Nullstone, 2002] Nullstone Corporation. (2002). *NULLSTONE Optimization Categories*. <http://www.nullstone.com/htmls/category.htm>
- [Nystrom, 1998] Nystrom, N. J. (1998) *Bytecode-level analysis and optimization of Java classes*. Master dissertation, Department of Computer Science, Purdue University. West Lafayette, Indiana. August, 1998.
- [O'Hanley, 2004] O'Hanley, J. (2004) *Java Practices - Home*, Canada. <http://www.javapractices.com/>
- [Ortiz, 2002] Ortiz, C. E. (2002) *A Survey of J2ME Today*. In Sun Microsystems' web site. Posted in November, 2002. <http://developers.sun.com/techtopics/mobility/getstart/articles/survey/>
- [PBP 1.0] Sun Microsystems. *Personal Basis Profile Specification, v1.0*. JCP Specification, JSR 129. <http://jcp.org/aboutJava/communityprocess/final/jsr129/>
- [PDAP] Sun Microsystems. *PDA Optional Packages for the J2ME Platform*. JCP Specification, JSR 075. <http://jcp.org/aboutJava/communityprocess/final/jsr075/>
- [PersonalJava] Sun Microsystems. *PersonalJava*. <http://java.sun.com/products/personaljava/>
- [Pessoa, 2001] Pessoa, C. *wGEM: um Framework de Desenvolvimento de Jogos para Dispositivos Móveis*. Master dissertation. Centro de Informática. Universidade Federal de Pernambuco, Recife, Pernambuco. November, 2002.
- [PP 1.0] Sun Microsystems. *Personal Profile Specification*. JCP Specification, JSR 62. <http://jcp.org/aboutJava/communityprocess/final/jsr062/>
- [RetroGuard] Retrologic. *RetroGuard 1.1.9 User's Guide*. Copyright 1998-2004. <http://www.retrologic.com/>
- [Serrano, 1997] Serrano, M. (1997) *Inline expansion: when and how?*. In Proceedings of the 9th International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97). Southampton, New York.
- [Tip et al, 1999] Tip, F. Laffra, C. Sweeney, P. F. (1999) *Practical Experience with an Application Extractor for Java*. In Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming

Systems, Languages, and Applications (OOPSLA'99), Denver, Colorado.

[Tip & Palsberg, 2000]

Tip, F. Palsberg, J. (2000) *Scalable propagation-based call graph construction algorithms*. In Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00). Minneapolis, Minnesota.

[Whitlock, 2000]

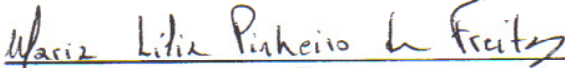
Whitlock, D. M. (2000) *Persistence-Enabled Optimization of Java Programs*. Master dissertation, Department of Computer Science, Purdue University. West Lafayette, Indiana. May, 2000.




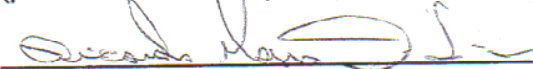
SERVIÇO PÚBLICO FEDERAL  
UNIVERSIDADE FEDERAL DE PERNAMBUCO  
CENTRO DE INFORMÁTICA  
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Ata de Defesa de Dissertação de Mestrado do  
Centro de Informática da Universidade Federal  
de Pernambuco, em 30 de agosto de 2004.

Ao trigésimo dia do mês de agosto do ano dois mil e quatro, no Centro de Informática da Universidade Federal de Pernambuco, às catorze horas, teve início a defesa de dissertação do Mestrado em Ciência da Computação intitulada **"Otimizações Bytecode Java na Plataforma J2ME"** do candidato **Tarcísio Pinto Câmara**, o qual já havia preenchido as demais condições exigidas para a obtenção do grau de mestre. A Banca Examinadora composta pelos professores Paulo Henrique Monteiro Borba, pertencente ao Centro de Informática desta Universidade, Ricardo Massa Ferreira Lima, pertencente à Escola Politécnica de Pernambuco da Universidade de Pernambuco, e André Luis de Medeiros Santos, pertencente ao Centro de Informática desta Universidade, sendo o primeiro presidente da Banca Examinadora e o último orientador do trabalho de dissertação, resolveu: **Aprovar por unanimidade e dar o prazo de trinta dias para a entrega da versão final do trabalho.** E para constar lavrei a presente ata que vai por mim assinada e pela Banca Examinadora. Recife, 30 de agosto de 2004.

  
\_\_\_\_\_  
Maria Lília Pinheiro de Freitas  
(secretária)

  
\_\_\_\_\_  
Prof. Paulo Henrique Monteiro Borba  
(primeiro examinador)

  
\_\_\_\_\_  
Prof. Ricardo Massa Ferreira Lima  
(segundo examinador)

  
\_\_\_\_\_  
Prof. André Luis de Medeiros Santos  
(terceiro examinador)

**Camara, Tarcisio Pinto**  
**Otimização bytecode Java na plataforma J2ME /**  
**Tarcisio Pinto Camara. – Recife: O autor , 2004.**  
**76 folhas : il., fig., tab.**

**Dissertação (mestrado) – Universidade Federal de**  
**Pernambuco. CIN. Ciência da Computação, 2004.**

**Inclui bibliografia.**

**1. Compressão de Dados. 2. Ofuscadores de**  
**Bytecode J2ME. 3. Method Inlining. 4. Boas Práticas de**  
**Programação. I. Título.**

**005.746**

**CDD (22.ed.)**

**CIN2006-024**