# Hold the Sources: A Gander at J2ME Optimisation Techniques

Patrik Mihailescu, Habin Lee, and John Shepherdson

Intelligent Systems Lab, BT Exact
B62 MLB1/PP12 Adastral Park, Martlesham Heath, IP5 3RE, UK

**Abstract.** With the advent of the Java 2 Micro Edition platform, developers have the ability to develop Java based mobile applications that benefit from features such as device independence, and memory abstraction. However, applications also inherit many limitations such as slow execution, and excessive memory usage that impact on overall application performance and usability. The aim of this paper is to present and evaluate six known optimisation techniques for improving the performance and usability of Java based mobile applications. Some of these techniques are dependent on features provided within the IBM WebSphere Studio Device Developer IDE. A real life multi-agent based mobile application is presented to demonstrate the performance and usability improvements that have been gained through applying these optimisation techniques.

## 1 Introduction

In the past, developing applications for mobile computing devices has typically been difficult due to proprietary development tools that required developers to have an intimate knowledge of the device and/or Operating System (OS) functionality. This is now changing with the advent of the Java 2 Micro Edition (J2ME) platform [1]. The J2ME platform provides developers with an alternative approach for developing mobile applications through the use of a common object-oriented programming language that hides the complexity and features of the underlying OS and device.

The J2ME platform is targeted at a wide spectrum of mobile computing devices such as mobile phones, household appliances, and Personal Digital Assistants (PDAs). To cater for the different levels of device functionality, where for instance, devices may provide anywhere between 64KB to 1MB of heap memory, the J2ME platform defines the concept of profiles [1]. A profile defines a set of APIs that are applicable for a similar group of devices, such as mobile phones. Using this approach, each profile can be optimised for a specific device group by only including APIs that are relevant to the available features and functionality of the device. However, each profile must also provide support for a common set of APIs that are core to the Java programming language.

Due to the interpreted nature of the Java programming language, the J2ME platform suffers from a number of limitations that affect the overall application performance. This includes slow execution, low heap memory, non-predictable garbage collection, memory fragmentation, and varying performance on different mobile computing devices.

These limitations, when combined with others, can have a confounding effect on the usability of mobile applications. Typically users of mobile computing devices only interact for very short periods of time with applications, therefore any start-up delays or slow application responses have a negative impact on users. The aim of this paper is to present and evaluate six known optimisation techniques for improving both the performance and usability of J2ME based mobile applications. Although several of these optimisation techniques are dependent on the features provided within the IBM WebSphere Studio Device Developer (WSDD) IDE, they can still be applied to other IDEs that provide equivalent features. We apply these optimisation techniques to a real life multi-agent based mobile application to measure the actual performance improvements.

The outline of this paper is as follows; in the next section we provide an overview of related work and in section 3 we provide information regarding each of the six optimisation techniques. In section 4, we apply these techniques to the real-life multi-agent based mobile application, and evaluate the performance improvements. Finally, in section 5, we conclude this paper.

## 2 Related Work

There is a wealth of information available on general optimisation techniques for the Java programming language, and those specific to the J2ME platform. Techniques include [11] [13]: minimizing inheritance, reusing objects, using alternative approaches to method synchronisation, eliminating inner classes, avoiding string concatenation, using shorter method/class names, using lazy class loading, etc. The difficulty in applying these optimisation techniques is the ability to pinpoint exactly which technique has improved performance and by how much.

The majority of these techniques are typically applied during the design and coding stages of application development. Whereas the optimisation techniques presented in this paper are applied during the testing and deployment stages of application development and each can be individually measured to work out the actual performance gains achieved.

[8] provides a performance comparison between the CDC specification (part of the J2ME platform), Java 2, and Java 1.1. The static and dynamic footprint of each VM was measured, including six individual tests that measured areas such as object creation, threading, and I/O. Our work differs in that we provide optimisation techniques that are designed to improve the performance of a J2ME based mobile application.

## 3 Optimisation Techniques Overview

The optimisation techniques presented within this paper are aimed at optimising an application without modifying the source code. These techniques can be grouped into three levels: 1) *VM*, tailoring a VM for a particular target environment, 2) *Deployment*, optimising an application through techniques such as code reduction, and 3) *Runtime*, fine-tuning the operational performance of both the VM and application. All three levels can be viewed as part of an optimisation stack, where each level focuses on specific optimisation techniques that can be applied independently or combined together. An additional level can be added that applies the techniques mentioned in the related work section.

Several of the techniques presented within this paper are dependent on the features provided within the IBM WSDD IDE [4]. The WSDD IDE enables the development of applications based on the J2ME platform. This IDE supports development for a diverse set of mobile OS's such as the Palm OS, and the QNX OS. For each OS, a customisable VM is provided which can be tailored to suit both the target environment, and application requirements.

The WSDD IDE supports the majority of the J2ME profiles currently defined within the Java Community Process (JCP) such as the Foundation profile [3], and the Mobile Information Device Profile (MIDP) [2]. In addition, a supplementary set of custom profiles (which have no relationship to the J2ME profiles) is included. Custom profiles can be used within environments, which require full customisation over the APIs bundled with the VM due to the limited availability of computing resources.

The optimisation techniques that are not dependent on the context of an application have been tested on an O2 XDA running the Pocket PC OS. Before an optimisation technique was tested, the XDA was reset. Each test was performed forty times, unless otherwise stated. Finally, we used version 5.5 of the IBM WSDD IDE. We will now discuss the optimisation technique within the VM level, followed by those within the deployment level, and conclude with the runtime level techniques.

## 3.1 VM Level Optimisation: Customising the VM

Mobile computing devices differ significantly from each other in terms of not only their hardware, and OS functionality but also their physical appearance, and usability properties. Therefore, a general purpose VM that contains a set of generic APIs is not suitable, and will impact on the overall performance and usability of an application. This optimisation technique is focused on customising the VM for both the target environment and application.

As mentioned, the WSDD IDE provides a customisable VM for each OS, which can be modified in two ways: i) *VM functionality* and ii) *API set*. Depending upon the target OS, each VM is comprised of an initial set of files that provide minimum level functionality such as dynamically loading Java classes.

By default, a VM is not configured to use any particular profile. Each profile is comprised of two parts: i) a file of classes (in JAR format) containing all the Java APIs, and ii) a corresponding DLL containing the native implementation. There is no fixed limit on the number of profiles that can be bundled with a VM.

To demonstrate the benefits of this optimisation technique we measured the start-up time and static footprint size of three customised VMs. Two of the VMs have been customised for a single profile, while the third VM has been set-up to use any profile type. All three VMs contain the same level of VM functionality.

The results of this test are presented in Table 1. The first two rows contain the results for a VM that has been customised for a single profile: 1) *JCL Xtr*, (a custom profile), and 2) *Foundation*. The last row contains the results for a VM that can use any type of profile; in this case, it has been bundled with the Foundation profile. The results show that customising the VM can not only save storage space, but also significantly reduce an application's start-up time.

**Table 1.** Test results from customising the VM optimisation technique.

| Profile type | Static footprint | VM start-up |
|---|---:|---:|
| JCL Xtr | 656KB | 799ms |
| Foundation | 1689KB | 1962ms |
| Foundation (Generic VM) | 1986KB | 2172ms |

By using a smaller profile (JCL Xtr), a storage space saving of 61% was achieved, and the VM start-up time was also reduced by 59%, when compared to using the Foundation profile (row two). If it is not possible to use a smaller profile, a saving can still be gained by customising the VM to a specific profile when compared to just using a generic VM (row three). A saving of 15% on storage space, and a 9.5% reduction in VM start-up was achieved.

## 3.2 Deployment Level Optimisation (1): Application Deployment

Effectively deploying applications on mobile computing devices is an important issue that not only affects an applications performance but also its usability. Lack of storage space, and application start-up delays are just two issues that need to be addressed.

Instead of using the JAR format, this optimisation technique utilises an alternative deployment format provided by the WSDD IDE, JXE (J9 Executable format) [7]. The JXE format reduces the delay when launching an application, and lowers the resource usage of the VM when loading Java classes. This is achieved initially by converting all the classes required by an application into a JXE specific format, and then ROMizing them into a single image. Within this image, all the classes have been resolved, and their locations (address) and those of their methods have been determined. Therefore, at runtime the level of processing required by the VM during the class loading process can be substantially reduced.

An additional benefit of the JXE format is the ability to create Execute-In-Place (XIP) applications. XIP applications can be placed into ROM, where they can be directly executed, thereby eliminating the need to keep two copies of the same class in memory. To demonstrate the benefits of this optimisation technique, we compared the start-up time of a VM that is customised for the

Foundation profile which has its classes packaged in both JAR and JXE format. The results of this test are presented in Table 2, which shows that while the JAR format achieved a better compression ratio by 19.5%, the JXE format reduced the VM start-up time by 25%.

**Table 2.** Application deployment optimisation number one test results.

| Deployment format | Profile class storage space | Start-up time |
|---|---|---|
| JAR | 990KB | 1963ms |
| JXE | 1149KB | 1464ms |

## 3.3 Deployment Level Optimisation (2): Code Optimisation

The second deployment level optimisation is aimed at reducing the overall size of an application, and improving (in certain parts) its execution. This is achieved by applying known code reduction and code optimisation techniques.

Code reduction involves removing unused classes, methods or fields from an application. Furthermore, techniques such as code obfuscating, compression, and stripping debug information from classes can also be applied. Code optimisation involves improving the performance of certain parts of an application's code through techniques such as:

– In-lining methods [12], to reduce the overhead during method invocation.
– Call site devirtualization [6], which can be applied to reduce the overhead when invoking interface methods by replacing them with static methods. This can only be applied to methods, which at compile time always refer to the same methods at runtime.
– Reducing the level of sub-classing and method overloading through analysis techniques to determine which methods/classes can be declared as final.
– Pre-compiling classes/methods to native code using Ahead of Time (AOT) compilation [10]. This is discussed further in section 3.5.

The WSDD IDE supports all of the techniques presented above.

## 3.4 Runtime Level Optimisation (1): Persistent VM

To enable 'instant on' applications, this technique focuses on further reducing the start-up delay experienced when launching an application. This technique ensures that only one instance of the VM is loaded, and that all applications are executed within this single instance. Typically on mobile computing devices, Java applications are launched via a shortcut that invokes the VM executable and passes additional parameters such as the name of the main application class.

This optimisation technique introduces an alternative approach to launching a Java application. This involves developing a native application to act as a Java application launcher. This native application will ensure that only one instance of the VM is created, and that all Java applications are executed within this single instance. To work out if an instance of the VM is available, a dedicated background Java process will need to be launched within the VM, the first time the VM is created. This running process is known as the VM registry, and is responsible for loading Java applications when requested by a Java application launcher.

Both the Java application launcher and the VM registry process use the Windows messaging API to communicate with each other. An example of how this optimisation technique works is provided below:

1. The first time an application (A) is started, its application launcher (La) will need to create an instance of the VM (as none is available), and provide it with two arguments: 1) the main class of the VM registry process, and 2) the main class of the Java application.
2. Once the VM registry process is started, it will use the second argument provided by application launcher (La) to load the requested Java application.
3. If another application (B) is launched, its application launcher (Lb) will attempt to find a running instance of the VM by using the 'FindWindow' API to obtain a handle to the VM registry process.
4. Once the VM registry process is located, application launcher (Lb) sends it a message using the 'SendMessage' API requesting that it loads application (B). Within this message, the main class of application (B) is provided.
5. Upon receiving this message, the VM registry loads application (B).

## 3.5 Runtime Level Optimisation (2): Ahead of Time Compilation

A common technique for improving the performance of a Java application is through the use of a Just-in-Time (JIT) compiler. A JIT compiler is used to dynamically convert frequently interpreted bytecode into native code during the execution of an application. However, in certain environments, a JIT compiler may be unsuitable due to the additional computing resources required for its inclusion and subsequent execution. For example on the Pocket PC OS, the JIT compiler requires at least 1046 kb of storage memory.

This optimisation technique makes use of an alternative approach for converting bytecode into native code, which is known as Ahead of Time (AOT) compilation [10], available as part of the WSDD IDE. AOT can be used at build-time to convert specific parts of an application's code into native code, which can then be directly executed at runtime by the VM without the need for a JIT equivalent compiler. Typically, only a small portion of an application should be converted into native code using AOT compilation, as with each converted method the overall application size increases. Therefore, this may be unsuitable for environments where storage space is a premium.

Unfortunately due to a problem with the current version of the VM for the Pocket PC OS, AOT compiled classes are not handled, and as such we are unable to measure the benefits of this technique.

### 3.6 Runtime Level Optimisation (3): VM Runtime Settings

Under each OS, a set of default runtime settings is provided to the VM that determines its operational settings. The default values provided are general in nature and may hinder application performance. Examples of the type of settings that can affect application performance include:

**-Xmca:** This setting determines the unit of growth for RAM memory when more memory is required to load classes. A high value is useful for applications with a large set of classes.

**-Xmco:** This is similar to the -Xmca setting, except it is used for ROM memory.

**-Xmx:** This setting represents the maximum amount of RAM memory that can be consumed by the VM during its execution.

**-Xmoi:** This setting determines the unit of growth for RAM memory when more heap memory is required.

**-Xmo:** This setting represents the initial size of heap memory available for an application. Once this value is reached, garbage collection is performed, and if no memory can be freed then the size of this memory space is increased based on the value of the -Xmoi setting. Depending upon the application context/requirements, a high value may be set to avoid garbage collection, thereby maintaining application responsiveness.

**-Xmn:** This setting determines the memory space size where an application's newly created objects are placed within. Once an object survives the garbage collection process ten times, it will be placed within the application's heap memory. Setting this to a high value will increase the length of time taken by the garbage collector to free unused objects.

**-Xiss:** This setting represents the initial stack size of a Java thread. This is useful for applications that create a large number of threads.

## 4 Real Life Application

We have applied these optimisation techniques to improve the performance and usability of a real life multi-agent based mobile application. This application provides a team-based approach for job management in the field of telecommunications service provision and maintenance. Jobs are assigned to teams of engineers based on pre-defined business rules which use information such as an engineer's geographic location and skill set. A variety of services are available that include: real-time job updating, job trading, job delivery (both push and pull mode), and travel planning. Further details regarding this application are provided within [9].

The underlying multi-agent platform used within this application is the JADE-LEAP platform [5]. The JADE-LEAP platform is an optimised version of the JADE platform that has been designed to run on a variety of mobile computing devices. An agent execution environment known as a lightweight container is provided that includes services such as messaging, service management, and task scheduling to locally executing agents.

Engineers run this application on a mobile computing device such as an XDA, and connect back to an Intranet via a secure Virtual Private Network (VPN) connection over a GPRS network where they can access services such as job retrieval and job update. In the following sub-sections, we evaluate the performance improvements gained as a result of applying the presented optimisation techniques.

## 4.1 Application and VM Deployment

We applied the deployment level one (JXE format) and two (code reduction) optimisation techniques to our application. For the deployment level two technique, we used most of the code reduction/optimisation techniques on two files: 1) JADE-LEAP platform classes, and 2) application classes. We did not optimise the Foundation profile's classes.

For both files we applied three common code reduction techniques: 1) compression, 2) omitting debug information, and 3) removing unused classes, methods and fields. We also applied one code optimisation technique, in-lining methods. We only applied the Call site devirtualization and class/method final declaration technique to our application's classes. Approximately 68 methods were inlined within our application, and 368 within the JADE-LEAP platform. Lastly, 61 methods were declared final and 826 methods were devirtualized within our application's classes.

Table 3 provides details as to the storage requirements for our application, the JADE-LEAP platform and the Foundation classes. The first row shows the storage size for each file that has been optimised through the above mentioned techniques. The second row shows the storage size for each file when deployed in JAR format. Both the JADE-LEAP platform and our application classes have been optimised through two common code reduction techniques: 1) compression, and 2) omitting debug information. We also removed unused classes/methods from the application classes. NOTE: When applying the VM level optimisation technique one, we customised the VM specifically for the Foundation profile.

## 4.2 VM Runtime Configuration

To meet both the operational and usability application requirements, we made a number of modifications to the VM runtime settings. These include:

**-Xmca:** This value was doubled to 32K, as both the underlying JADE-LEAP platform and our application load a number of classes dynamically.

**Table 3.** Storage requirements for our application.

| | Application | JADE-LEAP platform | Foundation profile |
|---|---|---|---|
| JXE | 353KB | 633KB | 1149 KB |
| JAR | 284KB | 756KB | 900KB |

**-Xmx:** This value was increased from 8MB to 32MB, to improve the responsiveness of the user interface, and to ensure that adequate memory was available.

**-Xmoi:** This value was increased from 64K to 96K for similar reasons as the previous setting.

**-Xmno:** This value was increased from 256K to 328K, to provide adequate memory for newly created objects.

**-Xmn:** This value was increased from 256K to 328K, to minimize the number of times required to grow the application's heap memory.

**-Xiss:** This value was doubled to 4K as the underlying network subsystem of the JADE-LEAP platform is heavily multi-threaded.

### 4.3 Application Start-up Reduction (1)

To measure the start-up performance improvements gained through applying the optimisation techniques mentioned in the previous section, we compared the time taken to load the two versions of our application. The results from this test are shown in Table 4. To reduce the affects of network fluctuations for each result, we eliminated the highest two values, and lowest two values. Through applying the deployment level one, two and VM level optimisation technique we reduced the application start-up time by approximately 9.6%, equivalent to 1.9 seconds.

**Table 4.** Results for improving the start-up time of our application.

| Type | VM start-up | Platform start-up | Application start-up |
|---|---|---|---|
| JAR | 2234ms | 13904ms | 3677ms |
| JXE | 1953ms | 12261ms | 3684ms |

### 4.4 Application Start-up Reduction (2), and Usability Improvement

As shown in Table 4, a significant amount of time of the overall application start-up time is spent in loading the underlying JADE-LEAP platform. Therefore to eliminate this delay, we not only applied the runtime level optimisation technique number one, but we also modified our application launch routine. Within the application launch routine, a check is performed to locate a running instance of the JADE-LEAP platform, and if an instance is found the application is attached to it (else one is created). Using this technique, we were able to reduce the start-up time of our application to approximately under four seconds.

An additional advantage of this optimisation technique was that it improved the usability of the application by ensuring that only a single instance of the application was executed. Within the Pocket PC OS, there is no easy way for a user to switch between multiple running applications. Typically users will re-select the application icon, which for a Java appplication results in launching another application instance. Within the Java application launcher if the user re-selects the application icon the launcher will attempt to locate a running instance of the application and if found, the application's window is displayed.

## 5 Conclusion

Within this paper we have presented a set of optimisation techniques for improving the performance and usability of J2ME based mobile appliations. Within the set, individual optimisations were provided that can be used independently or combined. We applied these techniques to a real world mobile application and demonstrated the performance improvements that were gained.

Future work includes evaluating the benefits of AOT compilation, and investigating other techniques that can address performance issues such as memory usage, network bandwidth, and battery power.

## References

1. Java 2 platform, micro edition, Frequently Asked Questions, January 2004. http://java.sun.com/j2me/reference/faqs/index.html.
2. JSR 118 Mobile Information Device Profile 2.0, January 2004. http://jcp.org/en/jsr/detail?id=118.
3. JSR 46 J2ME Foundation Profile, January 2004. http://jcp.org/en/jsr/detail?id=46.
4. Websphere Studio Device Developer Websphere software, January 2004.
5. M. Berger, S. Rusitschka, M. Schlichte, D. Toropov, and M. Watzke. Porting Agents to Small Mobile Devices - the Development of the Lightweight Extensible Agent Platform. *EXP in search of innovation special issue on JADE*, 3(3):32–41, 2003.
6. K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 294–310, 2000.
7. M. Kok. Developing a DB2 Everyplace Java Application using WebSphere Studio Device Developer., January 2002. http://www-106.ibm.com/developerworks/websphere/registered/tutorials/0212_kheng/kheng.html.
8. M. Laukkanen. Java on Handheld Devices - Comparing J2ME Cdc to Java 1.1 and Java 2, January 2004. http://citeseer.nj.nec.com/473890.html.
9. H. Lee, P. Mihailescu, and J. Shepherdson. A Multi-Agent System to Support Team-Based Job Management in a Telecommunications Service Environment. *EXP in search of innovation special issue on JADE*, 3(3):96–105, 2003.
10. G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, 1997.
11. J. Shirazi. *Java Performance Tuning (2nd edition)*. O'Reilly and Associates, 2003.
12. P. Tyma. Tuning Java Performance, January 2004. http://www.ddj.com/documents/s=962/ddj9604e/.
13. S. Wilson and J. Kesselman. *Java Platform Performance: Strategies and Tactics*. Addison-Wesley Pub Co, 2000.