Proceedings of the 5th WSEAS Int. Conf. on APPLIED INFORMATICS and COMMUNICATIONS, Malta, September 15-17, 2005 (pp181-186)

Optimisation Techniques for J2ME based Mobile Applications

PATRIK MIHAILESCU, HABIN LEE, and JOHN SHEPHERDSON
Intelligent Systems Research Centre, BT Group
B62 MLB1/PP12 Adastral Park, Martlesham Heath, IP5 3RE
UK
http://cigserver3.info.bt.co.uk/isrc/IBSR/

Abstract: - Now days there are a wealth of mobile application development tools available for developers to use, which are not limited to a particular device type. One of the most well-known development tools is the Java 2 Micro Edition platform. Using this platform, mobile application developers can benefit from the same features such as device independence and memory abstraction enjoyed by desktop application developers. However due to the interpreted nature of the Java programming language, applications also inherit its limitations such as excessive memory consumption, and slow execution. The aim of this paper is to present and evaluate six known optimistaion techniques for improving the performance of a Java based mobile application. These techniques are then applied to a real life multi-agent based mobile application to demonstrate the performance and usability improvements gained.

Key-Words: - J2ME, Performance, Optimisation, Multi-agent system, PDA

1 Introduction

Previously, when an application developer wanted to build an application for a mobile computing device, their options were limited. Typically they were forced to use proprietary development tools that required them to have an intimate knowledge of both tool and operating system, which inevitable restricted their application to a particular device type. Now days, application developers have greater choice through the advent of new development tools that free developers from dealing with low level hardware/operating system details, and enabling them to focus on enriching their applications regardless of device type.

The Java 2 Micro Edition (J2ME) platform is one example of the new generation mobile application development tools. The J2ME platform is targeted at a wide spectrum of mobile computing devices such as household applications, personal digital assistants (PDA), and mobile/smart phones. 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[4]. 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, such as those provided within the java.lang package (e.g. Object, String, System).

Due to the interpreted nature of the Java programming language, the J2ME platform suffers from a number of limitations that affect overall application performance. These include, execution speed, memory management, and implementation differences.

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 give a brief introduction into the WSDD IDE in section three. Section four contains detailed information regarding each of the six optimisation techniques. In section five we apply these techniques to a real-life multi-agent based mobile application, and evaluate the performance improvements. Finally in section six 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][14]:

limiting the level of inheritance, minimizing the level of object creation, using alternative approaches to method synchronisation, eliminating inner classes, avoiding string concatenation, using shorter method/class names, using lazy class loading, etc. The majority of these techniques are typically applied during the design and coding stages of application development.

The difficulty in applying these optimisation techniques is the ability to adequately measure the true performance improvements achieved. For example, it is difficult to pinpoint exactly which optimisation has resulted in an increase in application performance or a reduction in memory usage. The optimisation techniques presented within this paper are applied during the testing and deployment stages of application development. Each of the optimisation techniques presented within this paper, 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 mobile based application.

3 Overview of Websphere Studio Device Developer IDE

Before we present the six optimisation techniques, we briefly provide an overview of the IBM WSDD IDE [13], as several of the techniques are dependent on features provided within this IDE. The WSDD IDE enables the development of applications based on the J2ME platform. This IDE supports development for a diverse set of mobile computing OS's such as the Palm OS, the QNX OS, and the MontaVista Linux OS.

The WSDD IDE supports the majority of the J2ME profiles currently defined within the Java Community Process (JCP) such as the Foundation profile [5], and the Mobile Information Device Profile (MIDP) [6]. 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. Developers are able to tweak the APIs provided within custom profiles by excluding classes/methods that are not required. Developers are not permitted to modify any of the APIs contained within J2ME profiles.

4 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, 2) Deployment, and 3) Runtime. The VM level optimisation focus on tailoring a VM for a particular target environment, and application type. The deployment level optimisations focus on tailoring an application for its target environment through techniques such as code reduction. Lastly, the runtime level optimisations focus on fine-tuning the operational performance of the application.

All of the optimisation techniques presented within this paper have been tested on an XDA running the Pocket PC OS. The hardware properties of the XDA are listed in Table 1. Before an optimisation technique was tested, the XDA was reset (soft reset). Each test was performed forty times, unless otherwise stated. Finally, we used version 5.5 of the IBM WSDD IDE.

Device Property	Value
Model no	PW10A1
ROM version	3.17.03 ENG
Radio version	4.20.00
CPU type	ARM SA1110
Speed	206 Mhz
RAM size	64 MB
ROM size	32 MB

Table 1 Hardware properties of the XDA used to test each optimisation technique.

We will now discuss the optimisation technique that falls within the VM level, followed by those within the deployment level, and conclude with the runtime level techniques.

4.1 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 in section three, a fully customisable VM is available for each OS, which can be customised in two ways: A) VM functionality and 2) API set. Depending upon the target OS, each VM is comprised of an initial set of files that provide minimum level functionality. For example, the ability to dynamically load Java classes from the file system. Additional features can be installed as required, although each new feature added will consume certain computing resources.

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 specifically 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 2. 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, a J2ME profile. The last row contains the results for a VM that can handle any type of profile; in this case it has been bundled with only 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.

Profile Type	Static Footprint	VM Start-up time
JCL Xtr	656 kb	799 ms
Foundation	1689 kb	1962 ms
Foundation (Generic VM)	1986 kb	2172 ms

Table 2 Test results from customising the VM optimisation technique.

4.2 Deployment Level Optimisation (1): Application Deployment

Effectively deploying applications on mobile computing devices is an important issue that not only affects an application's performance but also its usability. Lack of storage space, and application start-up delays are just two issues that need to be addressed. Typically Java applications are deployed in JAR format, which provides a means of grouping Java classes within a compressed file.

This optimisation technique utilises an alternative deployment format provided by the WSDD IDE, JXE (J9 Executable format) [2]. The JXE format is designed to reduce the delay when launching an application, and lower 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 that 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. Compression is also supported.

To demonstrate the benefits of this optimisation technique, we compared the start-up time of a VM that is customised for a single profile (Foundation), which has its classes packaged in both JAR and JXE format. The results of this test are presented in Table 3.

Deployment Format	Storage Space	Start-up Time
JAR	900 kb	1963 ms
JXE	1149 kb	1464 ms

Table 3 Test results from the application deployment optimisation technique number one.

4.3 Deployment Level Optimisation (2): Code Redution/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. The WSDD IDE supports these techniques through a deployment tool called Smartlinker.

Code reduction involves removing any unused classes, methods or fields from an application. This can be performed automatically (by Smartlinker), based on working out, from an initial set of classes, the required references, or the classes can be manually specified. 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 applications code through techniques such as in-lining methods [12], call site devirtualization [3], and pre-compiling classes/methods to native code using Ahead of Time (AOT) compilation [10].

No test will be provided to measure the benefits of this optimisation technique, as it is dependent on the context of an application.

4.4 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. Therefore, only the first executed application needs to load the VM, all subsequently executed applications do not.

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 running, a dedicated background Java application will need to be executed within the VM, the first time the VM is created. This dedicated background Java application is known as the VM registry, and is responsible for executing Java applications when requested by a Java application launcher.

Both the Java application launcher and the VM registry application use the Windows messaging API to communicate with each other. Figure 1 provides a sample scenario, which demonstrates how this optimisation technique works when executing two different Java applications. When the first application (A) is launched an instance of the VM is created, including the VM registry, which will load application (A), and any other new applications (such as (B)).

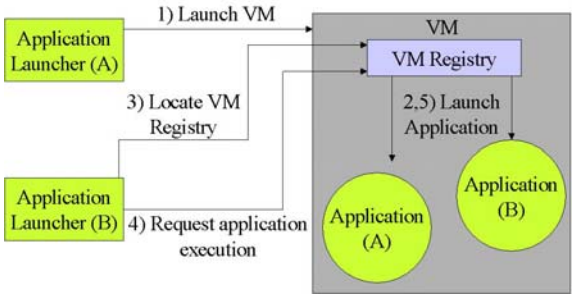

Fig. 1 Sample scenario demonstrating the use of the runtime level optimisation technique number one.

No test will be provided to measure the benefits of this optimisation technique.

5 Real Life Application

We have applied these optimisation techniques to improve both the performance and usability of a real life multi-agent based application that runs on mobile computing devices. 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 a set of pre-defined business rules 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 [1]. 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. To cater for the different types of mobile computing devices, two tailored versions of the JADE-LEAP platform are available, based on two separate J2ME profiles: 1) **MIDP**, and 2) **Foundation**. The JADE-LEAP platform provides an agent execution environment known as a lightweight container that provides services such as messaging, service management (e.g. discovery, hosting), and task scheduling to locally executing agents. Further information on the JADE-LEAP platform can be found at [1].

This application uses the Foundation profile version of the JADE-LEAP platform, which comprises of 693 classes (includes inner classes), while our application (the part that runs on the device) consists of 183 classes (includes inner classes). In the following sections we evaluate the performance improvements gained as a result of applying the presented optimisation techniques.

5.1 Application and VM Deployment

We applied the deployment level one and two optimisation techniques to optimise our application to the intended target environment. When applying the deployment level two technique, we applied most of the code reduction/optimisation techniques to 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 two common code reduction techniques: 1) Compression, and 2) Omitting debug information. We only removed unused classes, methods and fields from just the application classes. For both files 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.

Table 4 provides details as to the storage requirements for our application, which also includes both the JADE-LEAP platform and Foundation classes. The first row shows the storage size for each file when deployed in JXE format without applying any code reduction/optimisation, whereas row two does apply these techniques. Finally row three shows the storage size for each file when deployed in JAR format, which have applied two common code reduction techniques: 1)

Compression, and 2) Omitting debug information. We also removed unused classes, methods, and fields from the application classes.

	Application	**JADE-LEAP**	**Foundation**
JXE	360 kb	970 kb	1149 kb
JXE (optimized)	353 kb	954 kb	1149 kb
JAR	284 kb	756 kb	900 kb

Table 4 Storage requirements for our application.

5.2 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 two versions of our application. One version contains all the applied optimisation techniques, and the second version contains its classes (JADE-LEAP platform, and application) in JAR format, and use a generic VM that is bundled with the Foundation profile (classes in JAR format). The results from this test are shown in Table 5.

To reduce the affects of network fluctuations for each result we eliminated the highest two values, and the lowest two values. Through applying the deployment level one, two and VM level optimisation technique we managed to reduce the application start-up time by approximately 9.6%, equivalent to 1.9 seconds.

Type	**VM Start-up**	**Platform Start-up**	**Application Start-up**
JAR	2234 ms	13904 ms	3677 ms
JXE	1953 ms	12261 ms	3684 ms

Table 5 Results for improving the start-up time of our application.

5.3 Application Start-up Reduction (1) and Usability Improvement

As shown in table 5 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 routinue. Within the application launch routinue, 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.

6 Conclusion

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

Future work we plan to undertake 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] Berger, M., Rusitschka, S., Schlichte, M., Toropov, D., & Watzke, M., Porting Agents to Small Mobile Devices - The Development of the Lightweight Extensible Agent Platform. *EXP in search of innovation special issue on JADE*, Vol. 3, No. 3, 2003, pp. 32-41.

[2] Kok, M. Developing a DB2 Everyplace Java Application using WebSphere Studio Device Developer. (2002). Retrieved January 7, 2004, from http://www-106.ibm.com/developerworks/websphere/registered/tutorials/0212_kheng/kheng.html.

[3] Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H. & Nakatani, T., A study of devirtualization techniques for a Java Just-In-Time compiler, *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2000, pp. 294-310.

[4] JAVA 2 PLATFORM, MICRO EDITION FEQUENTLY ASKED QUESTIONS, (n.d). Retrieved January 7, 2004, from http://java.sun.com/j2me/reference/faqs/index.html.

[5] JSR 46 J2ME Foundation Profile, (n.d). Retrieved January 7, 2004, from http://jcp.org/en/jsr/detail?id=46.

[6] JSR 118 Mobile Information Device Profile 2.0, (n.d). Retrieved January 7, 2004, from http://jcp.org/en/jsr/detail?id=118.

[7] Just-In-Time Compilers. (n.d). Retrieved January 7, 2004, from http://www.bytecodes.com/techJITC.html.

[8] Laukkanen, M. (n.d). Java on Handheld Devices - Comparing J2me Cdc to Java 1.1 And Java 2. Retrieved January 7, 2004, from http://citeseer.nj.nec.com/473890.html.

[9] Lee, H., Mihailescu, P., & Shepherdson, J., A Multi-Agent System to Support Team-Based Job Management in a Telecommunications Service Environment, *EXP in search of innovation special issue on JADE*, Vol. 3, No. 3, 2003, pp. 96-105.

[10] Muller, G., Moura, B., Bellard, F., & Consel, C., Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, 1997.

[11] Shirazi, J. (2003). Java Performance Tuning (2nd Edition).

[12] Tyma, P. Tuning Java Performance. (n.d). Retrieved January, 7, 2004, from http://www.ddj.com/documents/s=962/ddj9604e/ .

[13] WebSphere Studio Device Developer WebSphere software, (n.d). Retrieved January 7, 2004, from http://www-306.ibm.com/software/wireless/wsdd/.

[14] Wilson, S., and Kesselman, J. (2000). Java Platform Performance: Strategies and Tactics.