Slim Execution on Distributed Mobile Environment

Heungsoon Rim, Youil Kim, Heeseung Jo, Seonggeon Kim, Hwansoo Han, and Jin-Soo Kim

CS/TR-2005-237
May 17, 2005

KAIST
Department of Computer Science

This work was supported by SK Telecom through Mobile Media Research Center (MMRC) at KAIST.
ABSTRACT: In this paper, we present slim execution for distributed mobile environment. To relieve resource constrained mobile devices of CPU and power constraints, resource demanding applications are executed on nearby powerful servers and only interactive parts of applications are executed on mobile devices. Using nearby servers as backing storage, we can overcome memory constraint as well.

The DiET, our bytecode transforming tool, automatically transforms a Java application into a distributed mobile code that is appropriate for target mobile devices. When applications include intensive computation, the DiET separates computation intensive parts of code, which are executed on servers. Mobile devices download only the rest of the generated code from servers and execute it with limited resources. Through our early implementation and experiments, we find our goal is promising.

Keywords: Code partitioning, computation offloading, distributed mobile environment, resource constrained device

1. INTRODUCTION

In recent years, mobile devices play important roles in our life. Cellular phones and personal digital assistants (PDAs) are representative examples of such devices. Even though recent mobile devices are remarkably more advanced than early versions, users and developers feel much more enhancements are needed mainly due to the resource limitations in them.

In this paper, we assume an environment where many powerful networked servers are deployed and users walk around with resource constrained mobile devices. Wireless networks connect mobile devices to nearby servers and provide accesses to the Internet. By executing some of expensive jobs on nearby local servers, we can relieve mobile devices of resource limitations. Many of mobile devices already support wireless communication using IrDA, Bluetooth, and Wireless LAN. We believe that such mobile devices will become popular in the near future.

Assuming such environment, we propose slim execution, a novel method to enhance resource constrained mobile devices:

- Computation intensive pieces of application code are executed on nearby servers, so we can relieve mobile devices of CPU or power limitations.
- Memory intensive pieces of code are executed on nearby servers, so we can overcome memory limitation of mobile devices.
- When the main memories of mobile devices are limited, we utilize nearby servers as secondary storage.
- An application is automatically transformed into the distributed code for slim execution before it is transmitted.

In the literature, the term offloading is often used to represent the techniques that are analogous to our approach. Slim execution differs from the approaches based on offloading in that a mobile device may not have the entire code. Therefore the transformed code can be smaller than the original code.
Our final goal is developing a system which automatically analyzes a mobile application and transforms it into the distributed code for the *slim execution*. Our final goal targets no intervention from application developers. The main domain of our work is applications written in Java. Java becomes increasingly popular in the development of mobile applications due to its *Write-Once-Run-Anywhere* characteristic. Java programs are typically compiled into the intermediate form called Java bytecode, which can run on any platform equipped with the Java virtual machine. Since Java bytecode retains most of high level structures of source programs, we can effectively analyze and transform Java class files.

In this paper, we present the design and implementation of Distributed Execution Transformer (DiET), which is our prototype tool to analyze and transform Java applications for *slim execution*. DiET is distinguished from existing works in that it does not require any special virtual machine.

The remainder of this paper is organized as follows: In Section 2, we discuss the design of our system. In Section 3, we describe some of implementation details. In Section 4, we present our experimental results. We summarize related work in Section 5, and conclude in Section 6.

### 2. SYSTEM OVERVIEW

To deal with the problem due to the limited computing capabilities of mobile devices, we propose DiET, a code transformation system for *slim execution*. In this section, we present the main idea of our proposed system. First, we address how mobile devices work in our slim execution environment. Second, we describe the basic operation and the primary design policy of DiET.

#### 2.1 Slim Execution Environment

Figure 2.1 is depicting slim execution environment. Standalone applications are deployed to the Internet as Java bytecode. If a user wants to execute one of the applications, the mobile device downloads the application from Internet first. In the proposed environment, however, the mobile device does not download the application from the Internet directly, but requests it to a nearby local server which is wired to the Internet. Then the local server downloads the requested application from the Internet, and transforms it to the distributed application suitable for the slim execution. When the transformation is completed, the local server sends the part of the distributed application for the mobile device. Finally, the application cooperatively runs on the mobile device and the local server. The application dynamically exploits computing resources of the local server such as memory, CPU, and power. Even on a mobile device with small amount of memory and low-end CPU, the transformed application can be executed more efficiently.

![Figure 2.1: Slim Execution Environment](image)

There can be some optimizations on the process;

- If the mobile device has an application, the local server can retrieve the application from the mobile device instead of Internet.
- Once the local server has transformed an application, it can keep the transformed application in its cache, and avoid redundant transformation overhead later.

This paper will not delve into various optimization issues in our system. Rather than that, this paper focuses on code transformation issues. Other than optimization issues, the following assumptions are made not to digress from our main topic:

- The connection between the mobile device and the local server is relatively reliable.
- The mobile device’s authority to download the application can be delegated to the local server.
- The local server has enough resources to serve all the requests of the mobile devices.
• The mobile device can discover the presence of the local server.

2.2 Distributed Execution Transformer (DiET)
DiET transforms a Java bytecode application into slim bytecode which is better fitted for mobile devices.

2.2.1 Basic Operation
The basic operation of DiET is given in Figure 2.2. DiET reads Java bytecode as an input and performs appropriate code partition on an original Java bytecode with the direction from users. It then generates the bytecode for a mobile device. On the server, the original bytecode is stored with a few modifications.

At current stage of DiET implementation, we use user description to decide which methods will be executed on servers. Final goal, however, we plan to automate the code partition process based on dynamically collected profile.

2.2.2 Design Policy
As a primary design policy, we consider transparency in deployment and development.

Transparent Deployment: We do not want to modify any parts of Java Virtual Machine (JVM). If we modify JVM in order to implement our idea, users have to install modified JVMs instead of standard JVMs to execute the applications generated by our system. Since our modified Java bytecode is still runnable on any standard JVMs, all the DiET’ized codes can run even on heterogeneous machines only if they are equipped with standard JVMs.

Transparent Development: We design our system not to alter application development process. Programmers just develop applications without concerning themselves with available system resources and network communications. The developed applications are automatically transformed into a mobile application cooperating with servers by DiET.

3. IMPLEMENTATION
The granularity of resource-intensive jobs is determined by the unit of method. If our system decides that a certain method is a resource-intensive job, then DiET generates proxy methods for original methods. These proxy methods are delegated to request servers to execute original methods.

Refer to the example code in Figure 3.1. Note that DiET actually works on Java bytecode but the example is written in Java source code for better understanding. From the user description, DiET decides the method sum() of the class A is a CPU-intensive job. It then makes a new class AProxy that extends class A in order to generate the proxy method. The AProxy class contains the proxy method sum(), which requests servers the invocation of the original method, sum() of class A.

```java
class A { // implements Serializable
    public int x;
    public int sum(int x, int y) {
        return x + y;
    }
}
class AProxy extends A implements Serializable {
    public int sum(int x, int y) {
        send (class_name, method_name, method_signature, this_object, arguments);
        read (exception_flag, return_value, this_object, arguments_to_be_updated);
        restore (this_object);
        return return_value;
    }
}
```

Figure 3.1: Example of Proxy Extension
DiET also automatically changes class $A$ to implement the `Serializable` interface class. When a method is remotely executed, the object itself referenced by `this` is required to be sent to the server side. Making serializable objects reduces the programming efforts to pass back and forth the objects between mobile devices and local servers.

In addition to generating proxy methods for remote execution, DiET needs to modify original bytecode to use the generated proxy methods instead of original ones. Since DiET creates subclasses for remote execution, it searches through bytecode instructions that create objects of interest types and replaces their types with the proxy classes (subclasses).

As shown in the example of Figure 3.2, the `test()` method of the class $B$ uses the proxy method `sum()` of the class $A$. In this case, DiET finds the instructions that are related to the creation of objects type `class A` and it modifies the instruction to use the `class AProxy`.

By using proxy classes, the methods that need to be executed remotely are automatically overridden by the virtual methods of proxy classes. In Figure 3.2, we just find a new bytecode instruction in the method `test()` and modify the class type $A$ into the class type `AProxy`. After this modification, the instructions that are related to the object $a2$ of `class A` type such as $a2.x = 1$ and $a2.sum(a2.x, 2)$ do not need additional modifications. Therefore, we can save the effort to modify many Java bytecode instructions. The modified Java code is shown in Figure 3.3.

The server is designed as a general server model for multi-clients. During the execution of Java applications, Java class files can be dynamically loaded with the application code. Inspired by this technique, we implemented a general server, which can provide the service for multi-clients by loading class files on-demand. This generic server code is presented in Figure 3.4.

4. EXPERIMENTAL RESULTS

4.1 Evaluation Environment

We perform our evaluation on client-server environment. From the original bytecode, we extract the client bytecode. Then we run the client bytecode on a PDA and the server bytecode on a Linux server.

Our client device is a HP iPAQ Pocket PC h5500. It has the Intel PXA255 400MHz CPU with 128 MByte RAM. It uses the Microsoft Pocket PC 2003 for OS and Jeode Virtual Machine [7] for Java. The server machine is a 1.8GHz Pentium 4 PC with 512 MByte RAM. It is operating with the RedHat Fedora core 3 and SUN Java2 v1.4.2. The embedded wireless LAN (802.11b) of the iPAQ connects to the Linux PC server.
In order to transform, modify, and insert Java bytecode instructions, DiET uses ByteCode Engineering Library (BCEL) [8].

4.2 Performance

We use the SciMark[9] for benchmark applications. Since the SciMark is designed to evaluate the performance of Java Virtual Machine, it performs multiple runs for five floating-point kernels. In order to measure execution time including the network transmission overhead, we add timing routines before and after the invocations of main kernels. Figure 4.1 shows the execution times of five kernels of SciMark. The left bar on each kernel shows the execution time when we run on PDA alone and the right bar on each kernel shows the execution time when we run on PDA-Server environment. We could improve the overall performance by 10%. FFT, SOR, and MC show performance improvement, while SM and LU degrades their performance on PDA-Server environment mainly due to the overhead of network transmission. However, we believe the performance benefit is scalable as we use much faster PC servers.

4.3 Code Size

As explained before, DiET modifies the original bytecode and generates the client parts of applications. Though some extra bytecode was added, client device does not need some part of original bytecode. For example, the codes that run on servers are not needed on client side.

Table 4.1 shows the analysis of SciMark code size. We can see that the size of the client bytecode was decreases by 15%, compared to that size of the original bytecode. Though the amount of reduction depends on the configuration and application structure, the size of client bytecode will not be larger than that of original bytecode.

Even if the size of server bytecode increases, it is tolerable since servers generally have much larger capacity than client devices. Besides, there are two common library classes for all client applications: the socket library class and the returned object update class. We excluded the size of common library classes, since they are regarded as system classes shared among all applications. The total size of two libraries is 3,422 bytes.

4.4 Network Traffic

The network traffic which is induced by dividing an application into the client part and server part is the main overhead of our scheme. We used the ntop[10] for the network traffic evaluation.
Table 4.2 shows the size of bytes for communication between client device and server machine. The amount of the traffic is depends on the size of the called object, the called method parameters, and the return value. Even though the traffic is an additional overhead, it is reasonable because it can be minimized by reducing above parameters and it is expected that the wireless network capacity will greatly increase in the near future.

5. RELATED WORK

Computation offloading [3, 6] is the strategy to overcome CPU limitation of mobile devices by executing computation intensive jobs on nearby powerful servers. Energy offloading [2, 4, 5, 6] describes similar strategies to computation offloading, but the goal is biased towards reduce energy consumption. Memory offloading [1] is to overcome memory limitation of mobile devices by using nearby servers as secondary storage during execution.

6. CONCLUSION

In this paper, we propose slim execution, a novel approach to relieve mobile devices of resource constraints by using wireless LAN connection to powerful servers. Slim execution is efficient in the sizes of applications and even in the execution times for some applications. Slim execution is transparent as well; it requires no special modification to standard virtual machines.

We described the design and implementation of the DiET, our prototype tool for automatic code transformation. Experimental results with SciMark show that the performance improves by 10% and the code size reduces by 15% on average.

REFERENCES