Distributed Execution for Resource-Constrained Mobile Consumer Devices

Seonggun Kim, Heungsoon Rim, and Hwansoo Han

Abstract — Mobile consumer devices take increasingly important roles, more closely and personally interacting with users. As users get used to mobile devices, they often want the same level of computing experience as they can have from desktop PCs, but still in small and light form factors. Considering current technology, we find the limitations of the processor and the memory are still too big in current mobile devices to satisfy demanding mobile users. To alleviate resource limitations, many researchers explored techniques to share the resources of powerful surrogate servers nearby. In that line of research, we propose slim execution for an effective mobile computing paradigm. To experimentally verify our execution model, we develop a code transforming tool, Distributed Execution Transformer (DiET). The DiET takes original Java bytecode and replaces the bodies of heavy methods with remote procedure calls to surrogate servers. Since the modified bytecode is still a legal Java bytecode, mobile devices can download and run the modified bytecode on standard JVMs, cooperating with surrogate servers. Our experiments with the SciMark 2.0 show our distributed execution scheme reduces the execution time by up to 71%.1

Index Terms — Distributed computing, method offloading, field access analysis, resource-constrained computing.

I. INTRODUCTION

Many people manage their daily life with the assistance of many kinds of computing devices. Whether they are supercomputers, desktop PCs, MIDs, PDAs, or mobile phones, people use those computing devices to access available information in a fast and easy way. Every morning people listen to the weather forecast computed by powerful climate simulations on supercomputers. People use desktop PCs in their offices to increase the productivity of their business and also use them at home for entertaining purposes. MIDs, PDAs, and mobile phones are recently added to the list of daily used computing devices. The most prominent difference of MIDs, PDAs, and mobile phones is that they are mobile, which means they are designed in small and light form factors and battery-powered. Current technology, even though we can manufacture very powerful desktop PCs, fails to provide as powerful mobile devices as desktop PCs. Mobile users, who are already exposed to versatile and graphic-rich applications of their desktop PCs, often find applications on mobile devices are very primitive for their tastes.

As the prices of desktop PCs become cheaper, many public places such as coffee shops, airports, and hotels provide PCs for the public. In the near future, those PCs deployed in public space will be a great leverage to the future of mobile computing. In this article, we propose slim execution as one way to overcome resource limitations in mobile devices by taking advantage of those public PCs. Our principle idea in slim execution is to execute the computationally heavy parts of applications on nearby powerful PCs and only the rest parts on mobile devices. Existing works [1]-[6] have similar goals to ours. By partitioning applications at class level, they can distribute heavy computations on servers, but they manage them at the class granularity. Those researches focused more on distributing objects and their class methods altogether. In contrast, we focus on offloading heavy individual methods. We explicitly select only the heavy methods to offload instead of the whole classes and all of their methods. We then replace the bodies of selected methods with function calls to request remote execution on nearby PCs.

Computation offloading at procedure level has been also studied [7], [8]. They provide a cost model to reduce power consumption [7] and a parametric analysis to make an offloading decision with the runtime parameters [8]. Our approach is similar in that we also find the methods to offload, but the focus of our work is to propose an overall execution framework where computation partitions are decided using a cost model (currently, our cost model is based on timing profile and pointer analysis) and programs are transformed into distributed programs according to the partitioning decision made by the cost model. Another issue to discuss is whether we need custom JVMs to deploy our goal. Some works in [1], [4] require special JVMs to handle the field accesses of remote objects. Such JVMs internally check if objects are remotely located and provide field accesses through communicating with remote JVMs. On the other hand, other works in [2], [3], [5], [6] modify the original bytecode to replace the field accesses with the method invocations from the proxy objects. The field access methods in the proxy objects then communicate with the remote servers to acquire the values of the requested fields. By using the proxy objects, they are able to run on any JVMs without modifications, but they inevitably increase the size of the bytecode to include additional classes for the proxy objects. One of the critical design issues in our approach is to build a transparent execution environment by using unmodified JVMs.

1 This work was supported by grant No. R01-2006-11196-0 from the Basic Research Program of the Korea Science & Engineering Foundation (KOSEF).
S. Kim is with the Department of Computer Science at KAIST, Korea.
H. Rim was with the Department of Computer Science at KAIST, Korea.
He is currently with Samsung Electronics, Suwon, Korea.
H. Han (corresponding author) is with the Department of Computer Engineering at Sungkyunkwan University, Korea (e-mail: hhan@skku.edu).

Contributed Paper
Manuscript received April 9, 2009
Distributing objects may force us to modify JVMs or use inefficient proxy method calls. Instead, we decide to send the copies of the necessary objects required by remote methods and receive back any modified objects after the method execution. To minimize the network traffic, we use a static analysis to decide which fields of which objects are necessary to send back and forth. Under this scheme, objects can also be created locally on remote servers if their lifetimes are within the scopes of the remote methods. Otherwise, those objects are sent back to clients and newly created on clients.

To verify our idea, we developed the Distributed Execution Transformer (DiET), which modifies Java bytecode to offload selected heavy methods. Since Java becomes increasingly popular among application developers, we believe Java applications are good candidates for the purpose of our study. In addition, we can easily analyze and transform Java class files, since Java bytecode retains most of the high level structures of source programs. Since the DiET transforms the original bytecode into a distributed one, application developers are free from the details of offloading. Moreover, the transformed applications are also legal Java bytecode. Thus, they can run on normal, unmodified Java virtual machines (JVMs), which save extra efforts for users to install custom JVMs on their devices.

The remainder of this article is organized as follows. We begin by introducing the slim execution and the DiET. Then, we discuss the field analysis to minimize the amount of the data to exchange between user devices and servers. We next describe the details of our transparent method offloading. We finally present our experimental results and conclude this article.

II. DIET FOR TRANSPARENT SLIM EXECUTION

The computing environment we expect to have in the near future consists of resource-constrained devices carried by mobile users and many powerful servers connected to the Internet. User devices are connected via the wireless network to have an access to the Internet. Many of recent mobile devices already support wireless communication through Bluetooth and Wireless LAN. Many new technologies are developing for a better quality of service in the wireless communication. The DiET is our tool to transform the bytecode for such computing environment. The output of our transforming tool is a distributed bytecode that can cooperatively run on user devices and surrogate servers. In this section, we address what the slim execution is and how mobile devices work in the slim execution environment. We then describe the detailed organization of the DiET.

A. Slim Execution

Fig. 1 depicts the slim execution approach we adopted from our early works [6], [9]. The primary goal of the slim execution is to extend the computing power of the mobile devices by exploiting the computing powers and memories of more resourceful servers. This approach presumes the existence of surrogate servers, which are able to communicate with mobile devices via wireless networks. Surrogate servers not only provide their computing resources to mobile devices by executing the computationally heavy parts of the applications but also perform the bytecode transformation as a part of the DiET framework.

The slim execution is initiated by users. Users look up the list of the applications they can execute. We assume this list is made available by the service providers. In the meantime, mobile devices silently look for surrogate servers and register the capabilities of the servers. All the necessary procedures to discover surrogate servers are performed in the background while users stay in the WiFi hotspot. Using a similar discovering scheme presented in [10], [24], we can discover a proper surrogate servers and find their capabilities. Once a user decides to launch a certain application, the mobile device sends a request to an appropriate surrogate server to cooperatively execute the selected application. If the surrogate server has enough capability to handle the requested application, the mobile device downloads only the client part of the bytecode, which we call the slim bytecode, where computationally heavy methods are replaced with remote method invocations. The surrogate server stores the server bytecode in their file system and launches a dispatcher process that listens to the request from the client side. During the execution of the slim bytecode on the client side, remote method invocations are requested to invoke heavy methods on the surrogate server. The dispatcher takes the requests and executes the corresponding original methods by loading class files from the server bytecode in its file system. Since the transformation of bytecode may require extra time, service providers prepare the transformed bytecode before hand and the surrogate servers spontaneously download transformed bytecode for popular applications before users actually request them. When the actual user request is received, the surrogate server can respond with the already downloaded bytecode. Depending on the load of the surrogate server and the bandwidth of wireless network, we may fall back to the original bytecode. Thus, surrogate servers also download the original bytecode for such occasions.

B. Distributed Execution Transformer (DiET)

Our bytecode transforming framework, the DiET, is designed with the following design principles. It helps application developers as well as application service providers
by providing a transparent way of program development and deployment.

- **Transparent Deployment**: Our modified bytecode runs on standard JVMs without any modifications. If we modify JVMs to implement our idea, users need to install customized JVMs before they can execute the slim bytecode generated by our transformer. Since our modified Java bytecode is still a legal bytecode, all the resulting bytecode can run on any standard JVM.

- **Transparent Development**: Our transformer can change existing bytecode to run in a distributed way. Program developers need not to alter the development process of mobile applications. Programmers develop monolithic applications which run on a single machine. All transformations needed for partition and communication can be done with our bytecode transformer.

- **Beneficial Offloading**: Ultimately, we want to apply our transformation to a broad range of applications. By offloading applications at method-level granularity, we can provide enough flexibility to find many beneficial cases.

![Fig. 2. Overview of DiET](image)

The DiET consists of three components as shown in Fig. 2: field access analyzer, environment monitor, and bytecode transformer. The field analyzer collects the information on which fields are actually read and/or written inside each method. Using the analysis result, we can generate the code that sends only the needed data and receives only the modified data. By reducing the amount of the data transfer, we can minimize the cost of the data communication between mobile devices and surrogate servers. During the field access analysis, we also estimate the amount of data to transfer.

Program developers in the DiET framework inspect the time spent in each method. Depending on the target environments, profiling runs are repeated on multiple architectures. We currently adopt the Eclipse Profiler (also called as Eclipse Colorer) [11] in our framework. Combined with the estimated amount of data transfer, the timing information can provide a good guideline to service providers who are willing to adopt the slim execution. Partitioning directions can be manually provided, but these should be based on the profiling information, the field access analysis and the monitored information of computing environment. All the components of the DiET except the environment monitor reside on the service provider side, while the environment monitor performs monitoring on the surrogate server. The monitor gathers information on the available bandwidth of the wireless network and the capacity of the surrogate server. Based on gathered information, it decides whether it deploys the slim execution for the requested application.

The bytecode transformer reads the list of the methods to offload that is provided from the program developers or from the guideline automatically determined based on the profiling result and the field analysis. The DiET is responsible to generate two pieces of bytecode for distributed execution: the slim bytecode and the server bytecode. Application providers are the most eligible party to produce the partitioned bytecode to support the slim execution. Before the actual transformation, we need to preprocess the original bytecode to make necessary classes serializable. Since we use the serialization APIs for the communication between user devices and servers, these modifications in the preprocessor are necessary. In the actual transformation, the slim bytecode is generated by replacing the bodies of the methods to offload with the remote method invocations and by adding the required data transfers before and after the remote method invocations with the serialization APIs. These server bytecode, which are actually the output of the preprocessing, will be downloaded in to the surrogate server. Meanwhile, the slim bytecode will be passed on to the mobile device and cooperatively executed with the server bytecode later.

### III. Field Access Analysis

The object serialization [12] is a convenient mechanism that can change the internal representation of an object into a byte stream. By making a stream, we can easily transfer the object over the network and allow exchanging objects between mobile devices and surrogate servers. The original serialization APIs, however, are very slow for class objects. Moreover, if an object contains references to other objects, the Java serialization mechanism is recursively applied to send the referenced objects. If this object is not used within the offloaded method, this is a waste of the effort to serialize in terms of CPU cycles and network bandwidth. To avoid such wasted efforts, we develop the field analyzer that inspects which class fields are actually used in each method. By using the field access analysis, we can selectively exchange only necessary fields.

The field access analyzer is implemented with the Soot, a Java optimization framework [13]. The Soot is a useful tool to optimize and analyze bytecode. Since it provides various APIs related to optimizations and analysis, we can conveniently make use of those APIs to implement our analyzer. For example, we can use the result of points-to analysis [14] and call graphs from the Soot APIs such as `PointsToAnalysis`, `getCallGraph`, and so on. It also provides four intermediate
representations to analyze bytecode. The Jimple [15] among those intermediate representations is a typed 3-address code suitable for analysis and optimizations. If we directly analyze the stack-based bytecode, we would have to face with several difficulties due to the implicit usage of the operand stack [15]. Since the statements in the 3-address code explicitly refer to the operand variables, the Jimple intermediate representation is much easier for us to analyze the field accesses of Java bytecode.

A. Overview of Analysis Algorithm

We perform our field access analysis after the phase of whole Jimple transformation in the Soot framework. Our analysis consists of two steps. The first step collects the initial summary of field usages from the Jimple representation for each method. By visiting each statement of a method but skipping any method invocations, the analyzer initially records which object fields are read and/or written. Since objects can be pointed to by aliases, our analyzer must check all aliases to gather precise field access information. We rely on points-to analysis, which conservatively determines the set of the objects pointed by each reference variable [14].

The second step completes the field usage summary with the consideration of method invocations. The second step works on the call graphs provided by the Soot framework. By propagating the field usage summary upward along the call graph, the summary is accumulated from the callees to their callers. The second step is implemented with the work list algorithm. The work list algorithm processes one method by another, appending only the necessary methods to process again due to the changes from the successor methods in the call graph.

```
void f(obj1, nat2, obj3) {
    tmp = obj1;
    y = tmp.x + nat2;
    g(tmp, obj3);
}

void g(obj4, obj5) {
    obj4.x = 1;
    obj5.x = obj4.x;
}
```

B. Inter-procedural Field Access Analysis

Using the example shown in Fig. 3, we will explain more about the second step of the field access analysis. Visiting all the methods of the input bytecode, our analyzer investigates all the statements within each method and summarizes the result of the field accesses for each method. In the next step, our analyzer merges the method summaries from the descendant methods in the call graph. In the method \( f \), the field \( x \) of \( tmp \) and the variable \( \text{nat2} \) are read, and the field \( y \) of \( \text{this} \) object is written. In this case, the analyzer inspects the aliases of the base objects. In the example, the alias of \( tmp \) is \( \text{obj1} \), the first argument of the method \( f \). Thus, we record the access information of \(<\text{arg1}, \text{x}, \text{read}>\) in the method summary. Similarly, \(<\text{arg2}, \text{whole}, \text{read}>\) and \(<\text{this}, \text{y}, \text{write}>\) are written to the method summary. The objects to analyze include static class objects as well as the objects whose references are passed by arguments including \( \text{this} \) object. We can apply the same analysis algorithm to the method \( g \). The resulting method summary is shown in Fig. 3.

Assuming that we offload the method \( g \), we only need to send the value of the field \( z \) of \( \text{obj5} \), instead of all the fields of \( \text{obj4} \) and \( \text{obj5} \). Moreover, we only receive the updated value of the field \( x \) of \( \text{obj4} \) on the return of the method \( g \). By sending only the necessary fields and receiving only the updated fields, we can reduce the amount of data to communicate. This optimization can reduce the communication delay and increase the performance as a result. Suppose that we offload the method \( f \), the situation is a little bit complicated. Since the method \( f \) invokes the method \( g \), the method \( g \) is executed on the same side as the method \( f \). That is the server side. In order to correctly execute the subsequent methods invoked from the method \( f \), all the data accessed by those subsequent methods, such as \( g \), need to be on the server side. Consequently, we need to know not only the fields accessed by the method \( f \) itself but also the fields accessed by the method \( g \) in this example. In general, the method summary needs to include all the method summaries of the descendant methods in the call graph. Fig. 4 shows the method summaries for the example code from Fig. 3. All the summaries of the descendant methods are accumulated after the second, inter-procedural step. When we accumulate the method summaries from the descendant methods, we consider the aliases and the relations between the actual parameters and the formal parameters. The first argument of the method \( g \) is actually \( \text{tmp} \) from the method \( f \) and \( \text{tmp} \) is the alias of the first argument of the method \( f \). Thus, we insert \(<\text{arg1}, \text{x}, \text{write}>\) to the method summary of \( f \). With a similar process, we insert \(<\text{arg3}, \text{z}, \text{read}>\) to the method summary of \( f \).

IV. TRANSPARENT METHOD OFFLOADING

The granularity of offloading is a key factor to transform monolithic applications into fast running distributed ones. Since not all class methods are often good candidates for method offloading, we have a supposition that partitioning at the class level may not be flexible enough to achieve
beneficial method offloading. Partitioning at the method level granularity enables the DiET to selectively transform the computationally heavy and/or memory intensive parts. Once certain methods are chosen as offloading candidates, the DiET replaces the bodies of those methods with the instructions to call SlimMethod, which is delegated to request a remote surrogate server to execute original methods on that server.

In this section, we describe how we implement our method offloading. We first describe the mechanism for the transparent method offloading. We address the overall process about how the DiET generates the slim bytecode and the server bytecode. We then present the details of the slim bytecode and the dispatcher in turn. Finally, we discuss issues and limitations in our implementation.

![Fig. 5. Transformation for Method Offloading](image)

```java
class A {
    class A implements java.io.Serializable {
        public double intensive (args) {
            /* heavy computation */
            DIETLib.SlimMethod(method_id, args);
        }
    }
}
```

**Fig. 6. Slim Method and Dispatcher**

### A. Method Offloading Mechanism

Fig. 5 shows how the DiET transforms the candidate methods with an example. It first changes the class `A` to implement `java.io.Serializable` interface. By making objects serializable, we can provide a convenient way to exchange the objects between mobile devices and surrogate servers [12]. The body of original method (`intensive`) is replaced with a call to `SlimMethod`. This `SlimMethod` is our class library routine that takes the `method descriptor` and the arguments of the original method. The method descriptor is passed to select the method to offload. All the arguments of the original method are passed to the `SlimMethod`, but only the part of them are sent to the server and the part of the sent data are returned for the updates on the client side. As we described in previous section, the field access analysis decides which fields of which arguments are sent and returned. If the argument is a primitive type, i.e., non-object, it can be sent by using a corresponding serialization API. If the argument is an object type, the result of the field access analysis is used. We only send the fields that are accessed in the offloaded method. Individual fields are sent in a similar way to primitive type arguments using corresponding serialization APIs. As for the object type field, we may analyze further to identify which fields are used inside the object. This analysis, however, may take a long time to complete, since object fields can appear each level we analyze further down. Thus, we limit the depth of the field analysis up to one level. When the `SlimMethod` receives the updated results, it uses a similar approach. According to the field access analysis, it identifies the primitive type arguments and the one-level fields of the object type arguments to update.

After executing the original method on the server side, the server sends a boolean value to the `SlimMethod`. This value indicates if any exception has occurred during the execution on the server side. If the boolean value is true, the `SlimMethod` will throw the received exception. Otherwise, the `SlimMethod` receives the updated values. To cooperate with the surrogate server, the slim bytecode and the `SlimMethod` on the client side require a supporting class, called the DiET class. Using the methods in the DiET class, mobile devices communicate with servers to send requests for remote method invocations and receive the results. The DiET class also provides the APIs to look up the result of the field access analysis.

### B. Transformation to Slim Bytecode

Suppose the `intensive` method is listed as a computation and memory intensive method, the DiET transforms the body of the method into a library call to the `SlimMethod` that is customized to send and receive the required fields and the method descriptor to a surrogate server. The `SlimMethod` takes the `method_id` as the first argument which consists of the class name and the method signature to uniquely identify the method to offload. All the arguments of the original method are passed to the `SlimMethod`, but only the part of them are sent to the server and the part of the sent data are returned for the updates on the client side. As we described in previous section, the field access analysis decides which fields of which arguments are sent and returned. If the argument is a primitive type, i.e., non-object, it can be sent by using a corresponding serialization API. If the argument is an object type, the result of the field access analysis is used. We only send the fields that are accessed in the offloaded method. Individual fields are sent in a similar way to primitive type arguments using corresponding serialization APIs. As for the object type field, we may analyze further to identify which fields are used inside the object. This analysis, however, may take a long time to complete, since object fields can appear each level we analyze further down. Thus, we limit the depth of the field analysis up to one level. When the `SlimMethod` receives the updated results, it uses a similar approach. According to the field access analysis, it identifies the primitive type arguments and the one-level fields of the object type arguments to update.

Listening to the connection requests from multiple clients, a dispatcher process runs on the server. If a client connects to this dispatcher, the dispatcher process forks a child process and redirects the connection to the child process. Then the
child process runs as a designated worker process for the client. The worker process handles the method offloading requests from the corresponding client. The worker receives the method_id from the client. After uniquely identifying the original method code, the worker instantiates an object of the given class type and the necessary fields of the object are initialized with the field values sent from the client. This is the initialization work for this object. Similar initialization works are performed for the other arguments. After the initialization, the method of the original bytecode is invoked. Exceptions are properly handled to transfer the exceptions to the client of the remote method invocations. After the execution of the original method, the values of updated fields are identified through the field access analysis again. Only the values of updated fields are sent back to the client along with the return value. All of these works are done with the help of the Java reflection APIs.

V. ISSUES AND LIMITATIONS

A. Object Consistency

When we apply method offloading, we make duplicate copies of objects on the server side. Some of those copies are updated and returned back to the client side for restoring updated values. Data communications and value updates are extra operations in order to preserve the consistency of objects. Suppose a field of an argument (arg.field) is a reference type of an object and that field is analyzed to be updated on the server side, the client will pass the object and the server will update the fields of the object. The updated object will be passed back to the client and copied back to the original object. We maintain all the objects on the client side where the actual owners of applications reside. The objects on the server side are the temporary copies only during the computations of offloaded functions.

When the client side receives the updated object, we may update the reference field (arg.field) to point to the new copy instead of copying the whole contents of the object. This is only possible if there is no other reference than arg.field. If other variables (alias) have references to the original object, we need to update all those references. Point-to analysis usually tell you which objects may be referenced by a variable but does not tell which variables are exactly pointing to a particular object at a certain program point. Thus, copying the whole object is inevitable without a context-sensitive precise alias analysis, which is hardly feasible.

Moreover, there may be a problem with the referential integrity in the current version of our system. With the standard Java serialization, referential integrity is only preserved within the transmission. If a reference relation is changed in the server side, the client cannot know the change. To be conservative, we exclude those methods from the offloading candidates, if an alias relation among the arguments (and their fields) occurs. As computation-heavy methods usually use primitive arrays or simple composition of those, our limitation in referential integrity is not prohibitively strict.

B. Arrays and Collections

Similar to the case in object consistency, restoring the elements of arrays and collections is also necessary. Even if only one element is actually updated, we copy all the elements. Since our field access analysis does not distinguish the elements of arrays, we do not know which elements are updated. We may increase the depth of analysis, but still we limit our analysis to a fixed depth in order to make our analysis complete within a reasonable amount of time. Suppose we need to send and receive large arrays and collections, we may need to selectively increase the depth of analysis instead of suffering from communication overhead.

C. Reducing Communication Overhead

Frequent communications of small amounts of data incur too much overhead. If we apply method offloading inside the loop, we may need to invoke a remote method every iteration. Since all remote method invocations involve quite a large overhead, we can transform the source code to aggregate communication. The argument used in remote procedure call is expanded to a vector (an array) and computation results are also accumulated to the vector. The remote method invocation, which originally exists inside the loop, is hoisted outside the loop. This optimization can reduce the overhead of the remote method invocation. This technique is extensively used in the distributed message passing systems [17], [18].

If a method selected to offload is located within a loop, the DiET first tries to offload the outer method that contains the loop. It may not be possible, if the outer method has any anchor method (e.g., native methods, I/O operations, etc.). In that case, the transformer checks whether hoisting the inner method outside the loop may violate any dependency. If the dependency check proves the optimization is safe, the transformer applies the communication aggregation. We tested an interactive graphical puzzle which rotates 3D objects on mouse drag of users. To achieve smooth rotating objects, many coordinate calculations are performed during a 90° rotation. Those computations are typically heavy floating-point operations, which are very slow on mobile devices. By offloading the method for coordinate calculation, we could complete the calculation more than 1,000 times faster on a server, but the communication overhead due to many remote invocations cancels out all the benefits of the fast computation. In such cases, aggregating communication is critical to achieve beneficial offloading.

D. Restrictions in Methods to Offload

If a method calls native functions on the client side, that method cannot be offloaded to the server. Since the server does not have equivalent functions, any method which uses native functions of the mobile devices should stay at the client side. User interface functions also have similar characteristics. To interact with users, such methods should be executed on the client side. Using a similar approach to J-Orchestra [3], we can determine the methods to stay at the client side based on the call graph analysis. In our implementation, we do not offload any
method that resides in the Java classpath. In addition, users can provide the exact list of native functions and user interactive functions to prevent the methods from offloading.

Another restriction is applied to synchronized methods. If a method itself is synchronized or any portion of code within this method including subsequent methods invoked by this method is synchronized, the whole method invocation to server should be synchronized with appropriate objects. This may widen the synchronized range too much, if synchronization is originally needed in a small fraction of the offloaded method execution.

### VI. EXPERIMENTAL RESULTS

We evaluate our method offloading scheme by measuring execution times and code sizes. We performed our experimental evaluation with the SciMark 2.0 [19]. We use an HP iPAQ hx4700 on the client side and a desktop PC on the server side. The iPAQ has an Intel PXA270 running at 624MHz with 64MB RAM. The desktop PC runs Microsoft Windows XP on a 3.0GHz Intel Pentium4 and 1GB RAM. The Java virtual machine used in this experiment is the Jeode Java VM for iPAQ and Java HotSpot VM v1.4.2-08 for the Windows XP on a 3.0GHz Intel Pentium4 and 1GB RAM. The Java virtual machine used in this experiment is the Jeode Java VM for iPAQ and Java HotSpot VM v1.4.2-08 for the desktop PC. Most ARM processors on mobile devices currently perform floating-point operations as software emulated, since a coprocessor called VFP is required to ARM processors for hardware floating-point acceleration [20]. The client devices and the server PCs communicate through the Wireless LAN (IEEE 802.11b). To isolate the experimental environment, we use our own WiFi access point and admit only the mobile devices participating in our experiment.

The offload decision was made based on the information from profiling and monitoring. We select the methods to offload that have higher self seconds per call, the average time spent in the method itself per call. In our experiment, we select just one method per each computational kernel for the SciMark 2.0. The SciMark 2.0 is composed of the five computationally intensive kernels frequently found in scientific computing and multimedia processing. Complex applications usually need plenty of computation power, but the capabilities of mobile devices are often insufficient to satisfy the requirements. Since we try to expand the computing capability of mobile devices, executing computationally intensive applications is an important issue for the slim execution model. Since most of the ARM processors in mobile devices lack the hardware acceleration for floating-point instructions, an application that has many floating point computations would be a suitable benchmark for our experimental purpose. The SciMark 2.0 is one of such applications. By experimenting with the SciMark 2.0, we can estimate the benefit of our method offloading scheme on computationally intensive applications. TABLE I describes the characteristics of each kernel along with the sizes of input data.

#### A. Impact on Execution Times

We measured the execution times of original bytecode and slim bytecode for the five benchmark kernels. To further investigate the execution behavior of the slim bytecode, we separately measured the computation times on the surrogate server and also measured the data transmission times on the client PDA. Since we use kernels, we assume a situation where data are transmitted to the surrogate server for computations and transmitted back to the PDA to resume the rest of the application. The results are presented in TABLE II.

The column labeled as **Original** shows the execution time of the original bytecode on the PDA alone (Torg). The column labeled as **Slim** shows the execution time of the slim bytecode, which is the elapsed time when we cooperatively run the slim bytecode and the server bytecode. The numbers within a parenthesis show the breakdown of the slim execution time, which consists of the computation time of the server bytecode (Tcomp) and the data transmission time between the client PDA and the server PC (Ttrans). Tcomp includes the time spent on the process fork and the connection redirection. Among the five kernels, FFT, LU, and MC show the reductions of 16-71% in the execution times. Meanwhile, SOR and SM show the severe increases in the execution times. MC is an ideal case. The transmission time for MC is very small and the execution time on the server is improved by a large amount. The data transmission times are the dominant components for FFT and LU, but the computation times on the server are improved much enough to cancel out the transmission overhead. Meanwhile, SOR and SM run relatively fast on the PDA alone. Even though we can reduce the computation times, the data transmission times are too much overhead to overcome. In such cases, the slim execution has narrow chances to excel.

Since a large portion of the execution time is spent on the data transmission in the slim execution, the network throughput is a key factor for the performance of the slim execution. The following equation shows the condition where the slim execution excels the original execution.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Offloaded Method</th>
<th>Original Torg (sec)</th>
<th>Slim Tslim (Tcomp+Ttrans) (sec)</th>
<th>Reduced (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>transform</td>
<td>22,059</td>
<td>18,363 (984 + 17,617)</td>
<td>16%</td>
</tr>
<tr>
<td>LU</td>
<td>factor</td>
<td>34,518</td>
<td>11,237 (730+10,746)</td>
<td>67%</td>
</tr>
<tr>
<td>MC</td>
<td>integrate</td>
<td>1,004</td>
<td>282 (277 + 19)</td>
<td>71%</td>
</tr>
<tr>
<td>SOR</td>
<td>execute</td>
<td>444</td>
<td>4,382 (196 + 4,362)</td>
<td>-927%</td>
</tr>
<tr>
<td>SM</td>
<td>matmult</td>
<td>214</td>
<td>6,078 (209 + 6,080)</td>
<td>-2839%</td>
</tr>
</tbody>
</table>
the execution times of original bytecode ($T_{\text{org}}$) through the above equation. In this calculation, we measured the size of transmitted data at the application levels and the required network execution, which we can calculate using Equation (2). To estimate the required network throughput for the slim bytecode, we choose slim execution with method offloading. With profiling, we can roughly find the cases for beneficial offloading. The third and forth column in TABLE III compare the size of server bytecode along with the original bytecode. The combined size of all bytecode becomes larger than the original bytecode. That is, however, acceptable, as surrogate servers generally have much larger storage space than mobile devices.

To achieve beneficial offloading, the network throughput needs to be at least a certain level. This is a minimum network throughput required, which we can calculate using Equation (2). To estimate the required network throughput for the slim execution, we measured the size of transmitted data at application levels and calculated the required network throughput according to the above equation. In this calculation, the execution times of original bytecode ($T_{\text{org}}$) and server bytecode ($T_{\text{comp}}$) are provided by service providers who measured these times on a standard machine. Thus, these times need to be adjusted depending on the workload and the performance of mobile devices and surrogate servers. Thus, the times should be adjusted based on the current capability of the surrogate server from the environment monitor and the monitoring environment at run-time, we can decide whether offloading is beneficial for each application.

By profiling, we can roughly find the cases for beneficial offloading at the development stage of applications. By monitoring environment at run-time, we can decide whether we choose slim execution with method offloading. With matured quality of service (QoS) in wireless network, our slim execution can expect a certain level of throughput and find more stable cases for offloading.

### TABLE III

**REQUIRED NETWORK THROUGHPUTS**

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Time (msec)</th>
<th>Data to Transfer (byte)</th>
<th>Required Throughput*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>($T_{\text{org}} - T_{\text{comp}}$)</td>
<td>(no analysis)</td>
<td>(Kbps)</td>
</tr>
<tr>
<td>FFT</td>
<td>21,075</td>
<td>8,388,675</td>
<td>8,388,675</td>
</tr>
<tr>
<td>LU</td>
<td>33,788</td>
<td>8,408,221</td>
<td>6,311,069</td>
</tr>
<tr>
<td>MC</td>
<td>727</td>
<td>81</td>
<td>29</td>
</tr>
<tr>
<td>SOR</td>
<td>248</td>
<td>4,199,492</td>
<td>2,102,340</td>
</tr>
<tr>
<td>SM</td>
<td>5</td>
<td>6,291,456</td>
<td>3,205,242</td>
</tr>
</tbody>
</table>

* The realizable throughput was about 500 KBps during the experiment.

which yields

$$T_{\text{comp}} + T_{\text{trans}} < T_{\text{org}}$$

(1)

$$\frac{\text{Data Size}}{T_{\text{org}} - T_{\text{comp}}} < \text{Network Throughput}$$

(2)

where, $\text{Network Throughput} = \frac{\text{Data Size}}{T_{\text{trans}}}$

(3)

To achieve beneficial offloading, the network throughput needs to be at least a certain level. This is a minimum network throughput required, which we can calculate using Equation (2). To estimate the required network throughput for the slim execution, we measured the size of transmitted data at application levels and calculated the required network throughput according to the above equation. In this calculation, the execution times of original bytecode ($T_{\text{org}}$) and server bytecode ($T_{\text{comp}}$) are provided by service providers who measured these times on a standard machine. Thus, these times need to be adjusted depending on the workload and the performance of mobile devices and surrogate servers. Thus, the times should be adjusted based on the current capability of the surrogate server from the environment monitor and the types of mobile devices in the DiET framework.

The third and forth columns in TABLE III compare the amount of data to transmit. Using the field access analysis, we can reduce a large amount of data ranging from 25% to 65%. Only FFT does not decrease the size of data even after we apply our field access analysis. The last column in TABLE III shows the required throughput for each kernel. In our experimental setup, the realizable throughput at application level was about 500 KBps. While the required throughputs for FFT, LU, and MC are below the realizable throughput, those for SOR and SM are far above the realizable throughput. This is why the slim executions for SOR and SM poorly perform.

### VII. RELATED WORK

Method offloading can be implemented using Java RMI [21], which enables programmers to make distributed Java applications. In contrast to our DiET system, programmers need to explicitly design distributed applications. That means many of the monolithic Java applications need to be reprogrammed or abandoned in order to develop new distributed applications. Meanwhile, our DiET can transform Java bytecode, lessening the burdens of programmers.

Addistant [2] and J-Orchestra [3] can partition monolithic Java applications into distributed applications. The common goal of those systems is to provide transparent development of distributed applications. Those systems focus on the distribution of objects by transparently partitioning programs at Java class level. In contrast, our system is designed to partition at method level not class level. The goal of JavaParty [5] is similar to Addistant and J-Orchestra but it also produces distributed bytecode using programming language extensions. Java DSM [22] employs a software distributed-shared memory (DSM) technique to allow distributed execution of monolithic Java applications. It also handles the distribution transparently, but requires a special implementation of the JVM.

The Coign system [23] is designed for automatic partitioning of applications based on COM components. It constructs a graph model of the communication among components. The system then uses a min-cut algorithm to minimize the execution delay due to the overhead from network communication. Our DiET also uses a partitioning algorithm based on the profile of the computation time and the data transmission overhead and the monitored information on computing environment.

Previous researches on offloading among mobile clients and surrogate servers also find the performance improvements [1] [4]. Those systems are, however, implemented with custom JVMs to handle object distribution. These approaches may delay the deployment of offloading, since mobile devices need to be equipped with custom JVMs.
Service discovery is an interesting and important research challenge of distributed computing. S. Goyal and J. Carter [10] proposed a service discovery protocol using a central discovery server. In this protocol, clients can locate a surrogate server with the help of the central discovery server. The discovery server manages the registration list of surrogates and finds a suitable surrogate for the request from a client.

VIII. CONCLUSION

In this article, we propose the slim execution that adopts the transparent method offloading for resource-constrained mobile devices. Our approach relieves mobile devices of resource limitations. To experiment with our approach, we developed the Distributed Execution Transformer (DiET), a tool to transform the original bytecode for method offloading. By replacing the bodies of heavy methods with the custom remote method invocations, the DiET transparently transforms existing Java applications into distributed applications. Resource-constrained mobile devices download only the client portions of the bytecode and cooperatively run them with surrogate servers. Since we transform Java applications at bytecode level, we are able to run the transformed applications on standard JVMs. Experiments with the SciMark 2.0 show that we can reduce the execution times for three kernels of the SciMark 2.0. The reductions are up to 71%. As for the code size, we increased the size of the bytecode to download by 22% for the SciMark 2.0, but this increase is for the better performance. Through our study with the DiET, we find our transparent method offloading can relieve mobile devices of resource constraints and accelerate the computing performance.

REFERENCES


Seonggun Kim received the BS degree in electrical engineering from KAIST in 2004. He is currently a PhD student majoring in computer science at KAIST. His research interests are in the field of compiler techniques to improve the memory locality for a broad range of applications such as embedded software and scientific applications as well. Currently, he investigates automatic data management for explicitly managed memory hierarchies for manycore processors.

Heungsoon Rim received the BS in information and computer engineering from Ajou University in 2004 and the MS degree in computer science from KAIST in 2006. After graduation, he joined Samsung Electronics, Korea. His research interests include software optimization and application platforms for mobile devices. He explores and evaluates techniques from compilers and operating systems to adopt them for consumer mobile devices.

Hwansoo Han (M’03) received the BS and the MS degrees in computer engineering from Seoul National University, Korea in 1993 and 1995, and the PhD degree in computer science from the University of Maryland at College Park in 2001. He is currently an associate professor at Sungkyunkwan University. Previously, he served as an associate professor at KAIST and as a senior engineer at Intel. His research interests include compiler technology for high-performance computing, embedded computing, and secure computing.