ProxyVM *

A Network-based Compilation Scheme for Resource-Constrained Devices

Vasanth Venkatachalam, Lei Wang, Andreas Gal, Christian Probst, Michael Franz
Department of Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{vvenkata,leiw,gal,cprobst,franz}@uci.edu

ABSTRACT
Both interpreters and JIT compilers have inherent problems when used in resource-constrained mobile devices such as mobile phones and PDAs. We propose a network-based virtual machine infrastructure to offload the resource-intensive mobile code compilation and optimization to a powerful proxy server. This paper describes the motivation, architecture and implementation of the ProxyVM prototype system.

1. MOTIVATION
The computing power of mobile devices suffers from two major drawbacks compared to that of desktop machines – battery life and memory constraints. For example, while a typical handheld device has roughly 64 MB RAM and 16 MB ROM, the JDK 1.4 runtime library alone requires approximately 14 MB of storage space in compressed form. The full Java development kit amounts to well over 50 MB. Specialized virtual machine implementations for handheld devices exist, but due to resource limitations, they usually just support a limited subset of the whole Java platform. Of course, none of the above numbers include the extra memory that running applications need. Even simple AWT applets will consume additional 10-20 MB of RAM at runtime.

The goal of our ProxyVM architecture is to increase the usability of handheld devices through software techniques.

In traditional virtual machines, all files associated with the Java program execution (class files, java libraries, VM binaries) are required to reside on the target device. Most virtual machines execute bytecode using interpretation, just-in-time compilation (JIT), ahead-of-time compilation (AOT), or a combination of these approaches. Each approach has advantages and drawbacks with respect to mobile devices.

While interpreters have the advantage of a small memory footprint, code interpretation is too slow to be acceptable for many real world applications. JIT compilation breaks the performance barriers of interpretation by compiling bytecode to native code before execution. JIT compilation occurs while a program is already running (mixed-mode execution). The application pauses whenever it needs a method compiled and resumes execution once the native code is available. However, building an effective JIT compiler for a mobile device is a non-trivial task given the resource limitations of handheld devices. Traditional JIT compilers such as JikesVM [1] or the HotSpotVM [7] have a large memory footprint. Memory saving hybrid JIT compilers typically used in embedded systems [13] have a smaller footprint but generate less optimized code. Besides memory consumption the compilation time also impacts the overall execution performance, since CPU power is a limited resource on handheld devices as well.

AOT compilation involves compiling an entire program to native code and then executing the program. The application need not pause for compilation during execution. While this gives AOT an advantage over JIT, good JIT compilers can sometimes produce better code than AOT compilers based on additional profiling information obtained through runtime monitoring.

The ProxyVM architecture captures the advantages of all the traditional approaches while minimizing resource consumption on the handheld. Using our framework, handheld users can, for example, request Java applets from the internet as they normally would with a web browser on a desktop machine. However, a powerful server infrastructure (the proxy) intercepts Java class file requests, downloads the bytecode and compiles all necessary classes to native code.

Unlike interpretation and JIT compilation, our framework does not require any Java support in the form of runtime libraries or VM specific files on the handhelds themselves. By pushing all compilation work onto the more powerful server, our approach saves resources and enables optimizations that would be too costly in terms of memory consumption and CPU time to perform on handheld devices.

*Parts of this effort are sponsored by the National Science Foundation under ITR grant CCR-0205712 and by the Office of Naval Research under grant N00014-01-1-0854.

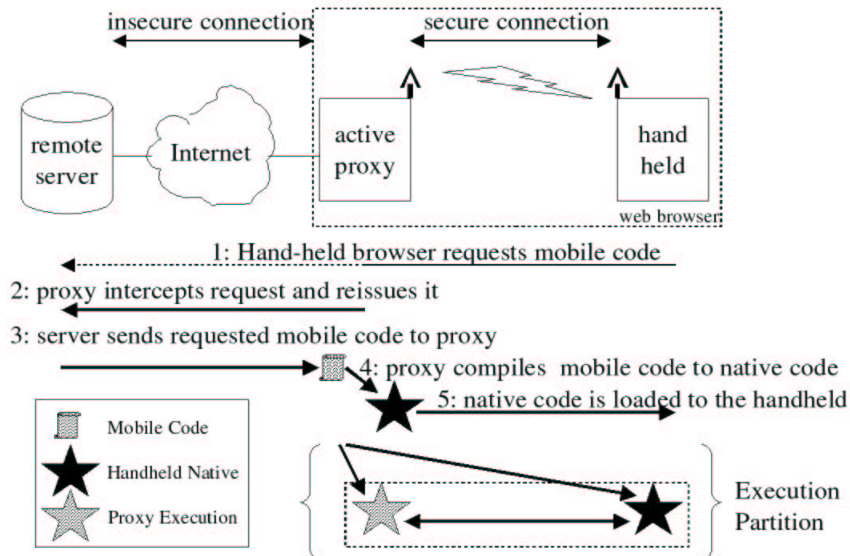

Figure 1: The ProxyVM Framework

The remainder of this paper is organized as follows. In Section 2 we present the ProxyVM architecture overview. Section 3 discusses the implementation issues of the prototype system. Section 4 discusses related work. Section 5 presents our conclusions, while Section 6 proposes possible future extensions.

2. INFRASTRUCTURE OVERVIEW

The ProxyVM framework enables resource-efficient mobile code execution on handheld devices. The framework consists of a proxy bytecode compiler on a server and a runtime system for the handheld device.

So far we have implemented a ProxyVM prototype system, using the Sharp Zaurus PDA as a target device. The Sharp Zaurus is a handheld device with 64 MB of RAM, 16 MB of ROM, and a 206 MHz StrongARM CPU.

Figure 1 shows the overview of the ProxyVM architecture. In the current scenario, the handheld device requests a Java program from the proxy (number 1 in Figure 1). The proxy loads (2,3), parses and compiles all necessary class files into native code (4) and sends the native code to the handheld (5). Our runtime system on the handheld maps the native code into memory and executes it. Thereafter, the program executes without using any Java support on the handheld. Beyond shifting compilation to the proxy, the execution of the mobile code might also be partitioned between the proxy and the handheld, in order to achieve less power consumption and faster execution, or even possibly to break the memory limits on the handheld.

The proxy mediates all communication between the internet and the mobile device. When the server receives a class file request, it sends it to the code generator that in turn notifies the class file loader. The loader searches available paths for the class. These paths include the local file system, JAR archives, and remote URLs.

Initially, the parser analyzes a class file and retrieves information about the logical structure of the class. The code generator then uses this information to create a data structure (class descriptor) that captures object oriented aspects of the class.

After loading the main class and its dependent classes, the code generator analyzes these classes to determine what methods to compile. It builds a tree of callable methods and proceeds to compile every node in the tree. The resulting binary code includes a minimal set of methods needed for runtime execution. The proxy sends the binary code to the handheld, and the handheld maps it into memory for execution.

After mapping the code into memory, the handheld sends the code address back to the proxy. The proxy uses this information to update method tables so that they contain absolute method addresses. It then sends a list of class descriptors to the PDA. While executing the program the runtime loader can refer to these class descriptors whenever it needs object-oriented information specific to a class (e.g., virtual method addresses).

3. IMPLEMENTATION

In this section, we cover the implementation of the ProxyVM prototype system. The discussion consists of two parts. Section 3.1 to Section 3.4 cover issues specific to Java. These issues include support for the Java object model, garbage collection, multithreading and exception handling. Section 3.5 to Section 3.6 cover other implementation issues specific to the ProxyVM, including selective compilation, runtime services, and Java library support for the handheld.

3.1 Object Model

To represent object-oriented aspects of a program, the proxy builds a list of data structures called *class descriptors*. Ev-

ery loaded class has its own descriptor that stores information specific to that class. This information includes the class instance size, super-class reference, static fields, virtual method addresses, class initializer offset, reflection handler offset, and interface method table offset. The proxy sends the descriptors to the handheld after sending the binary code. The runtime system consults these descriptors whenever it needs class-specific information, such as method addresses. The ProxyVM compiler embeds in every object a pointer to the associated class descriptor table. The runtime system uses this pointer to find the virtual method table for the class.

At compile time, every method of a class receives an offset in the method table. Since Java supports only single inheritance, the offset is the same for every derived class.

As usual, interface method dispatch is more difficult. Interfaces are the closest approximation to multiple inheritance in Java. An interface is a class having only abstract methods and final constants. A class implements an interface by providing definitions for all the interface methods. A class can implement several interfaces, and different classes can implement the same interface method differently by giving different definitions for the method body. Traditional strategies for virtual method resolution do not apply in a straightforward way to interface methods. Virtual methods in Java receive the same offset in the method tables of their defining classes and subclasses. This consistency allows fast lookup times. In contrast, an interface method does not necessarily have the same offset in every implementing class's method table.

```
interface I {
  public void f();
};

class A {
  public void g() {};
};

class B
extends A
implements I {
  public void f() {};
};

class C
implements I {
  public void f() {};
};
```

Figure 2: Example for interface method offset inconsistency problem

Consider the example in Figure 2. Class A has a single method g. This method takes up the first available slot in A's virtual method table (VMT). A has a subclass B that additionally implements the interface I. Interface I demands that all conforming classes implement the method f. In B, the method f will appear in the next available slot, which is 1, since slot 0 already contains the method g inherited from A. If a class C implements interface I but does not derive from A, the method f would appear in slot 0 instead of slot 1 as was the case for B.

We handle the interface dispatch problem in our ProxyVM using selector indexing with collision resolution [2]. Each class has a fixed size interface method table (IMT) and every interface method has a unique global id. Loading a class includes hashing all the interface methods to the IMT based on id.

Under this approach, two or more interface methods may hash to the same IMT slot. A slot to which only one method hashes stores a pointer to that method. A slot to which two or more methods hash stores the address of code that the compiler generates whenever it detects a collision.

This collision resolution code consists of pairs of comparison and branch instructions. The instructions compare the called procedure's id with the id of every method that hashes to the IMT slot and jump to the method with matching id.

Selector indexing with collision resolution solves the problem of assigning consistent offsets to interface methods. By allowing all interface method tables to retain a fixed size, it also guards against the memory consumption problem. In the average case, the cost for the collision resolution code is negligible [2].

3.2 Garbage Collection

For shorter response times, the ProxyVM uses an incremental mark-sweep tracing garbage collector. Unlike atomic garbage collectors, incremental collectors execute concurrently with the program, thereby removing the need to pause the program for garbage collection. Our garbage collector uses a Tricolor marking scheme with a write-barrier approach.

One problem with incremental tracing is that the reachability graph may change during tracing due to concurrent execution of the program. The Tricolor marking scheme addresses this problem [20]. In a simple mark-sweep garbage collection algorithm, all objects are initially white. The algorithm colors reachable objects black while tracing down the graph from the root set. The collector finally retains all black objects and reclaims white objects. Incremental tracing uses a third color gray, to prevent the tracing failure that arises from graph changes. Under incremental tracing, the reached objects are initially gray. A gray object becomes black only after the collector traverses its internal reference pointers. Thus there are no pointers directly from black objects to white objects, which we call tricolor invariant.

A garbage collection algorithm can maintain the tricolor invariant with either a read-barrier approach or a write-barrier approach. We choose the latter, since heap writes are far less common than heap reads, thus making write-barrier techniques cheaper. We implement the write-barrier by notifying the garbage collector after every $PUTFIELD$ or $PUTSTATIC$ instruction. White objects caught by the write barrier turn gray immediately.

3.3 Threads and Synchronization

Thread support is another essential feature of Java. A multi-threaded environment allows independent threads to operate on data in a shared address space. Proper communication among threads requires synchronization mechanisms.

Java uses *monitors* for synchronization, thereby ensuring that only one thread at a time can execute a critical region of code. The language supports thread functionality through the methods of *Thread* and *Object* classes, and support synchronization through synchronized methods and synchronized blocks that are marked by *synchronized* keyword. Every JVM must provide support for threads and synchronization, and our implementation uses the *Pthreads* library routines [4].

The Pthreads API subroutines generally provide the following three major functions:

Thread management This group of subroutines work directly on threads - creating, detaching, joining, etc. They also include functions to set or query thread attributes.

Mutexes The second group of subroutines deal with synchronization using a mutex (mutual exclusion). Mutex routines enable creating, destroying, locking and unlocking mutexes. They also include routines that set or modify mutex attributes.

Condition variables This third group of subroutines deal with condition-based communication between threads that share a mutex. Routines to create, destroy, wait and signal based upon specified variable values, and routines to set or query condition variable attributes all belong to this group.

The ProxyVM implements the Java thread functionality by writing native wrapper procedures to invoke appropriate Pthreads subroutines.

For thread management, a central data structure is maintained to hold information about all the created threads, including their pthread handles, liveness, interrupted-ness, current stack boundaries and local heaps[1]. This centralized structure is very useful for obtaining information about a thread that is not the current thread. It also helps services such as the garbage collector traversing the stacks of all threads.

Another central data structure in the ProxyVM implementation is for mutexes. According to the JVM specification, every object has its own mutex for locking and synchronization. An object's mutex is accessed whenever there is a synchronization request using this object. While it is natural to have a separate mutex field embedded within each object, this would cause excessive memory overhead since only a very low percentage of all created objects are actually used in synchronization. Instead, we use a hash table of linked lists with mutex elements, indexed by the instance address of the object. To prevent unnecessary overhead, a mutex for an object is created and inserted into this mutex structure only when an attempt is made to use it. An object's mutex, if it exists in the mutex structure, is destroyed when the garbage collector reclaims the object.

[1]Each thread has a separate heap of its own in ProxyVM implementation.

3.4 Exception Handling

Java's exception handling mechanism provides a clean way to check for and deal with errors at runtime. Efficient exception handling is essential for performance.

Two common exception handling techniques include:

Stack cutting Set the stack pointer and the program counter to point directly to the exception handler.

Stack unwinding Unwind the stack frame one at a time to search for the corresponding exception handler.

The first technique can be very fast in a native implementation that saves two pointers. The *setjmp* and *longjmp* instructions in C code are also useful for stack cutting, though they are more expensive since they normally save and restore necessary states [16]. To cut the stack directly to the exception handler, this technique uses a dynamically growing linked-list of data structures representing try blocks. Upon entering a *try* block, the algorithm adds a new node to the list. Upon exiting the block, it deletes the node. Stack cutting will not automatically restore the callee-saved registers in method calls. Thus the exception handling data structures can also save these registers. Stack cutting has been used in exception handling implementations for C++ [3] and Ada [6].

The second technique unwinds the stack by one frame at a time until it reaches the exception handler. It restores callee-saved registers after each unwinding iteration. The approach uses exception tables to find the appropriate exception handler at runtime. Stack unwinding has also been used to implement C++ exception handling [9] and is the model adopted by the JVM [12]. In Java, an exception table is associated with each method if necessary. Each table entry stores the program counter values for start and end of the guarded *try* block, the program counter of the exception handler, and the class type of the exception the exception handler is about to catch. At runtime, when an exception occurs, the JVM searches the exception table and directs program execution to the exception handler. If no matching exception handler is found, the JVM unwinds the stack by one frame and searches the exception table at the upper level method. This process continues until the exception is caught by an exception handler, or until the topmost level of method is hit and the JVM default exception handler catches the exception.

Both techniques are not universally perfect in terms of efficiency. Stack cutting has the overhead to dynamically maintain the linked data structure regardless of whether the exception really happens. Stack unwinding can incur more overhead than stack cutting if unwinding occurs several times. Thus stack cutting is more efficient in exception handling paths and stack unwinding is more efficient in normal paths.

In ProxyVM implementation, a modified CACAO stack unwinding approach [10] is used, since we believe efficiency in the normal path has the best payoff. The reason is that exceptions are rarely raised in real world Java programs. For example, in Java SPEC benchmarks, 5 out of 8 benchmarks do not generate any exceptions and only 2 benchmarks generate notable exceptions [11]. By choosing the stack unwinding approach we avoid the unnecessary penalty in the normal path caused by stack cutting, and by choosing a naive but fast scheme, as is used in CACAO, we also make the exception handling path efficient.

Our exception handling implementation uses a mechanism similar to that of a typical JVM interpreter. Each method has an exception table, which is searched whenever an exception is thrown. This native exception table is generated from the Java exception table, with all the bytecode pointers translated into native code pointers.

3.5 Selective Compilation

Selecting methods to compile is an important decision that can significantly impact compile time and code size. It is an especially critical decision for the ProxyVM infrastructure: Since the proxy does all work associated with loading and compiling classes and must also send information back to mobile devices, it must not become a bottleneck. A naive approach is to compile every method in every loaded class. This approach does not scale since Java programs are full of dependencies. When a method in one class calls a method in another unloaded class, the JVM must load the second class, too. Java programs often have lengthy chains of method calls that require loading several hundred classes. This class explosion problem is most apparent in the case of GUI programs. The statement, *Frame f = new Frame( )*, requires loading over 800 classes. By no means are all methods of all loaded classes callable. If the JVM were to compile every method of every loaded class, it would waste time compiling unused methods and in most cases the resulting binary would be too large for the handheld device. Thus it makes sense to compile only only callable methods. In addition, the identification of callable methods allows the runtime system to skip the dynamic dispatch for call sites having only one possible target.

The ProxyVM implementation uses standard techniques to identify methods that might be called at runtime. The approaches implemented so far include *Class Hierarchy Analysis (CHA)* and *Rapid Type Analysis (RTA)* [19]. While these are ad hoc approaches, they are very fast since they only require parsing the bytecode stream once. The analysis results computed by CHA and RTA already allow us to significantly minimize the number of callable methods.

When deleting uncallable methods, there are some special cases to consider. We discuss these cases in more detail.

The JVM invokes class initializers whenever it loads a class. These initializers assign values to static fields and execute any user defined initialization code inside a static block. Initially the JVM invokes the class initializers of the main class and its superclasses. Thus, the ProxyVM adds these initializers to the set of callable methods after adding `main`. Also, there are specific JVM instructions that prompt class initialization. These include `NEW`, `GETSTATIC`, `PUTSTATIC` and `INVOKESTATIC`. Whenever the VM encounters these instructions, it must add the initializers of the target class and superclasses to the set of callable methods.

The Java Native Interface (JNI) is a programming environment that allows Java and C programs to exchange information. Generally, designers implement native methods in C using the JNI library and store the code in a library. Since the C code might have callbacks to Java code, the final set of callable methods must include any Java method a native method calls. A comprehensive but costly solution is to parse the C library files containing native method implementations in search of callbacks to Java code. Fortunately, these callbacks are rare, allowing us to adopt an easier solution. We add native methods to the set of callable methods based on external information.

As mentioned above, interface methods pose an interesting challenge since their resolution happens only at runtime through the interface method table. In some cases, we can identify the exact target of an interface method call after deleting uncallable methods. In these cases, the runtime system can invoke the method directly instead of using the IMT.

3.6 Runtime services

Some bytecode instructions depend on services available only at runtime. For example, there is no ARM instruction for integer division. Thus the proxy must compile the division bytecode instruction into native code that calls a C library division routine. Native methods also rely heavily on runtime services since they are platform dependent.

Our runtime system includes C routines that, depending on the handheld device, provide functionality that otherwise would be unavailable. At compile time, the ProxyVM does not know the absolute addresses of these routines, but only the distance between the current program counter and a special *jump table* that it builds to invoke the routines. The native code for a division instruction generated by the proxy for an ARM target then has a branch to the jump table entry for the division runtime service.

The jump table holds an entry for each type of invoked runtime service. An entry takes up two slots. The first slot has an ARM instruction for jumping to the address in the second slot. The proxy leaves the second slot empty. The runtime system fills this slot with the runtime service address.

3.7 Java Runtime Libraries

Traditional VMs for embedded devices usually store their libraries on the target device. As a result, they must provide stripped-down versions of these libraries that use minimal resources of the device. Consider Sun's PersonalJava Application Environment, which targets miniature devices such as cell phones. The online documentation for this runtime environment shows that numerous libraries are either optional or not fully supported [18].

Since all Java libraries in our infrastructure reside on the more powerful proxy, our library implementations are less subject to resource limitations of the embedded environ-

ment. We are able to support more libraries more thoroughly than traditional VMs that target mobile devices. We use the complete Sun JDK 1.3.1 versions for most libraries. An exception is the Abstract Window Toolkit (AWT), a fundamental library for GUI support.

To implement the AWT, we have re-engineered the version in Kaffe[8], using the Qt-Embedded library to manipulate the Zaurus framebuffer. This version has slightly more functionality than Sun's AWT and is also resource efficient.

4. RELATED WORK

To our knowledge, there has been little prior work on server-based compiler infrastructures for Java. The recent work most similar to the research we are proposing is the Mojo virtual machine, developed at the University of Sussex [14]. The basic Mojo framework consists of a proxy server containing a Java-bytecode-to-C compiler and a C compiler for the target platform. Initially, Mojo statically compiles the main Java class and its transitive closure to ISO-C source[2]. The C source includes implementation for all class methods and additional code that adds class-specific information to a table of loaded classes at runtime. The loading and compilation of additional classes occurs on demand during program execution. The runtime system checks the global class table to see if a required class has been loaded. If the class has been loaded, the runtime system can instantiate it using the class table information. Otherwise, the system requests the class from the proxy, which loads and compiles the class to C source, compiles the C source to native code and sends the native code to the target.

While the Mojo VM reduces resource usage on the handheld through proxy compilation, it introduces an additional overhead by compiling class files first to C source and then to native code. Our ProxyVM avoids this overhead by directly compiling class files to native code and building class descriptors in the process. Mojo also does not include runtime profiling or application partitioning, which are two areas that we intend to explore.

Another remote compilation system is the Java Compiler On Demand (JCOD) system developed at the Silicomp Research Institute [5]. Under this framework, a Java application is initially downloaded to the target device, which contains a modified version of Hewlett Packard's Chai VM that supports dynamic profiling. The application first runs through an interpreter. While the program is running, the VM on the target profiles it to collect statistics on execution frequency. When the frequency of a method call reaches a threshold, the VM requests a remote compile server to compile that method. In turn, the server compiles the method and sends native code back to the target. Like Mojo, JCOD reduces resource usage and improves code generation by shifting compilation work to a more powerful proxy. However, it incurs the memory costs of storing a VM and required class files on the target. Our ProxyVM avoids these costs by storing class files and all VM related files on the proxy.

[2]The transitive closure of a class A consists of all classes that A references, excluding classes loaded through the *java.lang.class.forName*() method.

There have been related efforts to build distributed virtual machines. The Kimera project at the University of Washington is one such effort [17]. The goal of Kimera is to split a virtual machine into functional components and distribute these components over organization-wide network servers to amortize the overall costs of VM services.

The Kimera VM architecture consists of static components (verifier, security-enforcer, compiler) that execute once before a program runs, and dynamic components (network manager, library manager, runtime type-checker) that service a program while it is running. Static components typically reside on one or more centralized proxies that form gateways to an organization's local network. These proxies append digital signatures to ensure that all code entering an organization passes verification and security checks. The system may redirect internally modified code to the centralized servers for verification.

While the Kimera architecture uses ideas similar to what we propose (remote compilation, runtime monitoring), it targets the local area networks of companies. Though the framework involves partitioning at the virtual machine level, it places less emphasis on Java application partitioning and continuous optimization.

Kimera also does not address all issues arising in a more general setting where the distances between servers and users may possibly span up to several hundred miles. Our research must consider such issues. In the scenarios we are exploring, it is less efficient to spread a single JVM's functionality across multiple servers, since this would significantly increase communication costs of the interacting JVM components. Network management is another complex issue that we intend to address. Rather than centralizing network services on a single server that may be thousands of miles from the nearest handheld device, we may need to store a complete JVM on each proxy and make devices interact with the nearest proxy. Of course, we will also have to formulate appropriate cost models for migrating applications between servers in response to changing user locations.

5. CONCLUSIONS

We have presented a framework for network-based compilation of Java programs for resource-constrained handheld devices. Using this framework, a powerful proxy server intercepts class file requests, downloads and verifies all necessary class files, and compiles the bytecode to executable native code. This approach supports the offloading of resource-intensive compilation and optimization phases to the proxy, so that it is not necessary to store a JIT compiler on the handheld. At the same time server-based compilation enables more sophisticated optimizations due to the additional resources available at the proxy. With the prototype implementation of our ProxyVM we have also demonstrated that our architecture supports a much broader set of library functions for mobile device applications since the handheld only needs to download required parts of the runtime system.

6. FUTURE WORK

We have identified three main areas that we will focus our future work on. First, we plan to introduce a monitoring and profiling subsystem on the handheld to enable dynamic

compilation and dynamic optimization. The revised compiler will also include a more sophisticated selection mechanism for selecting callable methods based on *Rapid Control Flow Analysis* [15], which in contrast to the currently used techniques is flow-sensitive and promises significantly better results. Our second major area of interest is partitioning. Currently the whole program executes on the handheld device. Applications that are too resource intensive to execute entirely on the handheld will execute jointly on the handheld and the more powerful proxy server. We plan to explore various partitioning strategies for our framework to minimize resource usage at the target device and communication costs. Finally, we intend to port our ProxyVM to more handheld devices to prove the universal applicability of this approach. This phase is going to target especially highly resource-restricted devices such as mobile phones on the one end of the spectrum and more powerful ones like more powerful handheld devices on the other end.

7. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V.Litvinov, M. F. Mergen, T.Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Virtual Machine. In *IBM System Journal*, 2000.

[2] B. Alpern, A. Cocchi, S. J. Fink, D. Grove, and D. Lieber. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Conference on Object-Oriented*, pages 108–124, 2001.

[3] D. L. D. Cameron, P. Faust and M. Mehta. A Portable Implementation of C++ Exception Handling. In *C++ Conference*, pages 225–243, Aug. 1992.

[4] POSIX Threads Programming. http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/main.html.

[5] B. Delsart, V. Joloboff, and E. Paire. JCOD: A Lightweight Modular Compilation Technology for Embedded Java. *Lecture Notes in Computer Science*, 2491, 2002.

[6] E. W. Giering, F. Mueller, and T. P. Baker. Features of the GNU Ada Runtime Library. In *TRI-Ada*, pages 93–103, 1994.

[7] The Java Hotspot Virtual Machine, Technical White Paper, Sun Microsystems, Inc. 2001.

[8] KAFFE's Homepage http://www.kaffe.org/.

[9] A. Koenig and B. Stroustrup. Exception Handling for C++. In *C++ Conference*, pages 149–176, 1990.

[10] A. Krall and M. Probst. Monitors and Exceptions: How to Implement Java Efficiently. *Concurrency: Practice and Experience*, 10(11–13):837–850, 1998.

[11] S. Lee, B.-S. Yang, S. Kim, S. Park, S.-M. Moon, K. Ebcioglu, and E. Altman. Efficient Java Exception Handling in Just-in-Time Compilation. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 1–8. ACM Press, 2000.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addision-Wesley, 2nd edition, 1999.

[13] G. Manjunath and V. Krishnan. A Small Hybrid JIT for Embedded Systems. *SIGPLAN Notices*, 35:44–50, 2000.

[14] M. Newsome and D. Watson. Proxy Compilation of Dynamically Loaded Java Classes with Mojo. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems*, 2002.

[15] C. W. Probst. *A Demand-Driven Solver for Constraint-Based Control Flow Analysis*. PhD thesis, Universität des Saarlandes, 2002.

[16] N. Ramsey and S. P. Jones. A Single Intermediate Language that Supports Multiple Implementations of Exceptions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 285–298. ACM Press, 2000.

[17] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and Implementation of a Distributed Virtual Machine for Networked Computers. In *Symposium on Operating Systems Principles*, pages 202–216, 1999.

[18] Personal Java Application Environment Specification, Sun Microsystems, Inc. 2000.

[19] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, Oct. 2000.

[20] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proc. Int. Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag.