SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:129–146

EJVM: an economic Java run-time environment for embedded devices

Da-Wei Chang[*,†] and Ruei-Chuan Chang

Department of Computer and Information Science, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsinchu 30050, Taiwan

SUMMARY

As network-enabled embedded devices and Java grow in their popularity, embedded system researchers start seeking ways to make these devices Java-enabled. However, it is a challenge to apply Java technology to these devices due to their shortage of resources.

In this paper, we propose EJVM (Economic Java Virtual Machine), an economic way to run Java programs on network-enabled and resource-limited embedded devices. Espousing the architecture proposed by distributed JVM, we store all Java codes on the server to reduce the storage needs of the client devices. In addition, we use two novel techniques to reduce the client-side memory footprints: server-side class representation conversion and on-demand bytecode loading. Finally, we maintain client-side caches and provide performance evaluation on different caching policies. We implement EJVM by modifying a freely available JVM implementation, Kaffe. From the experiment results, we show that EJVM can reduce Java heap requirements by about 20–50% and achieve 90% of the original performance. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: Java virtual machine; embedded system; memory footprint

INTRODUCTION

In recent years, Internet technologies together with the rapid development on embedded systems techniques have made many network-enabled embedded devices become prominent. According to the previous research [1], these devices such as smart handheld devices, cellular phones and pagers are poised for rapid growth in the market. More importantly, consumers are driving the shift of embedded devices from static, fixed-function systems to more dynamic and extensible ones [2].

One of the most critical technologies in this trend is Java [3]. Java is not only a programming language, but also a run-time environment. The excellent features provided by Java, such as platform

[*]Correspondence to: Da-Wei Chang, Department of Computer and Information Science, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsinchu 30050, Taiwan.
[†]E-mail: gis86805@cis.nctu.edu.tw

Copyright © 2001 John Wiley & Sons, Ltd.

Received 20 December 1999
Revised 31 August 2000
Accepted 31 August 2000

independence, automatic memory management and so on, make it quickly accepted by programmers, including embedded system developers.

Unfortunately, it is not easy to apply Java technology to resource-limited embedded devices due to the large resource requirements of Java platforms. Generally speaking, the resource consumption of a Java platform comes from two sources:

1. *Storage overheads of the virtual machine and the Java system class libraries*. To make a device Java-enabled, not only a Java virtual machine (JVM) [4–6] but also the Java system class libraries have to be stored on that device. However, the huge size of the class libraries makes it difficult to store them completely on resource-limited embedded devices. For example, the size of the class libraries shipped with JDK 1.1 is about 9.5 MB [7]. Assuming that an embedded device always performs fixed functions, one may store only the required applications and libraries on that device. However, this approach limits the extensibility of the device and prevents it from supporting diverse applications [2].
2. *Ineffective use of memory*. Current JVM implementations treat a Java class file as a non-separable unit. When a Java method (i.e. procedure) is invoked, it can not be executed until the class containing it is totally loaded into the memory. This results in ineffective use of memory since many unused Java methods are also loaded. As indicated by the previous research [8], it is possible that more than one-half of the codes in a Java program are unused. The benchmarks for our experiments also reveal the same thing (see Table III). Therefore, the memory footprints may be reduced by only loading Java codes that are actually used.

Distributed JVM [9,10] (DJVM) provides an approach to save the resource consumption of the JVM and the system class libraries. It factors out some components (e.g. bytecode verification) from a JVM (which resides on a client host continuously connected to the server) and stores the Java class libraries on a server host. All Java classes are dynamically loaded from the server on demand. This approach trades the storage needs on the clients to the network delay. With the increase in network bandwidth and reduction in transfer latency, it becomes more feasible for these continuously connected clients to defer their functionality to servers.

In order to use memory more effectively, many research projects including DJVM make use of the class file splitting technique. In their approaches, infrequently used Java methods, which are identified by profile data, are factored out from the original class (i.e. the 'hot' class) to one or more new classes (i.e. the 'cold' classes). In the hot class, the bytecodes[‡] of these infrequently used methods are rewritten. They are replaced with stubs that forward method invocations to the corresponding method implementations in the cold classes. For most of the cases, only the hot class will be used and loaded to the memory. As a consequence, memory consumption can be reduced.

In this paper, we focus on making effective use of memory in Java environments. We propose EJVM (Economic Java Virtual Machine), an economic way for executing Java programs on resource-constrained embedded devices. It is a Java run-time environment that follows the client–server model. Espousing the architecture proposed by DJVM, EJVM allows the applications to use the full Java API

[‡]A Java program contains a set of Java methods. A Java method in turn contains a sequence of bytecodes. In the rest of this paper, 'bytecodes of a method' represents the bytecode sequence contained in the Java method, and 'bytecodes of a class' stands for bytecodes of all methods in that class.

without the need for storing the Java class libraries on the embedded devices. However, our work is unique in the following three aspects.

First, we factor out the job of converting representations of Java classes to the servers. Java programs are originally represented in the form of class files (i.e. file representations of classes [11].) The traditional way to load a class is to read the class file into the memory, and then convert it to the run-time class representation. This results in larger memory footprints since both kinds of class representations have to be stored during the loading of the class. Therefore, it is not suitable for memory-limited embedded devices. To address this problem, we move the job of converting class representations to the server hosts and transfer only the run-time representations of classes to the clients. This saves memory space not only for the file representations, but also for codes that perform class-representation conversion.

Second, we load bytecodes of Java methods on demand. As we described earlier, current JVM implementations treat a Java class as a single non-separable unit. When a class is loaded into memory, all of its bytecodes are also loaded. This lowers the effectiveness of the memory usage. For example, if a device has no input equipment, all bytecodes dealing with inputs will be useless. Therefore, to reduce the memory footprints, we load bytecodes only when they are actually needed. This is similar to the demand paging technique [12] used in virtual memory management systems. Unlike the class splitting technique, this approach requires neither bytecode-rewriting nor profile data.

Third, and most importantly, we cache Java bytecodes on the clients and evaluate the performance of different caching policies. According to the performance results, EJVM saves about 20% to 50% of the Java heap requirements, and achieves 90% of the original performance.

The remainder of this paper is organized as follows. We describe the design and implementation of EJVM in the next section. Following them are the performance measurement and analysis. Next, we describe the related works. Finally, the conclusions and future work are presented.

DESIGN AND IMPLEMENTATION

The design goal of EJVM is to save the memory footprints of a Java platform. We use two techniques to achieve this: server-side class representation conversion and on-demand bytecode loading. In this section we shall describe these techniques after an overview of the EJVM architecture. In addition, we also describe the implementation details of the server-side cache as well as the different caching policies of the client-side cache.

Architecture overview

Based on the idea of DJVM, EJVM stores all the application codes and Java system class libraries on the server. Figure 1 shows the architecture of EJVM and the control flow of requesting Java classes and bytecodes. A typical Java application runs on the client host. When a class is needed, the class loader on the client sends a request message to the server to ask for that class (step 1). After receiving the message, the server frontend parses it and queries the cache manager to see whether the requested class is in the cache (step 2). In case of cache hit, the cache manager sends the requested class (in the form of a message) to the server frontend (step 7), which forwards it to the client (step 8). However, if cache miss occurs, the cache manager has to invoke the class file reader to read the class file from the file system (steps 3 and 4). After the reading is finished, the class file reader hands the class to the

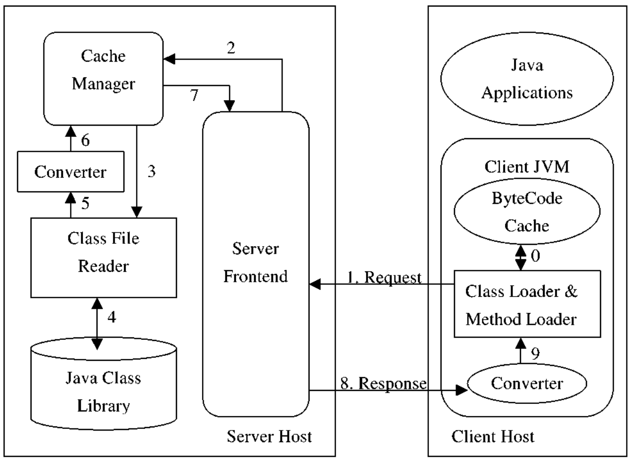


Figure 1. EJVM system architecture.

converter. The converter removes all the bytecodes within the class, marshals the resulting class to a message with ‘class’ type (step 5) and inserts the message into the cache (step 6). At this time, the cache manager is able to send the message to the server frontend (step 7), which in turn forwards it to the client (step 8). After receiving a class type message, the client JVM unmarshals it to the run-time format of that class and hands the result to the class loader (step 9). After the conversion, the class is totally the same as a loaded-from-disk class except that it contains no bytecodes.

When a method in the class is invoked, the client JVM checks whether the corresponding bytecode sequence is attached to the class (specifically, the run-time data structures of the class). If not, the JVM will load it in a way similar to the class-loading procedure we have described above. However, there are two things worth noting. First, before loading the bytecodes (i.e. bytecode sequence) from the server, the bytecode cache on the client will be searched in advance. A request for the bytecodes is sent only when the cache misses. Second, only the bytecode sequence of the invoked method is transmitted[§]. Other pieces of bytecodes are transmitted when their corresponding methods are invoked.

To implement EJVM, we modify Kaffe (version 0.10.0) [13], which is a freely available JVM implementation. In the following, we shall describe our design and implementation techniques in a more detailed way.

Server-side class representation conversion

A major part of the class-loading procedure is to convert the representation of a class from the class file format to the run-time format. This conversion requires memory overheads such as room for the class file, codes and data structures to parse that class file, etc. To minimize these overheads, we offload

[§]In fact, the bytecode sequence is transmitted together with its exception table and line number table.

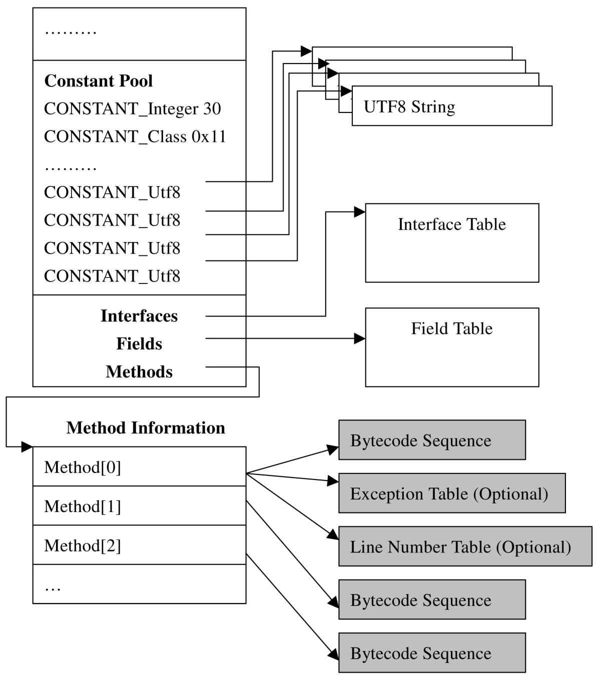


Figure 2. Run-time class format.

the conversion task from the client JVM to the server (specifically, the server-side converter). The conversion task performed on the server is the same as that performed by a traditional JVM. Therefore, we omit the details of the conversion task in this paper. Figure 2 shows the result of the conversion. The constant pool is the symbol table of the program. The interface table records the interfaces this class implements, and the field table stores all the fields in the class. Each method in the class occupies an entry in the method information table. This entry contains links to the bytecode attributes, which consist of the bytecode sequence, the (optional) exception table, and the (optional) line number table. Once the conversion is done, we have to transmit to the client the pointer-rich data structures illustrated in Figure 2, except for the areas colored gray. To achieve this, we marshal (i.e. serialize) the data structures and use relative addressing (from the start of the message buffer) for swizzled pointers. After receiving the message, the client JVM is able to unmarshal it to reconstruct the run-time format of the class.

It should be noted that, because bytecode attributes will be used only when the corresponding Java method is invoked[¶], we split out all of them (from the original class) before marshaling the class.

[¶]In EJVM, we assume that all classes are preverified so that the client JVM does not have to perform verification during the loading of the class. Therefore, bytecode attributes will not be used unless the corresponding method is invoked.

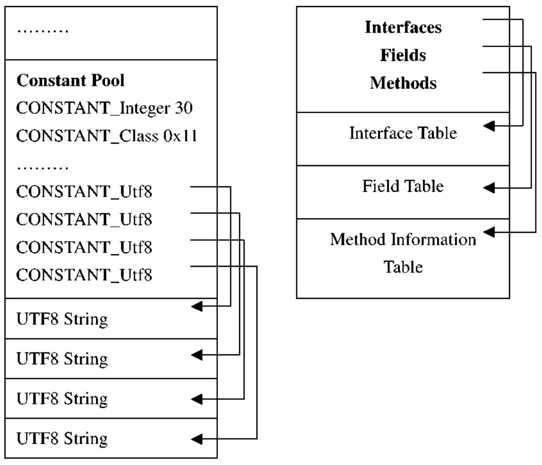


Figure 3. The class type message.

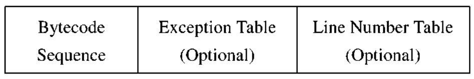


Figure 4. The bytecode type message.

Figure 3 shows the result of the marshaling. Bytecode attributes, which are colored gray in Figure 2, are not included in Figure 3. Instead, they are marshaled separately and transmitted when the client explicitly requests them. Figure 4 shows the result of marshaling on bytecode attributes. Note that the bytecode attributes of a Java method will be treated as a single non-separable unit for transmission and caching[‖].

Server-side cache management

To improve the server performance, the server-side cache manager caches all the marshaled classes and bytecodes in the server's memory. This prevents us from reading, converting and marshaling the same classes. The cache manager maintains two memory caches: the class cache and the bytecode cache. Both of them are implemented as hash tables. The hash key of the class cache is the class

[‖]Although bytecode attributes contain not only bytecodes but also exception and line number tables, these tables are not much related to the discussion in this paper. Thus, for simplicity, we refer bytecode attributes as bytecodes in the rest of the paper.

Table I. Bytecode-caching policies.

Policy	Description
CACHE_NONE	Dropping all bytecodes
CACHE_ALL	Holding all bytecodes
CACHE_HOT	Holding hot bytecodes only
CACHE_LRU	Performing LRU replacement policy on the cache

name. However, the hash key of the bytecode cache is a 3-tuple: <class name, method name, method signature>. This is because only this 3-tuple uniquely identifies a Java method[**].

In case of cache-hit, the cache manager returns the requested class or bytecodes (in the form of a message) to the server frontend. Otherwise, it invokes the class file reader, which eventually makes the class and all of its bytecodes be marshaled and cached. At this time, the cache manager is able to return the requested object.

On-demand bytecode loading

As we described earlier, EJVM loads bytecodes only when the corresponding methods are invoked. In order to achieve this, it performs the following two tasks. First, it separates bytecodes (i.e. bytecode attributes) from classes. As we mentioned above, this is done by the server-side converter before it marshals classes and bytecodes. Second, it checks the existence of the bytecode sequence when the corresponding method is invoked. If it does not exist, EJVM will search for it in the client-side bytecode cache. A miss on that cache results in a bytecode request being sent to the server.

Bytecode cache policies

To prevent from reloading bytecodes from the server, we maintain a bytecode cache on the client. The goal of the cache is to minimize the memory footprints as well as to keep the performance acceptable.

We designed and evaluated four bytecode-caching policies, which are shown in Table I. The CACHE_NONE policy drops all the bytecodes immediately after they are completely executed. In contrast, the CACHE_ALL policy keeps all the bytecodes that it ever loads. Obviously, they are two extremes of all the possible bytecode-caching policies. The former uses the least memory space but results in the longest execution times; the latter leads to the least execution times but consumes the largest memory footprints.

[**]Strictly speaking, a class C is uniquely identified by the pair <class name of C, class loader>, where the class loader is a class used to load C. However, for simplicity, we assume that all classes in EJVM are loaded by the default (i.e. system) class loader. Therefore, we can identify a class by using its name. The same assumption is also made on Java methods.

The third policy (i.e. CACHE_HOT) caches ‘hot’ bytecodes only. A piece of bytecode is identified as ‘hot’ if its access count is larger than a predefined constant, MAXDROPS. In order to implement this policy, we maintain per-method access counts, which are increased by one whenever the corresponding methods are invoked.

The CACHE_LRU policy makes use of a small, fixed-size cache to keep bytecodes. In addition, as the policy name indicates, we use LRU as our cache replacement policy to prevent the cache from overflowing. All the bytecodes removed due to the cache replacement are actually dropped so that they need to be reloaded from the server next time. Whenever a piece of bytecode, say B, is completely executed, it is unlinked from its method information table entry and inserted into the cache. When the method corresponding to B gets invoked later, a cache-lookup operation is performed. After B is found, it can be unlinked (i.e. removed) from the cache and relinked on that method information table entry again.

PERFORMANCE MEASUREMENT AND ANALYSIS

In this section we make performance comparisons, in terms of client-side memory footprint and execution time, between EJVM and Kaffe. We show that EJVM reduces the memory footprints and keeps the execution times acceptable.

Benchmarks

Table II shows the benchmarks that we used to measure the system performance. ‘Hello’ is a simple Java application that prints ‘Hello, World!’ on the console. ‘SMTP’ acts as an SMTP client, sending an email to a predefined receiver. We obtain this program by modifying the SendMail version 1.0 [14], which is originally a Java applet, to a Java application. ‘POP3’ is a POP3 client that reads mail from a server and stores them in a file. ‘Java2HTML’ colors all Java keywords in a Java source file and translates it into HTML format, making it more ‘readable’ in WWW browsers. The Java class names of the ‘POP3’ and the ‘Java2HTML’ benchmarks are *VasFireWallMailReader.class* and *VasJava2HTML.class*, respectively. They are both available at Vasile’s homepage [15]. ‘CaffeineMark’ is an embedded version of CaffeineMark 3.0 [16]. It contains 6 non-graphical tests, and is used for measuring the performance of Java-enabled embedded devices. ‘SciMark’ [17] is a Java benchmark, which is used for measuring the performance of numerical codes occurring in scientific and engineering applications. Among the three problem sizes it provides, we choose the ‘default’ problem size in our experiments.

It is worth noting that all the benchmarks are Java applications instead of applets. Therefore, it is pointless to download Java class files in the original case. This facilitates measuring the effects of moving all the Java codes, which are originally located on the client, to the server and loading them on demand.

Table III summarizes some of the characteristics of our benchmarks. Column 2 shows the numbers of classes loaded during the execution of these benchmarks. Note that the classes include not only application codes but also classes in the Java system class libraries. Column 3 presents the total numbers of methods in these classes and the numbers of methods that are actually accessed. Column 4 shows the access counts for these methods. We can see that a small number of methods account for a

Table II. The benchmarks.

Benchmark	Description
Hello	A ‘Hello, World!’ application
SMTP	An SMTP client
POP3	A POP3 client
Java2HTML	A Java source to HTML converter
CaffeineMark	An embedded version of CaffeineMark 3.0
SciMark	A benchmark for testing numerical codes

Table III. Characteristics of the benchmarks.

Benchmarks	Classes	Used/total methods	Method access counts	Bytes of used/total bytecodes
Hello	94	99/1171	2329	70 751/126 245 (56%)
SMTP	113	152/1358	4060	73 828/138 652 (53%)
POP3	120	169/1443	18 314	75 767/144 258 (52%)
Java2HTML	174	148/2331	103 581	76 992/230 404 (33%)
CaffeineMark	117	202/1455	637 295	75 026/146 094 (51%)
SciMark	105	160/1257	329 630	77 959/141 103 (55%)

large amount of accesses. Column 5 presents the total sizes of bytecodes in these classes, as well as the sizes and percentages of the bytecodes that are actually used. We can see that only about 30–60% of the bytecodes are used. This is evidence that loading bytecodes on demand may be beneficial.

Experimental environment

Our experimental environment consists of a client and a server host that are connected via a 10 Mbit/s Ethernet link. The server host is a Pentium 300 MHz machine with 64 MB RAM, running Linux 2.2.5. It also acts as the mail server for the ‘SMTP’ and ‘POP3’ benchmarks. The client host is a Pentium 133 MHz machine with 32 MB RAM, running Linux 2.0.36.

Performance results

In the following experiments, we assume that the MAXDROPS is 1 for the CACHE_HOT policy, and the cache size is 4 kB for the CACHE_LRU policy. We shall discuss the impact on performance at the end of this section, when these two parameters are varied.

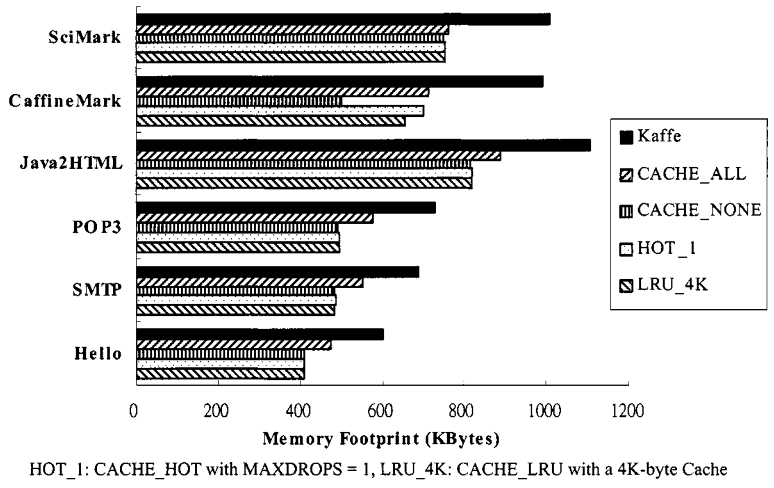


Figure 5. The memory footprints of Kaffe and EJVM.

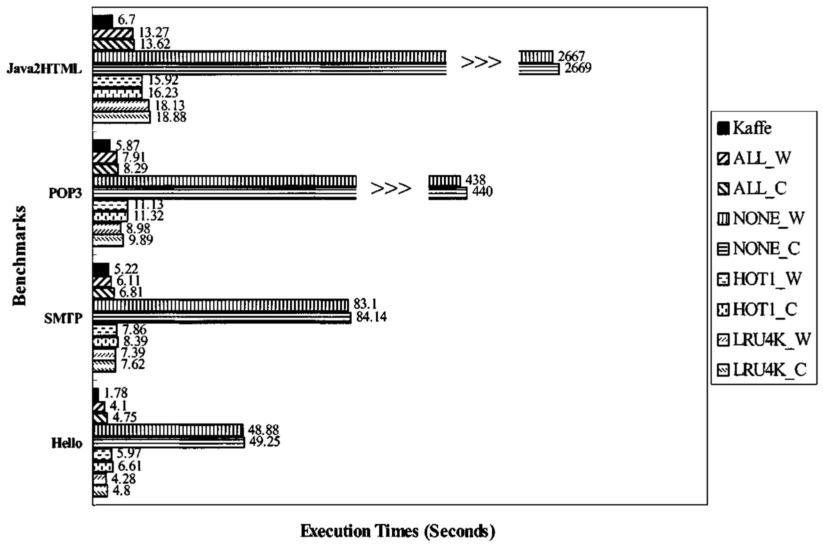
Client memory footprint

Figure 5 shows the client-side memory footprints of the benchmarks under Kaffe and EJVM. It is worth noting that the 'memory' we mention here is actually the Java heap from which a JVM allocates Java classes, objects, bytecodes, and so on. We measure these values by inserting codes into the memory allocation routines (i.e. gcMalloc() and gcFree()) of Kaffe to keep track of the high watermarks of memory consumption. This figure illustrates that all of the policies of EJVM result in smaller memory footprints, and the savings range from 20–50%. In addition, the CACHE_NONE policy results in the smallest memory footprints. This is obvious since it does not cache any bytecodes on the clients. Finally, the CACHE_HOT and CACHE_LRU policies have moderate improvements on memory footprints over the CACHE_ALL policy for four (out of the six) cases. They also lead to nearly the same memory footprints as the CACHE_NONE policy does, except for the CaffeineMark benchmark. Given that the MAXDROPS is 1 and the cache size is small (i.e. 4 kB), this indicates that most bytecodes are accessed only once and the 'working set' of bytecodes is small.

Execution time

We measure the execution times under two configurations—local and remote ones—to get a clearer view of the performance of EJVM. In the local configuration, the client and server processes both reside on the server host and communicate with each other via UNIX domain sockets. In the remote configuration, they use TCP/IP sockets on top of a 10 Mbit/s Ethernet interface for communication.

Figure 6 shows the execution times under the remote configuration when the server-side cache is either warm or cold. Figure 7 shows the number of messages transferred between the client and the



W indicates the server-side cache is warm; C indicates the server-side cache is cold

Figure 6. Execution times under the remote configuration.

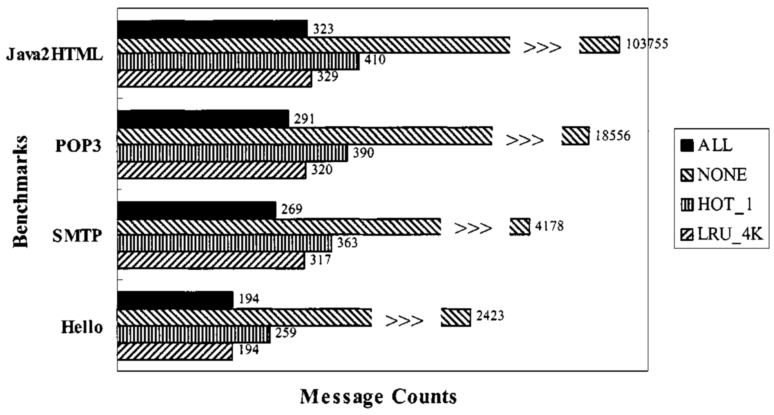


Figure 7. Message counts for different caching policies.

Table IV. Execution times (in seconds) under the local configuration.

		ALL		NONE		HOT_1		LRU_4K	
	Kaffe	Warm	Cold	Warm	Cold	Warm	Cold	Warm	Cold
Hello	0.29	**0.26**	0.69	0.95	1.62	0.31	0.97	0.30	0.90
SMTP	1.50	**0.93**	1.78	2.22	2.68	**1.49**	2.06	**1.37**	2.02
POP3	1.53	1.74	2.25	14.18	14.69	1.76	2.37	1.83	2.45
Java2HTML	1.88	2.06	2.54	254.5	255.4	2.19	2.68	2.91	3.52

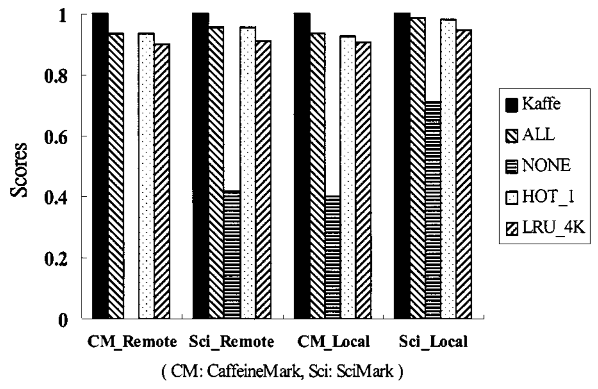


Figure 8. Normalized performance scores of EJVM and Kaffe.

server during the benchmarks run. These figures reveal four things. First, all of the policies, except for the CACHE_NONE, have performance comparable to the original Kaffe VM. The CACHE_NONE policy performs badly since it results in too many messages between the client and the server. Second, maintaining a server-side cache gains little profit in this configuration, since there are no obvious differences in execution times between the 'warm' and the 'cold' cases. This is due to the performance being limited by the network capacity, not the processing times of the server. Third, the CACHE_ALL policy has the minimum execution times and client–server communication. This is natural since it never drops bytecodes that it ever gets. Fourth, the CACHE_LRU policy results in fewer client–server interactions than the CACHE_HOT does, so it has smaller execution times for most of the cases. However, there is an exception. For the Java2HTML benchmark, the CACHE_LRU policy has a smaller message count but a longer execution time. This happens due to the high method invocation overheads of the CACHE_LRU policy. Under this policy, each method invocation involves in a cache lookup operation and a bytecode-relinking step (which relinks bytecodes from the cache to the run-time data structures of the corresponding class). In addition, another bytecode-relinking operation, in the reverse direction, is required when the method exits. Generally speaking, when the network

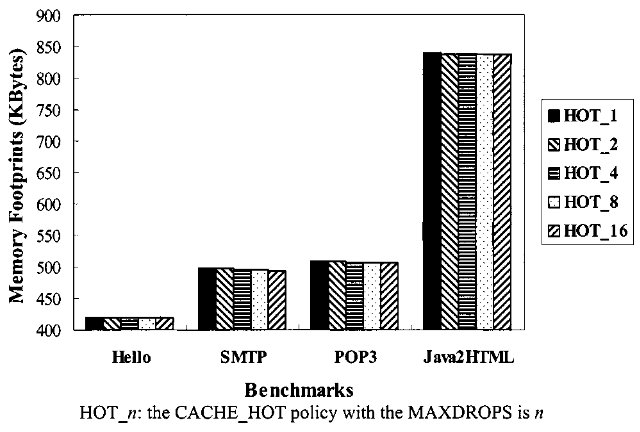


Figure 9. Memory footprints for different MAXDROPS values.

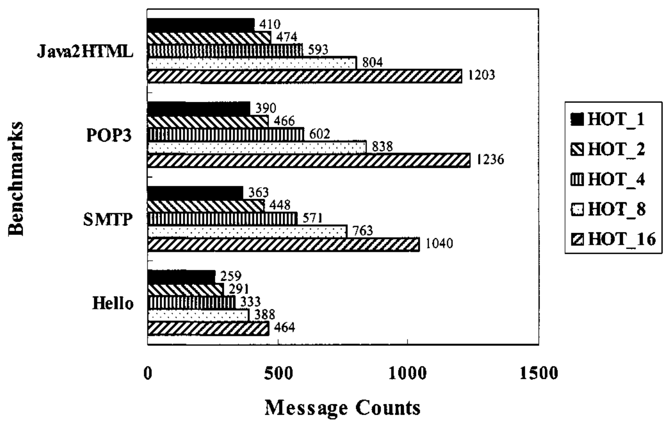


Figure 10. Message counts for different MAXDROPS values.

latency is not small, the CACHE_LRU policy tends to have better performance than the CACHE_HOT policy, provided that the method invocation frequency is not too high. In the future, we will make in-depth investigations into the impacts of the network latency and method invocation frequency on the performance.

Table IV shows the execution times under the local configuration. The 'warm' ('cold') columns present the execution times when the server-side cache is warm (cold). From the table we can see that the performance differences between Kaffe and EJVM become smaller (when compared to those in Figure 6). Moreover, there are cases, indicated in bold, that EJVM outperforms Kaffe when the server-

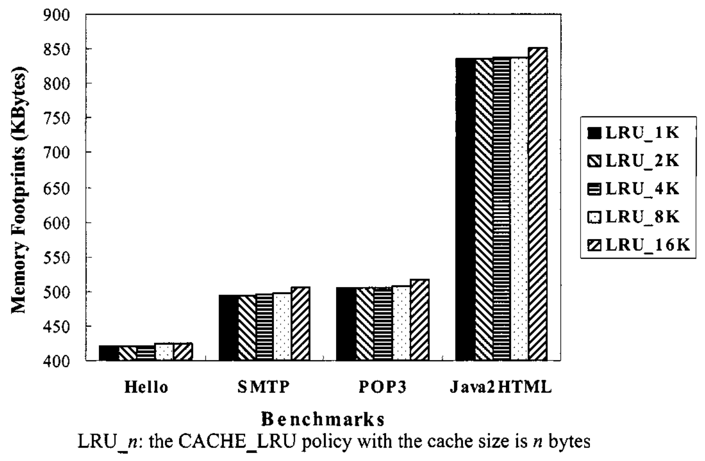


Figure 11. Memory footprints for different cache sizes.

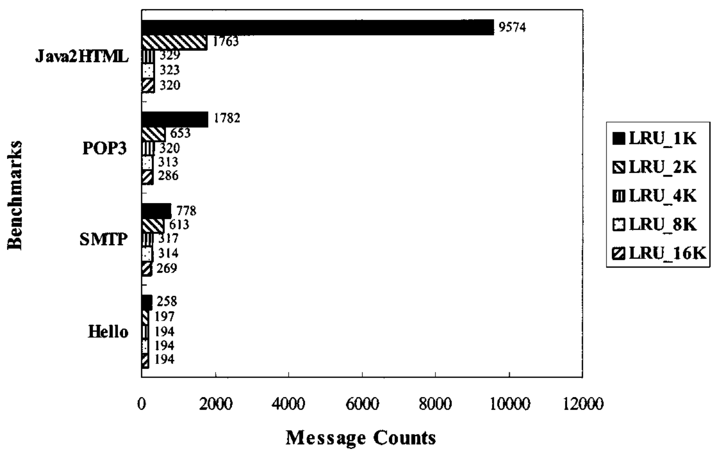


Figure 12. Message counts for different cache sizes.

side cache is warm. This happens because the network times do not dominate the execution times any more due to the low network delay in this configuration. As a consequence, the server-side cache has more effects on the system performance.

Figure 8 shows the performance scores, which are normalized to 1, of the CaffeineMark and SciMark under both configurations. From the figure we can see that all of the policies except for the CACHE_NONE achieve more than 90% of the original performance (i.e. the performance of Kaffe).

The CACHE_NONE policy performs badly under the remote configuration. For the CaffeineMark benchmark, the score even equals zero[††]. As we described earlier, this is due to the heavy client–server traffic occurring in this policy.

At the end of this section we shall discuss the influence of the two parameters, MAXDROPS and cache size, on the system performance. Figures 9 and 10 show the memory footprints and message counts, respectively, under the CACHE_HOT policy when the values of MAXDROPS are varied from 1 to 16. Although larger MAXDROPS values lead to better memory footprints, the improvements are not obvious. Larger MAXDROPS values, however, do result in larger message counts since fewer bytecodes are regarded as hot and cached. Therefore, according to these figures, setting MAXDROPS to 1 leads to the best performance. This indicates that most bytecodes in our benchmarks will continue being accessed when they are accessed twice.

Figure 11 and 12 show the memory footprints and message counts, respectively, under the CACHE_LRU policy when the cache sizes are varied from 1 kB to 16 kB. These figures reveal two things. First, heavy client–server traffic will occur if the cache is not large enough. In our experiments, a 4 kB cache is sufficient since larger cache sizes will not cause obvious reductions on the message counts any more. Second, larger caches may cause more seldom-used bytecodes to be cached, and therefore lead to larger memory footprints. For example, under the Hello benchmark, increasing the cache size from 4 kB to 8 kB results in exactly the same message count, but leads to an increase in memory footprint. This happens because a large piece of bytecodes, which is larger than 4 kB, is cached when the cache size becomes 8 kB. Unfortunately, it is never used again. If the cache size is 4 kB, it can not be cached due to its large size. Thus, the memory footprint is smaller.

RELATED WORKS

Previous studies on reducing the resource consumption of the Java platforms fall into several categories: function extraction, class file splitting, non-strict execution, procedure reordering and distributed virtual machines.

Function extraction

Assuming that an embedded device always performs a fixed set of functions, a common idea for decreasing the resource needs is to extract the code and data structures that are used to support these functions from the JVM and the Java system class libraries. The result of the extraction is then installed on the target devices. Many research projects [18–25], including those formed by Sun and HP, take advantage of this idea to implement their embedded Java platforms.

The limitation of this approach is that it makes the embedded devices hard to extend. Adding new functionality to these devices requires another round of function-extraction and installation. Furthermore, once the system vendors decide to make their devices extensible, they have no choices

[††]The performance score reported by CaffeineMark is already zero before normalization. This happens because the CACHE_NONE policy results in poor performance, and CaffeineMark records scores in integers instead of real numbers.

other than installing the whole class libraries on their devices. This is infeasible due to the shortage of resources on these devices.

EJVM does not try to limit the functionality of JVMs and system class libraries. Instead, by espousing the architecture of DJVM, EJVM stores the full set of the functions (i.e. Java classes and methods) on the server and loads them only when they are actually needed. Thus, it is more extensible and suitable for diverse applications.

Class file splitting

Sirer *et al.* [26] propose splitting out infrequently used methods of a Java class file (i.e. either an application or a system class) to form other new class files (i.e. the 'cold' classes). Due to the behavior of dynamic class loading, the cold classes may not always be loaded. As a consequence, memory consumption can be reduced. Krintz *et al.* [8] take a similar approach except that they apply the technique only to application classes.

In their approaches, the infrequently used methods are identified by profile information, which is gathered during the previous runs of the applications. In addition, bytecodes are rewritten to forward method invocations. Our approach differs from theirs in that we require neither profile information nor bytecode-rewriting.

Non-strict execution

Traditional JVMs use *strict execution* semantics. When a class is loaded, the loading thread can not continue to run until the class is completely loaded, verified and linked. Krintz *et al.* [27] propose *non-strict execution*, which reduces the invocation delay of a Java mobile program by overlapping the code transmission with execution. Basically speaking, our technique uses the idea of non-strict execution, too. However, the authors put efforts on reducing the startup latency of Java mobile programs instead of saving the memory consumption of them. In their approaches, the whole class files are eventually downloaded to the clients. In contrast, EJVM downloads only the used bytecodes.

Lee *et al.* [28] extend non-strict execution to desktop and web applications. They use procedure reordering to pack application codes more effectively and demand paging to reduce the memory consumption. These techniques require modifications to the page fault handler of the underlying operating system on the clients. In contrast to their work, we pay attention to Java applications. In addition, we cache bytecodes on the clients and evaluate different caching policies.

Procedure reordering

Chen and Leupen [29] describe an approach to decreasing the memory footprints of both applications and libraries. In their environments, procedures are split out from the original executable file and stored in another data file. During execution, these procedures are copied into the memory when they are actually used. The memory addresses of these procedures are determined in the order that they are first invoked.

In contrast to this technique, which applies on UNIX and Win32 applications, EJVM focus on Java applications. In addition, except for procedure reordering, this technique has similar effects to one of the caching policies (i.e. the CACHE_ALL policy) used in EJVM. However, we implement four different caching policies and provide performance comparisons on them.

Distributed virtual machines

Sirer *et al.* [9,10] propose some techniques to reduce the consumption of resources on the clients. First, some system services, such as verification, security enforcement and optimization, are factored out of clients to reduce the memory footprints. In addition, the Java system class libraries are placed on network servers and loaded to the clients on demand. This reduces the consumption of storage on the clients. Finally, a profile-based, class file splitting technique is used to reduce the memory footprints.

Similar to DJVM, EJVM places Java system class libraries on network servers. However, it is different in the following aspects. First, the job of converting class representations is performed on the server to reduce the client-side memory footprints further. Second, the bytecodes of Java methods are loaded when they are actually needed. Different from the class splitting technique used in DJVM, our approach needs neither profile data nor bytecode-rewriting. Third, and most importantly, we propose caching Java bytecodes on the clients to reduce the number of client–server interactions, and we provide evaluation on different caching policies.

CONCLUSIONS AND FUTURE WORK

In this paper, we propose EJVM, which aims at reducing the resource requirements on Java-enabled embedded devices. Espousing the architecture proposed by DJVM, EJVM stores all Java class files on the server to reduce the storage needs of the client devices. However, we use two novel techniques—server-side class representation conversion and on-demand bytecode loading—to reduce the client-side memory footprints further. More importantly, we maintain a cache on each client and provide performance evaluation on different caching policies. We implement EJVM by modifying a freely available JVM implementation, Kaffe. According to the experiment results, we show that it reduces the Java heap requirements by about 20–50% and achieves about 90% of the original performance.

As we described earlier, the CACHE_LRU policy may result in larger execution times even when it has fewer client–server interactions, compared to that of the CACHE_HOT policy. This is due to the high method invocation overheads of the former. In the future we shall make in-depth investigation of the relationship between method invocation frequency, network capacity and execution time. In addition, the ‘on-demand loading’ mechanism applies only to Java bytecodes now. It will be interesting if JVM components, or even the underlying libraries and operating system services, can all be loaded on demand. In the future, we will investigate its feasibility.

REFERENCES

1. Hennessy J. The future of systems research. *IEEE Computer* 1999; **32**(8):27–33.
2. Sun Microsystems Inc. Java embedded server: A white paper.
http://www.sun.com/software/embeddedserver/whitepapers/index.html.
3. Gosling J, Joy B, Steele G. *The Java Language Specification*. Addison-Wesley, 1998.
4. Franz M. The Java virtual machine: a passing fad? *IEEE Software* 1998; **15**(6):26–29.

5. Li L. Java virtual machine—present and near future. *Proceedings of Technology of Object-Oriented Languages*, 1998; 480–485.
6. Lindholm L, Yellin F. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
7. Mulchandani D. Java for embedded systems. *IEEE Internet Computing* 1998; **2**(3):30–39.
8. Krintz C, Calder B, Holzle U. Reducing transfer delay using Java class file splitting and prefetching. *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999; 276–291.
9. Sirer E, Grimm R, Bershad B, Gregory A, McDirmid S. Distributed virtual machines: A system architecture for network computing. *Proceedings of Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, August 1998.
10. Sirer E, Grimm R, Gregory A, Bershad B. Design and implementation of a distributed virtual machine for networked computers. *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP'99)*, 1999; 202–216.
11. Jensen T, Metayer D, Thorn T. Security and dynamic class loading in Java: a formalisation. *Proceedings of International Conference on Computer Languages*, 1998; 4–15.
12. Vahalia U. *Unix Internals: The New Frontiers*. Prentice Hall, October 1995.
13. Wilkinon T. Kaffe—A virtual machine to run Java code. http://www.kaffe.org/.
14. Back G. SendMail version 1.0. http://www.freecode.com/cgi-bin/viewproduct.pl?4114.
15. Calmatui V. Vasile's Java classes and applets. http://www.chez.com/vasile/vasjavaus.html.
16. PenDragon Software Corporation. The Embedded CaffeineMark.
http://www.pendragon-software.com/pendragon/cm3/embed.zip.
17. Pozo R, Miller B. Java SciMark 2.0. http://math.nist.gov/scimark2/index.html.
18. Guthery S. Java Card: Internet computing on a smart card. *IEEE Internet Computing* 1997; **1**(1):57–59.
19. Hewlett Packard. *ChaiVM Technical Summary*. October 1998. See also http:// www.chai.hp.com/.
20. Rayside D, Kontogiannis K. Extracting Java library subsets for deployment on embedded systems. *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, 1999; 102–110.
21. Sun Microsystems Inc. Embedded Java Specification. http://java.sun.com/products/embeddedjava/spec/.
22. Sun Microsystems Inc. Java 2 platform, micro edition. http://java.sun.com/j2me/.
23. Sun Microsystems Inc. PersonalJava Technology White Paper, July 1998.
See also http://java.sun.com/products/personaljava.
24. Sun Microsystems Inc. PersonalJava 1.1 Application Environment Memory Usage. *Technical Note*, July 1998. Available at http://java.sun.com/products/personaljava/MemoryUsage.html.
25. Sun Microsystems Inc. The K virtual machine: A white paper. http://java.sun.com/products/kvm/wp.
26. Sirer E, Gregory A, Bershad B. A practical approach for improving startup latency in Java applications. *Workshop on Compiler Support for Systems Software*, 1999.
27. Krintz C, Calder B, Lee H, Zom B. Overlapping execution with transfer using non-strict execution for mobile programs. *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998; 159–169.
28. Lee D, Baer J, Bershad B, Anderson T. Reducing startup latency in web and desktop applications. *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.
29. Chen J, Leupen B. Improving instruction locality with just-in-time code layout. *Proceedings of the USENIX Windows NT Workshop*, 1997.