A Practical Approach for Improving Startup Latency in Java Applications

Emin Gün Sirer, Arthur J. Gregory, Brian N. Bershad
{egs,artjg, bershad}@cs.washington.edu
Dept. of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350

February 26, 1999

Abstract

In emerging application domains, such as thin client computing and hypertext systems with embedded objects (e.g. the World Wide Web), the process of downloading application code is in the critical path of users. We observe that in these domains, compliance with existing standards and minimizing the impact on the clients is crucial. We argue that the fundamental problem for mobile code is that the units of code distribution in networked object systems, such as Java, are not suited for efficient utilization of network bottlenecks. In this paper, we propose a separate optimization step, between compilation and loading, whereby application code is restructured to more effectively use the available network bandwidth for program download. We have designed and implemented such an optimization step as a binary rewriting service for Java applets and applications. Our implementation does not require any modifications to existing Java virtual machines, compilers or clients. We have found that restructuring of Java applications can improve program startup times by up to 30%.

1. Introduction

Recent increases in network connectivity, emergence of standard Internet protocols and the evolution of the World Wide Web towards an embedded object model have fostered mobile code. Systems based on mobile code, such as virtual machines like Java [Lindholm&Yellin96], Inferno [Inferno] and OmniWare [Adl-Tabatabai et al. 96], and embedded object systems like ActiveX [Brockshmidt 94], inherently require that clients fetch applications over the network prior to their execution. In such systems, the time spent transferring application code is a significant source of user-perceived delays for typical applications, especially on networks with medium to low (less than 1 Mb/sec) bandwidths.

In order to decrease time spent loading mobile code, to speed up network applications and to reduce user-wait time, mobile code systems need to be designed to maximize the *effective bandwidth* of the network. We define the effective bandwidth to be the portion of the raw link bandwidth that is used to transfer the bits that affect the computation performed on the client. Any code that is transferred to the client, but not used in computation, contributes to parasitic loss of network capacity. Surprisingly, state of the art commercial virtual machines and mobile code systems pay little attention to optimizing network transfers and to developing binary distribution formats that maximize effective network bandwidth. For instance, the ActiveX distribution model stipulates that an entire application be packaged as a single unit of transport and shipped as a monolithic dynamically-linked library, while the user waits for the entire library to download before commencing computation. Java offers two separate modes of transport, one in which the whole application is shipped as a single unit, and another where entire object implementations are fetched at first reference to any method or field of the object. Even in the latter case, where lazy object loading filters out unused object definitions, roughly 10-30% of all downloaded code is never invoked.

Fundamentally, the problem is that the units of code distribution in these systems are not suited for efficient bandwidth utilization. The granularity at which code is transferred in all of these systems corresponds either to source-derived logical abstractions, such as classes, or artifacts of the compilation and linking process, such as object files. These coarse units of code transfer fail to capture or to imitate the dynamic execution path for an application. A client must request the complete class implementation even when it requires only a single method from the class during the entire execution. The resulting transfer of unneeded code and data incurs runtime costs, which include:

- Delays in program startup and execution from transfer of unused code,
- Increased memory consumption from storing unused code components,
- Interference with other threads of execution which share or serialize on common resources during code download.

To help solve these problems, we have developed a practical approach for restructuring mobile code in the context of the Java virtual machine. We propose a separate optimization step, between compilation and loading, whereby application code is split up into smaller transfer units, based on a profile, to more effectively use the available network bandwidth for program download. Our approach uses binary rewriting to repartition application components at method granularity such that frequently used, related code units are grouped together, while less frequently used methods are factored out into chunks that can be transferred separately and independently. This repartitioning is performed late in the software distribution chain, after the code has been released but before it has been shipped to the users. The server uses profiling to discover common application paths, and repartitions the application through binary rewriting to make these paths execute faster. The primary advantage of our approach is its practicality and simplicity. We do not require any changes to the Java virtual machine, client software, or the Java compiler, yet achieve up to 28% improvement in application performance over 28.8 Kb/sec links.

The benefits of profile-driven code restructuring include:

- Reduced transfer time: Applications download only those components that affect their execution, conserving bandwidth and reducing latency.
- Reduced resource requirements on the client: Client memory that would be dedicated to holding unused code can be used for other purposes.
- Backward compatibility: Code repartitioning using the existing class file format to transport method code requires no modification to the JVM specification, and no changes to the clients.

Fundamentally, our approach addresses the abstraction mismatch in current state of the art mobile code systems by decoupling the method implementations from the units of code transfer. By enabling applications to be repartitioned at will, a greater degree of freedom is possible in code location and transfer scheduling. While we chose to do our implementation in the context of the Java virtual machine because of its pervasiveness, our technique would apply equally well to any other mobile code scheme that provides sufficient semantic information for executable editing.

The rest of this paper is organized as follows. The next section examines related work. Section 3 describes our approach and implementation in detail. Section 4 quantifies the performance benefits of our approach, and Section 5 concludes.

2. Related Work

Previous work on decreasing application transfer times broadly spans traditional compiler optimizations, code compression, overlapped I/O and lazy loading. The common thread among all but the first of these approaches is that they require significant changes to clients to be effective, and therefore face a large implementation barrier in the current World Wide Web with millions of already deployed browsers.

Traditional compiler techniques for reducing the static size of applications have centered on instruction selection. Such optimizations often take place in the back end of a compiler, where the emitted instructions are selected to reduce the memory footprint of the instruction segment using standard techniques [Aho et al. 86,Pelegri&Graham 88]. Typically employed to generate code for embedded or other memory-constrained systems, traditional space optimizations are complementary to our approach as they reduce the initial application size.

Code compression is another field that examines the effective bandwidth of application transfers. Code compression relies on a post compilation step to represent applications using a space-efficient encoding. Recent work in this area has shown that grouping semantically related components of a binary together and compressing these groups with an adaptive compression scheme can achieve large space savings. [Clausen et al. 98] shows that factoring common code sequences can reduce code size by 30%. Slim Binaries [Franz and Kistler 97] use predictive compression on abstract syntax trees to reduce complete application size by a factor of 3. [Ernst et al. 97] shows that move to front encoding followed by Lempel-Ziv compression on separate streams for opcodes and operands can reduce code size by a factor of 4.9. [Fraser&Proebsting 98] proposes customizing instruction sets for applications using a separate, space efficient encoding for each application. All of these techniques require changes to the clients in order to support decompression, and thus are not compatible with the millions of currently deployed virtual machine clients. However, code compression techniques are complementary to our approach, and could be used in conjunction.

The most direct treatment of transmission delay in mobile code appears in [Krintz et al. 98]. The authors propose modifying Java's execution semantics in order to overlap code transmission with execution. Such an approach offers increased application performance by allowing I/O operations of a thread to be scheduled concurrently with the computation of the same thread. This technique requires relaxing the execution semantics of Java, such that methods can be executed before the class file they belong to has been transferred in full. The authors report simulation results that this scheme can achieve 25-40% performance improvement. This technique is complementary to ours, but requires significant changes to the Java virtual machine. Krintz et al. are exploring a separate code repartitioning strategy substantially similar to ours.

Profile guided code positioning has previously been examined in the general context of program optimizations [Pettis&Hansen 90]. The authors propose an optimization based on a call graph profile, where the dynamic call graph is annotated with weights based on call frequency, and heavily weighted callees are located close to their callers. This colocation policy is suited for whole program optimization, and may increase the transmission delays in mobile code. [Chen&Leupen 97] provides an adaptation of this scheme, where code positioning is driven just in time by a first-use graph derived from the execution of a program instead of a profile. Their simulation study indicates that just in time code layout can halve the memory footprint of applications. [Lee et al. 99] propose combining code reordering and demand paging to improve the startup of ActiveX applications. The authors show that code reordering via binary rewriting and demand fetching of pages through the memory fault handler can improve the startup times of x86 applications by up to 58%. Since x86 binaries lack semantic information, binary rewriting is more difficult, and code movement is performed at the instruction level. Modifications to the memory fault handler require changing at least the browser, and possibly the operating system, on the clients.

3. Approach

Our goal is to decrease transmission delays and to increase effective bandwidth for mobile Java code. In addition, we restrict our implementation to operate on unmodified Java virtual machines, to be transparent to the application programmer, and to be amenable to automation. Without these properties, new optimization techniques would have difficulty in being adopted into the millions of currently deployed existing Java virtual machines and hundreds of thousands of Java applications.

Our approach is based on repartitioning Java applications into modules that utilize network bandwidth more effectively. Existing Java virtual machines use two separate techniques for distributing application code, both of which limit the effective bandwidth. The first technique uses *classfiles* to store and distribute code [Lindholm&Yellin96]. Each class file contains the interface and implementation for a single object type. The client JVM requests classes lazily as they are required during execution, and thus avoids transferring unused classes, but may download unused code within a class. The second technique for distributing code bundles all classfiles for an application into a single unit. The client then pays up-front to download all of the code for the application in one transmission. Execution commences after all of the bundled classes are linked into the JVM. In a 1997 survey we performed of publicly available, indexed applets on the Internet, the second technique was less popular by a few orders of magnitude. We attribute this to the high user-interface latency necessitated by this approach. Overall, both of these techniques suffer from the same problems; namely, the units of code transfer do not correspond to the execution path of an application, and indiscriminately bundle unrelated code into a single transfer unit.

We address these shortcomings through profile-driven code repartitioning. Essentially, we decouple method implementations from class files, thereby enabling distribution of code in units that best utilize available network bandwidth. We specifically focus on optimizing application startup time, as it is the most visible and limiting source of delays in the user-interface.

Our implementation consists of two phases. The first phase generates a first-use graph by collecting a profile of the methods used by the application. This profile is generated automatically either by inserting tracing code into method entry points or by placing every method of a class in a separate classfile and tracing the client request stream on the server side. We picked the latter approach, because security constraints in clients or in network proxies may prohibit the application from transmitting the profile back to the server. The server initially places every method in a separate class using the splitting technique described below, and then simply concatenates the client request stream to generate a time-sorted graph of used methods. This graph identifies the unused methods in a class, which are partitioned out in the second phase of optimization.

Figure 1 illustrates how code partitioning is performed in our implementation. In order to decrease startup latency, we identify all unused methods based on the profile, and place their implementation in separate classfiles. The original methods are replaced with stubs that forward method invocations to appropriate separated implementations. More specifically, an unused method **M** in class **C** is relocated into a new **final** class **C$M**. Its signature is modified, if necessary, to make it a **static** method and possibly to take an extra first argument corresponding to the old **this** pointer. **C.M** is replaced with a proxy that simply invokes **C$M.M** with the original arguments, including the object instance pointer. The access modifiers for fields in class **C** are modified to grant access to **C$M**. Name conflicts for newly created classes are minimized, though not completely eliminated, by using the inner-class naming convention in Java. New names are derived from the original class name using a special separator, and Java convention dictates that classes introduced dynamically into the system from other packages will not share the same prefix.

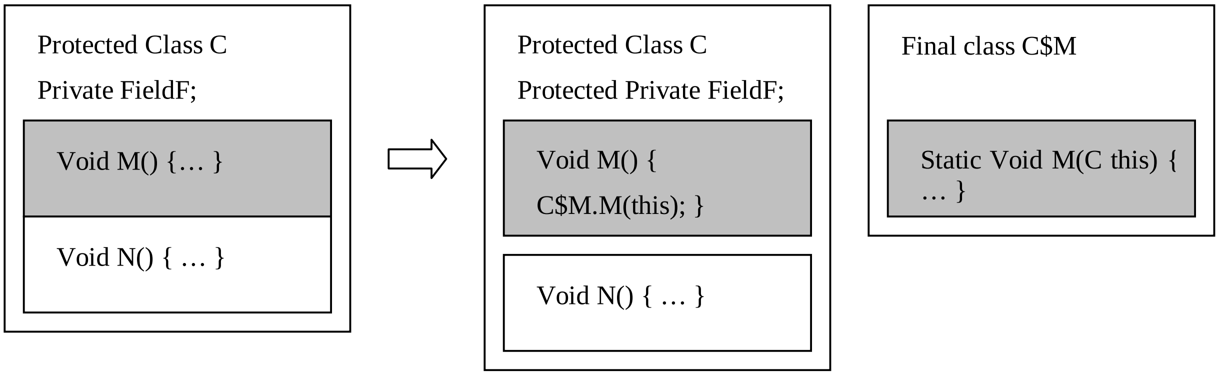

Figure 1. An illustration of code repartitioning. Unused method M is factored out into a separate classfile, reducing the code size of the original class. While the figure illustrates the transformation using Java source code, the actual implementation operates directly on Java bytecodes.

Code repartitioning schemes are highly dependent on the verifier implementation found on the clients, because the verifier determines the class fetching policy. Verifiers may fetch classes that make up an application eagerly or lazily. In eager verification, the transitive closure of all classes referenced by the initial class is downloaded onto the client at startup, and the whole set is verified all at once. Lazy verification, on the other hand, enables the client to defer the transfer of classes by delaying portions of verification to execution time. In the example in Figure 1, an eager verifier would fetch both the class **C** and **C$M** at startup, negating any performance gains from code repartitioning. A lazy verifier, on the other hand, will note that the class **C$M** is not needed to prove the correctness of class **C** unless method **M** is actually invoked, and will therefore avoid transferring **C$M**. The JVM specification explicitly leaves the style of verification, and the timing of verification exceptions, undefined to enable the transfer optimizations supported by lazy verification. Consequently, JVMs deployed in latency and bandwidth-limited settings as well as JVMs that cannot make assumptions about the quality of their network connections, such as all major browsers, exclusively use lazy verification. Our code-partitioning scheme takes advantage of the ubiquity of lazy verification by assuming the existence of a verifier that does not eagerly load classes.

Our approach conforms to all existing Java virtual machine semantics, and therefore does not require any changes to clients. Seamless integration with existing JVMs poses some challenges, however, especially with respect to verification. Since we repartition applications at method granularity, type-safety checking is not affected by our repartitioning scheme. However, for method implementations in segregated classes to access the fields of the original class, we downgrade field access modifiers from private to package access. Allowing other classes in the package to access all the fields of the class may at first seem like a potential security hole. However, this change does not alter the access controls on the program because it is applied only to classes that are not visible outside a package (i.e. they are private or protected, or the program does not allow any extensions to be installed), and because other classes within the package are verified prior to any partitioning. An alternative, more powerful, approach is to combine code repartitioning with distributed virtual machines [Sirer et al. 98]. Distributed virtual machines enable a verification service in a proxy to track the intended, as opposed to the modified, state of the access control flags for each client, and impose restrictions accordingly.

Structuring the code repartitioning service as a binary rewriting service provides three advantages over traditional, compiler based optimization schemes. Namely, binary rewriting can be applied late in the code distribution chain, is transparent to clients and applications, and does not require programmer assistance. Further, rewriting Java virtual machine applications at the bytecode level is particularly powerful, because high-level semantic information about the program, such as method boundaries, is visible within the bytecode. The structure of Java classfiles, where all jumps are PC-relative and all code is re-locatable, makes binary rewriting easy and fast to perform.

4. Performance Analysis

We examine the effects of profile-driven code repartitioning on six interactive applications. These benchmarks consist of three traditional standalone applications, and three applets, designed to operate in the context of a web browser. The benchmark programs were selected because they embody significant functionality and are based primarily on a user-interface for their operation. We focus on interactive applications instead of SPEC-like benchmarks, because performance delays are hard to mask and thus are immediately visible to the user in an interactive domain. Table 1 summarizes the applications we used for performance analysis.

Table 1. List of Benchmarks

Benchmark	Type	Description
HotJava v1.1.5	Application	Internet browser
Java Studio v1.0	Application	Graphical environment for Java development
Java Work Shop v2.0	Application	Programming environment for Java development.
CQ	Applet	Calendar queue sorting algorithm demo
Net Charts	Applet	Chart displaying applet
Animated UI	Applet	Animated user interface

We first examine the behavior of these applications during their startup. We define startup to be the time between the initial invocation of the program and the entry into the application event loop, when the application can first start responding to user activity. Table 2 shows the total code size for the application, the total size of code loaded to reach the main event loop, and the reductions in size after code repartitioning.

Table 2. Benchmark Code Sizes (in bytes)

Benchmark	Total Code Size	Size of Startup Set	Size of Repartitioned Startup Set	Improvement
HotJava	1,768,144	818,762	713,578	13%
Java Studio	9,574,809	2,140,932	1,592,742	26%
Java Work Shop	5,954,310	1,241,010	885,527	29%
CQ	94,748	80,663	67,620	16%
Net Charts	1,794,306	406,166	319,390	21%
Animated UI	271,890	175,646	136,674	22%

The savings above results from Java's use of classes as the transfer unit for distributing code. For example, Java Work Shop contains over 400K bytes of code bundled in its startup classes that is not used to initialize the application. The client is required to wait for these bytes to come over the network, delaying the startup of the program unnecessarily. Reducing the startup set not only will improve startup speed, but should also result in a reduction of memory requirements on the client that correspond to the space improvement shown in the last column. The amount of memory saved through code repartitioning can be bigger, in terms of absolute bytes, than shown if the client translates class files into an expanded internal or native form.

We next examine the effect code repartitioning has on the running times of mobile applications. To quantify the effects of profile-driven code repartitioning, we ran each of the benchmarks through different simulated network bandwidths. Our evaluations were run on a 200 MHz Intel Pentium Pro platform with 128Mb of memory using Sun JDK v1.1.6 with just-in-time compilation. The network delay that each class incurs was simulated by a pause in each class' initialization method. The length of the pause was determined by dividing the length of the class file by the bandwidth available from the network. We simulated a slow network instead of using an actual slow link in order to provide the appearance of persistent connections. The Sun JDK uses separate TCP/IP connections for each request, and TCP/IP does not sufficiently utilize the available bandwidth for short connections, thereby inflating the performance improvement for our scheme. We compare our simulated numbers against a real slow link later in this section and show that they are conservative.

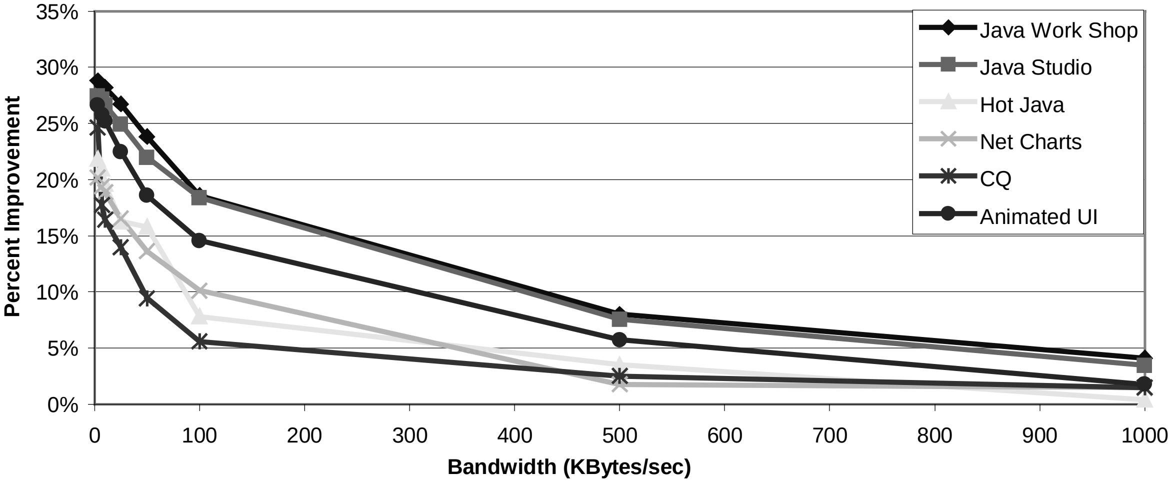

Chart 1. Percent improvement in startup time as a function of bandwidth.

Chart 1 summarizes the percent improvement of code restructuring on the startup times of our benchmark applications. Eliminating the overhead from transferring unused code in class files decreases the startup delay observed for the client. The improvements are modest, up to 4%, for a relatively high bandwidth of 1MB per second, comparable to Ethernet speeds. For moderate to low bandwidths, the gains are markedly higher, for an average of 25% for 28.8K baud speeds.

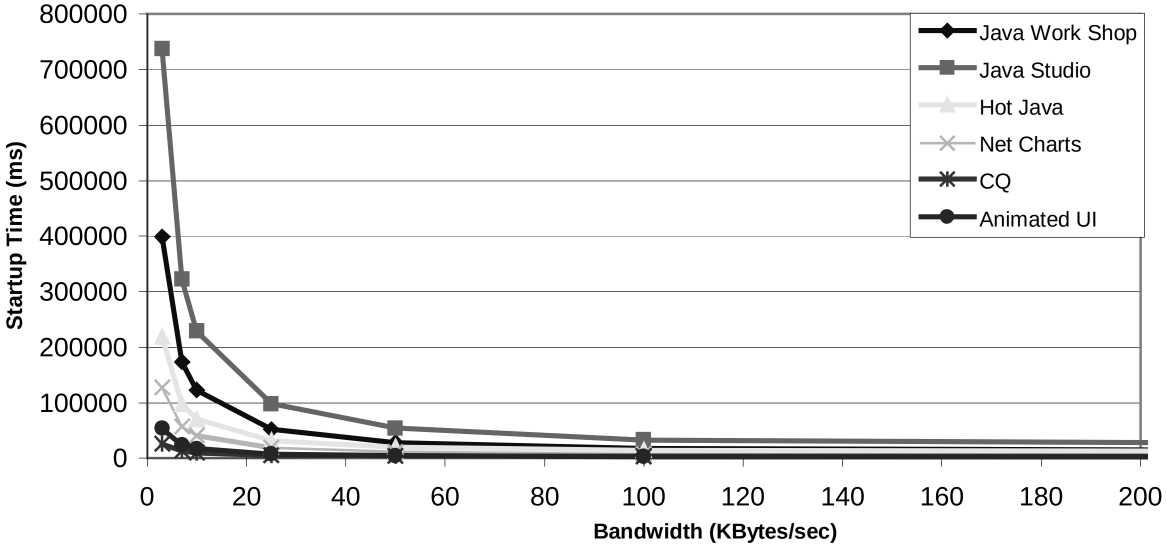

Chart 2. Absolute time spent for startup as a function of bandwidth.

We next examine the absolute startup times for applications as a function of available bandwidth. Chart 2 shows that the relationship between network bandwidth and benchmark startup latency is non-linear. This relationship follows a simple model dependent on a constant computational time needed to run the code (T), the size of the bytes needed for startup (S), and the bandwidth of the network (BW):

$$\text{Startup Latency} = T + \frac{S}{BW}$$

This simplified model fits our data within 10% for the bandwidth range shown. The model indicates that as available bandwidth decreases, the transfer delay quickly becomes the bottleneck and dwarfs the computation time, making any latency masking from available coarse grain parallelism negligible. Most of our benchmarks reach this point between 50K to 100K bytes per second, which correspond roughly to DSL speeds, with a rapid increase in startup latency thereafter.

Applying our model to calculate percent improvement, we find that the following equation relates the original size of the application (OrigSize), the size of the repartitioned application (NewSize), the compute time needed to run the code (T), and the bandwidth of the network (BW):

$$\text{Percent Improvement} = \frac{\text{OrigSize} - \text{NewSize}}{(BW * T) + \text{OrigSize}}$$

All factors in the equation are constant for a given benchmark except bandwidth. Since bandwidth is magnified by the time the application spends computing, applications with shorter think times will exhibit greater speedup through code repartitioning.

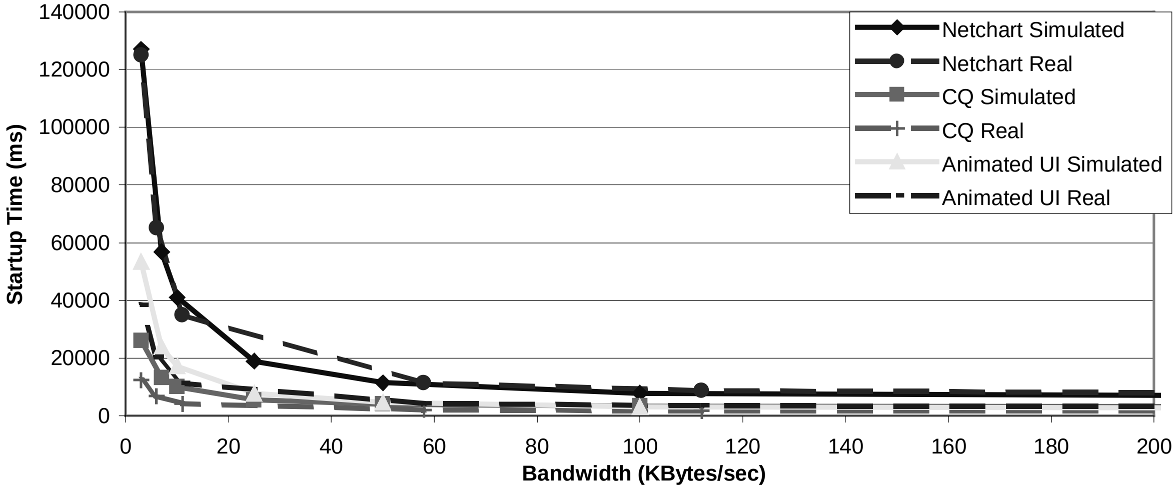

Chart 3. Comparison of simulated delay to actual network delay.

Finally, we compare our simulated network to a real network. We set up a 200 MHz Pentium Pro processor running FreeBSD with 128 MB of memory as a web server. The client is a 200 MHz Pentium Pro with the same amount of memory. The FreeBSD packet filter mechanism was used to limit the rate at which the server transmits pages. The actual bandwidth observed on the wire was verified using bulk file transfer through the ttcp utility. We examine the behavior of our applets across different bandwidths. The industrial applications override the ClassLoader facility in Java, and are not amenable to automatically loading across an HTTP connection. Chart 3 shows that the results on the simulated network are generally a conservative estimate of the results on the real network.

5. Issues

While repartitioning applications can improve execution times and reduce memory requirements in the clients, it may also pose some problems. First, the increased activity between the client and server may increase the load on the server. The time to create, service and tear down connections for automatically generated class files may, in aggregate, negatively impact the load on the server. Second, small transfers over TCP/IP, with its slow-start and

MTU discovery algorithms, are unlikely to achieve the maximum possible throughput, and may be affected disproportionately badly by network congestion and loss. Persistent connections [Fielding et al. 97] mitigate both of these problems by obviating the need to create and tear-down extra connections, and by aggregating sufficient traffic for the network layer to adapt to network conditions. The extra fetch overhead per request, which consists of request processing on the server side and a round-trip latency from the client's perspective, may still pose problems.

6. Conclusions

The state of the art for mobile code uses units of code transfer that do not match the needs of applications. By profiling and repartitioning application code, application startup time can be improved significantly, especially over low-bandwidth connections. We provide a practical approach for profiling and repartitioning in the context of the Java virtual machine that does not require any modifications to the clients, and yet achieves up to 30% reductions in startup time for interactive applications.

7. Acknowledgements

We would like to thank Neal Cardwell, Robert Grimm, Eric Hoffman and Dennis Lee for many valuable discussions. Dennis Lee also helped us with our experimental setup.

8. References

[Adl-Tabatabai et al. 96] Adl-Tabatabai, A., Langdale, G., Lucco, S. and Wahbe, R. "Efficient and Language-Independent Mobile Programs." In *Conference on Programming Language Design and Implementation*, May, 1996, p. 127-136.

[Aho et al. 86] Aho, A., Sethi, R. and Ullman, J. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, New York, 1986.

[Brockshmidt 94] Brockschmidt, K. Inside OLE. Microsoft Press.

[Chen&Leupen 97] Chen, J. B. and Leupen, B. D. D. "Improving Instruction Locality with Just-In-Time Code Layout." 1997 USENIX Windows NT Workshop, August 1997.

[Clausen et al. 98] Clausen, L. R., Schultz, U. P., Consel, C. and Muller, G. "Java Bytecode Compression for Embedded Systems." IRISA Technical Report-1213, December 1998.

[Custer 93] Custer, H. Inside Windows NT. Microsoft Press, 1993.

[Dean et al. 97] Dean, D., Felten, E. W., Wallach, D. S. and Belfanz, D. "Java Security: Web Browsers and Beyond." In Internet Besieged: Countering Cyberspace Scofflaws, D. E. Denning and P. J. Denning, eds. ACM Press, October 1997.

[Ernst et al. 97] Ernst, J., Fraser, C. W., Lucco, S. and Proebsting, T. A. "Code Compression." SIGPLAN '97 Conference on Programming Language Design and Implementation, June 1997.

[Fielding et al. 97] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T. RFC-2068 Hypertext Transfer Protocol -- HTTP/1.1. Request for Comments, Internet Engineering Task Force, January 1997.

[Franz and Kistler 97] Franz, M. and Kistler, T. "Slim Binaries." Communications of the ACM, 40:12, 87-94; December 1997.

[Fraser&Proebsting 98] Fraser, C. W. and Proebsting, T. A. "Custom Instruction Sets for Code Compression." http://www.research.microsoft.com/~toddpro/papers/pldi2.ps

[Gosling&Yellin 96] Gosling, J. and Yellin, F. The Java Application Programming Interface, Volumes 1 & 2. Addison-Wesley, 1996.

[IBMVM86] IBM Corporation. Virtual Machine/System Product Application Development Guide, Release 5. Endicott, New York, 1986.

[Inferno] Lucent Technologies. Inferno. http://inferno.bell-labs.com/inferno/

[Krintz et al. 98] Krintz, C., Calder, B., Lee, H. P. and Zorn B. G. "Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs." In Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[Lee et al. 99] Lee, D., Baer, J. L., Bershad, B. N. and Anderson, T. "Reducing Startup Latency in Web and Desktop Applications." Submitted for review.

[Lindholm&Yellin96] Lindholm, T. and Yellin, F. The Java Virtual Machine Specification. Addison-Wesley, 1996.

[Pelegri&Graham 88] Pelegri-Llopart, E. and Graham, S. L. “Code Generation for Expression Trees: An Application of BURS Theory.” in Conf. Record of the 15th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages, San Diego, CA, Jan. 1988, pp. 294--308.

[Pettis&Hansen 90] Pettis, K. and Hansen, R. C. Profile guided code positioning. In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), pages 16-27, White Plains, New York, 20-22 June 1990.

[Sirer et al. 98] Sirer, E. G., Grimm, R., Bershad, B. N., Gregory, A. J. and McDirmid, S. “Distributed Virtual Machines: A System Architecture for Network Computing.” European SIGOPS, September 1998.

[Stata&Abadi 98] Stata, R. and Abadi, M. “A Type System for Java Bytecode Subroutines.” In *Proceedings of the 25th Symposium on Principles of Programming Languages*, January 1998, p. 149—160.