Dynamic Distribution of Java Applications

Gita Alagbhand and David Gnabasik
University of Colorado at Denver and Health Sciences Center
Department of Computer Science and Engineering
Campus Box 109, P.O. Box 173364, Denver CO 80217-3364, USA
E-mail: Gita.Alaghband@cudenver.edu P:303-556-2940 F:303-556-8369
E-mail: DavidGnabasik@comcast.net P:303-994-2740 F:303-617-7877

Keywords: Java, component streams, class loader, mobile devices

Received: April 10, 2007

This paper describes a streaming mechanism that distributes Java class bytecode streams to a client from a database server. The class server uses a 1st-order Markov probability model to effectively predict the client's next class request. Experimental results demonstrate that class prediction can deliver a class cache hit ratio of up to 54% using a modest cache size of 64kb on the client, whereas a 16kb cache delivers a hit ratio of 37%. The model is designed to mitigate the distribution and deployment problems of monolithic application software and is useful for applications running on resource-constrained, mobile computing devices.

Povzetek: Članek opisuje postopek dinamičnega porazdeljevanja aplikacij v Javi.

1 Introduction and problem description

Dynamic component streams can address several client software distribution and deployment issues, including the automated update of applications from a centralized software repository, as well as the delivery of application streams to resource-constrained mobile devices. We submit that the process of application deployment can take advantage of the dynamic linking and class loading [3] mechanisms in Java compilers to support a distributed component model that does not require the streaming of large monolithic applications to these devices. Java class hierarchies are stored as Java bytecode streams into database rows by a class author. A streaming mechanism then transmits a virtual application to a client's process space from a class server. Our class server uses a 1st-order Markov transition probability model to effectively predict the client's next class request based upon the statistical analysis of historical application runs. This permits for the effective overlapping of client-server communication with client processing. The Java linking model and class file format are suited for managing this problem since the class file is analyzed and stored as the unit of compilation [2]. We manage and optimize system performance by:

- decomposing and storing class hierarchies into a database;
- defining an effective asynchronous streaming component model;
- retrieving the most probable class subgraphs to be activated next;
- effectively managing limited available client memory.

This paper is organized as follows. In section 2, we describe each architectural component of the system and the underlying theory. Section 3 outlines its design and implementation. Section 4 describes our experimental measurements. Section 5 discusses related work, and section 6 presents our conclusions and summarizes the contributions of this paper.

2 System architecture

2.1 Class authoring and client access roles

The role of the class author is to correctly compile and store Java components or entire applications as binary streams of Java bytecode into a database. By doing so, the class authoring interface encapsulates much of the complexity of class relations and their distribution. The primary function of the interface is to scan each class for referenced classes and to store this list into the database as well. The authoring interface also manages the following information:

- appropriate class / application authorization and security measures;
- any class digital signatures or certificates;
- what foundation set of Java classes are required locally on the client;
- the set of client deployment preferences that indicate how and when the classes in a particular application trace should be updated.

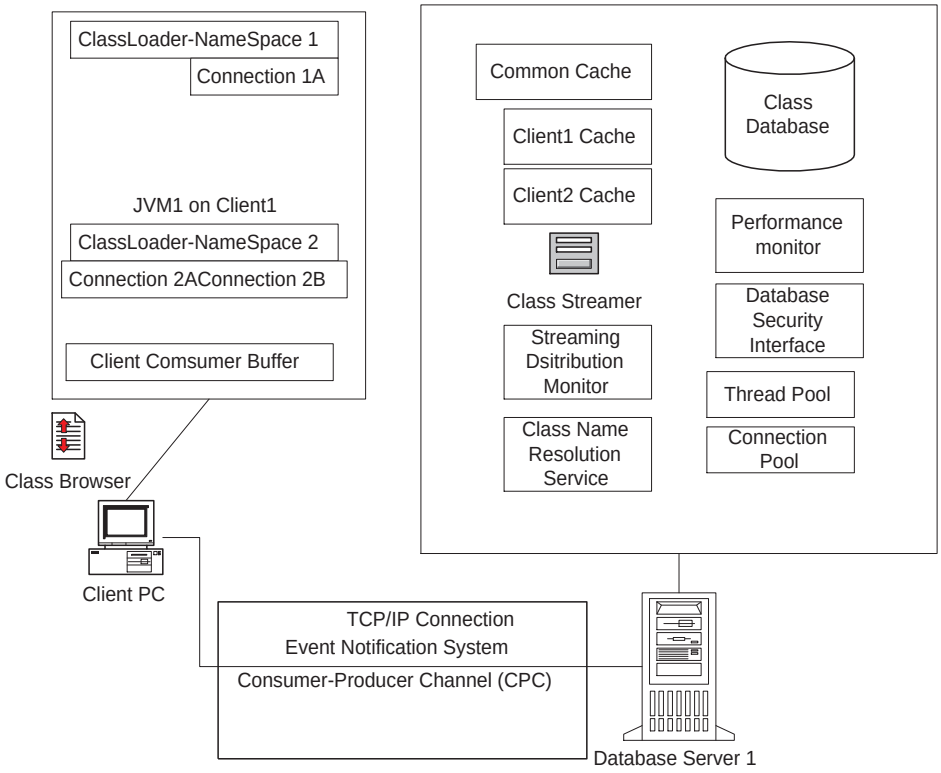

Figure 1: Overall architectural schema.

Clients use a class browser or the Java Naming Directory Interface (JNDI) to request a particular application through a well-known URL address. The client request establishes a consumer-producer channel with the class server. Once the client is authorized, the server retrieves and transmits a custom class loader to the client who loads it into his Java virtual machine (JVM). It is this custom class loader which initially loads an application's main class and which makes any further class requests on behalf of the application. No other changes to existing Java classes are necessary. Figure 1 diagrams our system architecture.

2.2 The class loading and linking process

Class loaders are responsible for importing binary data that define a running program's classes and interfaces. The JVM's flexible class loader architecture provides for dynamically extended applications. Linking a class stream into a JVM's run time state is divided into three steps: verification, preparation, and resolution [4]. Verification ensures that the stream is properly formed, preparation allocates memory needed by the stream, and resolution transforms symbolic class references into direct machine references for the sake of run time speed and efficiency. During the class activation process, the JVM must give the impression that it loads classes as late as possible, a process called *lazy activation* [3]. This on-demand activation process permits the transmission of individual Java classes to a client by a class server.

2.3 Class transmission mechanism

The server retrieves a requested set of classes, or class subgraphs, according to a class transition probability model. This model, which forms the basis for prediction, is described in section 1.4. Based on this model, the class server attempts to prefetch and transmit the next set of expected classes while the client is busy executing. Even though some classes are prefetched and delivered that may not be loaded by the client into his JVM, this network traffic occurs while the client is busy processing the current class, in effect, overlapping operations. Any successful access to a prefetched class is a measurable performance gain. Overall system performance is maintained by an efficient class prefetching algorithm, client and server caching, a simple database schema, and judicious threading.

When the client attempts to load another class, the custom class loader searches for classes in the following order: the client JVM, the client's class cache (CCC), the client's standard Java class libraries, and the class server. If the class is found in the client's cache, it is decompressed by the JCL or built-in classes of the JVM, loaded into the running JVM, and removed from the cache. If the requested class is not in the CCC, the JCL requests the class from the class server. The class is retrieved from the database along with its previously parsed list of symbolically referenced classes, which was generated when the class was inserted into the database by the class author. The list of classes is compressed and streamed to the CCC in the order in which a JVM internally resolves all of its referenced

classes. The amount of data transmitted is limited by the size of the client's cache, which is managed by the client's JCL.

Classes from the Java libraries are loaded by the native, primordial class loader. Classes in the class cache are loaded by the custom object class loader. The different class loaders are related by a policy called *delegation*. For each class it loads, the JVM keeps track of which class loader, whether primordial or object, loaded the class [4]. Our model uses a separately threaded class loader derived from the SecureClassLoader Java class to load all subsequent classes. SecureClassLoader extends ClassLoader with additional support for defining classes with an associated code source and permissions. The protection domain in the model is the class server database itself, which has permissions assigned to it as a code source. It also uses the *parent-delegation model* introduced in Java 1.2 to determine which class loader actually loads a given class. The rule is that the JVM uses the same class loader that loaded the referencing or calling class to load the referenced or called class. The custom class loader is the first class the client receives in an application stream.

A Java class is initialized in a certain order. In stage I, all the class's superclasses are recursively initialized, then the class itself, and then its inner classes, followed by the class's static variables. Once the class is initialized for active use, then the class's private variables are initialized followed by all classes referenced in any constructors. These references are necessary to initialize a class; therefore their invocation probability is 1. Class initialization is described in detail in the next section. Stage I classes are always transmitted to the client in their own package.

Stage II classes include any method argument classes, method return types, and all classes referenced in any method. These referenced classes are conditionally invoked by an application. Stage II classes are retrieved from the database, ordered by invocation probability, and transmitted to the client in a separate package. This strategy allows the class server to act as a predictive look-ahead class feeder for the client. Figure 2 diagrams these separate stages.

2.4 Transition probability model and class invocation

The delivery effectiveness ratio H is defined as the number of requests satisfied by a particular cache divided by the total number of requests during an entire application trace. Given that C_C is the number of class requests satisfied by the client cache, S_C is the number of class requests that had to be satisfied by the server, and C_R is the total number of class requests made by the client's class loader, such that $C_C + S_C = C_R$, then $H_C = C_C/C_R$. The effectiveness of prediction is then the ratio of class requests satisfied by the CCC to requests satisfied by the class server. The modeling question becomes: How can the server more effectively predict which classes to stream to the client?

The proposed model generates a set of invocation probabilities, one for each context $class_j$ that invokes a particular $class_i$. Given a class, the model enumerates all the invocation probabilities of the classes that it symbolically references. Since the probability of transmitting a specific class depends upon the class that calls it, the model establishes a conditional probability vector of $P(class_i|class_j) = P(class_i \cap class_j)/P(class_j)$ probability values, where $class_j$ is the context class for $class_i$. Each $class_i$ instance may be invoked by different classes; hence each class has multiple $class_j$ context classes. Each class also maintains a total count of class invocations per context class. These class values are accumulated during the execution of an application. Class invocation prediction implies distributing both the globally most frequently accessed classes as well as the class most likely to be invoked next at any point in the program. Good prediction means transmitting those classes that are most likely to be consumed by the client. Given a particular context class, stage I classes are always transmitted, but stage II classes are transmitted according to their invocation probabilities.

To illustrate, Table 1 lists the set of classes that are invoked by the class TelnetDriver over 20 program runs of a Telnet application. Stage I classes are DialerAccess, Plugin, Common and ReturnFocusRequest because TelnetDriver always invokes them. The invocation probabilities for the other classes are lower because they were conditionally invoked depending upon the flow of execution. Clearly, TelnetDriver is a stage I class because it was also invoked the same number of times the application was run. Each invocation probability is calculated as the number of invocations divided by the number of program runs (e.g., 20). For example, P(invoking Class OnlineStatusListener within Telnet) $= 2/20 = 0.1$. The 103 total classes and interfaces in this Telnet application range in size from 124 to 26158 bytes, with an average size of 2083 bytes for all classes.

2.5 Markov chain modeling

A 1st-order, finite state, probabilistic Markov model is proposed because a Markov model is suitable to the local dependencies embedded in Java class invocation structure, and the finite-state machine model accurately reflects the necessary and unique set of state transitions that occurs in an application trace. Since classes are conditionally invoked in an application trace, the model is able to characterize their invocations by probability values. The Markov model has an additional advantage in that it reveals an application's locality of reference, since "the performance of demand-driven caching depends on the locality of reference exhibited by the stream of requests made to the cache" [13](abstract). Vanichpun et al [13] further claim that "the two main contributors to locality of reference are temporal correlations in the streams of requests and the popularity distribution of requested objects", which are both accounted for by the model. The "temporal correlations" cor-

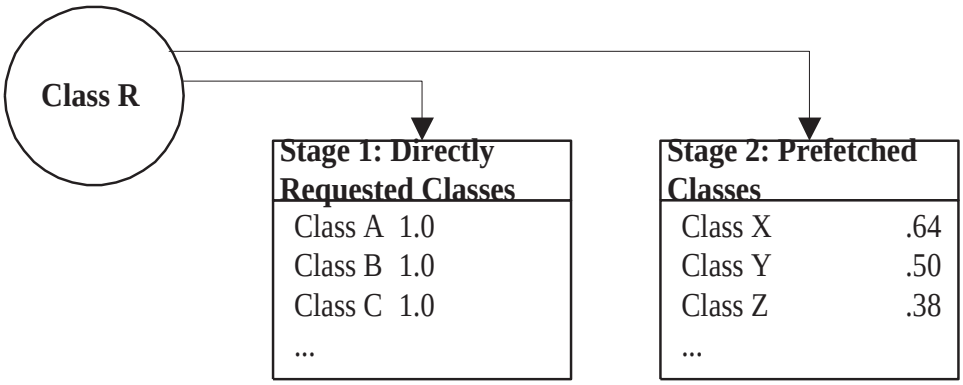

Figure 2: Invocation probability stages.

Table 1: Conditional probability vector (CPV) for class TelnetDriver.

Invoked Class Name	Invocation Probability	Number of Invocations
DialerAccess	1.0	20
Plugin	1.0	20
Common	1.0	20
OnlineStatusListener	.10	2
FocusStatusListener	.10	2
SocketRequest	.80	16
VisualTransferPlugin	.66	13
ReturnFocusRequest	1.0	20

respond to the sequential flow of class invocations and the "popularity distribution" corresponds directly to the probability of invocation for a particular class.

Markov chains can dynamically model these class invocation patterns found in applications. A discrete Markov chain model is defined as the tuple $< S, A, \lambda >$ where S corresponds to the state space, A_{ij} is a matrix representing the invocation probabilities from one state to another, and λ is the initial probability distribution of the states in $S(0)$. In our model, S represents every possible application trace, A_{ij} represents the invocation probabilities for each class, and λ is the initial class invocation distribution retrieved from the database. If the vector s_c denotes the probability vector for all the subsequent possible class invocations during execution of a specific class c, where $c \in S$, then the overall set of expected transition state values for class c is $\hat{s}_c(j) = s_c(i)A_{ij}$. The A matrix is recalculated during each application trace and stored in the database. The class request mechanism simply selects the largest probability values from the vector $\hat{s}_c$ either until a threshold probability value is reached or their cumulative class sizes are greater than the client's class cache size. The conditional probability vector for a class directly supports the calculation of $\hat{s}_c$.

For example, table 2 calculates $\hat{s}_c(j$ =TelnetDriver$\geq$ .75) for $s_c(i$ =TelnetDriver= .50). It is these combined stage I and II classes that are actually delivered to the client because the invocation probability for $\hat{s}_c(j)$ is $\geq .75$. $\hat{s}_c(i = .50)$ is the probability that the application would invoke TelnetDriver in the first place.

Given the semi-hierarchical structure of nearly all applications, this Markov chain model is not *irreducible*, that it is possible to get to any state from any other state. Some states would be *transient*, such that given a starting state there is a non-zero probability that the application would never return to that particular state. Most states would instead be *recurrent* or *persistent*, that at some point in time the application would return to that state. Most application states also avoid the Markov property of *absorbing*, where it is impossible to leave a particular state. Since our Markov chain model is not irreducible, there is no guarantee that the model provides a steady-state or equilibrium distribution. In practice, however, the probability transition matrix quickly approached a set of relatively stable values.

2.6 Threaded queue representation

Although we do not present a detailed queuing model in this paper, it is important to note that our system can also be represented by *queuing circuits* because of Java's stringent class loading requirements and the way the system is architected around several queue components. Relevant queuing centers are the *client request queue* (CRQ) and the more complicated *class server queue* (CSQ), each of which are distinguished by their own average service times. See sections 2.4 and 2.5 for complete descriptions of the queuing mechanism. The CSQ is accessed by two threads per client: one thread for handling class requests to the server and the

Table 2: Expected transition state values s_c for class TelnetDriver.

Class Transition	Transition Prob.
DialerAccess	$1.0 \times .50 = 0.50$
Plugin	$1.0 \times .50 = 0.50$
Common	$1.0 \times .50 = 0.50$
ReturnFocusRequest	$1.0 \times .50 = 0.50$
SocketRequest	$.80 \times .50 = 0.40$

second thread for receiving predicted, prefetched classes. The third client thread accesses locally referenced classes. The same queuing components are managed by each thread including a common cache, a client delivery cache, and a database fetch component. Figure 3 diagrams the fundamental queuing centers of the system. Operating system and database-specific queues are not included for the sake of simplicity.

Following Gunther [9], our system is characterized by a first-in, first-out (FIFO) service policy which assumes an exponential service time distribution for both queues, the custom and native class loaders. Under FIFO, the service time can only depend upon the queue length at each queuing center. The system is considered open because the server queuing center can access a possibly infinite number of elements or classes, even though only a limited and indeterminate number of classes are actually invoked in an application trace. Highleyman [11] argues that an open queuing center model is a reasonably accurate approximation if N is at least 10 times larger than the average queue length, which is indeed the case. The two queuing streams are also *separable* and *mergeable*. The streams are separable because it is possible to evaluate the performance measurements of the complete set of queuing centers as though each of the centers were evaluated separately in isolation. The streams are mergeable because the performance of the entire system is then built by combining the separate solutions. These queuing characteristics establish two different servers in an open, overlapping configuration with an infinite population, which we describe as a **M/M/2/FIFO** delay queue model.

3 Design and implementation

3.1 Optimizing overall system performance

Our system architecture attempts to minimize overall network traffic by delivering only the immediately needed classes of an application trace, since the model claims that overall delivery throughput is increased by incorporating and limiting the Markov probability model to the 1st-order when predicting the client's next class request. Network delivery time is reduced by writing the class subgraph into a single *Java ARchive* (JAR) or ZIP package for transmission.

The major problems to resolve in order to maintain adequate system performance are (along with

– decomposing and storing class hierarchies into a database where each Java class file is stored as precompiled Java bytecode in a database row;

– defining an effective asynchronous streaming component model that overlaps execution with network communication as implemented in the CSQ and CRQ queueing centers;

– retrieving the most probable class subgraphs to be activated next from the class server which uses a 1st-order Markov probability model to effectively predict the client's next class request to reduce invocation latency;

– caching the most frequently invoked bytecodes at the client while prefetching and caching class bytescodes at the server to reduce database access delay;

– managing class elements as discrete database rows, which allows for database access optimization;

– avoiding slow file system accesses and disk paging by storing the class bytecodes in database rows;

– effectively managing limited available client memory by allocating a minimum client cache;

– the use of judicious I/O threading.

Other design goals include:

– not modifying the client's virtual machine executable;

– not allowing the class server to manage client state beyond minimal client authentication, authorization and initialization, which reduces the overall complexity of the application;

– expending time and effort at the class-producing or -authoring stage instead of the class-consuming stage.

When non-duplicated classes are inserted into the CCC, they are associated with the class that invoked them. If a class is extracted from the CCC for activation, the discard policy marks the list of explicitly associated classes for discard, too. These classes are either sequentially pushed out of the CCC to make room for new classes or they are extracted for activation. Note that any inherited or parent classes or interfaces have themselves already been extracted from the CCC since a class's parents must be completely activated before the class itself. Since only the

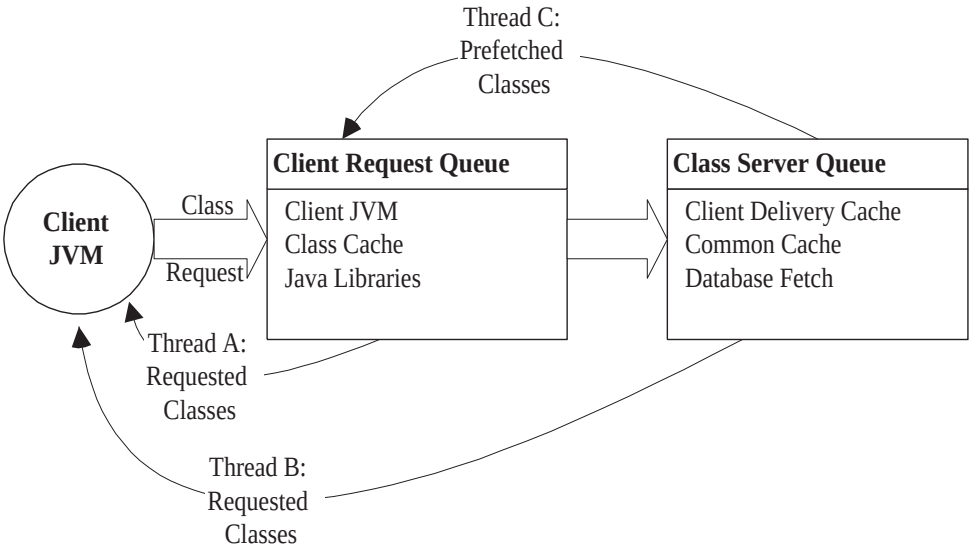

Figure 3: Queuing request centers.

client removes classes from the CCC, there is no need for the client to inform the server that the client has unloaded a class. The client JCL is responsible for cache overflows.

Storing Java bytecodes instead of source text offers several advantages: the bytecode does not have to be recompiled for each client; unlike machine code, the Java bytecode supports a heterogeneous computing environment. In addition, the source analysis and compilation is performed by several powerful class authors instead of by the clients themselves.

3.2 Server caching strategy

The server implements the following caching strategy. A common cache is maintained by the server which contains classes or resources that have recently been used by two or more active clients, as shown in Figure 1. A smaller, client-specific delivery cache is also established on the server for each client which receives the anticipated set of classes referenced in the class subgraph. These caches are informed and populated by the application's class invocation probabilities.

Using these previously computed probabilities for the requested class, the database keys of the requested class and its subgraph are fetched from the class server database as shown in Figure 3. Then the class fetching algorithm works as follows:

```
while requested classes not in client-specific
buffer do
  THREAD C:
    receive prefetched classes and buffer them
    locally
  while requested classes not in common buffer
    concurrently do
     THREAD A:
       fetch, buffer, transmit requested class
       stream
     THREAD B:
       fetch, order, buffer, transmit subgraph
       stream
  endwhile
endwhile
```

The Java environment also supports the effective use of multiple threads for asynchronous events. As shown above, each connected client allocates at least three threads: two to process class requests and one to receive class data. The server allocates and manages a separate thread from a thread pool per connected client. This coarse-grained parallelism, coupled with the use of dedicated socket ports, permits for effective overlapping of client-server communication with client processing. Note that a client does not establish a direct connection to the server database.

Determining the proper size of the client cache is of critical importance. This size must balance the critical constraints of limited client memory with the fact that class prediction may be incorrect, which therefore may transmit unneeded classes to the client. The viability of class prediction is demonstrated only if these constraints are effectively reconciled; i.e., if the delivery effectiveness ratio H can be increased.

4 Performance measurements

4.1 Experimental setup

In the experiments, a 600MHz computer simulates a resource-constrained client with a specified cache size. It is connected to a 1.7GHz server computer over a 10Mb TCP/IP network through a router. The server pro-

gram executes within a single Java JVM instance, version 1.4.1.02.b06. The representative Telnet client application is designed to exercise the two different stages of classes to be delivered. The application has a reasonably rich class hierarchy including superclasses, inner classes, static and private variables, class variables and constructors.

The goal of this paper is to present the feasibility and effectiveness of the proposed methods. Experiments with more applications and benchmarks running on more suitable hardware, say one of the newer mobile handsets, will be the subject of future publications.

4.2 Verifying the effectiveness of a common cache

A performance-experiment was conducted in order to test the effectiveness of a common cache on the server (as defined in the Server caching strategy section) under the assumption of multiple client JVMs. A common class package was transmitted to a set of 4, 16 and 32 clients all running under their own JVM instance on the same client computer. Figure 4 shows the effect of averaged package instantiation time with and without the common 256k server cache operating. The performance effect of using the 256k common cache is not significant until 32 clients are being serviced simultaneously, at which point the average package instantiation time is cut in half from 15 seconds to 7 seconds. We will use a common cache of 256K for the next set of experiments.

4.3 Experimental results

After exercising the client source application under multiple and various traces using different class method calls, the delivery effectiveness ratio H is calculated as a function of client cache size. Table 3 presents the class transfer data gathered for a complete application run using a 32 kb cache. In the table, the effectiveness ratio is calculated as $H_C = C_C/C_R$.

The server records the stream size it delivers to the client. Previous experiments [7] had demonstrated that activating the class prediction mechanism delivered twice as many bytes to the client as opposed to simply delivering every class stream on-demand. The separately threaded client class loader, threads B and C, records the total stream transfer time it took to receive requested data as the amount of time the class loader blocks on the server, not including actual class instantiation. However, the two-stage packaging mechanism that transmits the two separate class package streams does not double the amount of transfer time because the predicted classes are transmitted asynchronously to the client. The additional network traffic occurs while the client is busy processing the current class, in effect, overlapping operations. By convention, the custom class loader itself and the application's main class are counted in S_C. We do not count those classes that are already accessible within the client's JVM.

Recall that multiple stage I classes are usually required to initialize any particular class for active use, which simply activates the class in the client JVM. Depending upon the application's structural class hierarchy, these implicitly loaded classes and interfaces may be satisfied by either the CCC or the server. In either case, $C_R = 52$ reflects the actual number of classes the application trace instantiates, both implicit and explicitly named. Without prediction, the server has to deliver 27 out of 52 classes (52%) to the application stream while the client blocks. Note that the custom class loader makes a fewer number of explicit class requests than what is actually delivered to the CCC, and so does not produce a reliable number for comparison.

With prediction, the number of classes S_C satisfied by the server, where the client is forced to request, block and wait for a class stream, is 4 less because the server has anticipated their use, then prefetched and transmitted them to the CCC. Now the server delivers only 23 out of 82 classes (28%) to the application stream responding to a blocked request. The total client class cache count, $C_R = 82$, is larger because the server has transmitted additional, predicted stage II classes. However, the prediction process increases C_C, the number of classes fetched from the local client cache. We claim that any successful local access by the client to a prefetched class is a significant performance gain. The end benefit is that the client effectiveness ratio H_C, has increased from 0.48 to 0.72, an improvement of 50%.

Figure 5 illustrates the averaged instantiation times and H_C ratios, prediction over no prediction, under four different cache sizes: 8k, 16k, 32k, and 64k. A client cache size of 16k accommodates most class requests relatively efficiently. A client cache size of 32k nearly approximates immediate class activation, as measured by averaged time of class activation. The averaged class cache hit ratios approached 54% with a cache size of 64k, revealing the effective limits of the prediction mechanism as well as demonstrating the effective parallelization of class delivery with program execution. For a modest increase in client memory, say 32k, a large application can be effectively delivered to an otherwise resource-constrained client.

To show the effect of communication overlap of the required additional data transfer to the client's overall execution time, we compare estimates of the total application trace execution time t_e with and without prediction. Because we are interested in client I/O-bound applications, it is fair to factor out the common duration of in-memory class execution time and to concentrate on the I/O parameter of total blocking time t_b, which includes total class transmission time. We ask at what client class cache size, if any, does the ratio of total blocking time over trace execution time ever become less with prediction than without prediction? Tables 4 and 5 present the following averaged results for 1 client at various client class cache sizes. Again, the total size of the stream transmitted was 84746 compressed bytes without prediction and 169104 bytes with prediction.

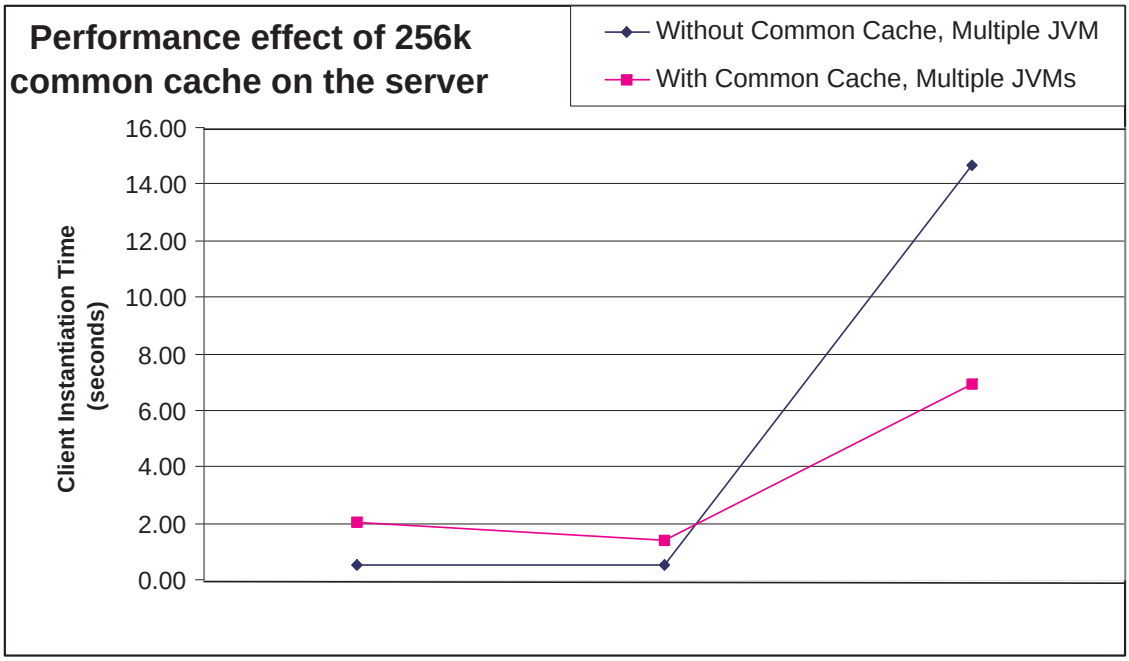

Figure 4: Common class package instantiation for 4, 16 and 32 clients.

Table 3: Effectiveness of class prediction for 1 client.

32kb Client Cache: 1 Active Client	**No Prediction**	**No Prediction**	**P/NP**
.	Stage I Only	Plus stage II	.
Total stream size (compressed bytes)	84746	169104	1.995
Total stream transfer time (secs) = t_b	14.2	19.3	1.36
Classes satisfied by server = S_C	27	23	0.85
Classes satisfied by client cache = C_C	25	59	2.36
Total client class cache count = C_R	52	82	1.58
Effectiveness ratio $H_C = C_C/C_R$	0.48	0.72	1.50

Table 4: Total execution t_e and blocking times t_b (sec) without prediction.

CCC Size	**8k**	**16k**	**32k**	**64k**
t_b	23.7	17.5	15.2	15.0
t_e	38.1	34.1	31.8	31.2
t_b/t_e	62%	51%	48%	48%

Table 5: Total execution t_e and blocking times t_b (sec) with prediction.

CCC Size	**8k**	**16k**	**32k**	**64k**
t_b	29.5	18.9	12.1	12.0
t_e	39.9	31.5	28.1	28.6
t_b/t_e	74%	60%	43%	42%

As shown in Figure 6, we conclude that prediction becomes more effective than not in terms of reducing total client execution time at a client cache size of 32k, even though twice as many class bytes are delivered to the client.

5 Related work

Arnold's recent survey [14] of adaptive optimization in virtual machines presents the three major developments of adaptive optimization technology in virtual machines over the last thirty years: 1) selective optimization, 2) feedback-directed code generation, and 3) other feedback-directed optimizations in order to improve VM performance. The survey discusses the benefits and drawbacks of just-in-time compilers, synchronized thread management, dynamic class loading, native code caches, class splitting and dynamic class caching. Most of these techniques exploit some form of temporal locality to be effective.

Krintz's work [5] proposes Java class file splitting and prefetching optimizations as an effective latency-hiding optimization for mobile programs which also does not require

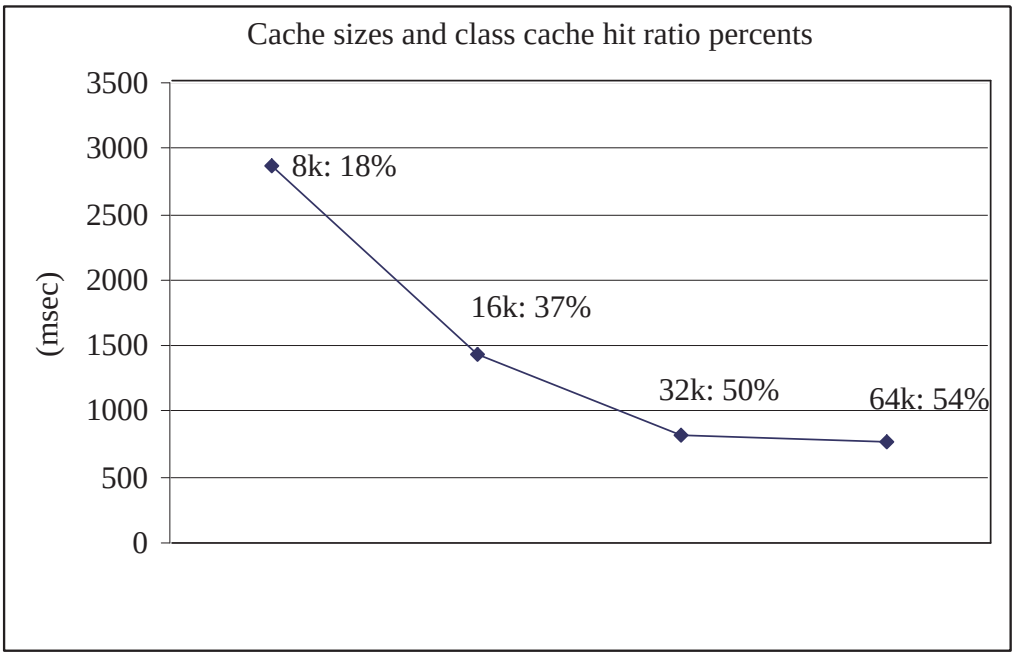

Figure 5: Client cache size vs. average time of class instantiation.

redefining the Java Virtual Machine specification. The combination of techniques reduces the overall transfer delay encountered during a mobile program's execution by 25% to 30% on average. However, Krintz's method requires inserting prefetch statements into the Java source code as well as compilation using a binary modification tool.

Bartels et al [1] describe an adaptive fault-based prefetching scheme based on a one-level Markov net that achieved high prediction accuracy for some classes of scientific applications.

Thiebaut's work [10] with synthetic traces demonstrated the structural importance of the locality of reference in real programs and its impact on hit and miss ratios.

Chilimbi [12] proposes a data reference framework and an exploitable locality abstraction called *hot data streams* as a quantitative basis for understanding and optimizing data reference locality.

Our general approach is indebted to Patterson's seminal paper [6] on informed aggressive disk prefetching and caching (*TIP*), where it is shown that prefetching can not only mask latency with asynchrony, by overlapping I/O with computation, but also expose parallelism for the sake of greater throughput.

6 Conclusions

Our paper describes a streaming mechanism that distributes Java class bytecode streams to a client from a class server using a 1st-order Markov probability model to predict the client's next class request. The experimental results demonstrate that a simple class prediction mechanism significantly reduces client blocking using a dedicated client cache size of 32k. At the cost of the client receiving twice as many bytes over the network and a modestly-larger cache, the client is able to execute rich and complex applications not otherwise possible. We acknowledge that this extra network processing is clearly a concern for power-sensitive devices.

Using the Java architecture requires writing and delivering a custom class loader for mobile devices. Sun's Java 2 Micro Edition (J2ME) CLDC, targeted to cell phones, requires 128K to 512K total memory available with less than 256K ROM/Flash and less than 256K RAM (JSR-000030). However, it currently does not support user-defined class loaders or native method access. The mobile device manufacturers would themselves have to embed modified class loaders into these devices in order to handle the proposed streaming mechanism.

To summarize, this paper makes the following contributions:

- The Java linking process permits dynamic class loading that delivers application streams to clients.
- A 1st-order Markov class invocation probability model effectively predicts the client's next class request in order to reduce invocation latency and transfer delay by overlapping program execution with network communication. Prediction is worth the effort.
- The current implementation does not require the modification of the Java Virtual Machine definition. However, due to the lack of a dynamic class loading mechanism, the current version of J2ME/CLDC would require significant modification.

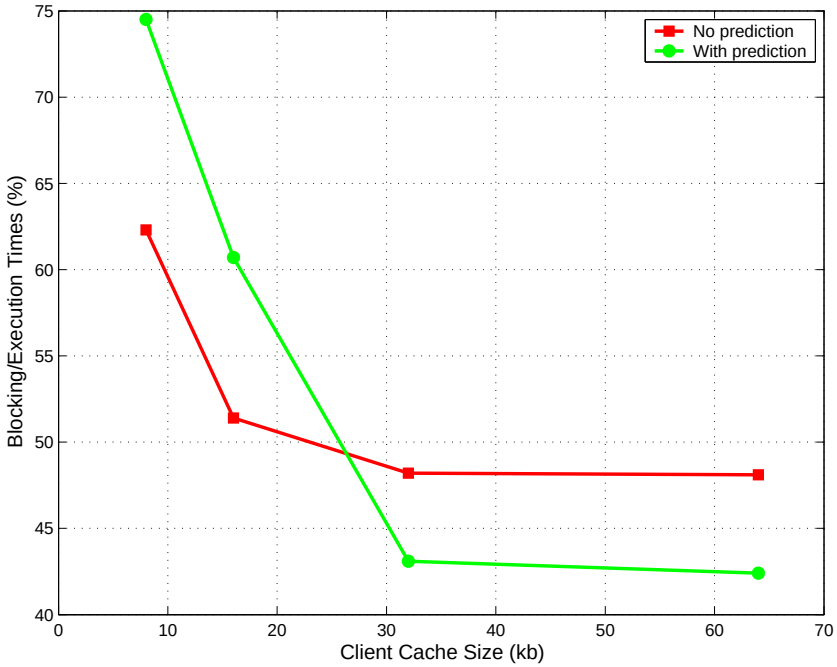

Figure 6: Execution and blocking time percent vs client cache size.

– We also plan to carry out the methods developed in this work with various benchmarks on appropriate hardware, such as mobile handsets.

References

[1] Bartels, G., Karlin, A., Levy, H., Voelker, G.; Anderson, D., Chase, J. (1999). *Potentials and Limitations of Fault-Based Markov Prefetching for Virtual Memory Pages*, ACM SIGMETRICS Performance Evaluation Review, Volume 27 , Issue 1, June 1999.

[2] Gosling, J., B. Joy, G. Steele, and G. Brach. (2000). *Java Language Specification, 2nd Ed*. Boston: Addison Wesley.

[3] Liang, Sheng and Bracha, Gilad. (1998). *Dynamic Class Loading in the Java Virtual Machine*, in Proceedings of OOPSLA '98, published as ACM SIGPLAN Notices, Volume 33, Number 10, October 1998, pages 36-44.

[4] Venners, Bill. (1999). *Inside the Java 2 Virtual Machine, 2nd Ed.*, New York: McGraw-Hill Companies.

[5] Krintz, C., Calder, B., and Holzle, U. (1999). *Reducing Transfer Delay Using Java Class File Splitting and Prefetching*, UCSD Technical Report, CS99-615, March 1999.

[6] Patterson, R.H. Gibson, G.A., Ginting, E., Stodolsky, D. and Zelenka, R. (1995). *Informed Prefetching and Caching*, J. Proc. of the 15th Symposium of Operating Systems Principles, Copper Mountain Resort, CO, December 3-6, 1995, pp. 79-95.

[7] Alagbhand, G., Gnabasik, D. (2004). *Streaming Java Applications to Mobile Computing Devices*, Proceeding of the 2004 International Conference on Wireless Networks, Monte Carlo Resort, Las Vegas, Nevada, June 21-24, 2004, pp. 637-643.

[8] Sun Microsystems. (1998). *Java Object Serialization Specification*. Available online: http://java.sun.com/products/jdk/1.2/docs/guide/serialization/spec/serialTOC.doc.html

[9] Gunther, N., (1998). *The Practical Performance Analyst: Performance-By-Design Techniques for Distributed Systems*, New York: McGraw-Hill Companies.

[10] Thiebaut, D., Wolf, J.L., and Stone, H.S. (1992). *Synthetic Traces for Trace-Driven Simulation of Cache Memories*. IEEE Trans. Computers. 41(4):388-410.

[11] Highleyman, W.H., (1989). *Performance Analysis of Transaction Systems*. Englewood Cliffs, N.J.: Prentice-Hall

[12] Chilimbi, Trishul. (2001). *Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality*. Microsoft Research, One Microsoft Way, Redmond, WA

[13] Vanichpun, S., Makowski, A.M., (2004). *Comparing strength of locality of reference - Popularity, ma-*

jorization, and some folk theorems. To appear in Performance Evaluation and Planning Methods for the Next Generation Internet, A. Girard, B. Sanso and F.J. Vazquez-Abad, Editors, Kluwer Academic Press.

[14] Arnold, M., Fink, S.J., Grove, D., Hind, M., Sweeney, P.F. (2004).*A Survey of Adaptive Optimization in Virtual Machines*. IBM Research Report, RC23143 (W0312-097).