Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs

Chandra Krintz Brad Calder

Dept. of Computer Science and Engineering
University of California, San Diego
{ckrintz,calder}@cs.ucsd.edu

Han Bok Lee Benjamin G. Zorn

Dept. of Computer Science
University of Colorado
{han.lee,zorn}@cs.colorado.edu

Abstract

In order to execute a program on a remote computer, it must first be transferred over a network. This transmission incurs the overhead of network latency before execution can begin. This latency can vary greatly depending upon the size of the program, where it is located (e.g., on a local network or across the Internet), and the bandwidth available to retrieve the program. Existing technologies, like Java, require that a file be fully transferred before it can start executing. For large files and low bandwidth lines, this delay can be significant.

In this paper we propose and evaluate a non-strict form of mobile program execution. A mobile program *is any program that is transferred to a different machine and executed. The goal of non-strict execution is to overlap execution with transfer, allowing the program to start executing as soon as possible. Non-strict execution allows a procedure in the program to start executing as soon as its code and data have transferred. To enable this technology, we examine several techniques for rearranging procedures and reorganizing the data inside Java class files. Our results show that non-strict execution decreases the initial transfer delay between 31% and 56% on average, with an average reduction in overall execution time between 25% and 40%.*

1 Introduction

The computational paradigm of the Internet is such that applications are retrieved from remote sites and processed locally or are transfered for remote execution. These applications are referred to as *mobile programs*. The performance a mobile program achieves is determined by processor speeds and the rate at which the application can transfer to the remote site. As the gap between processor and network speeds continues to widen, mechanisms to compensate for network latency are required to maintain acceptable performance of mobile programs.

Performance is most commonly measured by overall program execution time. Additionally, in a mobile environment, performance is measured by invocation latency. *Invocation latency* is the time from application invocation to when execution of the program actually begins. Research has shown that invocation latency is crucial in how users view the performance of an application. Early work investigated the effect of time-sharing systems on productivity (e.g., see [6]), and concluded, among other things, that increased system response time disrupted user thought processes. More recent work focuses on the effect of transmitting video over the Web, and contrasts the negative impact of unpredictable Web latency with earlier systems in which latency was more predictable [15].

This paper appeared in the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, October, 1998.

Network transfer delays can result in significant invocation latency and the communication delay can dominate execution time of mobile applications. To amortize the cost of network transfer to the execution site, code execution should occur concurrently with (i.e., overlap) code and data transfer. However, existing mobile execution facilities such as those provided by the Java programming environment [11] typically enforce *strict execution* semantics as part of their runtime systems. *Strict execution* requires a program and all of its potentially accessible data to fully transfer before execution can begin. The advantage of this execution paradigm is that it enables secure interpretation and straightforward linking and verification. Unfortunately, strictness prevents overlap of execution with network transfer, and little can be done to reduce the cost of transfer latency.

In this paper we investigate the efficacy of *non-strict execution*: Procedures execute at the remote site without the restriction that the files the procedures are contained in transfer prior to execution. To examine the potential of non-strict execution, we use Java as our execution environment because of its widespread use for Internet computing. We show substantial performance benefits (in terms of reduced invocation latency and decreased program execution time when the transfer time is included). We also identify file restructuring techniques that take advantage of non-strict execution. In doing so, we propose compiler-based and profile-based approaches for partitioning and restructuring programs so that strictness is enforced at the method level only. In addition, we investigate non-strict execution in the presence of two transfer methodologies: *parallel file transfer*, a technique that schedules the simultaneous transfer of multiple class files, and *interleaved file transfer*, in which the transfer of data and methods are interleaved among the different class files in a program.

We first summarize related work in Section 2. Section 3 describes our approach to non-strict execution. Section 4 describes the procedure reordering algorithms we considered for non-strict execution. Section 5 describes the two transfer methodologies we examine for using non-strict execution. In Section 6 we describe the methodology used to perform our studies, and in Section 7 we show the benefits of non-strict execution with program restructuring. We summarize our contributions in Section 8.

2 Related Work

While widespread computing with mobile programs is a relatively new phenomenon, there are three areas of research that are closely related to our work: code compression, program restructuring, and continuous compilation. In this section, we discuss each area in turn, and compare existing work with our own.

2.1 Code Compression

For non-strict execution, we advocate maximizing the overlap between execution and network transfer as a way to reduce the overhead introduced by network delay (i.e., latency tolerance). An alternative and complementary approach is to reduce the quantity of data transferred through compression (i.e., latency avoidance). Several approaches to compression have been proposed to reduce network delay in mobile execution environments, and we discuss those here.

Ernst et al. [7] describe an executable representation called BRISC that is comparable in size to gzipped x86 executables and can be interpreted without decompression. The group describes a second format, which they call the wire-format, that compresses the size of executables by almost a factor of five (gzip typically reduces the code size by a factor of two to three). Both of these approaches are directed at reducing the size of the actual code, and do not attempt to compress the associated data.

Franz describes a format called *slim binaries* in which programs are represented in a high-level tree-structured intermediate format, and compressed for transmission across a wire [8]. The compression factor with slim binaries is comparable to that reported by Ernst et al., however Franz' reports results for compression of entire executables and not just code segments. Additional work on code compression includes [9, 18, 27].

Our work is distinct from, and complementary to, code compression techniques as the approaches mentioned do not attempt to reorganize the code and data that is being compressed. Our methods will benefit from compression, just as the positive effects of these compression techniques can be enhanced by reorganization, restructuring, and non-strict execution.

2.2 Program Restructuring

Classical program restructuring work attempts to improve program performance by increasing program locality. Historically, because virtual memory misses have always incurred a very high cost, programs are reorganized to increase the temporal locality of their code. For example, if procedures are referenced at approximately the same time, then they are placed on the same page. Attempts to understand and exploit reference patterns of code and data have resulted in such algorithms as least recently used page replacement (e.g., see [2, 13]) and Denning's working set model [5].

More recently, as memory sizes have increased, interest has shifted to improving both temporal and spatial locality for all levels of memory. Many software techniques have been developed for improving instruction cache performance. Techniques such as basic block re-ordering [14, 22], function grouping [10, 12, 14, 22], reordering based on control structure [20], and reordering of system code [25] have all been shown to significantly improve instruction cache performance. The increasing latency of second-level caches means that expensive cache usage patterns, such as ping-ponging between code laid out on the same cache line, can have dramatic effects on program performance.

Most of the prior work in code reordering has focussed on improving overall program locality, since the physical memory or cache was a limited resource with a constrained size. As a result, replacement policies and the cost of replacement play a significant role in the algorithms. Our problem, that of reorganizing mobile programs for wire transfer, is substantially different, in that we are only concerned with the first use of an object and are (at least in the current work) not concerned with subsequent uses.

2.3 Continuous Compilation

The final area of related research is that of continuous compilation [23]. Continuous compilation is a method for improving the performance of Just–In–Time compilers. Just–In–Time compilation produces executable code just prior to when it is executed. Continuous compilation combines interpretation of code with compilation in order to reduce the overall execution time of the program as well as to compile the program for future execution: overlapping interpretation with compilation. Our project is similar in that our techniques can also provide a mechanism for improved remote Just–In–Time compilation performance; but we overlap transfer with execution.

3 Non-Strict Execution for Java

To study the benefits of non-strict execution we require a platform for mobile programs. Our methods can potentially be applied to any mobile program system (e.g., Omniware [1] or ActiveX [4]), but we choose to use Java because of its widespread use and built-in support for mobile programs. In this section we describe non-strict execution and its implications for Java.

Java is an object-oriented language that enables remote execution by providing a platform independent executable representation and object-oriented modularity through an abstraction called a *class file*. The class file contains information about the Java classes represented in an architecture–independent form, called bytecodes, which may either be compiled for a particular architecture or interpreted. The class file information enables various run time verification and security techniques to be implemented by an interpreter, describes global data structures, and provides access information through a set of access flags. The only requirement for execution of Java class files is the presence of the Java Virtual Machine (JVM) bytecode interpreter or Just-In-Time compiler. These are necessary since the bytecodes must be translated into architecture specific machine instructions for execution.

In existing implementations of the Java interpreter and the JVM, each Java class is in a separate file. Figure 1 provides a visual of the class files from an arbitrary application. There are two classes, A and B, containing three and two methods, respectively. The classes also contain global data (pictured) and local data (contained within the methods). The order of the methods in the class file is equivalent to that in the Java source file. We will use this example throughout the paper to clarify selected points.

The Java interpreter is invoked by providing the file name of the class to be executed. All classes required by the program are commonly loaded when the interpreter is invoked. In addition, in a mobile program context, all classes loaded are verified as part of the linking process. The JVM allows dynamic loading of classes but the entire class file must be loaded in order for any method within the class to execute. Such strict execution of classes imposes a major performance limitation. With existing network transfer delays, the invocation latency can be significant, and the communication delay can dominate the execution time of a Java application.

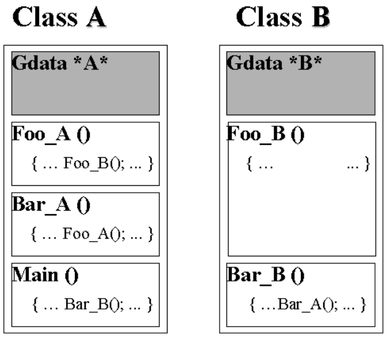

Figure 1: Example Java Application: Class A contains global data and three procedures, Foo_A, Bar_A, and Main; Class B contains global data and two procedures, Foo_B, and Bar_B.

When a program is executed remotely, the first class file to execute (the class containing the main routine: Class A in Figure 1) is transferred to the remote site. In some implementations, other non-local class files are not requested to transfer until the program executes code that uses those classes. In other cases, all of the class files in the package or application may be transferred concurrently in a non-specified order. The two restrictions to the JVM model of execution are that (1) each class cannot start to execute until the full class file has been transferred, and (2) the class file must transfer to completion once it has started to transfer.

To decrease the invocation latency and to allow idle cycles to be used by overlapping execution with transfer, we propose using a non-strict transfer and execution model at the procedure (method) level. In this model, a class file is partitioned into two parts: global data needed to begin execution of any method within the class, and code (with local data) needed to begin execution of each procedure. Transferring the global data first allows the JVM to incrementally perform the linking process (described below) when overlapping the execution with the transfer. In our non-strict version of Java, inside each class file a method delimiter is placed after each procedure and its data. The *method delimiter* is used to indicate that the data and procedure have transferred, so that the procedure can begin execution. During the execution of a Java program, if a procedure is called but it or its data have not completed transferring, the program is stalled until the procedure's delimiter has transferred.

3.1 Non-Strict Java Virtual Machine Linking

The goal of this paper is to examine the performance improvement achievable from non-strict execution. For this approach to be viable, we need to address the effect of non-strict execution on linking of class files for Java. In this subsection, we describe at a high level the JVM changes that are necessary to support non-strict execution.

To allow non-strict execution to work using Java, the JVM linking of class files is performed incrementally. The linking of a Java program is the process of taking a Java binary (expressed as bytecodes) and putting it into the runtime state of the JVM by performing (1) *verification*, (2) *preparation*, and (3) *resolution* on the bytecode stream. Verification is the process of verifying the structure of a Java class file to know it is safe to execute. Preparation involves allocation of static storage and any internal JVM data structures. For initialization, Java executes any class variable initializers (class constructors) in textual order. Resolution is the process of checking a symbolic reference before it is used. Symbolic references are usually replaced with direct references during this phase. While verification and preparation can be performed once the global data is transferred, resolution can be performed lazily as procedures are invoked.

3.1.1 Verification for Non-Strict Java

The JVM has five steps in verifying a class file as described in [19]. The first two verify the structure of the class file and the global data. Since the global data is the first to transfer, this verification can proceed as soon as the global data is transferred. Step 3 is performed as each procedure is transferred, and Step 4 is performed as each procedure is executed. Dependence analysis is performed at each step to determine whether each class can be trusted for interpretation. When classes are dynamically loaded, cross class dependences are resolved during linking of these classes.

In a non-strict execution environment, incremental cross class dependency resolution is extended to the procedure level. Within procedures, dependence analysis remains as it is. Interprocedural dependence analysis is performed as methods are loaded and verified.

The other responsibility of the verification process is to provide security for the underlying system in which the linked program is executed. The whole verification process can be avoided by providing a means of trust between the compiler that produced the Java class file and the JVM interpreter. For example, with security mechanisms for digital signatures [3] or software fault isolation [26], the verification step can be skipped completely.

4 First Use Procedure Reordering

To capitalize on the benefit achieved from non-strict execution, it is necessary to predict the execution order of the procedures in the program, and then to place them in the class file in this predicted order. This order is different from prior code reordering research. We need to predict a *First Use* ordering, that is, the order in which the methods are first executed. We examine the performance of two reordering techniques. The first approach uses static program estimation to predict the order of invocation for procedures, and the second approach uses first-use profiling to create a profile indicating the order of invocation.

In this study, we model non-strict execution of procedures because of the modularity that procedures provide. Non-strict execution can be performed at the basic block level; however, preliminary experiments show that checking for a delimiter at the conclusion of each basic block incurs additional overhead with little added benefit. Code reuse in object-oriented languages, like Java, results in small method sizes. The applications we use for our simulations support this claim. With method level support for non-strict execution, large procedures can still benefit by using the compiler to break the procedure up into smaller procedures. In this paper, we do not perform any procedure splitting since the procedures in our test programs are of reasonable size.

4.1 Static First Use Estimation

The first technique we examine for first-use order prediction uses a static call graph. To obtain this ordering, we construct a basic block control flow graph for each procedure with inter-procedural edges between the basic blocks at call and return sites. The predicted static invocation ordering is derived from a modified depth first search (DFS) of this control flow graph, using a few simple heuristics to guide the search.

A flow graph is created to keep track of the number of loops and static instructions for each path of the graph. When generating the first-use ordering, we give priority to paths with loops on them, predicting that the program will execute them first. When processing a forward non-loop branch, first-use prediction follows the path that contains the greatest number of static loops. In addition, looping implies code reuse, and thus increases the opportunity for overlap of execution with transfer. The order in which procedures are first encountered during static traversal of the flow graph determines the first-use transfer order for the procedures. When processing conditional branches inside of a loop, the first-use traversal traverses all the basic blocks inside the loop searching for procedure calls, before continuing on to the loop-exit basic blocks.

To process all the basic blocks inside of a loop before continuing on, first-use prediction uses a stack data structure and pushes a pair, (x,y), onto the stack when processing a loop-exit or back edge from a conditional branch. The pair consists of the unique basic block ID and the ID of the loop-header basic block. These pairs are place holders, which allow us to continue traversing the loop-exit edges once all the basic blocks within the loop have been processed. When all the inner-basic blocks have been traversed, and control has returned to the loop-header basic block, the algorithm continues the psuedo DFS on the loop-exit edges by popping the pairs off the top of the stack. Upon termination of the modified DFS algorithm, the static traversal of the procedures determines their first-use order, and the methods are reordered within each class file to match this ordering.

4.2 Profile Guided First Use Estimation

The second method we include for code reordering uses profile information to determine the first-use ordering of procedures. A first-use profile is generated by keeping track of the order in which procedures are invoked during a program's execution using a particular input. All procedures that are not executed are given a first-use ordering during placement using the static approach described above.

Since a program's execution path may be input dependent, we attempt to choose adequate sets of inputs in order to provide an execution path that is similar to most of the possible inputs. In our results section we include the efficacy of using the training (initial) input set to determine the profile for the first use ordering for executions on both the training input set (perfect prediction) and an arbitrary, more robust test input set.

We now use our example application to clarify static and profile driven first use estimation. In the simple application, there are no control flow constructs except for the procedure calls and thus, the static and profile driven method in this case produce the same first use call graph, pictured in Figure 2. We then restructure the class files using this first use information and reorder the class files as pictured in Figure 3. In the next section, we use this new ordering to determine when we transfer the classes over the network for execution.

5 Methods of Transfer

In this section we discuss transfer techniques. The techniques we present are not the only techniques that can be used with non-strict execution to transfer files; they are two possible examples of transfer methodologies that can take advantage of non-strict execution and program restructuring.

- Parallel File Transfer - multiple class files transfer independently and in parallel sharing fixed bandwidth.

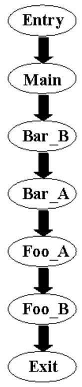

Figure 2: First Use Call Graph: The first use call graph is generated using the static first use estimation or the profile. It is then used to determine the order in which the files transfer for remote execution.

- Interleaved File Transfer - all methods in the application are interleaved and transferred as a single virtual class using all of the available bandwidth.

The transfer techniques we examine mask transfer latency by overlapping execution with transfer. More specifically, they are a form a prefetching; the techniques predict the order in which procedures execute, in attempt to transfer the code and data prior to the cycle in which execution of each initiates. The techniques do not reduce the time required to transfer the files to the destination, except for cases in which a program using non-strict execution finishes executing before transfer completes. For the results in this paper, if an application completes execution before all the methods have transferred, we terminate the remaining transfer.

5.1 Parallel File Transfer

Current Internet HTTP transfer technology allows multiple files to be transferred in parallel. The latest release of the HTTP 1.1 specification uses a single TCP connection. This allows up to four outstanding requests can be made (pipelining) [21]. In this vain, we model the transfer of multiple classes at once to assure that methods arrive as near to the predicted start of their execution as possible. The transferring files split the fixed amount of bandwidth available equally. In addition, classes are not preempted by the transfer of other classes; i.e., they transfer to completion once started.

Since bandwidth is shared, we require a schedule that indicates when class files should be transferred to obtain efficient overlap of the execution with transfer. A *Transfer Schedule* is created using the first-use procedure order determined by the reordering techniques. There are many factors that must be taken into account when developing a transfer schedule.

First, information about the size of each procedure and class file is required. The size of global and local data is also needed. For our experiments, we assume that all of the global data in a class file is transfered first. Each procedure is then transfered: local data and then code. Transfer completes once the entire class file transfers in this manner. With the size information, the scheduler can make an informed prediction of the time it will take to transfer the various parts of each file.

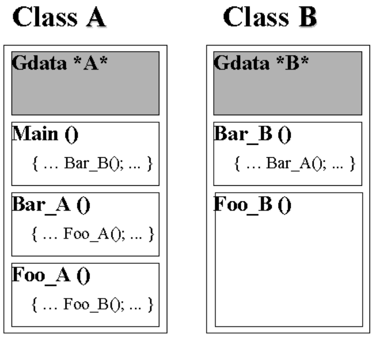

Figure 3: Reordered Class Files: The example application is restructured according to the static first use estimation or the profile. Restructuring reorders the procedures so that they appear in the class files in the order each is executed the first time.

Another key factor necessary for creating a schedule is determining the relative point in time that execution transfers for the first time from one procedure to another. Figure 4 is an example of a parallel file transfer schedule for the application introduced previously. Since the method main in class A calls method Bar_B of class B, Bar_B must have completed transfer when procedure main executes the call to procedure Bar_B in order for execution to continue uninterrupted. The schedule determines that class B must begin transfer prior to when class A begins in order to ensure that method Bar_B has completed transfer when it is called. It is apparent with this small example that a transfer schedule construction is complicated. The transfer order in the figure implies that method Bar_B will execute enough unique bytes to allow for method Bar_A of class A to complete transfer. In addition, the transfer schedule guarantees that all such first use dependency requirements are met.

We examined several algorithms for creating a transfer schedule and settled on a greedy algorithm that creates a schedule processing the class files in terms of their first usage, overlapping transfer of different class files to allow a procedure to transfer in time to switch from one class file to the the next without stalling execution.

The greedy algorithm establishes dependencies between files from the first-use procedure reordering. Class file B is dependent on a class file A if class file A executes a procedure prior to the execution of the first procedure of B. For example, in our sample application, class file B is dependent on class file A since main executes prior to Bar_B. The global data in class B and all code up to and including procedure Bar_B must have completed transfer when the method main in class A calls method Bar_B in class B. With the dependence information, the algorithm uses the number of unique bytes from each of the dependencies to determine the transfer schedule. If we are using the static first use estimation technique, the number of unique bytes is computed by accumulating the total static size in bytes of procedures that are executed before transferring to the dependent class file. For the profile driven estimation technique, unique bytes are accumulated using the total size of the instructions executed from the procedures that a class file is dependent on. In our example, method Bar_B is dependent on k unique bytes from A where k is the sum of the total bytes of class A's global data, and the (unique) bytes in method main (of class A).

During execution, a new class begins transfer once the predicted number of bytes from all classes that the new class is dependent on have transfered. Additionally, there must be bandwidth available for the class to begin transfer. If there is a restriction on the number of classes that can transfer at once (e.g., four in the recent specification of HTTP 1.1), then the class must wait until a currently transferring class completes transfer. In our results section we examine the impact of limiting the number of classes that can transfer at once.

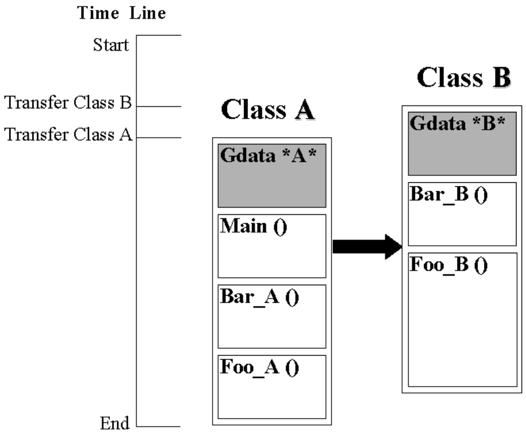

Figure 4: Parallel File Transfer Schedule: Class B must start earlier than class A so that first use dependencies are met according to the requirements determined from the first use procedure order and additional size information. The arrow indicates the place in the code where Bar_B is called by method main. Bar_B must have completed transfer at this point in order to prevent the execution from stalling.

If the restructuring techniques have mispredicted the execution order, parallel file transfer must dynamically correct for this during execution. A misprediction occurs when a procedure is invoked, but the class file the procedure is contained in has not been transfered and is currently not transferring. If there is available bandwidth, and the limit on the number of files that can be concurrently transfered has not been exceeded, the missing class file immediately starts to transfer. If the class file cannot start transferring because of the transfer file limit, it is queued up to be transfered next.

5.2 Interleaved File Transfer

In Java, an application is composed of multiple classes each containing global data, local data and code. This organization is similar to other programming languages for which multiple files comprise the executable program: for those languages the final program is typically a single binary. With interleaved file transfer, we consider a group of Java class files and compose a program as a single entity (an interleaved file), consisting of multiple procedures and data. This technique transfers the procedures and data to the destination in the order specified in this virtual interleaved file.

An interleaved file is a reordering of procedures. The transfer algorithm takes the application and the restructuring information as input. It generates an interleaved file from the input information and transfers it in the order dictated by the restructuring, e.g., methods from different classes may be interspersed for transfer. This transfer

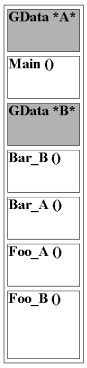

Figure 5: Virtual Interleaved File: This file is a combination of all of the class files in an application. The methods are interleaved to provide the most efficient transfer schedule according to the first use procedure order.

technique assumes that transfer proceeds at the method (procedure) level, in the order established by the restructuring algorithms. Figure 5 illustrates an interleaved file for our example application.

A method grouped together with its local data is a transfer unit. A single transfer unit is transferred at a time in the order specified by the first use procedure order. This allows each unit to acquire the total bandwidth available. As with parallel file transfer, global data is sent first and a procedure may execute if its code and data have all transferred. If this information has not arrived, then execution is stalled until the necessary transfer completes.

6 Experimental Methodology

To evaluate non-strict execution for Java, we used a bytecode instrumentation tool called BIT [16, 17]. The BIT interface enables elements of the bytecode class files, such as bytecode instructions, basic blocks and procedures, to be queried and manipulated. We use BIT to generate our first-use profiles, to perform the reordering, and to simulate the execution of the restructured class files.

For our Java implementation, we use the JDK version 1.12beta provided by Digital Equipment Corporation (DEC). The applications are instrumented and executed on a 500 MHz DEC Alpha running operating system OSF V4.0. We present results for the six Java programs described in Table 1. Five of the six programs are applications and the other is an applet (Hanoi). The programs were chosen based on the large number of bytecodes contained in each. The programs are well known and have been used in previous studies to evaluate tools such as Java compilers, decompilers, profilers, bytecode to binary, and bytecode to source translators [17, 24].

Table 2 shows the general statistics for the benchmarks. For each benchmark we use two inputs, the *test* input and a smaller *train* input. The static statistics shown in Table 2 apply to both inputs, and the dynamic statistics are shown for the test input, with the dynamic statistics for the train input shown in parenthesis.

6.1 Simulator Model

Our simulation results are in terms of the number of Alpha processor cycles needed to execute a program taking into account the cycles for transferring the program and the cycles for executing the program. To develop a baseline for the number of cycles it takes to execute a program with strict execution, we first timed each program to find out the number of cycles to execute the program on a 500 MHz Alpha 21164 processor. The number of cycles it took to execute each program and its program average CPI is shown in Table 3. Table 3 shows, the CPI (Alpha cycles per Java bytecode instruction) varies significantly with the application, ranging from 82 to 3830. The reason for this high variance is that some applications, such as Hanoi, invoke bytecodes that call uninstrumented implementations of some methods, e.g., window system calls. We use the average CPI for each program to model the number of cycles it takes to execute each bytecode instruction when performing our simulations. In our future work we plan to establish a more accurate measurement of the cycles required for each of the individual bytecode instructions in order to more accurately model this variance.

To evaluate non-strict versus strict execution we examine the performance of transferring the program over a T1 link (1Mb/sec) and a 28.8 Kbaud modem link (29Kb/sec). For a 500 MHz Alpha this equates to the T1 link taking approximately 3,815 cycles to transfer each byte, and 134,698 cycles to transfer 1 byte for the Modem link. These numbers are used during our cycle level simulation to determine how many bytes of each class file have finished transferring each cycle. This number is then used to determine when the global data or a procedure has finished transferring.

Table 3 shows the average CPI and the number of cycles (in millions) to execute the program. The fourth column shows the number of cycles (in millions) to transfer the complete program. The next column shows the total number of cycles to execute the program for strict execution, which is the sum of columns three and four. The fifth column shows the percent of strict execution due to the transfer delay. In reporting our results we computed a *normalized execution time*. The normalized execution time is calculated by taking the number of cycles for our current configuration and dividing that by the number of cycles for strict execution reported in Table 3.

7 Performance of Non-Strict Execution

To evaluate the impact of non-strict execution and program restructuring we present simulation results. The overall improvement is achieved by reducing the total number of cycles that an application waits for transferring code and data. This improvement results from the combination of non-strict execution and transfer schedule techniques. Our results measure the impact on invocation latency as well as on total execution time. In addition, we discuss the impact of global data restructuring to provide additional performance improvements.

7.1 Invocation Latency

Invocation latency is the number of cycles between the initiation of remote execution of an application and when execution of the application actually begins. We present these results to emphasize the importance of non-strict execution. If an application is allowed to begin execution upon receiving the first procedure (e.g., main()) instead of being required to wait for the entire first file to transfer, then invocation latency can be greatly reduced.

Table 4 shows the the number of cycles (in millions) from initiation until the program can start executing. For strict execution, this is the time it takes for the *first class file* to finish transferring. For non-strict execution, this is the number of cycles it takes for the global data and the *first procedure* to finish transferring. In parenthesis is the percent decrease in cycles provided by non-strict execution. Data partitioning is included with these results but explained in detail in Section 7.3. In essence, it is the restructuring of the global data throughout the class as it is needed, as opposed to transferring it all at the start of the class. The results show that

BIT	Bytecode Instrumentation Tool: Each basic block in the input program is instrumented to report its class and method name
Hanoi	(Java Applet) Towers of Hanoi puzzle solver: Solutions to 6 and 8 ring problems are computed
JavaCup	LALR Parser Generator: A parser is created to parse simple mathematics expressions
Jess	Expert System Shell: Computes solutions to rule based puzzles
JHLZip	PKZip file generator: Input is combined into a single file in PKZip format
TestDes	DES encryption/decryption algorithm: Encrypts a string then decrypts it

Table 1: Description of Benchmarks Used.

Program	Total Files	Size KB	Dynamic Instrs In Thousands Test (Train)	Static Instructions In Thousands Total	% Executed	Total Methods	Instrs Per Method
BIT	48	124	7763 (5582)	10.8	66	643	17
Hanoi	3	6	329 (68)	0.4	85	58	8
JavaCup	35	139	318 (126)	14.8	81	843	18
Jess	97	266	3116 (270)	15.1	47	1568	10
JHLZip	7	35	2380 (1023)	4.0	76	186	22
TestDes	3	50	310 (303)	8.9	98	51	174

Table 2: General Statistics for the Benchmarks. The columns represent the total number of files (classes), the size in KBytes of the application, the dynamic instruction count (in thousands), the static instruction count (in thousands), the total number of methods, and the average number of instructions per method.

			T1 Link (Millions of Cycles)			Modem Link (Millions of Cycles)		
Program	CPI	Execution Cycles in Millions (secs)	Transfer Exe Cycles (secs)	Total Strict Cycles (secs)	% Cycles Transfer	Transfer Exe Cycles (secs)	Total Strict Cycles (secs)	% Cycles Transfer
BIT	147	1141 (2.3)	776 (1.6)	1916 (3.8)	40.5	28404 (56.8)	27264 (54.5)	96.0
Hanoi	3830	1261 (2.5)	27 (0.1)	1289 (2.6)	2.1	2327 (4.7)	1066 (2.1)	45.8
JavaCup	1241	482 (1.0)	988 (2.0)	1471 (2.9)	67.2	35208 (70.4)	34726 (69.5)	98.6
Jess	225	700 (1.4)	1885 (3.8)	2585 (5.2)	72.9	66932 (133.9)	66232 (132.5)	99.0
JHLZip	82	194 (0.4)	258 (0.5)	452 (0.9)	57.0	9247 (18.5)	9053 (18.1)	97.9
TestDes	484	150 (0.3)	306 (0.6)	456 (0.9)	67.1	10952 (21.9)	10802 (21.6)	98.6
AVG	1001	655 (1.3)	707 (1.4)	1361 (2.7)	51.1	25512 (51.0)	24857 (49.7)	89.3

Table 3: Base Case Statistics. For each cycle count in this table, the equivalent number of seconds (on the 500Mhz Alpha) is provided in parenthesis. The columns represent the average cycles per instruction, total time in cycles for local execution, transfer time required in cycles, total execution time in cycles using strict execution, and the percentage of cycles due to transfer for both transfer rates, respectively.

	T1 Link			Modem Link		
Program	Strict	NonStrict	Data Part.	Strict	NonStrict	Data Part.
BIT	14	11 (19)	10 (26)	475	386 (19)	352 (26)
Hanoi	13	7 (42)	3 (77)	452	263 (42)	106 (77)
JavaCup	66	34 (49)	8 (88)	2333	1197 (49)	287 (88)
Jess	24	16 (32)	7 (72)	835	572 (32)	237 (72)
JHLZip	13	8 (43)	3 (76)	465	267 (42)	112 (76)
TestDes	71	70 (1)	70 (1)	2481	2459 (1)	2457 (1)
AVG	33	24 (31)	17 (56)	1173	857 (31)	592 (56)

Table 4: The Effect of Non-Strict Execution and Program Restructuring on Invocation Latency. For each transfer rate we show: a cycle count (in millions) to initiate strict execution, non-strict execution, and non-strict execution with data partitioning. The numbers in parenthesis are the percent decrease of strict execution cycles.

the invocation latency can be significant for strict execution, and non-strict execution can help to reduce this delay by 31% to 56%.

7.2 Overall Execution

The impact of non-strict execution can also be measured by the improvement of total remote execution time. The performance results presented in the remaining tables are normalized to the baseline model of strict execution. We compute the results as the percent of normalized execution time by taking the number of cycles to execute for a configuration and dividing it by the number of strict cycles to execute the program from Table 3. For example, a percent normalized execution time of 60 means that the number of cycles was 60% of the base, which resulted in a 40% improvement, so smaller numbers are better. For all of the results, our base execution was a simulation in which the application transferred one class to completion at a time and executed strictly: methods execute only when the entire class file in which they are contained has been transferred.

We present the results for the different transfer methods and for the T1 and modem link using procedure reordering guided by our estimated static call graph (*SCG*), using the train input profile to guide the ordering (*Train*), and using the test input profile to guide the ordering (*Test*). All results are shown for executing the test input. Therefore, the test results are perfect results since they use both the test input to profile and restructure the program and to gather the simulation results. Whereas, the Train results are more realistic since the train input is used to guide the first-use ordering and the simulation results are reported for the test input.

Table 5 and Table 6 show the results from our simulations using the parallel file transfer technique with restructuring for the two transfer rates, respectively. Results are shown for limiting the number of files that can be transferred in parallel to one, two, and four. Results for simultaneously transferring an infinite number of files are also shown. The results show that a maximum number of four parallel transfers is sufficient to provide most of the performance improvement for non-strict execution.

The benefits from having a single virtual file is shown in Table 7. Since a Java program consists of many files, these results model the effect of interleaving the transfer among the different files, or the performance that would be gained if the files were combined into one unified virtual file. This technique transfers one method at a time according to the predicted order of the restructuring. The results show additional performance gains can be achieved if all the class files were considered as one, or the transfer was interleaved between the different class files.

7.3 Partitioning of Global Data

Up to this point we have discussed restructuring the code section of the class files only. We now consider restructuring the global data. The global data section of each class file is divided into structures containing information about the global data. Table 8 shows the major parts of the class file pertinent to global data and the size of each as a percentage of the total global data size. These fields are described in detail in [19]. Since the constant pool takes up a majority of the class file, we also describe the parts of this structure in Table 8. These results show that if we are to optimize the data for class files, then we should concentrate on the Constant Pool and the Utf8 Java strings.

Table 9 shows the size of the global data in comparison to the local data for the programs we examined. Since the global data needs to be transferred before the first procedure, it can be advantageous to split the global data and to store the global data that is not needed at the method level. We examine breaking the global data into the global data that *must* be transferred before execution, the global data transferred with methods, and the global data unused as shown in Table 9. To break up the data at the method level we propose creating a JVM *GlobalMethodData* (GMD) structure. There is a GMD before each procedure in the new non-strict program. The GMD contains only the data in the constant pool and attributes that are needed to execute up to and including the procedure the GMD is placed before in the compiled bytecode file. This placement requires analysis to determine the first use of global data across the predicted ordering of procedures. This decomposition allows more efficient overlapping of computation and transfer, since we no longer have to wait for an entire global JVM `ClassFile` structure to be transferred before transferring the first procedure. Table 10 contains the results from partitioning the data with non-strict execution and code restructuring.

We include global data partitioning as an aside since implementing it increasingly complicates the existing linking and verification process in the JVM, as well as the incremental verification we suggest in this paper. All global data is currently required during JVM verification; techniques are needed that enable verification and security without requiring all of the global data at once. The results presented do not account for the overhead from a more complicated verification process. In our future work, we hope to establish techniques for such verification and to determine their performance impact so that we may more accurately determine the significance of partitioning the global data.

7.4 Summary of Results

Our results show that significant performance gains can be achieved by overlapping execution with transfer. Invocation latency is shown to decrease on average between 31% and 56%.

Figure 6 provides a visual summary of our results for normalized execution time. The Y-axis is the percentage of the execution time of the base case: strict execution with no restructuring. The results show that a 25% to 35% reduction in executed cycles is achieved when using the static call graph and 30% to 45% reduction in cycles is achieved when using training inputs to guide the testing inputs.

8 Conclusions

In this paper we present a non-strict model for transferring and executing programs for Internet computing. We present new techniques for rearranging the program in first-use order along with partitioning the global data for Java programs for more efficient non-strict execution. We also present two new methods for transferring Java programs to take advantage of non-strict ordering. The results show that non-strict execution combined with first-use code reordering and transfer methods significantly reduces the latency of invoking a remote application and the execution time for remote computing for the programs we examined. The reduction in invocation latency ranges from 31% to 56% on average, and the reduction in execution time ranges from 25% to 40% on average.

Although these latency hiding techniques are useful for improving Java performance, they may also be useful for Java related compilation, e.g., just in time, ahead of time, or way ahead of time compiling. If compilation can take place as the class files are being transferred, then the latency of transfer and compilation can overlap. In addition, non-strict execution techniques can be applied to other languages and mobile program technologies, such as ActiveX.

T1 Link												
	SCG				Train				Test			
Program	One	Two	Four	Inf.	One	Two	Four	Inf.	One	Two	Four	Inf.
BIT	99	96	94	90	94	88	79	79	90	87	79	79
Hanoi	100	99	99	99	100	99	99	99	100	99	99	99
JavaCup	82	81	76	76	63	61	61	59	61	56	55	55
Jess	97	93	86	77	94	90	78	70	89	64	64	64
JHLZip	97	82	74	74	82	79	72	72	75	73	72	72
TestDes	92	90	90	90	91	90	90	88	73	72	72	72
AVG	94	90	87	84	87	85	80	78	81	75	74	74

Table 5: Normalized Execution Time for Parallel File Transfer Using a T1 link. Results are shown for configurations where the transfer technology can only transfer one, two, four and an infinite number of class files at a time.

Modem Link												
	SCG				Train				Test			
Program	One	Two	Four	Inf.	One	Two	Four	Inf.	One	Two	Four	Inf.
BIT	95	92	88	76	57	55	53	53	56	54	53	53
Hanoi	90	90	90	90	90	88	88	88	90	87	88	87
JavaCup	69	69	67	65	63	60	58	56	54	54	54	54
Jess	72	70	69	69	57	57	56	55	54	53	52	51
JHLZip	56	55	55	55	56	53	53	53	54	53	53	53
TestDes	86	85	85	85	82	82	81	76	63	62	61	61
AVG	78	77	76	73	68	66	65	63	62	61	60	60

Table 6: Normalized Execution Time for Parallel File Transfer Using a 28 Kbaud modem link. Results are shown for configurations where the transfer technology can only transfer one, two, four and an infinite number of class files at a time.

	T1 Link			Modem Link		
Program	SCG	Train	Test	SCG	Train	Test
BIT	84	82	77	62	50	49
Hanoi	99	99	92	88	85	85
JavaCup	68	61	49	54	51	46
Jess	67	62	52	55	50	42
JHLZip	73	67	67	54	44	44
TestDes	74	72	72	63	60	60
AVG	78	74	68	63	57	54

Table 7: Normalized Execution Time for Interleaved File Transfer for both T1 and 28 baud modem transfer rates.

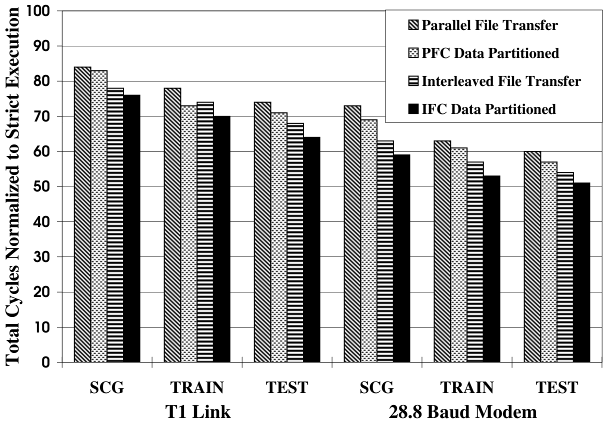

Figure 6: Summary of Results: Average normalized execution time for both transfer rates. Results are shown for parallel file transfer, parallel file transfer with data partitioning, interleaved file transfer, and interleaved file transfer with data partitioning.

	Percent of Global Data				Percent of Constant Pool										
Program	CPool	Field	Attrib	Intfc	Utf8	Ints	Float	Long	Double	String	Class	FRef	MRef	NandT	IMRef
BIT	88.2	9.2	0.7	0.0	80.1	2.2	0.0	0.0	0.0	1.8	2.4	2.6	4.5	0.1	6.3
Hanoi	93.5	3.3	0.8	0.1	75.1	0.0	0.0	0.0	1.2	0.2	3.0	4.3	6.3	0.0	9.9
JavaCup	95.3	2.9	0.5	0.0	80.3	0.3	0.0	0.0	0.0	2.3	1.7	1.8	6.1	0.1	7.3
Jess	95.6	2.0	0.6	0.1	81.9	0.2	0.0	0.0	0.0	1.1	3.7	1.3	5.4	0.1	6.2
JHLZip	94.2	4.0	0.5	0.0	63.2	17.0	0.0	0.0	0.0	1.0	1.6	3.1	6.0	0.1	8.0
TestDes	94.7	3.4	0.5	0.0	34.9	52.9	0.0	0.0	0.0	0.4	1.3	2.5	2.9	0.0	5.2
AVG	93.6	4.1	0.6	0.0	69.3	12.1	0.0	0.0	0.2	0.9	2.3	2.6	5.2	0.1	0.0

Table 8: Breakdown of Global Data and Constant Pool: Pertinent parts of ClassFile: Constant Pool (CPool), Fields, Attributes (Attribs), Interfaces (Intfcs). Parts of the Constant Pool: Utf8 (Java strings), Integers (Ints), Floats, Longs, Doubles, Strings, Classes, FieldRef Structures (FRefs), MethodRef Structures (MRefs), Interface MethodRef Structures (IMRefs). Data given is the percent of the total size of the containing structure.

Program	Local Data (KB)	Global Data (KB)	% Globals Needed First	% Globals in Methods	% Globals Unused
BIT	43.9	56.9	34	63	3
Hanoi	1.8	3.1	21	75	4
JavaCup	53.9	59.4	17	82	1
Jess	93.8	129.9	19	61	20
JHLZip	15.1	12.0	19	79	2
TestDes	29.7	5.0	15	84	1
AVG	39.7	44.4	21	74	5

Table 9: Breakdown of data in the class files into data local to methods and global data. The global data is further broken down into the data that must be transferred before execution, the data needed to be transferred with methods, and the unused data in the class files.

	Parallel File Transfer						Interleaved File Transfer					
	T1 Link			Modem Link			T1 Link			Modem Link		
Program	SCG	Train	Test	SCG	Train	Test	SCG	Train	Test	SCG	Train	Test
BIT	82	78	75	68	51	51	81	77	72	57	49	47
Hanoi	98	98	98	87	86	84	98	97	90	85	83	82
JavaCup	69	54	52	61	51	50	66	52	45	52	43	41
Jess	72	65	62	62	54	50	67	59	45	50	47	35
JHLZip	73	71	71	53	48	48	72	64	64	50	40	40
TestDes	89	71	71	84	76	60	73	70	70	61	58	58
AVG	81	73	71	69	61	57	76	70	64	59	53	51

Table 10: The Normalized Execution Time for Partitioning the Global Data with Parallel File Transfer and the Interleaved File Transfer technique for both T1 and 28 Baud Modem Transfer Rates. Results are shown for the parallel file transfer assuming a limit of four files for parallel transfer.

Acknowledgments

We would like to thank anonymous reviewers for providing useful comments. This work was funded in part by NSF CAREER grant No. CCR-9733278, UC MICRO grant No. 97-018, a DEC external research grant No. US-0040-97, and a NSF grant IRI-95-21046.

References

[1] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Programming Language Design and Implementation*, May 1996.

[2] Jean-Loup Baer and Gary R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Transactions on Software Engineering*, SE-2(1):54–62, March 1976.

[3] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology - Crypto 93 Proceedings, Lecture Notes in Co mputer Science*, 1994.

[4] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[5] Peter Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[6] W. J. Doherty and R. P. Kelisky. Managing VM/CMS systems for user effectiveness. *IBM Systems Journal*, pages 143–163, 1979.

[7] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 358–365, Las Vegas, NV, June 1997.

[8] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–103, December 1997.

[9] Christopher W. Fraser and Todd A. Proebsting. Custom instruction sets for code compression. Available at: http://www.cs.arizona.edu/people/todd/papers/pldi2.ps, October 1995.

[10] N. Gloy, T. Blockwell, M.D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *30th International Symposium on Microarchitecture*, December 1997.

[11] J. Gosling and McGilton H. The Java Language Environment: A white paper. In *Sun Microsystems, Inc., White Paper*, May 1995.

[12] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 171–182, Las Vegas, NV, June 1997.

[13] D. J. Hatfield. Experiments on page size, program access patterns, and virtual memory performance. *IBM Journal of Research and Development*, pages 58–66, January 1972.

[14] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the16th International Symposium on Computer Architecture*, pages 242–251, June 1989.

[15] Chris Johnson. *Human Factors and Web Development*, chapter 16: The Ten Golden Rules for Providing Video Over the Web or 0% of 2.4M (at 270k/sec, 340 sec remaining), pages 207–221. Lawrence Erlbaum Associates, Publishers, Mahwah, New Jersey, 1998.

[16] Han Bok Lee. BIT: Bytecode instrumenting tool. Master's thesis, University of Colorado, Boulder, Department of Computer Science, University of Colorado, Boulder, CO, June 1997.

[17] Han Bok Lee and Benjamin G. Zorn. BIT: A tool for instrumenting java bytcodes. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS97)*, pages 73–82, Monterey, CA, December 1997. USENIX Association.

[18] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *30th International Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.

[19] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1997.

[20] S. McFarling. Procedure merging with instruction caches. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26(6):71–79, June 1991.

[21] H.F. Nielsen, J. Gettys, A. Baird-Smith, E. Prudhommeau, H.W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *ACM Applications, Technologies, Architectures and Protocols for Computer Communication*, September 1997.

[22] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):16–27, June 1990.

[23] Michael P. Plezbert and Ron K. Cytron. Does just in time = better late than never? In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, January 1997.

[24] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications a way ahead of time (wat) compiler. In *Proceedings of the Third Conference on Object–Oriented Technologies and Systems*, 1997.

[25] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 360–369, January 1995.

[26] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 203–216, New York, NY, USA, December 1993. ACM Press.

[27] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded risc architecture. In *25th International Symposium on Microarchitecture*, pages 81–91, 1992.