# Reducing Transfer Delay Using Java Class File Splitting and Prefetching

Chandra Krintz Brad Calder

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

Urs Hölzle

Computer Science Department
University of California, Santa Barbara
urs@cs.ucsb.edu

**Abstract**

*The proliferation of the Internet is fueling the development of mobile computing environments in which mobile code is executed on remote sites. In such environments, the end user must often wait while the mobile program is transferred from the server to the client where it executes. This downloading can create significant delays, hurting the interactive experience of users.*

*We propose Java class file splitting and class file prefetching optimizations in order to reduce transfer delay. Class file splitting moves the infrequently used part of a class file into a corresponding cold class file to reduce the number of bytes transferred. Java class file prefetching is used to overlap program transfer delays with program execution. Our splitting and prefetching compiler optimizations do not require any change to the Java Virtual Machine, and thus can be used with existing Java implementations. Class file splitting reduces the startup time for Java programs by 10% on average, and class file splitting used with prefetching reduces the overall transfer delay encountered during a mobile program's execution by 25% to 30% on average.*

## 1 Introduction

The Internet today provides access to distributed resources throughout the world. The network performance available to an arbitrary user program in this environment is highly varied and relatively modest compared to the available processing power. For example, a typical user might be connected at less than 1 Mbit/s while running programs on a machine capable of executing several hundred millions of instructions per second.

In proceedings of 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), November, 1999.


Mobile applications use the resources of the Internet to perform computation; in this context they are programs that transfer over networks for remote execution. These programs, e.g., Java applets, are commonly downloaded on demand as opposed to being stored locally. The performance of mobile programs depends upon both the network latency and bandwidth available as well as the processor speed throughout the execution. Given the gap between processor and network speeds, mechanisms are needed to compensate for network performance in order to maintain acceptable performance of mobile programs.

Program performance is commonly measured by the time for overall program execution. However, Internet applications frequently include interactive participation by the user. For such programs, performance can be measured by the amount of delay a user experiences during the interactive session. For example, if a user presses an arbitrary button on an applet, execution should proceed accordingly as if the program is stored locally. When files must be downloaded in order to continue, the user instead experiences a delay in execution. If delays are long and frequent, as is common for applet execution, performance is greatly degraded. Research has shown that these delays are a crucial factor in a user's perception of overall application performance. Early work investigated the effect of time-sharing systems on productivity (e.g., see [2]), and concluded, among other things, that delays in system response time disrupted user thought processes.

In this paper, we focus on *Transfer Delays*, i.e., the delays created when an application is stalled waiting for code or data to arrive over a network connection. Our goal is to eliminate transfer delays in mobile Java programs by reducing the size of the class files transferred and by overlapping transfer with execution. To accomplish this, we propose the use of two new Java compiler optimizations – Java class file splitting and class file prefetching. Both optimizations are

implemented purely at the Java level and require no modification to the Java Virtual Machine.

*Class file splitting* partitions a class file into separate hot and cold class files, to avoid transferring code that is never or rarely used. Class file splitting helps reduce the overall transfer delay and invocation latency. *Invocation Latency* is the time required to begin execution of a program. In Java, this includes the time for transfer and loading as well as any additional file processing required by the execution environment, e.g. verification.

*Class file prefetching* inserts prefetch commands into the bytecode instruction stream in order to overlap transfer with execution. The goal is to prefetch the class file far enough in advance to remove part or all of the transfer delay associated with loading the class file.

In section 2, we describe the execution and verification model currently employed by the commonly available Java execution environments. Section 3 describes our implementation of class file splitting to reduce transfer delay. Section 4 then describes our approach for class file prefetching. Section 5 describes the programs examined and our simulation methodology, and section 6 provides performance results when using verification. Section 7 shows the performance of our optimizations if no verification is used in a trusted environment. Section 8 examines the distance between initialization of a class reference and the first time the reference is actually manipulated by the user program. Section 9 provides related work for reducing transfer delay, and section 10 summarizes the contributions of this paper.

# 2 Java Execution

Java programs are organized as independent *class files*, each of which contains the methods and state variables required to implement a class. Before any method can be executed or any state variable can be modified within a class file, the entire class must be transferred to the location where execution takes place.

Most JVM implementations load classes on demand at the time of the first reference to each class. This is called *dynamic loading*. References that cause a class to be dynamically loaded include object instance creation and static member (class method or field) access. Such accesses are called *first-use* references, since they cause a non-local class file to be transferred for dynamic loading. Dynamic loading causes a delay in execution each time a class file load request is issued, since the thread triggering the load stalls until the class has been loaded, verified, resolved, and initialized. Hence, users experience these *transfer delays* intermittently during execution as well as upon program invocation.

In this paper we concentrate on techniques to reduce these intermittent delays caused by class file loading. We introduce class file splitting to reduce the number of bytes that must transfer in order to continue execution and class file prefetching to enable file transfer to occur concurrently with program execution. Both optimizations decrease transfer delay in mobile Java programs.

## 2.1 Java Verification

Verification in Java is a security mechanism used to ensure that a program is structurally correct, does not violate its stack limits, implements correct data type semantics, and respects information hiding assertions in the form of `public` and `private` variable qualifiers. To reduce the complexity of these tasks, the verifier requires that each class file be present at the execution site in its entirety before the class is verified and executed for the first time. Verification may require additional classes to be loaded (without regard to whether or not they are executed) in order to check for security violations across class boundaries. We refer to this process as *verified transfer*. Verification is performed on each untrusted class in the class-loader prior to the first use of the class; this additional processing increases the delay in execution imposed by dynamic loading.

In contrast, verification can be turned off using the `-noverify` runtime flag if all classes are trusted. We refer to this option as *trusted transfer*. In this work, we focus on verified transfer but include the resulting performance using trusted transfer with our optimizations. For our results using verification, we modeled the verification mechanism in JDK 1.2. We clarify this process here with two small examples.

Java guarantees that types are used consistently during execution, i.e., each assignment of a variable is consistent with its defined type. If a code body contains variables with non-primitive types for which assignments are inconsistent, the verifier must check each class file used in the assignments. For example, in Figure 1, class `X` must be transferred and verified at program invocation. The class, however, contains a variable of class `ZSuper`, called `varZ`. This variable may be assigned an instance of class `Z` or of class `ZSuper` depending on the value of `j`. In order to verify class `X`, the verifier must transfer both class `ZSuper` and class `Z` in order to perform the necessary consistency checks on variable `varZ`.

Verification also requires loading and verification of an entire superclass chain in order to verify that a subclass (a class that extends another) is correct. For example, in the above scenario, when class `Z` is loaded, verification requires that its ancestors, class files `ZSuper` and `ZSuperSuper`, are loaded and verified.

Another example is shown in Figure 2. In this case, class file `A` will be transferred and verified at program invocation. Class file `B` will only transfer when it is first used (`new B()`), since all uses of `varB` consistently use the same type, class `B`, throughout the code in class file `A`. Class file `C` will also be transferred on its first-use; it transfers when the con-

```
public class X  {
  public static void main(String args[]) {
    ZSuper varZ = null;
    int j = Integer.parseInt(args[0]);

    if (j > 10) {
      varZ = new Z();
    } else if (j > 5) {
      varZ = new ZSuper();
    }

    if (j > 5) {
      int i = varZ.meth();
      System.err.println("answer: " + i);
    }
  }
}
```

```
class Z extends ZSuper {
  public int meth() {
    return 15;
  }
}

class ZSuper extends ZSuperSuper {
  public int meth() {
    return 10;
  }
}

class ZSuperSuper {
  public int meth() {
    return 5;
  }
}
```

Figure 1: First Java example to demonstrate class file transfer and its interaction with verification when using superclasses.

structor, `B()`, is executed. Each class in this example is transferred and verified on first use. Notice also that class `A` contains methods that are executed conditionally. For example, `error()` will only be executed if an error occurs. Despite this conditional execution, the method `error()` must still be transferred as part of class `A`. We will use this second example throughout the remainder of our paper.

The first example above indicates that verification may require many additional class files to be transferred for verification whether or not they are used during execution. The second example requires entire class files to transfer even though some fields or methods may never be used. These cases commonly occur in Java programs and thus motivate us to investigate optimizations that reduce the amount of code and data that must transfer in order to continue execution (thereby reducing transfer delay). We focus on optimizations that do not require modification to the Java Virtual Machine. In the next section, we examine class file splitting.

# 3 Splitting Java Class Files

Java class file splitting was recently proposed by Chilimbi, et. al., in [1] to improve memory performance. The goal of their research was to split infrequently used fields of a class into a separate class. When a split class is allocated, the important fields are located next to each other in memory space and in the cache for better performance. Separating fields in class files according to the predicted usage patterns improves data memory locality in the same manner as procedure splitting improves code memory performance [11].

The goal of our class file splitting is different than this prior approach, since we are optimizing to reduce transfer delay and not to improve memory performance [1]. With our techniques, a class is split into two: a hot class containing used fields and methods; and a cold class containing never or rarely used fields and methods. We create a reference in the hot class through which the cold class members are accessed. If a cold member is used, the use triggers the loading and verification of the cold class on demand. If the cold members are not used, they will not be transfered, which reduces transfer delay. In contrast, the class file splitting by Chilimbi et. al. [1] always transfers and loads cold class files.

## 3.1 Splitting Algorithm

Class file splitting occurs once Java programs have been compiled from source into bytecode. The splitting algorithm relies on profile information of field and method usage counts. With the profile information as input, a static bytecode tool performs the splitting. For this paper, we classify a field or method as cold if it is not used at all during profiling. In addition, we only perform splitting when it is beneficial to do so, e.g., when the total size of cold fields and methods is greater than the overhead for creating a cold class. The minimum number of bytes to represent an empty class file is approximately 200 bytes. In this section, we explain the primary steps for class file splitting using Figure 3 to exemplify the algorithm and to expose the potential benefits of our approach. The steps are:

1. Create execution profiles for multiple inputs and identify classes to split
2. Construct cold class files for each class selected for splitting
3. Move unused fields and methods from original (hot) class to cold class
4. Create references from hot class to cold class and vice versa

[1] Improved memory performance may be a side affect of our splitting but we leave this analysis to future work.

```
class A {
  public B varB;

  A() {...}

  main() {
    foo ();

    varB = new B();
    ...
  }

  foo () {...}

  mumble() {...}

  error() {...}
  ...
}
```

```
class B {
  public C varC;
  public int var1;
  private int var2;
  private int var3;
  protected int var4;

  B() {
   var1 = 0;
   var3 = 0;
   varC = new C();
  }

  bar() {
    var1 = var2*var4;
    varC.foobar();
  }
  ...
}
```

```
class C {

  C() {...}

  foobar() {...}

  ...
}
```

Figure 2: Second Java example to demonstrate class file transfer and its interaction with verification. This example will be used in the remainder of the paper.

5. Update variable usages in hot and cold class code to access relocated fields/methods via the new reference

The original code, shown in Figure 3(a), contains class `A` with a field reference to class `B`, and class `B` that references class `C` in its constructor. The first step of the algorithm profiles the use patterns of fields and methods during execution. Classes containing unused fields and methods are appended to a list of classes to be split. In the example, the profile determines that `mumble()` and `error()` in class `A` are rarely used, as well as method `bar()` in class `B`. Both class `A` and class `B` are added to the list of classes to split.

The next step of the algorithm, using the list as input, splits class `A` into class `A` and class `Cold$A`. A similar split is done for class `B` into class `B` and class `Cold$B`. The constant pool, method table, and field table entries are constructed for the cold classes, with any other necessary class file information. All cold code and data is then inserted into each cold class in the third step of the algorithm.

Next, a field `cldRef` is added to both original classes; this field holds a direct reference to the respective cold class. This field enables access to the cold class from within each hot class. In addition, the cold classes have a field `hotRef`, which holds a reference to the hot class for the reverse access. In the hot class, `cldRef` is assigned an instance of the cold class when one of the cold fields or methods is accessed for the first time. Upon each reference to cold fields and methods a check is added to determine if the cold object pointed to by `cldRef` has been instantiated. A new instance of the cold class will only be created during execution if one does not already exist. When the cold class is instantiated, the constructor of the cold class initializes `hotRef` to reference the hot class.

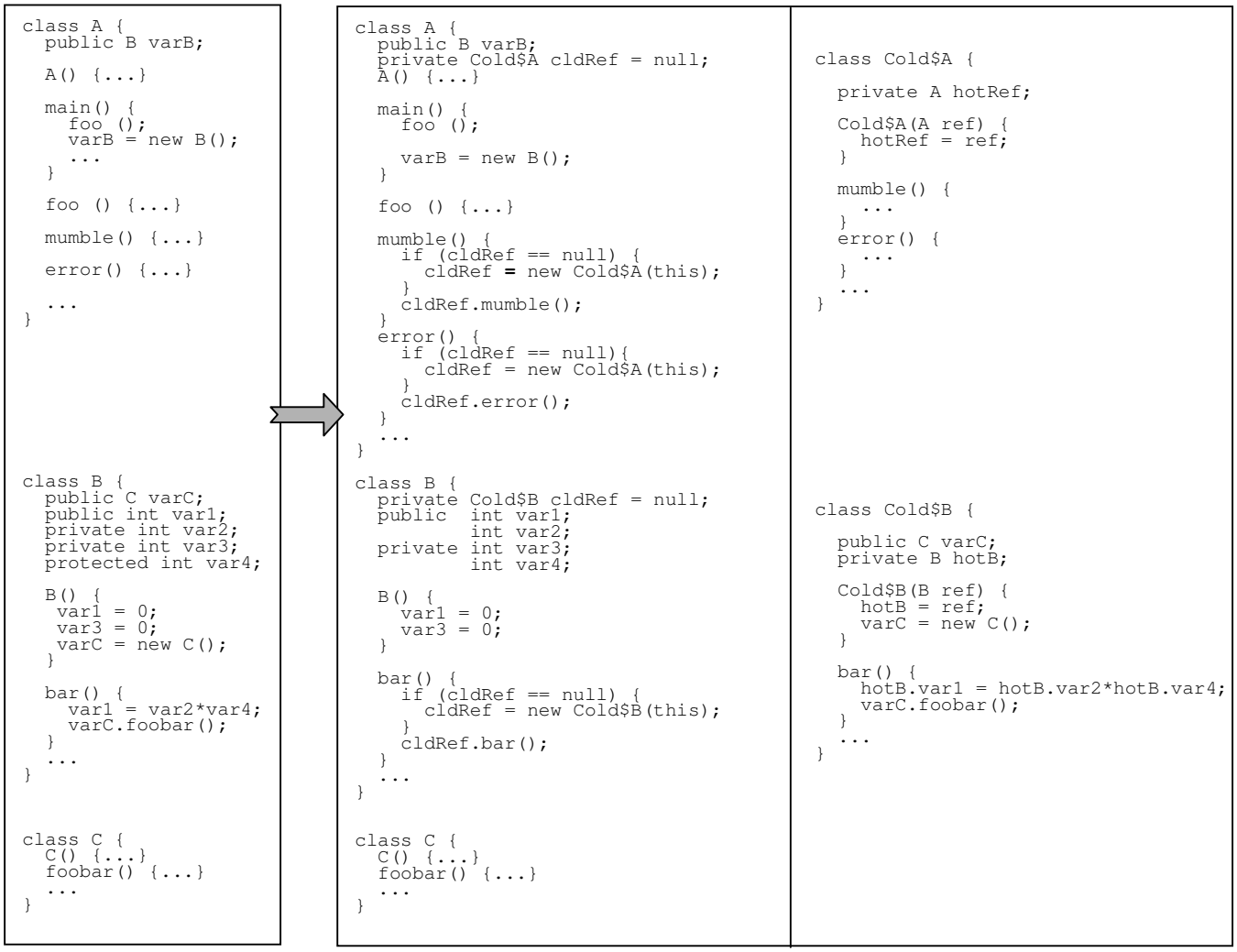
We emphasize that this new cold class reference is not created in the constructor of the respective hot class. If cold class instantiation is performed in the constructor, transfer of the cold class would be triggered prematurely (prior to the actual first use of the class), negating any benefit from splitting. Instead, we delay transfer of cold class files until first use (if it ever occurs). For example, in Figure 3(b), `Cold$A` will only be transferred if either methods `mumble()` or `error()` are executed. Likewise, `Cold$B` will only be transferred if method `bar()` is invoked.

In the final step of the algorithm, we modify the code sections of both the hot and the cold class. For each access to a cold method or field in the hot class, we modify the code so that the access is performed through the cold class reference. The same is done for the accesses to hot fields by the cold class. At this point the field and method access flags are modified as necessary to enable package access to private members between the hot and cold classes. For example, originally class `B` contained a private qualifier for `var2`. Since class `Cold$B` must be able to access `var2`, the permissions on the variable are changed to package access (public to the package). We address the security implications of this decision below.

In the example, our splitting algorithm also finds that the reference to class `C`, `varC`, in class `B` is only used in procedure `bar()`, which was marked and split into the cold class. Our compiler analysis discovers this, and moves `varC` to the cold class as shown in Figure 3(b).

## 3.2 Maintaining Privacy When Class File Splitting

As described above, a hot class must contain a reference to the cold class so that cold members can be accessed. The members of the hot class must be able to access the cold members as if they were local to the hot class. Likewise the object instance of the cold class must be able to reference all fields and methods in the hot class according to the semantics defined by the original, unmodified application.




(a) Original Classes

(b) Class Files after Class File Splitting

Figure 3: Class file splitting example.

The problem with this constraint is that if a class member is defined as private, it is only accessible by methods within the class itself. To retain the semantics of the original program during splitting, hot class members must be able to access cold class members and vice versa.

In our implementation, we change all cross referenced (cold members used by hot and vice versa) private and protected members to *package access*. This is accomplished by removing the private and protected access flags for these field variables as shown in Figure 3 for `var2` and `var4`. Package access means that members are public to all of the routines in the package, but not visible outside the package.

As previously stated, we apply our Java class file splitting optimization after compilation using a binary modification tool called BIT [9]. The original application has been compiled cleanly and is without access violations before splitting is performed. Therefore, changing the access of private or protected fields to package access happens after the compiler has performed its necessary type checking.

If package access is used during splitting, then splitting does not provide complete security, and may not be suitable for all class files in an application. For a secure application, we propose that the bytecode optimizer performing the splitting be given a list of classes for which splitting is disallowed. These are classes with private/protected fields that must remain private/protected for security reasons. The developer can then specify the classes for which splitting should not be used.

# 4 Prefetching Java Class Files

In this section, we introduce class file prefetching as an optimization that is complementary to class file splitting. Prefetching class files masks the transfer delay by overlapping transfer with computation, i.e., class files are transferred over the network while the program is executing. In the optimal case, this overlap can eliminate the transfer delay a user experiences. Effective prefetching requires (1) a policy for determining at what point during program execution each load request should be made so that overlap is maximized, and (2) a mechanism for triggering the class file load to perform the prefetch.

Figure 4 shows the benefit of prefetching, for the prior code example in Figure 2. The first class to be transferred is class `A`, and execution starts with the `main` routine. While executing `main`, a prefetch request initiates the loading of class `B`. We insert a prefetch request for class `B`, since it is needed when the first-use for class `B` is executed at the `new B()` instruction in `main`. If class `A` executes long enough prior to this first reference to class `B`, the statement `new B()` will execute without waiting on the transfer of `B`. On the other hand, if there are not enough useful compute cycles to hide class `B`'s transfer (that is, the time to transfer class `B` is greater than the number of cycles executed prior to `A`'s instantiation of `B`), then the program must wait for the transfer of class `B` to complete before performing the execution of `new B()`. In either case, prefetching reduces the transfer delay since without prefetching execution stalls for the full amount of time necessary to transfer class `B`.

## 4.1 Overview of Prefetching Algorithm

The prefetch algorithm contains five main steps:

1. Build basic block control flow graph
2. Find first-use references
3. Find cycle in which each basic block is first executed
4. Estimate transfer time for each class
5. Insert a prefetch request for each first-use reference

First, the algorithm builds a basic block control flow graph for the entire program, with interprocedural edges between the basic blocks at call and return sites. The next step of the algorithm finds all first-use references to class files. These are the first references that cause a class file to be transferred if it has not already. When a first-use reference to class `B` is found, the algorithm constructs a list of the class files needed in order to perform verification on class `B`; class `B`'s first-use reference causes these class files to be transferred.

The third step of the algorithm estimates the time at which each basic block in the program is first executed (measured in cycles since the start of the program). This start time determines the order in which first-use references are processed and the position at which to place a prefetch request for each class. Next we estimate the number of cycles required to transfer each class file. We use this figure to determine how early in the CFG the prefetches need to be placed in order to mask the entire transfer delay. The final step of the algorithm processes the first-use references in the predicted order of execution and inserts prefetch requests for the class file being referenced. The following sections discuss all of these steps in more detail.

## 4.2 Finding First-Use References

We use program analysis to find each point in the program where *first-use* references are made. A first-use reference is any reference to a class that causes the class file to be loaded. Therefore, for a class `B` reference to be considered a first-use reference, there must exist an execution path from the main routine to that reference, such that there are no other references to class `B` along that path. All of the first-use references to class files are found using a modified depth-first search of the basic block control flow graph (CFG). A description of this algorithm can be found in [6].

## 4.3 First-Execution Ordering and Cycle Time of First-Use References

Once all first-use references are found we need to order them so that prefetch requests can be appropriately inserted. Ideally, we should prioritize according to the order in which the references will be encountered during execution. This first-execution order is the sequential ordering of basic blocks (and thus first-use references in those basic blocks) based on the first time each basic block is first executed. Since we cannot predict program execution exactly, we need to estimate the cycle in which each basic block is first executed. To do this we use profiles to determine this *first-execution* order of references and cycle of execution.

In this paper we used profiles to determine the order of processing the first-use references. During a profile run, we keep track of the order of procedure invocations and basic block executions during program execution for a particular input. The order of the first-use references during the profile run determines the order in which we place prefetches for the Java class files. We also keep track of which class files are transferred using the JDK 1.2 verification mechanism, when executing each first-use reference. This provides us with a list of additional class files that need to be transferred to perform verification. All procedures and basic blocks that are not executed are given an invocation ordering and first cycle of execution based on a traversal of the control flow graph using the same static heuristics in [6].

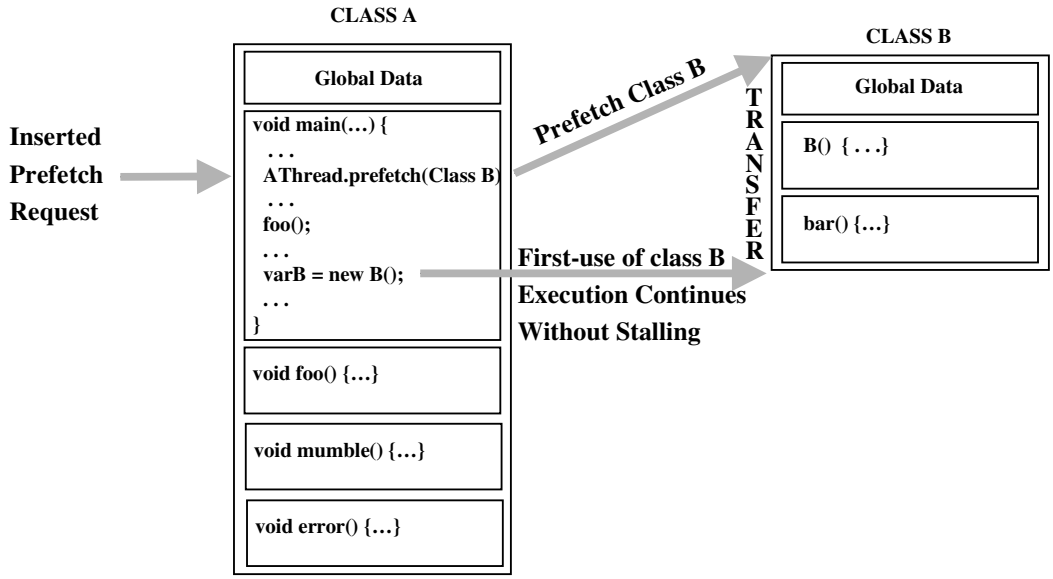



Figure 4: The benefit of Java class file prefetching. A prefetch to class file B is inserted into class file A. The full transfer delay will be masked if class file B has fully transferred by the time the command *new B()* is executed.

Profiles can accurately determine the relative distance in cycles between basic blocks. When calculating the first cycle of execution for each basic block, we use the average number of cycles per bytecode instruction (CPB) over the entire program execution to estimate the number of cycles required to execute the bytecodes in each basic block.

## 4.4 Prefetch Insertion Policy

In the fifth step of the prefetching algorithm, we determine the basic blocks in which to place the prefetch requests. Prefetch requests must be made early enough so that the transfer delay is overlapped. Finding the optimal place to insert a prefetch can be difficult. The two (possibly conflicting) goals of prefetch request placement are to (1) prefetch as early as possible to eliminate or reduce the delay when the actual reference is made, and (2) ensure that the prefetch is not put on a path which causes the prefetch to be performed too early. If a prefetch starts too early, it may interfere with classes that are needed earlier than the class being prefetched. In this case, the prefetch can introduce delays by using up available network bandwidth.

Figure 5 is the algorithm we use for this step. We clarify it with the example shown in Figure 6. In the example, we wish to insert two prefetches for the first-use references to class B and class C. Figure 6 shows part of a basic block control flow graph for a procedure in class A. Nodes are basic blocks with the name of the basic block inside each node. The dark edges represent the first traversal through this CFG during execution, and the lighter dashed edges represent a later traversal through the CFG. The first part of the prefetch placement algorithm determines the first-execution cycle and order of the basic blocks. This indicates that a prefetch for the first-use reference (in basic block Z) to class B needs to be inserted before the prefetch for first-use reference (in basic block Q) to class C. We process the classes in increasing order of first use reference.

The algorithm inserts a prefetch for each first-use reference (twice in our example). When placing a prefetch, the basic block variable `bb` is initially set to the basic block containing the first-use reference (node Z for class B, and node Q for class C), and `cycles_left` is initialized to the estimated number of cycles required to transfer the class files. The algorithm examines each parent of the current basic block to determine prefetch placement for each path in the CFG. The estimated number of cycles each basic block executes is subtracted from `cycles_left` during examination. The algorithm follows the edge from `bb` to each `parent` in the CFG until either (1) `cycles_left` is reduced to zero, or (2) the parent lies on a prefetched or already encountered path. Otherwise, we keep searching up the CFG and recursively call this routine on the parent of the current basic block.

For class B in our example, the algorithm starts at basic block U and performs a reverse traversal of the CFG processing the parents of each basic block. At each basic block encountered, `cycles_left` is decremented by the estimated cycle time of the current basic block. In our example, enough cycles execute during the loop between X and T to reduce `cycles_left` to zero. Since the relative distance in cycles between the first-use reference of B and basic block W is

```
Procedure: find_bb_to_add_prefetch(
      Reference ref,BasicBlock bb, int cycles_left)

   /* ref - a pointer to the first use reference for a class file X */
   /* bb - the current basic block to try and place the prefetch */
   /* cycles_left - number of cycles left to mask when prefetching
      the class files for this first-use */

   bb.processed = TRUE;
   bb.prefetch_path_name = ref.class_file_name;

   /* get one of the parent basic blocks of bb in the CFG */
   parent = bb.parent_list;

   while (parent ! = NULL) {

      if (parent.processed) {
         /* if parent basic block already is on a path for a prefetch
            then insert the prefetch at the start of basic block bb */
         insert_prefetch_at_start_bb(ref, bb);

      } else {
         /* parent is not yet on a prefetch path, so calculate the
            number of cycles that can be masked if the prefetch was
            placed in the parent basic block */
         cycles_between_bb = parent.first_cycle - bb.first_cycle;

         if (cycles_between_bb >= cycles_left) {
            /* all the transfer cycles will be masked by placing the
               prefetch at the beginning of basic block parent */
            insert_prefetch_at_beginning_bb(ref,parent);
            parent.processed = TRUE;
            parent.prefetch_path_name = ref.class_file_name;

         } else {
            if (cycles_between_bb > 0) {
               /* need to keep traversing up the CFG, because the
                  first time parent is executed is not far enough
                  in the past to mask all the transfer delay */
               find_bb_to_add_prefetch(
                     ref, parent, cycles_left - cycles_between_bb);

            } else {
               /* do nothing */
               /* the parent was first executed *after* the current bb,
                  so don't put a prefetch up this parent's path */
            }
         }
      }

      /* process next parent of basic block bb */
      parent = parent.next
   }
```

Figure 5: Algorithm for finding the basic block to place the prefetch.

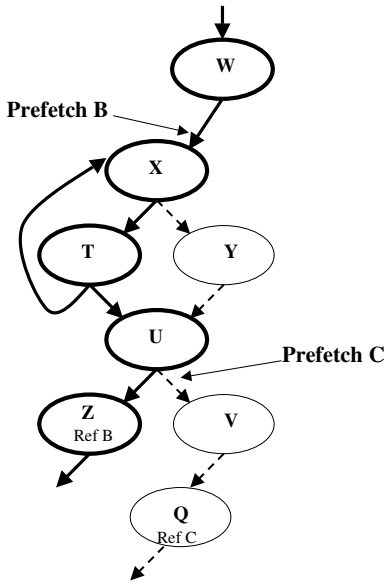



Figure 6: Prefetch Insertion Example. Nodes represent basic blocks in the control flow graph. Solid edges represent the basic blocks executed on the first traversal through the CFG. The dashed edges represent a later traversal through the CFG. Class B is first referenced in basic block Z, and class C is first referenced in basic block Q.

large enough to mask the transfer of B, the prefetch to class B is inserted immediately before basic block X.

The algorithm stops searching up a path when the basic block being processed is already on a prefetched path. A prefetched path is one that contains a prefetch request for a previously processed class. Placing a new prefetch on a prefetched path consumes available bandwidth for more important class prefetches and imposes unnecessary transfer delay on the class. When a prefetch is inserted onto a path, all of the basic blocks on that path are marked with the class file name of the prefetch and a processed flag. These flags are used to prevent later first-use prefetches from being placed on the same path. In our example, once the prefetch for first-use reference B is inserted, the algorithm continues with the next first-use reference for class C. When inserting the prefetch to class C, the prefetch does not propagate up into basic block U, since basic block U is on the prefetch path for B. Therefore, the prefetch to class C is inserted right before entering basic block V.

## 4.5 Prefetch Implementation

Once we determine all points in the program at which prefetch requests should be made, we insert prefetch instructions into the original application. For prefetching to be cost effective, the prefetch mechanism must have low-overhead and must not cause the main thread of execution to stall and

| | |
|---|---|
| BIT | Bytecode Instrumentation Tool: Each basic block in the input program is instrumented to report its class and method name |
| Jack | Spec Benchmark: Java parser generator based on the Purdue Compiler Construction Tool Set |
| JavaC | Spec Java Compiler Benchmark: javac-Java source to bytecode compiler |
| JavaCup | LALR Parser Generator: A parser is created to parse simple mathematics expressions |
| Jess | Spec Expert System Shell Benchmark: Computes solutions to rule based puzzles |
| JLex | Lexical Analyzer Generator |
| MPegAudio | Audio File Decompression Benchmark: Conforms to the ISO MPEG Layer-3 audio specification |

Table 1: Description of Benchmarks Used.

wait for the file being prefetched to transfer. To prefetch a class file B, we use the standard Java `loadClass` method.

When adding prefetching to a package, we create one separate *prefetch thread* to perform the loading and resolution of each class file. An inserted prefetch request then inserts a list of class files onto a prefetch queue, which the prefetch thread consumes. The prefetch thread prevents the main threads of execution from stalling unnecessarily while the class file is transferring. Therefore, this solution allows computation (performed by one or more of the main threads) and class transfer (initiated by the prefetch thread) to occur simultaneously.

Most existing JVMs (including the Sun JDK VM) only block the requesting thread when loading a class, and allow multiple threads to load classes concurrently. Therefore, our approach does not require any changes to these VMs. If the prefetch of a class is successful, the JVM will have loaded the class based on the request issued by the prefetch thread before any main thread needs that class. Alternatively, if a main thread of execution runs out of useful work before a required class is fully loaded, the JVM will automatically block this thread until the class becomes available.

A prefetch inserted for a first-use of class B may actually prefetch several class files as needed to perform verification for class B as described in section 2.1. Before each prefetch request, a flag test is used to determine if a class is local or has already been fetched. If the flag indicates that no prefetch is necessary than the overhead of our prefetch mechanism is equivalent to a compare and branch instruction.

# 5 Methodology

We implemented our prefetching and splitting optimizations using a bytecode modification tool called BIT [8, 9]. BIT enables elements of Java class files, such as bytecode instructions, basic blocks, and procedures, to be queried and manipulated. We use this tool for the simulation of our prefetching and splitting optimizations and to model their efficacy for programs in a mobile environment.

For our Java implementation, we use JDK version 1.2 to model the verification used in our simulations. We executed each benchmark with the `-verbose -verify` flags to force verification to occur as well as output to be generated about each file as it is loaded by the Java Virtual Machine. In addition, each benchmark was first instrumented to report each time a class is first used by the program. From the parsed output, we are able to determine the point in the program execution at which each file is loaded by the Java Virtual Machine. This provides us with a list of verification dependencies for each class file used. The list is then used by our simulator to determine when files must transfer.

Each benchmark simulation is executed on a 500 MHz DEC Alpha 21164 running operating system OSF V4.0. We present results for the Java programs described in Table 1. The programs listed in Table 1 are from the SPECjvm98 suite and other programs that have been used in previous studies to evaluate tools such as Java compilers, decompilers, profilers, bytecode to binary, and bytecode to source translators [9, 12].

Table 2 shows the general statistics for the benchmarks. For each benchmark we use two inputs, a larger input (Ref) used for all of the results and a smaller input (Test) used to construct our across-input profiles. The static data shown in Table 2 apply to both inputs, and the dynamic data apply to the Ref input, with the dynamic data for the Test input shown in parenthesis. The second column shows the number of class files in each program's package. The next column shows the total size of all the class files in kilobytes. The fourth column shows the number of bytecode instructions executed for each program. The next two columns show the number and percentage of static bytecode instructions executed. The final column shows the number of class files used during execution for each input.

Our simulation results are in terms of the number of processor cycles needed to execute a program, taking into account the cycles for transferring the program ("delay" or "lost" cycles in which no computation is performed) and the actual computation cycles. To establish a baseline for the number of cycles required to accomplish each computation

| Program | Total Classes | Size in KB | Dynamic Instrs In Thousands Ref (Test) | Static Instructions In Thousands Total | Static Instructions In Thousands % Executed | Total Classes Executed |
|---|---|---|---|---|---|---|
| BIT | 48 | 104 | 47163 (7814) | 11 | 65 | 32 (32) |
| Jack | 56 | 131 | 27448 (13729) | 19 | 80 | 46 (46) |
| Javac | 176 | 561 | 24689 (518) | 41 | 24 | 132 (59) |
| JavaCup | 35 | 116 | 381 (278) | 17 | 72 | 31 (31) |
| Jess | 151 | 397 | 15147 (3438) | 18 | 43 | 135 (133) |
| JLex | 20 | 87 | 28346 (12128) | 12 | 68 | 18 (18) |
| MPegAudio | 55 | 120 | 121998 (115514) | 34 | 88 | 28 (28) |

Table 2: Benchmark Statistics. The columns represent the program name, the number of class files, the number of bytes for all class files, the number of instructions executed for a given input, the number of static instructions in the programs and the percent of these executed, and the number of classes executed for a given input.

| | Verified Transfer: Time in Seconds for Execution and Transfer Delay | | | | |
|---|---|---|---|---|---|
| | Exe Time w/ No | T1 Link | | 28.8 Modem | |
| Program | Transfer Delay | Invocation | Overall | Invocation | Overall |
| BIT | 0.848 | 0.044 | 0.742 | 1.537 | 26.065 |
| Jack | 7.849 | 0.323 | 1.014 | 11.335 | 35.627 |
| Javac | 5.710 | 0.008 | 4.813 | 0.293 | 169.097 |
| JavaCup | 0.965 | 0.434 | 1.107 | 15.256 | 38.905 |
| Jess | 0.711 | 0.007 | 3.070 | 0.249 | 107.879 |
| JLex | 12.999 | 0.007 | 0.831 | 0.260 | 29.184 |
| MPegAudio | 0.080 | 0.013 | 1.288 | 0.442 | 45.259 |
| Average | 4.166 | 0.119 | 1.838 | 4.196 | 64.574 |

Table 3: Verified Transfer. The first column shows the execution time in seconds for each program if there was no transfer delay. The time (in seconds) is then shown for invocation latency and overall transfer delay for both the T1 link and the 28.8 baud modem.

(without any transfer delay) we first timed each program on a 500 MHz Alpha 21164 processor to calculate the average cycles per bytecode (CPB) for each program. We then use the average CPB to model the number of cycles it takes to execute each bytecode instruction when performing our simulations. To evaluate the performance of our prefetching model, we assume transfer of programs over a T1 link (1Mb/sec) and a 28.8 Kbits/sec modem. The T1 link takes approximately 3,815 cycles to transfer each byte, and the modem link takes 134,698 cycles per byte.

# 6 Results for Verified Transfer

To evaluate splitting and prefetching, we measure improvement in terms of the percentage of transfer delay eliminated. As mentioned before, the transfer delay is the total number of cycles an application spends waiting for class files to transfer during execution. Performance results are shown for only running the Ref input in Table 2. The results with *Diff* in the name use the Test input to perform profiling, and the results with *Same* use the Ref input to perform profiling.

Table 3 shows the time in seconds to execute each of the programs locally and remotely using verification. The first column shows the time for local execution. *Invocation* shows the time in seconds required to start executing the program. The invocation delay includes the time to transfer the first class file *and to perform its verification*. Therefore, this time includes the transfer of all of the class files needed to verify the first class file. The *Overall* columns shows the total transfer delay of the original program in seconds. The remainder of our performance results are shown in terms of the percent of invocation latency and overall transfer cycles that can be eliminated due to class file splitting and prefetching.

Figure 7 shows the reduction in invocation latency due to class file splitting, where larger percentages are better. The results show a 9% reduction in invocation latency on average, and that similar results are found between inputs.

Figure 8 shows the reduction in overall transfer latency for the 28.8 modem due to prefetching alone, class file splitting alone, and class file prefetching and splitting combined. Figure 9 provides the same scenarios for the T1 Link. These results show that prefetching by itself can provide a 6% reduction in transfer delay for the T1 link. Class file splitting on the other hand eliminated 25% of the overall transfer delay. Combining class file splitting with prefetching results in a 30% reduction in transfer delay. The Diff results for `BIT` show that the difference in the method usage patterns between the two inputs reduces the benefit of our algorithm by half. For the other benchmarks, methods that are unused by one input are most likely unused by other inputs for the benchmarks and inputs we studied.

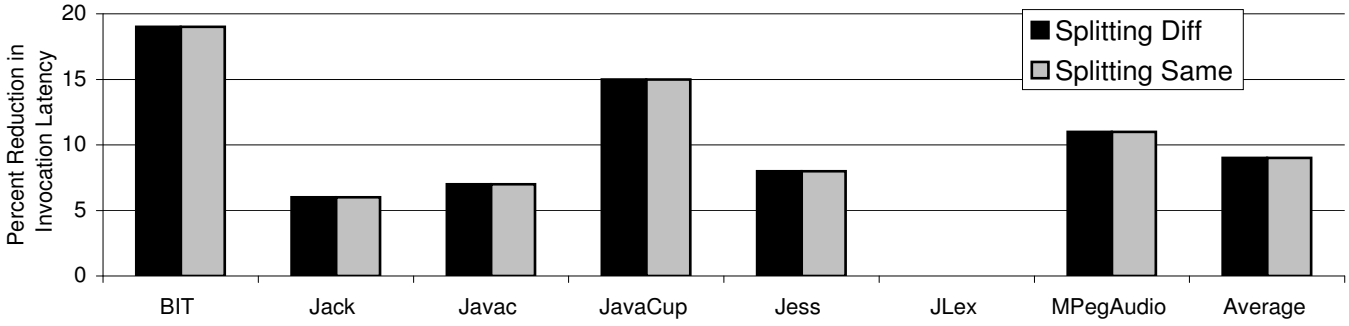



Figure 7: Percent reduction in Invocation latency after class file splitting when verification is used.

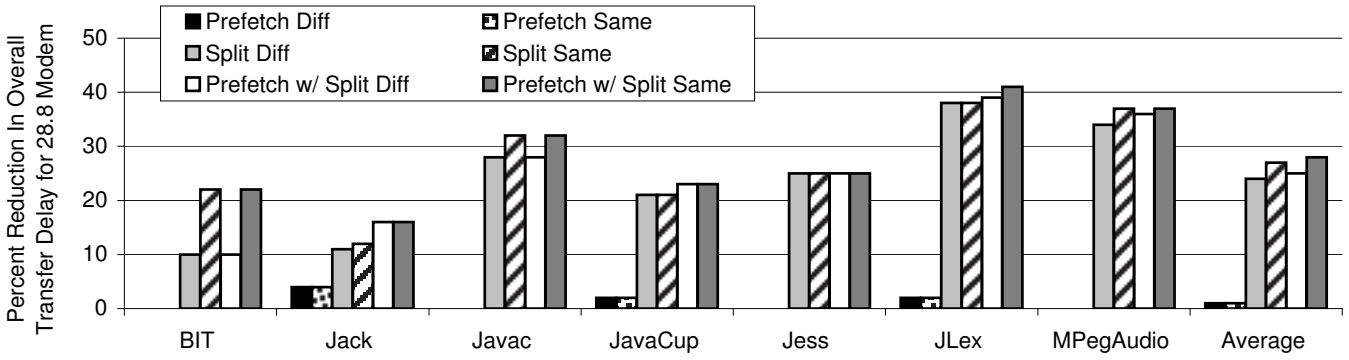



Figure 8: Percent of reduction in overall transfer delay for the 28.8 baud modem when verification is used.

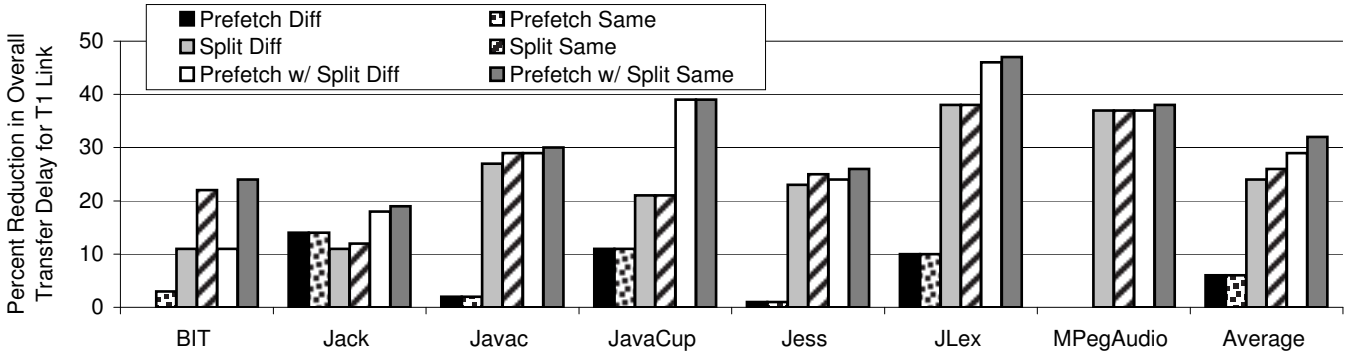



Figure 9: Percent reduction in overall transfer delay for the T1 link verification is used.

| Program | Trusted Transfer: Time in Seconds for Execution and Transfer Delay | | | | |
|---|---|---|---|---|---|
| | Exe Time w/ No Transfer Delay | T1 Link | | 28.8 Modem | |
| | | Invocation | Overall | Invocation | Overall |
| BIT | 0.848 | 0.044 | 0.742 | 1.537 | 26.065 |
| Jack | 7.849 | 0.014 | 0.998 | 0.490 | 35.056 |
| Javac | 5.710 | 0.008 | 4.357 | 0.293 | 153.092 |
| JavaCup | 0.965 | 0.133 | 1.102 | 4.667 | 38.720 |
| Jess | 0.711 | 0.007 | 3.070 | 0.249 | 107.879 |
| JLex | 12.999 | 0.007 | 0.831 | 0.260 | 29.184 |
| MPegAudio | 0.080 | 0.013 | 1.288 | 0.442 | 45.259 |
| Average | 4.166 | 0.032 | 1.770 | 1.134 | 62.179 |

Table 4: Trusted Transfer. The first column shows the execution time in seconds for each program if there was no transfer delay. The time (in seconds) is then shown for invocation latency and overall transfer delay for both the T1 link and the 28.8 baud modem.

For the programs we examined, the transfer delay using the modem link (Table 3) dwarfs the execution time of the programs. Prefetching improves performance when there are enough execution cycles between uses of class files available to overlap communication. This is the case for `Jack`, where prefetching alone does better than splitting alone for the T1 link results. For `Jack`, a large amount of computation occurs early in the program after only a small number of class files have been loaded, providing us with time to prefetch class files used later in the execution. However, most of the programs we studied initialize most of their classes/objects early in the program reducing the number of cycles available for overlap with transfer. We address this issue further in Section 8.

# 7 Trusted Transfer

In a trusted environment the Java interpreter can skip part of the verification phase. If either the Java class file is compiled without verification or the interpreter is run with verification turned off (the `-noverify` option), then the interpreter skips the type verification phase when loading a class file. We call this model of transfer *trusted transfer*.

In this mode, additional class files will not be transferred for verification. Only those class files used during execution will transfer upon each first use, thereby reducing the transfer delay imposed by those classes with verification dependencies. Using trusted transfer can also reduce the startup time for a Java program in which the first class file requires additional files to be transferred when verification is turned on.

We implemented a trusted version of our prefetching optimization, to model execution without verification. The same algorithm is used as described in section 4. The only difference is that when we process a first-use for a class, we insert a prefetch to load only that class. In comparison, the verified transfer algorithm prefetches all the class files needed to verify a class. The splitting algorithm used for trusted transfer is identical to that used for verified transfer.

## 7.1 Results for Trusted Transfer

Table 4 shows the time in seconds to execute each of the programs with and without transfer delay for trusted transfer. This table is the same as Table 3, but without verification. The difference in time between these two tables shows the reduction in transfer delay when class files are trusted as opposed to requiring verification.

Figure 10 shows the percent reduction in invocation latency (larger percentages are better) due to class file splitting for trusted transfer. The Diff bars show results when the Test input is used to profile the program and the Ref input is used to gather the simulation results. The Same results show the performance with the same input (Ref) is used to both profile and gather the simulation results. The startup latency is much smaller with trusted transfer and splitting provides a large reduction for `BIT` and `JavaCup`.

Figure 11 shows the reduction in overall transfer latency for the 28.8 modem due to prefetching by itself, class file splitting, and class file splitting with prefetching; Figure 12 shows the same results for the T1 link. Prefetching provides 6% improvement over splitting for the T1 link results shown in Table 4, since the overall transfer delay does not dwarf the execution time of the program. In general, splitting provides the largest gain in performance, but prefetching provides a large reduction in transfer delay for `JavaCup`, `Jack`, and `JLex`.

In the following section, we examine the potential of another compiler optimization to improve the impact of our prefetching optimization.

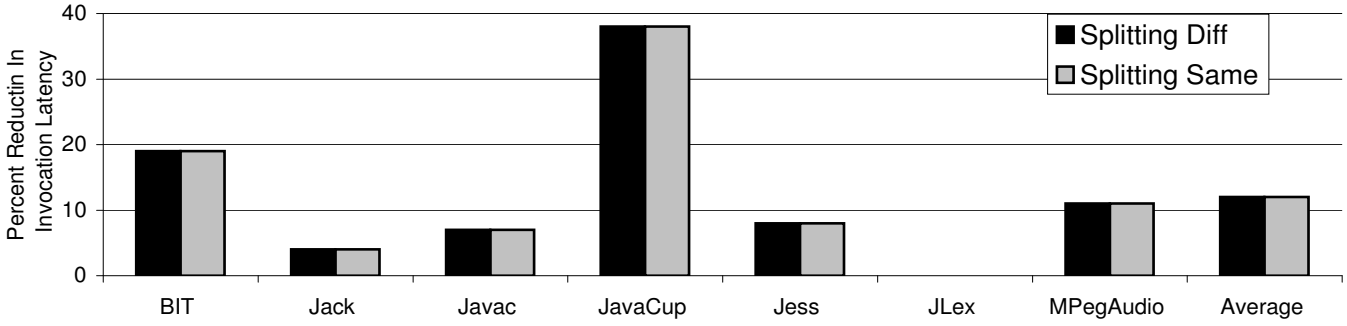



Figure 10: Percent reduction in Invocation latency after class file splitting for trusted transfer.

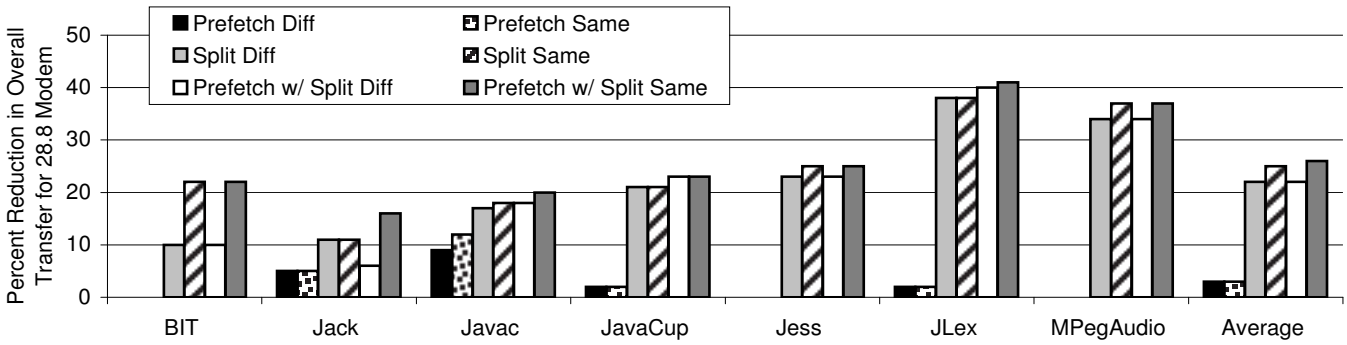



Figure 11: Percent reduction in overall transfer delay for the 28.8 baud modem for trusted transfer.

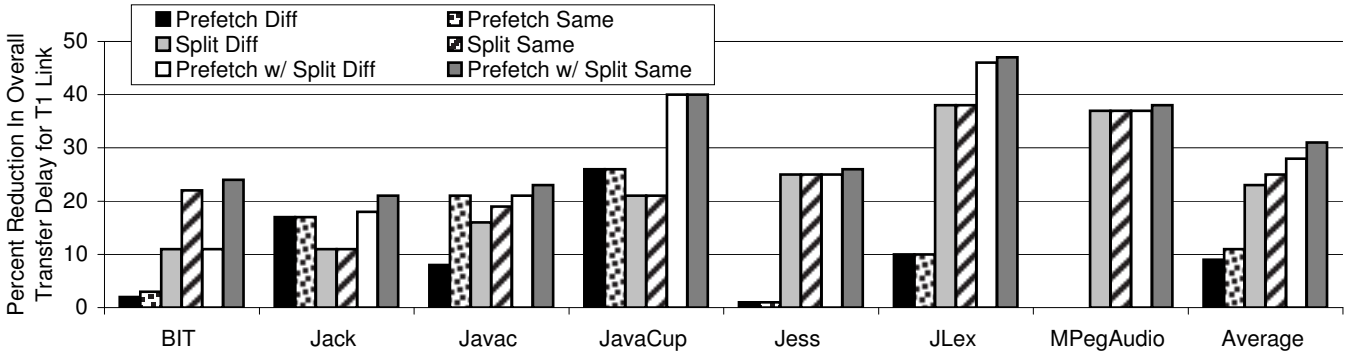



Figure 12: Percent reduction in overall transfer delay for the T1 link for trusted transfer.

| Program | Count | Average Distance | Median Distance |
|---|---|---|---|
| BIT | 25 | 22031.4 | 1075.5 |
| Jack | 36 | 319947.6 | 41.0 |
| Javac | 45 | 33445.1 | 3176.0 |
| JavaCup | 19 | 23575.3 | 4300.6 |
| Jess | 128 | 20526.7 | 17.6 |
| JLex | 15 | 210502.1 | 4.0 |
| MPegAudio | 21 | 192527.2 | 91776.6 |
| Average | 41 | 117507.9 | 14341.61 |

Table 5: Initialization to First-Use Distance. The first column contains the number of class files for which there is a distance in cycles between the initialization and the first-use of the class. Column two presents this distance as an average over all such classes in the application. The final column is the median value of the distance. The distance numbers are given in thousands of instructions.

# 8 Distance Between Initialization and First Real Use

Many times in Java, as well as in C++, constructors initialize all fields of the instance being created. In Java, this initialization may cause other class files to be transferred and loaded, even though the class members may not be used (except for initialization) until thousands or millions of cycles later in the program's execution. This was shown in Figure 2, where an instance of class `C` is allocated in class `B`'s constructor, even though variable `varC` might not be used at all or remain unused for many cycles. Optimizations may be able to reduce transfer delay by moving the instructions for the creation of instances of as-yet-unloaded classes to immediately before the first real use of the class. This postponement would avoid stalling the main thread of execution until it is absolutely necessary. It also potentially improves the efficacy of our prefetching optimization by making additional cycles available for overlap of transfer with execution.

To determine the potential of techniques that address the problem of premature initialization in Java, we measured the number of cycles executed between the initialization of a class file and the first real use of the class file. Initialization, in this context, is object allocation and creation; including assignment of initial values as necessary to fields. In Java, this is indicated by the execution of the `<init>` routine upon object creation. The *first real use* of an object is the instruction in the user's application that first manipulates the object, i.e., the first time a field is read or written, or the first time a method is invoked. In Figure 2, the first real use of class `C` by class `B` is the statement `varC.foobar()` in method `bar`, even though field `varC` is initialized in `B`'s constructor.

The results are shown in Table 5. The first column shows the number of class files that are initialized but are not used right away. The second column is the average number of cycles in thousands between initialization and the first real use for these classes. The last column is the median number of cycles (in thousands) between initialization and the first real use for these classes. The results show that for many class files, millions of cycles occur between class initialization and the first real use; approximately 50% of the class files in Javacup and BIT exhibit this behavior. The data assumes that classes incur no transfer delay, i.e., this distance is not caused by network delays. We are currently looking at compiler optimizations to move the initialization to the point in the program of first real use, and to use this in combination with prefetching to reduce transfer delay.

# 9 Related Work

Prior work has offered solutions to reducing the effect of transfer delay. The first is non-strict execution: a JVM modification that enables execution at the method level. This technique masks network latency by overlapping execution with computation. The second research area is code compression. Code compression reduces the amount of data transferred thereby avoiding network latency. Lastly there has been some recent effort to reduce the startup delay for applications.

## 9.1 Non-Strict Execution

In recent work [6], we proposed reducing the cost of transfer delays by overlapping class file transfer with execution at the procedure level using non-strict execution. The existing JVM imposes *strict* semantics. Each class file must transfer to completion prior to being executed. With *non-strict* execution, the unit of transfer is the procedure (method). When transferring a class file, execution is allowed to continue when the required method and data has finished transferring. To enable this, we added procedure delimiters and runtime checks to the JVM to identify when each procedure and its code has transferred. For short running programs, non-strict execution enables substantial performance improvements in both overall execution time (25% to 40% on average) and

program startup time (31% to 56% on average) [6]. Transfer schedules were used to indicate when to start transferring each class file to best overlap transfer with execution.

The use of the non-strict execution model can provide performance benefits, but it requires major modifications to the current design of the Java Virtual Machine (JVM). The modifications range from adding procedure delimiters to the wire-transfer format to modifying JVM verification to perform incremental verification at the procedure level.

Our splitting and prefetching work in this paper is very different. Prefetching represents a pull model of fetching class files to eliminate transfer delay, whereas non-strict execution uses a push model with a transfer schedule [6]. In addition, prefetching and splitting require no modification of the JVM or the server and work in existing Java execution environments.

## 9.2 Code Compression

For class file prefetching, 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 with compression (i.e., latency avoidance). Several approaches to compression have been proposed to reduce network delay in mobile execution and we discuss them here.

Pugh describes a wire format that reduces a collection of individually compressed class files 50% to 20% the size of compressed jar files on average [13]. The wire format uses the gzip compression utility but incorporates a very efficient and compact representation of class file information. In addition, it organizes the files into a single file that makes the gzip utility more effective. He determines when sharing can be performed within an application so that additional redundant information is not transferred.

Other compression techniques have been created for machine-specific binary representations. In one such project, Ernst et. al. [3] 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.

Other attempts to reduce the size of program code include work at Acorn Computers to dramatically reduce the size of a 4.3 BSD port so that it fits on small personal computer systems [15]. A major focus of this work is to use code compression to reduce disk utilization and transfers. Fraser and Proebsting also explore instruction set designs for code compression, where the "instruction set" is organized as a tree and is generated on a per-program basis [5]. Fraser has recently extended this work to incorporate machine learning techniques to automate decisions about instruction encoding in [4]. In other recent work, Lefurgy et. al. [10] describe a code compression method based on replacing common instruction sequences with "codewords" that are then reconstructed into the original instructions by the hardware in the decode stage.

Our optimizations are distinct from, and complementary to, code compression. Code compression can significantly reduce the transfer delay by decreasing the number of bits that need to transfer; it however, does not eliminate all of the delay. In contrast, class file prefetching can mask additional delay when the prefetch can be performed far enough in advance. Using compression, splitting, and prefetching together will allow prefetching to more easily mask the transfer delay, especially the transfer delay imposed by invocation. Examining the performance of using compression with prefetching and splitting is part of our future research.

## 9.3 Reducing Invocation Time

In research concurrent and independent of ours, Sirer et.al. [14] described an optimization for reducing the startup time for mobile programs executed on thin client and hypertext systems. The optimization repartitions Java class files to enable more effective utilization of the available bandwidth during transfer of such programs to remote sites. Methods distinguished as unused by profile information are split out to form new cold class files declared with Java's `final` modifier to enable optimization. In this study, the authors concentrate on improving the startup time of a mobile program. The optimization is quite similar to our splitting technique in that we reduce the invocation latency as well as the overall execution time. The results they achieve are similar to ours. Our technique differs in that we distinguish between verified and trusted transfer. In addition, we include class file prefetching to improve the overall performance of mobile programs by overlapping the transfer with useful work.

Other recent work by Lee et.al. [7], decreases program invocation time by packing application code pages more effectively for remote execution. The optimization extends Non-Strict execution (described above) to architecture-specific binaries of Web and desktop applications. Programs are reordered into contiguous blocks according to predicted use of procedures (using profiles). Programs are divided into a global data file and page-size files containing code. When a web engine executes a remote binary, it loads each file on demand and is able to continue execution once each page-size file arrives. The technique, when combined with demand paging, can reduce startup latency for the benchmarks tested by 45% to 58%. In contrast, our work concentrates on Java applications and applets and does not require a special execution engine to achieve performance. In addition, they do

not provide prefetching in their approach, so no overlap of computation and communication is performed and execution is stalled while each page is dynamically loaded.

# 10 Conclusions

The increased interest in the Internet as a computational paradigm has sparked demands for the immediate performance improvement of mobile applications. The performance of mobile programs depend both on the time to transfer the program for remote execution as well as the actual execution at the remote site. Since the reduction rate in network latencies has not paralleled that of processor cycle time, mechanisms for masking and eliminating transfer delays are important for the viability and performance of Internet applications.

In this work, we presented an optimization to split Java class files to reduce the size of class files transferred. In addition, we presented a latency-hiding technique that prefetches a Java class file prior to the first reference to the class by an application. Prefetching enables overlap of execution cycles with the transfer of class files. The results showed that class file splitting reduces the invocation latency by 10%. The overall transfer delay is reduced by 25% on average when using class file splitting. Prefetching provided its largest gains when using a T1 Link. Using prefetching with splitting for this configuration resulted in an average reduction in overall transfer delay of 30%.

The implementation of splitting and prefetching we present does not require any modification of the JVM. The optimizations use compile-time analysis and heuristics with profiles to guide selection of classes to split and placement of prefetch requests. Once the class files are modified, Java applications execute with improved performance and the same semantics of the original execution without optimization.

# Acknowledgements

We would like to thank Bill Pugh, Jim Larus, Emin Sirer, and the anonymous reviewers for providing useful comments. This work was funded in part by NSF CAREER grants No. CCR-9733278 and No. CCR-9624458, and a research grant from Compaq Computer Corporation.

# References

[1] T. Chilimbi, B. Davidson, and J. Larus. Cache-conscious structure/class field reorganization techniques for c and Java. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999.

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

[3] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 358–365, Las Vegas, NV, June 1997.

[4] C. Fraser. Automatic inference of models for statistical code compression. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

[5] C. Fraser and T. Proebsting. Custom instruction sets for code compression. Available at: http://www.cs.arizona.edu/people/todd/papers/pldi2.ps, October 1995.

[6] C. Krintz, B. Calder, H. Lee, and B. Zorn. Overlapping execution with transfer using non-strict execution for mobile programs. In *Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[7] D. Lee, J. Baer, B. Bershad, and T. Anderson. Reducing startup latency in web and desktop applications. In *Windows NT Symposium*, July 1999.

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

[9] H. Lee and B. Zorn. BIT: A tool for instrumenting Java bytecodes. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS97)*, pages 73–82, Monterey, CA, December 1997. USENIX Association.

[10] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. In *30th International Symposium on Microarchitecture*, Research Triangle Park, NC, December 1997.

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

[12] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. 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.

[13] W. Pugh. Compressing Java class files. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

[14] E. Sirer, A. Gregory, and B. Bershad. A practical approach for improving startup latency in Java applications. In *Workshop on Compiler Support for Systems Software*, May 1999.

[15] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *25th International Symposium on Microarchitecture*, pages 81–91, 1992.