UNIVERSITY OF CALIFORNIA, SAN DIEGO

Reducing Load Delay to Improve Performance of Internet-Computing Programs

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science and Engineering

by

Chandra Krintz

Committee in charge:

Professor Bradley Calder, Chairperson
Professor Andrew Chien
Professor Rene Cruz
Professor Urs Hölzle
Professor Joseph Pasquale

2001

The dissertation of Chandra Krintz is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2001

Dedication and Gratitude

I dedicate this work to two people that are immensely important in my life, Kristen and Jedd. I would not be the person I am today nor could I have made it to this point without them in my life. There is no one luckier than I to be able to receive such unconditional support, love, and friendship from these two exceptional individuals. I am also very grateful to Bobbie and Dick for their constant belief in me regardless of the wacky paths I considered taking. These two people mean more to me than I am able put into words. It is the freedom they gave to me and the strength they showed that enabled me to do all of the things I've done. On the most difficult days, just knowing they are part of my life makes everything easier. I am truly lucky to have them as such amazing role models and friends.

Many thanks to my advisor, Brad Calder, for venturing down this research path, supporting me to finish remotely, and for showing me the road to success. I am also extremely grateful to all of the people in my lab (Barbara, Beth, Eric, Glenn (my comrade in arms), John, Lori, Suleyman, Tim, Wei, as well as all of the new additions) for their friendship. They brought tremendous support, tireless listening, interesting conversation, and great fun to my life. They helped me make it through and to even enjoy the process - on certain days... ;). I wish them all tremendous happiness.

I am also extremely grateful to the University of Tennessee and its excellent system support staff (particularly Clay England and Brett Ellis) for their provision of patience as well as the invaluable resources that enabled the completion of this dissertation.

Above all else, I thank my best friend and partner in life, Rich. Rich, I thank you for your guidance and support, your patience and your love. You bring me such joy and I am eternally grateful to you for sharing your life and wonderful family with me. I look forward to a long life with you full of many more amazing moments.

Two roads diverged in a yellow wood
And sorry I could not travel both
And be one traveler, long as I stood
And looked down one as far as I could
To where it bent in the undergrowth,
Then took the other as just as fair
And having perhaps the better claim;
Because it was grassy and wanted wear,
Though as for that, the passing there
Had worn them really about the same.
And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet, knowing how way leads onto way
I doubted if I should ever come back.
I shall be telling this with a sigh,
Somewhere ages and ages hence:
Two roads diverged in a wood, and I -
I took the one less traveled by,
And that has made all the difference.
- Robert Frost

TABLE OF CONTENTS

Signature Page . . . iii

Dedication Page . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . ix

List of Tables . . . xiii

Acknowledgments . . . xiv

Vita, Publications, and Fields of Study . . . xv

Abstract . . . xvii

I Problem Statement . . . 1

II Background . . . 7
A. Implementation of the Java Language Specification . . . 8
1. Access Rights . . . 8
2. Class File Format . . . 9
B. The Java Virtual Machine (JVM) . . . 10
C. Applets v/s Applications . . . 12
D. The Java Execution Model . . . 12

III Related Work . . . 17
A. Transfer Delay Reduction . . . 17
1. Compression . . . 17
2. Startup Delay . . . 19
B. Compilation Delay Reduction . . . 20
1. Continuous Compilation . . . 20
2. Adaptive Compilation . . . 21
C. Other Related Work . . . 24

IV Experimental Methodology . . . 27
A. Benchmarks . . . 27
B. Transfer Delay Optimization Methodology . . . 30
1. Compression . . . 31
2. Simulation Model . . . 33
3. Verification . . . 36
4. Transfer Delay Optimization Metrics . . . 38
C. Compilation Delay Optimization Methodology . . . 40
1. Compilation Environments . . . 40
2. Compilation Delay Optimization Metrics . . . 43

V General Solutions for Reducing Transfer Delay . . . 45

VI Transfer Delay Avoidance and Overlap: Non-strict Execution . . . 47
A. Design and Implementation . . . 48
1. Transfer Schedules . . . 50
2. Program Restructuring . . . 51
3. Implications on JVM Verification . . . 57
B. Results: Non-strict Execution . . . 58
1. Trusted Transfer . . . 59
2. Verified Transfer . . . 75
C. Summary . . . 76

VII Transfer Delay Avoidance and Overlap: Class File Prefetching And Splitting . . . 81
A. Design And Implementation . . . 82
1. Class File Prefetching . . . 82
2. Class File Splitting . . . 89
B. Results: Class File Prefetching And Splitting . . . 93
1. Trusted Transfer . . . 94
2. Verified Transfer . . . 100
C. Summary . . . 100

VIII Transfer Delay Avoidance: Dynamic Selection of Compression Formats and Selective Compression . . . 113
A. Design and Implementation . . . 114
1. Dynamic Compression Format Selection . . . 114
2. Selective Compression . . . 117
B. Results: DCFS and Selective Compression . . . 118
1. Dynamic Compression Format Selection . . . 118
2. Selective Compression . . . 127
C. Discussion . . . 139
1. DCFS for Variable Bandwidth Connections . . . 140
2. Prediction of Network Characteristics . . . 140
D. DCFS Extensions . . . 143
E. Summary . . . 144

IX General Overview on Reducing Compilation Delay . . . 147

X Compilation Delay Avoidance and Overlap: Background Compilation . . . 150
A. Design And Implementation . . . 151
1. Lazy Compilation . . . 152
2. The Effect of Lazy Compilation . . . 153
3. Background Compilation . . . 159
B. Results: Lazy and Background Compilation . . . 162
C. Summary . . . 166

XI Compilation Delay Avoidance and Overlap: Annotation-guided Compilation . . . 168
A. Design and Implementation . . . 169
1. Framework . . . 170
2. Annotation Optimizations . . . 173
3. Security of Annotations . . . 178
B. Results: Annotation-guided Compilation . . . 179
1. The Effect on Startup Time . . . 181

2. Local v/s Remote Execution 186
3. Annotation Size 188
C. Summary 189

XII Conclusions 190

XIIIFuture Directions 198

Bibliography 200

LIST OF FIGURES

I.1 Load delay for an average mobile Java program. 3

I.2 The impact of transfer delay on startup time for an average benchmark. . . . 5

I.3 The impact of compilation delay on startup time for an average benchmark. 6

II.1 Verified class file transfer example 1. 15

II.2 Verified class file transfer example 2. 15

III.1 General depiction of an adaptive compilation environment. 22

IV.1 General depiction of our result generation model. 33

IV.2 Example of transfer delay and overlap simulation 36

VI.1 Example Java Application 48

VI.2 Strict v/s Non-Strict Execution 49

VI.3 Example application code 52

VI.4 An example of a first-use call graph 52

VI.5 Restructured class files 53

VI.6 NSE transfer schedule for method-level execution. 54

VI.7 NSE transfer schedule for MLE and intra-class reordering. 54

VI.8 NSE transfer schedule for MLE and global method reordering. 55

VI.9 NSE transfer schedule for MLE, MR, and global data reordering. 56

VI.10 Resulting non-strict transfer delay for benchmarks Bit and Compress 60

VI.11 Resulting non-strict transfer delay for benchmarks Jack and JavaCup 61

VI.12 Resulting non-strict transfer delay for benchmarks Jess and Soot 62

VI.13 SCG transfer schedule construction for benchmarks Bit and Compress. . . . 65

VI.14 SCG transfer schedule construction for benchmarks Jack and JavaCup. . . . 66

VI.15 SCG transfer schedule construction for the Jess benchmark. 67

VI.16 Performance degradation for the Soot benchmark using static estimation. . . 68

VI.17 The effect of NSE on program startup (Bit & Compress) and modem link. . 69

VI.18 The effect of NSE on program startup (Jack & JavaCup) and modem link. . 70

VI.19 The effect of NSE on program startup (Jess & Soot) and modem link. 71
VI.20 The effect of NSE on program startup (Bit & Compress) using a T1 link. . . 72
VI.21 The effect of NSE on program startup (Jack & JavaCup) using a T1 link. . . 73
VI.22 The effect of NSE on program startup (Jess & Soot) using a T1 link. 74
VI.23 Difference in transfer delay for trusted and verified execution. 75
VI.24 Resulting verified transfer delay for the Bit benchmark. 76
VI.25 Resulting verified transfer delay for benchmarks Jack and JavaCup. 77
VI.26 Resulting verified transfer delay for benchmarks Jess and Soot. 78
VI.27 Average transfer delay (in seconds) using non-strict execution. 79
VII.1 The potential of class file prefetching. 82
VII.2 First-use execution order of class files in a sample application. 85
VII.3 Algorithm for finding the basic block to place the prefetch. 87
VII.4 Prefetch insertion example. 88
VII.5 Class file splitting example. 90
VII.6 Code splitting example. 92
VII.7 Percentage of execution time overlapped with transfer. 94
VII.8 Percent reduction in transfer size. 96
VII.9 Transfer delay for Bit & Compress using prefetching and splitting. 97
VII.10 Transfer delay for Jack & JavaCup using prefetching and splitting. 98
VII.11 Transfer delay for Jess & Soot using prefetching and splitting. 99
VII.12 Program startup (Bit and Compress) using a modem link. 101
VII.13 Program startup (Jack and JavaCup) using a modem link. 102
VII.14 Program startup (Jess and Soot) using a modem link. 103
VII.15 Program startup (Bit and Compress) using a T1 link. 104
VII.16 Program startup (Jack and JavaCup) using a T1 link. 105
VII.17 Program startup (Jess and Soot) using a T1 link. 106
VII.18 Difference in transfer delay for trusted and verified execution. 107

VII.19 Verified transfer delay (Bit and Compress) using prefetching and splitting. . 108
VII.20 Verified transfer delay (Jack and JavaCup) using prefetching and splitting. . 109
VII.21 Verified transfer delay (Jess and Soot) using prefetching and splitting. 110
VII.22 Average transfer delay using class file prefetching and splitting. 111
VIII.1 The Dynamic Compression Format Selection (DCFS) Model. 117
VIII.2 Pct. reduction in total delay due to DCFS for Antlr and Bit. 120
VIII.3 Pct. reduction in total delay due to DCFS for Jasmine and Javac. 121
VIII.4 Pct. reduction in total delay due to DCFS for Jess and Jlex. 122
VIII.5 Total delay in (log) seconds using DCFS for Antlr and Bit. 123
VIII.6 Total delay in (log) seconds using DCFS for Jasmine and Javac. 124
VIII.7 Total delay in (log) seconds using DCFS for Jess and Jlex. 125
VIII.8 Average reduction in transfer delay enabled by DCFS. 126
VIII.9 Pct. reduction in total delay due to selective compression (Antlr). 128
VIII.10 Pct. reduction in total delay due to selective compression (Javac). 129
VIII.11 Pct. reduction in total delay due to selective compression (Jlex). 130
VIII.12 Pct. reduction in total delay due to selective compression (Jasmine). 131
VIII.13 Pct. reduction in total delay due to selective compression (Bit). 132
VIII.14 Pct. reduction in total delay due to selective compression (Jess) 133
VIII.15 Pct. reduction in total delay (across inputs) for the Bit benchmark. 135
VIII.16 Pct. reduction in total delay (across inputs) for the Jess benchmark. 135
VIII.17 Summary of results using PACK compression as base case. 136
VIII.18 Summary of results using JAR compression as base case. 137
VIII.19 Summary of results using TGZ compression as base case. 138
VIII.20 Raw data (left) and cumulative distribution functions (CDF) (right). 141
X.1 Percent reduction in methods compiled. 154
X.2 Reduction in compilation time due to lazy compilation. 155
X.3 Overall impact of lazy compilation application performance. 160

X.4 Example scenarios of background compilation. 164
X.5 Summary of total time (in seconds) for the Train input. 165
X.6 Summary of total time (in seconds) for the Ref input. 165
XI.1 ORP O3 (Optimizing) Compilation Time Breakdown. 171
XI.2 The histogram used to find the "Hot" methods important for optimization. . 177
XI.3 Seconds of compilation delay reduced. 180
XI.4 Total compilation time for O3, O1, & annotated compilation. 181
XI.5 Speedup over optimized (ORP O3) total time due to annotated execution. . 182
XI.6 The effect of annotated execution on startup time (for Jack and JavaCup). . 183
XI.7 The effect of annotated execution on startup time (for Jess and Jsrc). 184
XI.8 The effect of annotated execution on startup time (for Mpeg and Soot). . . . 185
XI.9 Total compilation overhead for O3, O1, and annotated compilation. 186
XI.10 Speedup over optimized (ORP O3) total time (Remote classes only). 187
XII.1 Summary of the effect of our optimizations on load delay. 192
XII.2 Summary of the effect of our transfer delay optimizations on startup time. . 193
XII.3 Summary of the effect of our compilation optimization on startup time. . . . 194

LIST OF TABLES

IV.1 Description of Benchmarks Used. 28

IV.2 Static statistics on the benchmarks used in this dissertation. 28

IV.3 Dynamic statistics on the benchmarks used in this dissertation. 29

IV.4 Compression characteristics of the benchmarks using PACK, JAR, and TGZ. 32

IV.5 Description of the networks used in this study. 34

IV.6 Jalapeño compilation statistics. 43

IV.7 Compilation characteristics using the Open Runtime Platform. 44

VIII.1 Total delay in seconds for the network bandwidths studied. 116

VIII.2 Pct. difference in sizes of complete and selective compression. 134

VIII.3 Compression-on-demand with DCFS. 145

X.1 Raw execution time data. 156

X.2 Dynamic execution count of dynamically linked sites. 159

XI.1 The added size in kilobytes due to annotations. 188

Acknowledgments

The text of Chapter VI is in part a reprint of the material as it appears in the proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). The dissertation author was the primary researcher and author and the co-authors listed on this publication ([49]) directed and supervised the research which forms the basis for Chapter VI.

The text of Chapter VII is in part a reprint of the material as it appears in the proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). The dissertation author was the primary researcher and author and the co-authors listed on this publication ([48]) directed and supervised the research which forms the basis for Chapter VII.

The text of Chapter VIII is in part a reprint of the material that has been submitted to the 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC). The dissertation author was the primary researcher and author and the co-authors listed on this publication ([47]) directed and supervised the research which forms the basis for Chapter VIII.

The text of Chapter X is in part a reprint of the material as it appears in the Journal of Software: Practice and Experience, Software: Practice and Experience, Volume 31, Issue 8, pp. 717-738. The dissertation author was the primary researcher and author and the co-authors listed on this publication ([50]) directed and supervised the research which forms the basis for Chapter X.

The text of Chapter XI is in part a reprint of the material as it is to appear in the proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). The dissertation author was the primary researcher and author and the co-authors listed on this publication ([46]) directed and supervised the research which forms the basis for Chapter XI.

VITA

May 23, 1970	Born Monticello, IN
1988	High School Diploma, St. Joseph's Academy Brownsville, TX
1996	Internship, NASA Goddard Space Flight Center Greenbelt, MD
1996	B.S. California State University Northridge, CA
1996	Platinum Solutions, Inc. Inglewood, CA
1997-1998	Teaching Assistant, Computer Science and Engineering Department, University of California San Diego, CA
1998	Internship, Microsoft Research Redmond, WA
1998	M.S., University of California San Diego, CA
1999	Internship, IBM T. J. Watson Research Center Hawthorne, NY
2001	Doctor of Philosophy, University of California San Diego, CA

PUBLICATIONS

"Using Annotation to Reduce Dynamic Optimization Time." Authors: C. Krintz and B. Calder. To appear in the Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2001.

"Using JavaNws to Compare C and Java TCP-Socket Performance." Authors: C. Krintz and R. Wolski. To appear in the Journal of Concurrency and Computation: Practice and Experience, 2001.

"NwsAlarm: A Tool for Accurately Detecting Resource Performance Degradation." Authors: C. Krintz and R. Wolski. To appear in the Proceedings of the IEEE/ACM Symposium on Cluster Computing and the Grid (CCGRID2001), May 2001.

"Reducing the Overhead of Dynamic Compilation." Authors: C. Krintz, D. Grove, V. Sarkar, and B. Calder. In the Journal of Software: Practice and Experience, Software: Practice and Experience, Volume 31, Issue 8, pp. 717-738, Dec. 2000.

"JavaNws: The Network Weather Service for the Desktop." Authors: C. Krintz and R. Wolski. In the Proceedings of JavaGrande, Oct., 2000.

"Reducing Transfer Delay Using Java Class File Splitting and Prefetching." Authors: C. Krintz, B. Calder, and U. Hölze. In the Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Nov. 1999.

"Running EveryWare on the Computational Grid." Authors: R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. In the Proceedings of Supercomputing, Oct., 1999.

"Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs." Authors: C. Krintz, B. Calder, H. B. Lee, and B. Zorn. In the Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct., 1998.

"Cache-Conscious Data Placement." Authors: B. Calder, C. Krintz, S. John, and T. Austin. In the Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct., 1998.

"AGAVE: A Visualization Tool for Parallel Programming." Authors: C. Krintz and S. Fitzgerald. In the Proceedings of The International Association of Science and Technology for Development (IASTED), Oct., 1995.

Field Of Study: Computer Science

ABSTRACT OF THE DISSERTATION

Reducing Load Delay to Improve Performance of Internet-Computing Programs

by

Chandra Krintz

Doctor of Philosophy in Computer Science and Engineering

University of California, San Diego, 2001

Professor Bradley Calder, Chair

Internet computing has been enabled by a mobile program execution model in which architecture-independent programs transfer to where they will be executed. The Java language model is designed to implement mobile execution by transferring bytecodes to a virtual machine which translates them into native machine instructions and then executes them on the target site.

Implementing Java's mobile execution model efficiently has proved challenging for two reasons. First, the time required to transfer program code from the place where it is stored to the Java Virtual Machine (JVM) that will execute it is perceived by the program's user as execution delay. Current levels of deliverable Internet performance can cause this delay to be substantial. Second, once the code has arrived it must either be interpreted or compiled "just-in-time" for its execution. Just-In-Time (JIT) compilation offers improved execution speed over interpretation by exploiting the opportunity for compile-time optimizations, but the compilation time is also perceived by the program's user as execution delay.

In this thesis, we define *load delay* as the unification of these two sources of overhead: transfer delay and compilation delay. We detail the causes of, describe the existing technology that contributes to, and show the degree to which load delay degrades performance of Internet-computing applications. We show that solutions to the problem of load delay in these *mobile* programs can be attacked in one of two ways regardless of the source: through *avoidance* and *overlap*. Avoidance is achieved by eliminating all or part of the cause of load delay and overlap by performing useful work concurrently with the delay. Both have the potential to reduce the effect of load delay and to improve performance of mobile programs. We present numerous solutions to load delay that implement either avoidance, overlap, or both. Our results show that both sources of load delay can be reduced substantially given currently available remote execution technology. In addition, our results suggest modifications that can be made to existing technology to further improve performance of Internet-computing applications.

Chapter I

Problem Statement

The Internet has been used traditionally for the provision of access. Its world-wide system of networks facilitates access to a diverse abundance of information and resources. An alternate use of the Internet that has become popular recently, is as a computational entity. The fundamental difficulty associated with such use is how to efficiently and effectively employ the processing power that is available and connected via the Internet. One methodology that has been developed to solve this problem is *remote execution* in which programs transfer over the network from the machine at which the code and data is stored to a target site for execution.

We refer to remotely executed programs as *mobile programs*. Commonly, these programs are transferred in an architecture-independent representation that enables portable execution at heterogeneous target sites. However, such formats must be converted to a native representation to enable execution at the destination. This translation is commonly performed by a compiler; a piece of software that not only converts programs to the native format but performs optimization on the code to enable efficient execution.

Performance of mobile programs is increasingly limited since it includes the time for transfer and compilation, as well as for execution; since transfer and compilation both occur while the program runs. We refer to these sources of overhead collectively as *Load Delay*. The widening gap between network and processor performance, highly variant network conditions, and the optimization complexity required for efficient execution speeds, make it exceedingly difficult to maintain acceptable mobile program performance. We propose to reduce the effect of load delay and to improve performance of remotely executed programs.

To summarize, we define the following terms that we use throughout this dissertation:

- *Remote Execution*: A methodology for the use of the Internet as a computational entity in which program code and data transfer from the machine at which it is stored to a destination (target) machine and execute upon arrival.
- *Native Code*: An architecture-*dependent* program format.
- *Mobile Program*: A program that is transferred in an architecture-*independent* format for remote execution.
- *Transfer Delay*: The time for network transmission of code and data from the source to the destination during remote execution.
- *Compilation Delay*: The time for translation (and possibly optimization) of a mobile program to native code by the destination machine as required for remote execution.
- *Load Delay*: The unification of two sources of remote execution overhead: transfer delay and compilation delay.

We use the Java Programming Language [28] as the experimental foundation of this dissertation work due to its remote execution functionality and its pervasive use in Internet computing. However, our techniques are general and can be used for other mobile program representations, e.g., MSIL, a.k.a. DotNet [40]. Performance of mobile Java programs is negatively affected by both sources of load delay. Transfer delay is experienced by the executing program since program files are loaded over the network as needed (on-demand). The execution stalls while the request completes and the non-local file is transferred. Compilation delay is imposed each time code within a file is invoked for the first time; execution is further interrupted waiting for compilation.

We exemplify the degree to which load delay impacts the performance of an average Java program in Figure I.1. The graph depicts load delay (both transfer and compilation overhead) as a function of network bandwidth. Load delay measurements consist of the time for transmission of the (non-library) code and data, the time to request program files required for execution, and the time for compilation of all executed methods. The average Java program used for this figure accesses 70 non-local classes requiring 178 kilobytes to be transferred, and compiles 238 methods (totaling 3 seconds) using the Open Runtime Platform (ORP) [15] as the Java execution environment. For a slow link, like that of a modem, load delay imposes almost 56 seconds. For a fast link, e.g., T1 (1Mb/s), load delay imposes over 10 seconds for an average benchmark. We have measured cross-country Internet performance and our results indicate that the bandwidth available to a common Java application falls between the 0.03Mb/s modem range and a T1 link (1Mb/s). In addition, Internet performance is highly variable and a single connection can experience this range of performance over a short duration.

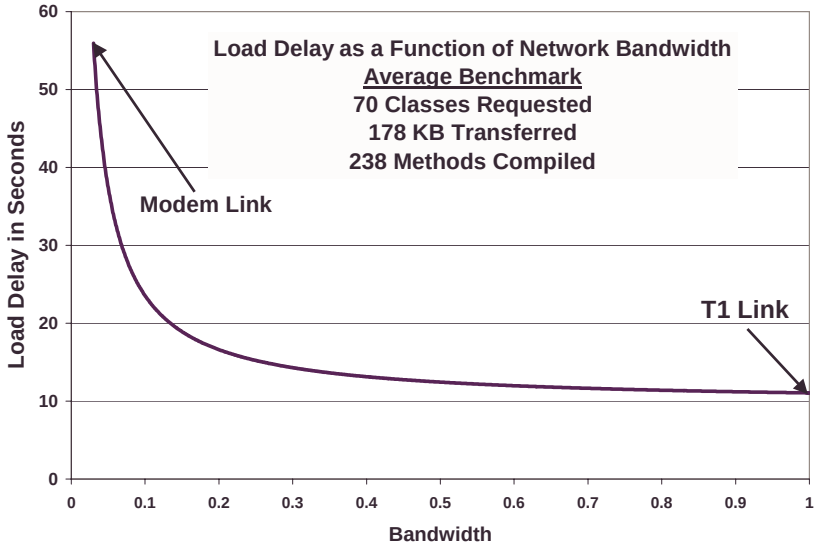


Figure I.1: Load delay for an average mobile Java program.
The figure shows load delay in seconds (x-axis) as a function of network bandwidth (y-axis). Compilation delay accounts for 3 seconds of total load delay in this graph. The remainder is due to transfer delay (the time for request and transmission of the program).

Figures I.2 and I.3 depict the effect of load delay on program startup time. The graphs show the average cumulative delay (y-axis) that is experienced during program execution. The x-axis is the percent of execution time completed by the average program (no transfer or compilation delay is included in this value). The average execution time for the programs used for this figure is 49 seconds. The two graphs in Figure I.2 are for transfer delay: the top is for a modem link (0.03Mb/s bandwidth) and the bottom is for a T1 link (1Mb/s bandwidth). This data assumes that a request for each class costs 100ms, a common (based on empirical data) cross-country round-trip time value. The graph in Figure I.3 is the average cumulative delay due to compilation, the second source of load delay. Each graph is read by taking an (x,y) position on the function; y seconds of delay (transfer or compilation) occurs during the first x% of program execution.

The function for the modem link (top graph in Figure I.2) indicates that 39 of the 56 seconds of transfer delay occur in the first 10% (5 seconds) of execution time and 90% of all transfer delay (50 seconds) occurs in the first 40% (44 seconds) of program execution. Similarly for the T1 link (bottom graph in Figure I.2), 5 of the 8 seconds of transfer delay is incurred during the first 10% of program execution and 90% (7 seconds) of the transfer delay occurs

in the first 30% of program execution (14 seconds). Compilation overhead is also incurred at program startup as shown by the graph in Figure I.3. The function indicates that 1.8 seconds of the 2.6 seconds (69%) of compilation delay occur during the first 10% of execution and 90% of it occurs in the first 30% of program execution. Almost 70% of *all* delay (transfer or compilation) occurs in the first 10% of program execution. By reducing the effect of load delay (transfer and compilation overhead) we improve the progress made at program startup as well as throughout overall execution.

Load delay is incurred at program startup and intermittently (distributed throughout execution) and substantially degrades mobile program performance regardless of where it occurs. Much work in industry as well as in academic research has focussed solely on the reduction of startup delay [53, 74, 78]. In addition, other work investigated the effect of time-sharing systems on productivity (e.g., see [21]), and concluded, among other things, that intermittent interruption reduced a user's perception of performance as well as their productivity. To improve mobile program performance, the effect of load delay must be reduced. With this work, we attack the problem of load delay in Internet-computing programs through the design and implementation of compiler and runtime techniques.

In this dissertation, we describe the execution model we assume for Internet computing, identify and detail the sources of load delay and articulate the degree to which load delay degrades program performance. Solutions to performance degradation resulting from load delay enable *overlap* of delay with useful work, *avoidance* of delay by directly limiting the cause of the delay, or both. The goal of our work is to develop compiler and runtime optimizations that both mask and avoid load delay imposed on Internet-computing applications. We present solutions first for transfer and then compilation delay and summarize the performance benefits in terms of load delay: The unification of these two sources of overhead.

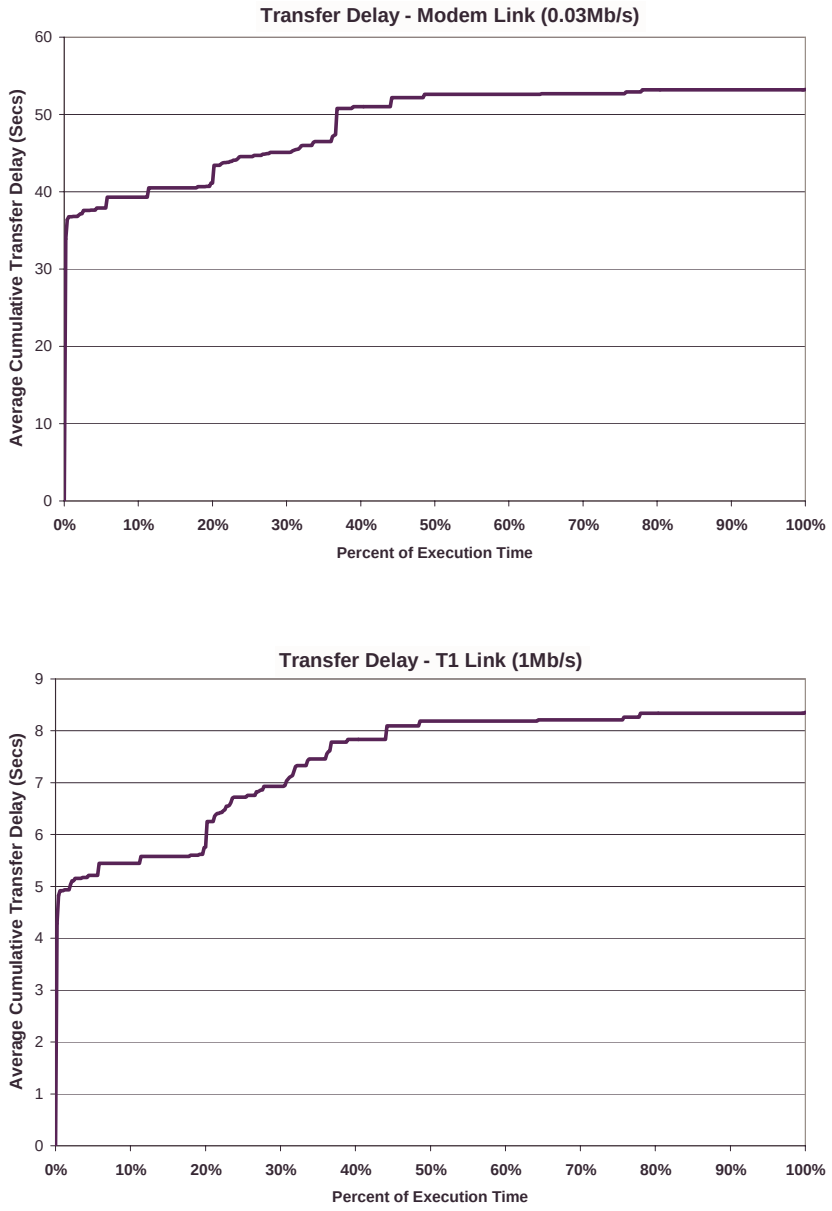


Figure I.2: The impact of transfer delay on startup time for an average benchmark. The graphs show the average cumulative delay (y-axis) that is experienced during program execution. The x-axis is the percent of execution time completed by the average program The average execution time for the programs used for this figure is 49 seconds. The top two graphs are for transfer delay: The top graph is for a modem link (0.03Mb/s bandwidth) and the bottom is for a T1 link (1Mb/s bandwidth). Transfer delay consists of time for request and transmission of class files for remote execution of an average Java program. Each graph is read by taking an (x,y) position on the function; y seconds of transfer delay occurs during the first x% of program execution. Almost 70% of *all* delay occurs during the first 10% of program execution.

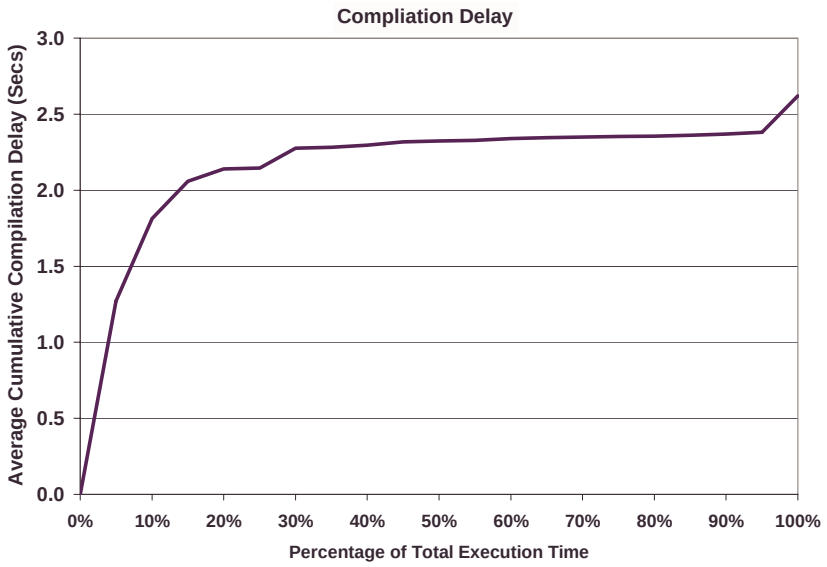


Figure I.3: The impact of compilation delay on startup time for an average benchmark. The graph shows the average cumulative compilation delay (y-axis) that is experienced during program execution. The x-axis is the percent of execution time completed by the average program The average execution time for the programs used for this figure is 49 seconds. compilation delay is the time spent compiling and optimizing the programs. Each graph is read by taking an (x,y) position on the function; y seconds of compilation delay occurs during the first x% of program execution. Almost 70% of *all* delay occurs during the first 10% of program execution.

Chapter II

Background

The goal of this dissertation is to improve the performance of applications that are remotely executed over the Internet. We facilitate experimentation in this study with the Java programming language [28] and execution environment [59]. Java programs implement remote execution with the Java applet execution model. Other languages also use remote execution for Internet computing but we restrict our studies to Java alone due to the duration of its use, its popularity, design, and current infrastructure availability. However, these other languages and their runtime implementations are similar to the decisions made for Java and hence, impose load delay in similar ways.

In this chapter, we detail the functionality and implementation of the Java language that is intended to support *mobile* (remote) execution. Specifically, we describe the existing state of Java technology in terms of its object-oriented framework, execution engine, and general execution model. Each of these language and runtime features impact load delay and the performance of Java applications for Internet computing.

The object-oriented abstractions that are implemented by the Java language facilitate file-level execution and transfer granularity. This allows remote files to be loaded (and thus transferred) independently as required by the executing program. Therefore, only remote files that are actually used are transferred. This reduces transfer delay since unused class files are not transferred. In this dissertation, we address the transfer delay that remains in this transfer and execution model and present optimizations that further reduce the effect of it.

A Java program is stored in an architectural-independent format called bytecode for portability purposes. A Java Virtual Machine (JVM) executes a Java program by converting

bytecode to the machine instructions of the underlying hardware. This conversion can be performed by an interpreter, an instruction-by-instruction translator, or by a compiler. We focus on the latter since it enables substantially faster execution times and exposes optimization opportunities unavailable to interpretation. However, compilation imposes load delay. In this chapter, we describe how compilation is performed on Java bytecode and how its use contributes to overall load delay.

In summary, we use this chapter to explain the object-oriented features of the Java programming language that effect load delay and impose constraints on modification of Java programs and execution environments. We then describe the execution engine and Internet-computing model for mobile Java application execution and detail the inherent performance limitations in terms of load delay.

II.A Implementation of the Java Language Specification

Java is an object-oriented [62, 75] language. Object-orientation is a programming methodology in which data and code for its manipulation are encapsulated together. A Java program consists of multiple files that define such encapsulations, called *class files*. A runtime instance of a class file is called an *object*. Data are referred to as fields, and code as methods; collectively they are called *class members*. Such modularity is ideal for Internet-computing programs since class files can be used as the unit of transfer and execution. That is, since the code and data contained in a class file is related, a class can be transferred and code within it can be executed independent of other class files.

II.A.1 Access Rights

An object-oriented language, among other things, provides functionality for the restriction of access to data members (data hiding). This enables security for the remotely-executed program at the destination; other class files with possibly destructive intent are unable to access class files, modify fields, or invoke methods for which access is restricted. We detail the functionality of Java access modifiers here since some of the techniques we present in this dissertation modify Java class files. In the course of describing such optimizations, we also address the security implications of our techniques in terms of the existing Java implementation described here.

In Java, access restriction is enabled through the inclusion of key-words [28]. in class

or member definition Members that are visible by all object instances of a class are called *static* or *class* members; those visible only to a single object are *non-static* members. Objects are allocated and non-static members are created and initialized within the object through the use of special methods called *constructors* defined in the class. Due to this restriction in visibility, or scope, static members are only able to access other static members; non-static members can access static members as well as other non-static members that are part of the same object instance. For further visibility control, the *public, private, and protected* key-words can be used. The public specifier indicates that any other class or object can access the member (according to the class and instance access rules). The private specifier restricts member access to only the other members of the class itself or an object instance of the class. Protected indicates that only the class itself (and its object instances) and any subclasses of class (and their object instances) can access the member. Subclasses are a standard feature in object oriented programming and we refer the reader to related texts in [72, 75, 22] for additional information on this, as well as, further description of these and other object oriented constructions.

Java implements a mechanism called *packages* to enable restriction of access to a group of files. The keyword package followed by the name of the package must be included in each file contained in the package. If there is no package indicated, the file is included in the default, *unnamed* package. Any members for which there is no access specifier explicitly set is considered to have package access. Package access restricts access of the member to be performed by only members from classes (and their instances) that are defined in the same package. In addition, protected members are accessible by class and object members in the same package. Note that this implies that class and instance members in the default, unnamed package can access all non-public members in any other class in the default, unnamed package.

II.A.2 Class File Format

A Java program is composed of separate files, each containing a class definition. Class files in Java source code are compiled into an architecture-independent representation called *bytecode*. Bytecode enables the "write-once-run-anywhere" methodology [72]. A Java program can be written by a programmer and it (in theory) can be executed on any architecture. Such a representation is ideal for program execution on the heterogeneous resources connected by the Internet. However, such abstraction and portability comes at a price. Since the representation contains verbose symbolic information to facilitate conversion to and execution on different hardware, the size of an application is much larger than that of a machine-specific binary.

Increased transfer size imposes transfer delay experienced by a mobile application as substantial startup delay and intermittent interruption. Transfer delay is one source of load delay that we attack with the work in this dissertation.

The bytecode format is a stream of 8-bit bytes that encode the information required for the secure [1] execution of a Java program. Method source code instructions take a pseudo-assembly code form. We refer to non-method bytecode as *Global Data* throughout this thesis. The global data and method code within the class file are organized into multiple data structures. The data structures for the global data include a constant pool which is a table of variable-length structures that represent string and other constants as well as names and types of all classes and members referred to by the class. The size of the constant pool can be quite substantial since all of the symbolic information is explicitly represented in Unicode [28], a character encoding that subsumes the ASCII standard to enable the inclusion of foreign language characters.

In addition to the constant pool, other global data structures include magic and version numbers, super (inheritance [62]) class information, class access rights, and information (access, names, types, size) about all fields and methods the class defines. The class file also contains attributes which are extra information about the class files or its members. For example, method code is represented by an attribute of a method member. Attributes are named and are commonly used for the inclusion of debugging information. Any attribute in a class file that is undefined is ignored by the runtime system [28]. In part of the research presented in this dissertation we exploit this specification feature so that we are able to include user-defined attributes within the class files. Specifically, we use this attribute structure to carry annotations that guide the compilation environment and reduce load delay. Since attributes are ignored if unrecognized by the Java Virtual Machine (JVM), our class file modifications remain backward compatible with existing JVM technology (systems that are not annotation-aware).

II.B The Java Virtual Machine (JVM)

The single requirement for execution of a Java class file in bytecode format is that there be a Java execution environment, or abstract computing machine, called a *Java Virtual Machine* (JVM) on the architecture upon which a Java program is to be executed. The JVM performs many functions including the conversion of bytecode to native code and the initiation

[1] More information about secure Java bytecode execution can be found in [59] and in our later discussions about bytecode verification.

of execution on the underlying hardware. Other functions include memory allocation and deallocation (garbage collection) and Java to native thread mapping and scheduling [59].

To convert bytecode to native machine code, the JVM originally only used interpretation in which each bytecode instruction is individually translated to equivalent native instruction(s) and executed. Interpretation is very simple to implement since it is a direct instruction-by-instruction translation. In addition, execution, as experienced by the user, makes immediate progress since the translation of a single bytecode instruction is performed very quickly. However, interpreted program execution time is notoriously slow. Since translated code is not stored in memory once executed, it is not reused; as such, instructions are re-translated when executed repeatedly. In addition, and more importantly, the native code generated by interpretation is very poor since the interpreter only considers a single instruction at a time. As such, there is redundancy and inefficiency in the resulting code.

To overcome the performance limitations interpretation imposes, the next generation of Java execution systems [79, 3, 15, 34] employ just-in-time (JIT) compilation. These new JVMs dynamically compile the bytecode stream of a method into machine code prior to executing the entire method. That is, each time a method is invoked for the first time, execution of the program stops and the method is compiled. Commonly, a single method is compiled at a time. The resulting execution performance is higher than for interpreted bytecodes since native method code is stored and reused each time a method is invoked repeatedly. In addition, compilation of an entire method at once exposes opportunity for optimization. Since the dynamic compiler analyzes multiple instructions at once and thus, more static program behavior is visible, a more compact and efficient set of instructions can be selected. In addition, optimizing algorithms can be applied to the code to further improve efficiency and overall program execution times substantially.

However, since compilation (and optimization, if any) must occur during execution of the program, there is an overhead incurred each time a method is invoked for the first time. This overhead cost must be amortized by the improvement in execution speed for optimization techniques to be practical and feasible. Since such optimization can be time consuming, this amortization proves to be a difficult task [3, 15]. Compilation overhead is another source of load delay since it occurs during runtime and imposes startup and intermittent interruption cost. We present multiple optimization for the reduction of the compilation portion of load delay in this thesis.

II.C Applets v/s Applications

There are two types of Java programs, applets and applications. In this section, we distinguish between the two since the former is the execution model for remote execution the the one we exploit and extend in this dissertation. An applet is a program that can be executed using one of two execution environments: an Internet browser, e.g., Netscape [63], Microsoft Explorer [41] or an appletviewer [61]. A browser is a piece of software that enables access to and visualization of distributed, collaborative, and hyper-media information via the httpd [38] protocol. Browsers commonly contain built-in JVMs for the execution of Java applets. An appletviewer is a tool that is packaged and distributed with JVM software that enables applet execution outside of the context of a browser; it is commonly used for debugging purposes.

Either execution environment uses a JVM to initiate program execution using a special entry point called <init> in the starting class file. When an applet is invoked via a browser (by a internal JVM) from a machine across a network, the applet is downloaded from its storage site to the local machine running the browser. The applet is then executed on the local hardware resources. Applets are highly restricted in terms of the actions it is allowed to perform. This is due to the Java language security policy [64] which is the subject of much research [5, 85]. Some examples of applet restrictions include disk access, communication to hosts other that the one from which the applet was downloaded, and program invocation.

Java applications are executed by invoking a JVM at the command line and passing the program name as a command line argument. The entry point of a Java application is the "main" routine which is similar to entry points of programs written in other languages. Applets and applications, once invoked, execute in the same manner. We describe this execution model in the next section. Since this work is focused on improving the performance of Internet-computing applications, we consider only Java applet invocation. We heretofore refer to Java applets as programs.

II.D The Java Execution Model

In this section, we describe the execution model that exists today for the execution of mobile Java programs. We describe how remote execution is implemented in the runtime system of the JVM. It is this system that we empirically measure and use as the base case for comparison of the techniques we present in this dissertation. Currently, Java programs are loaded into memory for execution using *dynamic class file loading*. The dynamic class loading

mechanism pauses the executing program and converts a newly accessed class file to the internal JVM class file representation. This occurs the first time the class is accessed by the execution.

Class files can be stored on and loaded from disk (located locally or across a network) as individual files or together as an archive that is possibly compressed. The most common archive/compression format for Java programs is the *Java archive (jar)* which uses the zip [67] compression format. Remote Java programs archived as jar files (that must be transferred over a network) commonly contain all classes in a program. The jar file is stored in memory and class files are decompressed from the archive individually upon first access by the dynamic class file loading mechanism. Any files not found in the archive are searched for locally using the JVM path list called the *classpath*. Class files that are not found locally are requested from the machine from which non-local program files (if any) were downloaded. A remote file (if it exists on the remote machine) is transferred upon receipt of the request. The cost of the request is part of the transfer delay imposed on the program. A small packet incurs the network latency for its round trip transfer. Transfer delay also includes the time for the transmission of the remote file from the machine on which it is stored to the execution site since execution is unable to continue until the class has transferred to completion.

Once the accessed class is loaded, the executing program is allowed to proceed until the next, as-yet-unaccessed class is accessed. Types of class access include read and write of fields, method invocation, and object creation. Class files can also be accessed by the JVM for verification of Java security policies.

Verification is a mechanism in the JVM which checks (prior to executing) that loaded class files are well formed. That is, that the files implement a set of constraints (described in the Java language specification [28]) and behave as expected to not adversely exploit the runtime system (and local machine). Verification ensures that the desired constraints hold on the class file it is attempting to incorporate. Examples of verification checks include structural validation, operand stack underflows/overflows, and validation of argument types. Since type checking is also performed during verification, all classes in the type hierarchy of the newly loaded class file must be loaded and verified to contain the expected types. For such cases, additional class files (not necessarily used by the executing program) must be loaded. Verification can be used on non-local class files only, on all class files, or on no class files as configuration or invocation options to the runtime system. All non-local class files executed using most browsers are commonly verified by default.

Verification is an important mechanism in the execution model for mobile Java pro-

grams since it impacts program transfer. Verification in some cases requires that class files be transferred regardless of whether or not they are used. It also effects the order in which class files are transferred. Both effect decisions about optimization of class files for transfer delay reduction. In this dissertation, we address the implications of program and runtime modification with and without verification.

We next explain the JVM version 1.2.x verification mechanism, the most current version and the one we assume in this thesis, 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 II.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 to verify that the type of 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 II.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 constructor, `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` since the transfer unit of a Java program is the class file.

Verification is an important factor in the performance of mobile Java programs. Since it requires that class files other than those executed, it causes additional transfer delay for each such class that is non-local. Class files required for verification may never be accessed by the executing program and therefore are purely transfer and verification overhead. In this dissertation, we consider optimizations that modify the verification mechanism to reduce transfer delay while continuing to ensure secure execution enabled by it.

```
public class X {
  public static void main(String args[]) {
    ZSuper varZ = null;
    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 {
  meth() {
    return 15;
  }
}
class ZSuper extends ZSuperSuper {
  meth() {
    return 10;
  }
}
class ZSuperSuper {
  meth() {
    return 5;
  }
}
```

Figure II.1: Verified class file transfer example 1.
This is the first Java example to demonstrate class file transfer and its interaction with verification when using superclasses. When class X is first used during execution, verification of it requires that class Z and class ZSuper be local to check that the type use of variable varZ is correct. When class Z is first accessed, its verification requires that both ZSuper and ZSuperSuper be local for type checking. Verification can substantially increase transfer delay since classes that are unused by the executing program are commonly required.

```
class A {
  public B varB;
  A() { . . . }
  main( . . . ) {
    bar();
    varB = new B();
    foo();
    varB.foo();
  }
  foo() { . . . }
  error() { . . . }
}
```

```
class B {
  public C varC = null;
  public int var1;
  private int var2;
  protected int var3;
  B() {
    var1 = var2 = var3 = −1;
  }
  bar () {
    (varC = new C()).foo();
    var2 = 0;
  }
}
```

```
class C {
  C() { . . . }
  foo() { . . . }
}
```

Figure II.2: Verified class file transfer example 2.
This is the second Java example to demonstrate class file transfer and its interaction with verification. When class A is first accessed, no other class files are required for its verification. Similarly, class B and C are transferred on-demand.

Each of the mechanisms described in this chapter impact the performance of remotely-executed Java programs. The object model enables applications to be partitioned to enable class file loading on-demand so that only accessed classes are loaded transferred (if non-local). This is important for the reduction of transfer delay the first source of overhead in load delay. However, dynamic class file loading still incurs substantial overhead for the transfer of non-local class files. Due to the execution model, this overhead can be experienced all at once at program startup (when the application is sent as an archive) or intermittently throughout execution (using dynamic class file loading). Neither is a good solution since the delay severely limits mobile program performance and hence, the wide use and acceptance of the Internet as a computational entity. We propose optimizations in this thesis that reduce the effect of this transfer delay. We also address any security implications due to modifications of class files and JVM mechanisms (described here) that arise from the incorporation of our techniques.

In addition, we describe existing execution engine (JVM) technology in this chapter. The JVM converts the architecture-independent bytecode format of class files to native machine code through compilation. Compilation is used over interpretation due to its potential to substantially reduce execution time of the resulting executable. Compilation exposes opportunity for optimization and at the same time imposes overhead since it occurs dynamically, while the program is executing. Code is compiled method-by-method (method-level) the first time a method is loaded (invoked), i.e., Just-In-Time (JIT). This introduces load delay in the form of execution interruption. Compilation delay is the second source of overhead (in load delay) that we attack and reduce in this dissertation research. Our goal is to achieve optimized execution times with greatly reduced compilation delay. With techniques to reduce the effect of both transfer and compilation delay, the two sources of load delay, we can substantially improve mobile program performance.

Chapter III

Related Work

Our work focuses on reducing the effect of load delay on Internet-computing applications. Work related to this includes research for the reduction of the two sources of overhead that collectively make up load delay: Transfer delay and compilation delay. As such, we divide this chapter into a discussion of the related work for each of these sources separately. Following this, we identify other existing techniques that we exploit and extend to reduce the effect of load delay.

III.A Transfer Delay Reduction

Many research and industrial groups have made a concerted effort to reduce the effect of transfer delay on programs that are remotely executed. The most common technique is compression; the compact encoding of files to reduce the amount transferred. In this section, we first detail related compression and other research that reduces overall transfer delay. Following this, we describe related work that focuses solely on reducing transfer delay to improve program startup time.

III.A.1 Compression

In this dissertation, we advocate maximizing the overlap between execution and transfer and avoiding transfer (by transferring less) to reduce the effect of transfer delay. Related work for the reduction of overall transfer delay implements the latter. The primary transfer delay avoidance mechanism is compression; such techniques are complementary to those presented

in this thesis. Compression reduces the amount of data transferred by compactly encoding the file that is to be transferred. Once at the destination, the files are decoded for use. Several approaches to compression have been proposed to reduce network delay in Internet-computing environments which we now discuss.

Ernst et al. [23] 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 et al. describe 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 [24]. 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 [25, 57, 87].

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 [87]. 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 [25]. In recent work, Lefurgy et al. 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[57].

The Jax [82] utility reduces Java class file size via renaming, name compression, static optimizations, and other techniques. A jar file (zip compression) is constructed from optimized class files that are reachable by the application, according to static analysis. Another Java-specific compression utility has been proposed by Pugh in [71] in which he describes a wire format that reduces a collection of individually compressed class files 50% to 20% the size of compressed jar files on average. 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. The compression algorithm determines when sharing can be performed within an application so that additional redundant information is eliminated.

III.A.2 Startup Delay

Startup delay can also be reduced through transfer avoidance. One way in which this can be done is to ensure that only those methods that will be executed are transferred across the network. Sirer et al. describe such an optimization in [74]. In this work, Java class files are repartitioned to enable more effective utilization of the available bandwidth during transfer. Profile information is used to identify methods that are unused during instrumented execution. Unused methods are then split out into new class files. Using existing Java class file loading techniques, the class files containing the methods that are used during execution are transferred. If methods predicted as unused and split out are not used during actual execution, the methods are never transferred and the transfer delay is reduced. If such methods are used, then the class files containing them are transferred.

In this thesis work we present an optimization, called class file splitting, that is a similar technique. The projects were implemented independently and concurrently. Sirer et al. describe a different implementation in which a single class file is created for all unused methods in a class. In our work, unused methods are each split out into separate classes. This reduces the overhead associated with transfer of a split class when usage predictions are incorrect. We detail this in Chapter VII. In addition, Sirer et al. do not consider the impact of the Java verification mechanism. Verification can cause additional class files to transfer; the degree to which this effects this related work is unclear since they only measure unverified, or trusted, transfer. Lastly, this related work does not address the security implications of the optimization presented.

Java class file splitting was originally described by Chilimbi, et al., in [14] 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 [66]. As a side-effect, we achieve advantages in memory performance using our class file splitting technique. However, since memory performance is not the focus of this thesis, we did not investigate it and do not provide measurements for it.

In other work by Lee et al. [53], the authors describe a technique that decreases startup time for x86 binaries by packing application code pages more effectively for remote execution. Programs are reordered into contiguous blocks according to predicted use of procedures. Predic-

tion is guided by usage profiles that are collected via off-line, instrumented execution. 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%. This work considers binary files only and do not suggest or implement extensions for Java bytecode. In addition, it requires a special execution engine to decode the packed files for execution at the destination.

III.B Compilation Delay Reduction

The performance of remotely executed programs is greatly improved through the use of dynamic compilation over interpretation. However, the compilation process is more complex and imposes longer, intermittent delays during execution since execution must pause waiting for compilation to complete. To compensate for compilation overhead, many systems use a combination of a very fast interpreter and an optimizing compiler or two compilers (one simple and very fast and the other optimizing).

The first compilation system we describe is continuous compilation [68] in which compilation is overlapped with interpretation. The other systems use adaptive compilation to amortize the cost of optimization by optimizing only frequently-executed pieces (*hot spots*) of the program.

III.B.1 Continuous Compilation

A project that attempts to improve program responsiveness in the presence of dynamic compilation is continuous compilation [68]. Continuous compilation overlaps interpretation with Just–In–Time (JIT) compilation. A method, when first invoked, is interpreted. At the same time, it is compiled on a separate thread so that it can be executed on future invocations. The authors of this work extend this idea to Smart JIT compilation: on a *single* thread, interpret *or* JIT compile a method upon first invocation. The choice between the two is made using profile or dynamic information. Interpretation and JIT compilation overlap is also used in the Symantec Visual Cafe JIT compiler, a Win32 JIT production compiler delivered with some 1.1.x versions of Sun Microsystems Inc. Java Development Kits [80].

III.B.2 Adaptive Compilation

Figure III.1 depicts the Java execution model in an adaptive compilation environment. Adaptive compilation systems first interpret or fast-compile (compilation with little or no optimization) a method when it is initially invoked. During this process the code is instrumented to measure various performance characteristics (invocation count, execution duration, loop count, etc.). When measured values reach a given threshold the method (or method-piece) is optimized using an optimizing compiler. Compilers in the system are invoked for initial method invocation (demarcated with [1] in the figure), by the on-line measurement system (profiler), or by the class loader (demarcated with [2] in the figure). The class loader can load classes from the local disk or across a network.

In the following sections, we first describe Self, an adaptive compilation system for programs written in the Self language. We then detail three popular, existing adaptive compilation systems for Java programs. All of these systems attempt to improve program responsiveness by only compiling those program sections that effect execution time as indicated by the profiler.

Self

In [33], Hölzle et al. describe an adaptive compilation system for the Self language that uses a fast, non–optimizing compiler and a slow, optimizing compiler. The fast compiler is used for all method invocations to improve program responsiveness. Program hot spots are then recompiled and optimized as discovered. Hotspots are methods invoked more times than an arbitrary threshold. When hot spots are discovered, execution is interrupted and the method, with possibly an entire call chain, is recompiled and replaced with optimized versions.

Open Runtime Platform (ORP)

A dual-compiler, adaptive compilation system for Java, called the Open Runtime Platform (ORP), was recently released as open source by the Intel Corporation [65]. The first compiler (O1) provides very fast translation of Java programs [1] and incorporates a few very basic bytecode optimizations that improve execution performance. The second (O3) compiler performs a small number of commonly used optimizations on bytecode and an intermediate form to produce improved code quality and execution time. O3 optimization algorithms were implemented with compilation overhead in mind, hence only very efficient algorithms are used [16]. Optimizations that are implemented at the time of this work include constant and copy propagation, global register allocation, dead code elimination, and basic loop optimizations. These

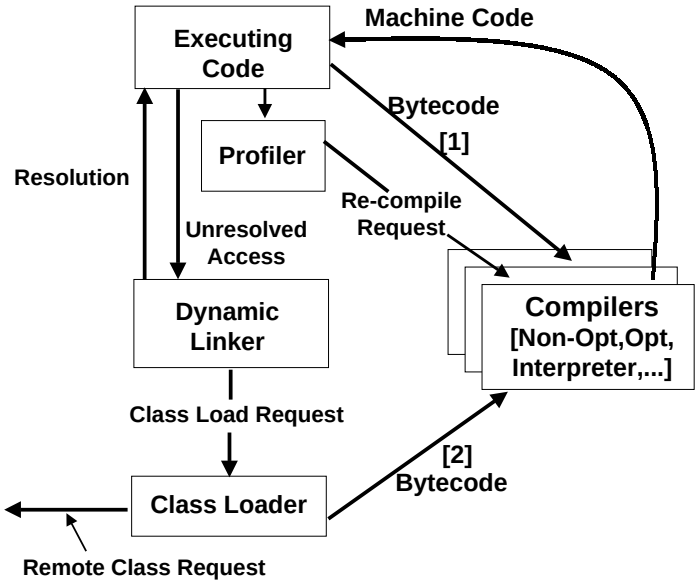


Figure III.1: General depiction of an adaptive compilation environment.
In an adaptive compilation environment multiple compilers are incorporated (or possibly a compiler and an interpreter). The compilers differ in compilation time imposed and resulting execution efficiency. The compilers can be invoked in multiple ways: [1] in which a method is invoked for the first time by the executing program, [2] when a class file is loaded into memory (the entire class may be compiled). In addition, the compiler may insert instruction into the execution stream so that the behavior of the executing code can be profiled. If the behavior of the program changes, the profiling mechanism can request that pieces of the code be recompiled to improve execution performance. Other parts of the figure depict the general Java class file loading mechanism as described in our background chapter (Chapter II).

optimizations, however efficiently implemented, impose compilation delay. O3 execution time is approximately 5% faster than O1 execution time while compilation time is 89% slower than O1 compilation for the programs studied. To compensate for the compilation delay, the O1 compiler is used first and inserts instrumentation into each method. On-line measurements of method invocation count are made. When a threshold for a method is reached, the method is recompiled with the O3 compiler and used on future invocations.

Jalapeño Virtual Machine

Jalapeño is an adaptive compilation system for Java programs that is written in Java (unlike ORP). It is also unique in that it is designed to address the special requirements of SMP servers: performance and scalability. Extensive runtime services such as parallel allocation and garbage collection, thread management, dynamic compilation, synchronization, and exception handling are provided by Jalapeño.

At the time of this work, there are two fully–functional compilers in Jalapeño, a fast *baseline* compiler and the *optimizing* compiler. The baseline compiler provides a near-direct translation of Java class files thereby compiling very quickly and producing code with execution speeds similar to that of interpreted code. Jalapeño, using the baseline compiler, performs in much the same way as an interpreted system. The second compiler is the optimizing compiler and builds upon extensive compiler technology to perform various levels of optimization [12].

The compilation time using the optimizing compiler is 50 times slower on average for the programs studied than the baseline, but produces code that executes 3–4 times faster. To warrant its use, compilation overhead must be recovered by the overall performance of the programs. To do this, on-line profiles are collected using instrumented method execution. Like other adaptive systems, when a threshold is reached, a method is recompiled using the optimizing compiler. The optimizing compiler incorporates multiple levels of optimization that include many simple transformations, inlining, scalar replacement, static single assignment optimizations, global value numbering, and null check and dead code elimination. On-line measurement continues even after initial optimization in case re-optimization using different levels is needed to further improve performance.

HotSpot

Another form of adaptive compilation for server systems is described in the Java HotSpot performance engine [34] from Sun Microsystems. The system analyzes an application

as it runs, identifying the areas that are most critical to performance, i.e., where the greatest time is being spent executing bytecode. Rather than compiling each method at initial invocation, the performance engine initially runs the program using an interpreter, and analyzes it as it runs to discover execution hot spots. It then compiles and optimizes only those performance-critical areas of code. This monitoring process continues dynamically throughout the life of the program, with the performance engine adapting to the ongoing performance needs of the application.

III.C Other Related Work

Other projects, that may not at first seem directly related to load delay reduction are those that concern program restructuring and Java bytecode annotation systems. Our work uses program restructuring to improve the performance of the algorithms and frameworks. In addition, we propose a bytecode annotation system for the reduction of compilation overhead. As such, we next describe related work on these two topics.

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 [7, 32]) and Denning's working set model [20].

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 reordering [37, 66], function grouping [26, 31, 37, 66], reordering based on control structure [60], and reordering of system code [83] 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.

Java Bytecode Annotation

Java bytecode annotation has been proposed to enable the use of complex, time-consuming optimizations at runtime. In existing systems, Array bounds check elimination and register allocation are performed off-line and the results are communicated to the compilation system at runtime. The communication is performed via bytecode annotation implemented using an existing class file data structure called an attribute. At runtime, the compiler uses the annotations to implement the optimizations in the generated native code. As part of this thesis work, we present a novel annotation framework for Java bytecode. In this section we describe what has been proposed for similar systems as well as the inherent limitations that we improve upon in this dissertation.

Pominville et al. in [69] describe a framework for Java bytecode annotation that enables implementation of an array bounds checks elimination optimization. The empirical data provided include the execution time with and without their optimization. The goal of this and all other prior bytecode annotation work is to enable time-consuming optimizations to be incorporated by the optimizing compiler to reduce execution time without substantially increasing compilation delay. However, like other such systems, no measurement of compilation delay (with or without their annotation optimization) is provided. The goal of these projects is not to reduce compilation overhead but to enable new and different, time-consuming optimizations. This distinction is subtle but important for comparison with the work we present in this thesis.

Two similar frameworks have been proposed for register allocation in Java programs. The implementation of register allocation and related optimizations is necessarily, very time consuming. The difficulty arises since the Java bytecode model is that of a stack-based architecture. Since most architectures on which Java programs are run are register-based it is difficult to achieve an efficient register mapping even without the time restrictions required for dynamic compilation and optimization. Hummel et al. in [36], show how static register allocation information in the Kaffe JIT can be conveyed using annotations. They use a virtual register scheme in which they assume an infinite number of registers, make virtual register assignments off-line, and communicate this information to the JIT compiler for register allocation. A similar bytecode annotation optimization for register allocation has been developed by Jones et al. in [42]. Th authors describe a system in which 256 virtual register numbers are assigned to the bytecode of each method to improve execution performance using the Kaffe JIT compiler.

All of these annotations substantially increase the size of the bytecode stream in an attempt to improve runtime performance with a single optimization. Pominville et al. increase

application size by 7% to 16% on average; Hummel et al. show an average increase of 33% to 97% (31% to 38% on average by Jones et al.). Since we consider load delay, the combination of transfer plus compilation overhead, we ensure that the improvements achieved by our annotation-guided optimizations are not negated by the increase in transfer delay due to annotated bytecodes as can occur using this prior work. In addition, these related projects do not consider the use of annotations to reduce optimization overhead as we do in Chapter XI. The goal of each of these prior works is to enable an expensive optimization to be performed, which is a side-effect of the framework we present The goal of our annotation work is to reduce compilation overhead (and thus load delay) for *all* optimizations not a specific few.

Another issue the must be addressed by any annotation implementation is security. For annotations to be trusted, they must be verified or implemented so as to guarantee safety of the JVM or machine on which annotated execution is performed. The annotation-guided optimizations presented in [69, 6] are unsafe since the annotations contain information that effect the semantics of the program and no verification is performed at the destination. For example, in [69], the authors present an annotation for array bounds check elimination. If this annotation in intercepted and modified by an untrusted party, a boundary check might be eliminated and cause illegal memory access. Likewise, in [6], the authors implement register allocation, and annotation manipulation can cause program behavior that can potentially harm the JVM in which it is executing as well as the underlying machine. For such annotations to be trusted, some mechanism for verification must be implemented. The annotations we present for remote execution (Section XI.B.2) are safe without requiring verification since their modification only affects program performance not semantic behavior.

Chapter IV

Experimental Methodology

In this chapter, we introduce the experimental methodology used for the results presented in this dissertation. We describe the benchmarks incorporated then detail the execution, simulation, and compilation environments we use to evaluate our techniques. In each of the following research chapters (Chapters V through XI), we articulate any additions to this methodological framework that are specific to the technique(s) presented in the chapter.

IV.A Benchmarks

Throughout this dissertation we present empirical results for the thirteen Java programs described in Table IV.1. The programs, which include the SpecJvm98 benchmarks, 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 [55, 70]. Subsets of this list are used for the different techniques we present due to the change in and diversity of the infrastructures used for our experimental results at the time the work was performed (1998-2001). In addition, as new benchmarks became available throughout this period we incorporated them into our studies.

Tables IV.2 and IV.3 show the general, static and dynamic statistics, respectively, of each benchmark. Column two of Table IV.2 is the number of class files in the application. Columns three and four show the total number of methods and instructions, respectively. The last column is the size (in KB) of the application; the percentage of this size that is global data is indicated in parentheses. On average, global data accounts for 63% of the total application size.

Table IV.1: Description of Benchmarks Used.

Antlr	Parser generator
Bit	Bytecode instrumentation tool: Each basic block in the input program is instrumented to report its class and method name
Compress	SpecJvm95 compression utility
DB	SpecJvm95 database access program
Jack	SpecJvm95 Java parser generator based on the Purdue Compiler Construction Toolset
Jasmine	Bytecode obfuscation tool
Javac	SpecJvm95 Java to bytecode compiler
Jcup	LALR parser generator: A parser is created to parse simple mathematics expressions
Jess	SpecJvm95 expert system shell benchmark: Computes solutions to rule based puzzles
Jlex	Lexical analyzer for Java
Jsrc	Java bytecode to HTML converter
Mpeg	SpecJvm95 audio file decompression benchmark: Conforms to the ISO MPEG Layer-3 audio specification
Soot	Bytecode processing tool: converts Java class files to an intermediate format: Jimple

Table IV.2: Static statistics on the benchmarks used in this dissertation.
For each benchmark, the first three columns provide the number of static classes, methods, and instructions. The last column is the size (in KB) of the application; the percentage of this size that is global data is indicated in parentheses. The average across all benchmarks is the last entry in the table.

	Total Number of			Total
Program	Classes	Methods	Insts (1000s)	KB Size (% GData)
Antlr	118	1318	49	418 (52%)
Bit	53	317	14	152 (57%)
Compress	12	44	2	18 (78%)
DB	3	34	2	10 (58%)
Jack	56	315	19	128 (53%)
Jasmine	207	1160	33	404 (79%)
Javac	176	1190	41	548 (70%)
Jcup	36	385	14	130 (59%)
Jess	151	690	18	387 (81%)
Jlex	20	134	12	85 (52%)
Jsrc	33	414	15	145 (52%)
Mpeg	55	322	34	117 (50%)
Soot	721	3607	65	1111 (74%)
Avg	126	764	24	281 (63%)

Table IV.3: Dynamic statistics on the benchmarks used in this dissertation.
For each benchmark, the first three columns provide the number of classes, methods, and instructions executed using input1. In each of these columns, the same data for input2 is shown in parenthesis. We provide data on two inputs since some of our optimizations are profile-guided. For these techniques we provide results for profiles generated with both inputs. The fourth column of data is the interpreted execution time for each program and input (input2 in parenthesis). The last column is the size (in KB) of the class files used by the application for each input. The average across all benchmarks is the last entry in the table.

	Characteristics of Programs (Ref) (Train in parenthesis)				
	Number of Executed			Interpreted	
Program	Classes	Methods	Insts. (100000s)	Execution TTime in Secs	Used Size (KB)
Antlr	67 (69)	538 (549)	87 (26)	15.37 (4.97)	268 (269)
Bit	40 (37)	158 (153)	599 (342)	56.03 (29.63)	137 (134)
Compress	12 (12)	32 (32)	1138 (954)	54.02 (46.12)	18 (18)
DB	3 (3)	27 (24)	1115 (25)	3630.34 (7.61)	10 (10)
Jack	46 (46)	265 (265)	233 (27)	227.07 (27.54)	120 (120)
Jasmine	165 (159)	714 (669)	461 (110)	49.27 (22.50)	326 (317)
Javac	139 (132)	740 (713)	911 (25)	276.43 (7.96)	472 (225)
Jcup	29 (29)	213 (213)	42 (4)	24.52 (2.18)	123 (123)
Jess	133 (135)	412 (412)	1554 (3)	206.10 (8.52)	351 (357)
Jlex	18 (18)	99 (97)	28 (12)	6.96 (2.49)	81 (81)
Jsrc	29 (30)	318 (329)	8 (20)	7.49 (16.97)	128 (128)
Mpeg	42 (42)	201 (200)	11489 (1220)	491.83 (51.52)	111 (111)
Soot	158 (158)	346 (346)	3(2)	16.63 (7.69)	317 (317)
Avg	68 (67)	313 (308)	1359 (213)	389.39 (18.13)	189 (170)

These static statistics (Table IV.2) apply to any inputs; the dynamic statistics in Table IV.3 show data for two inputs a *Ref* input and a *Train* input (in parenthesis). Columns two through four in the dynamic statistics table show the number of executed classes, methods, and instructions, respectively. The fifth column is the total execution time when the programs are interpreted and the final column is the size of class files that are used during execution.

We provide statistics for two inputs (Ref and Train) since many of our techniques use profile information to guide optimization. We report results from the use of the Ref input for all of our measurements. However, two sets of result data are shown for each profile-guided technique which are distinguished by *Ref-Ref* and *Ref-Train* labels. For these techniques, we generate profiles and guide our optimizations using both inputs. We denote results that use the profile generated using the Ref input by Ref-Ref (the first Ref indicates the input used for result generation, the second indicates that used for profile generation). Using the same data set for both profile and result generation provides ideal performance since we have perfect information about the execution characteristics of the programs. Results demarcated with Ref-Train are those that use the profile generated using the Train input to guide optimization. Ref-Train, or cross-input, results indicate realistic performance since the characteristics used to perform the optimization can differ across inputs and the input that will be used is not commonly known ahead of time. As mentioned above, results are executed using Ref regardless of the input used for profile creation.

We use two general experimental methodologies for the measurement of the benefits achieved on the benchmarks by the techniques and optimizations we present. The first is used for the transfer delay optimizations (Chapters V- VIII); the second is that used for the compilation delay optimizations (Chapters IX- XI).

IV.B Transfer Delay Optimization Methodology

Results presented for our transfer delay optimizations are generated on (and assume) a single processor, 300 MHz x86 platform, running Debian Linux version 2.2.15. The Java implementation we use is JDK version 1.1.8 for Linux provided by Blackdown Corp. [11].

As mentioned previously, many of our techniques are profile-guided: off-line measurements are made of execution behavior and are used to perform program optimizations. Profiles are generated by executing instrumented versions of the benchmarks. Instrumentation is performed using the Bytecode Instrumentation Tool (BIT) [54, 55]. The BIT interface enables

elements of the bytecode class files, such as bytecode instructions, basic blocks and methods, to be queried and manipulated. In particular, BIT allows an instrumentation program to navigate through the basic blocks of a bytecode program, collect information about the use of local and constant pool variables, opcodes, branch conditionals, etc., and perform control-flow and data-flow analysis. The type of information collected using profiles is specified in each of the the research chapters (V- XI) in which profiling is used to guide the technique presented.

IV.B.1 Compression

For this dissertation, some of the techniques for transfer delay reduction exploit and extend existing compression research. For these techniques, we consider three commonly used compression formats for of mobile, Java programs: JAR, PACK, TGZ. The Java archive (jar) format (referred to here forward as JAR), is the most common tool for collecting (archiving) and compressing Java application files. The format is based on the standardized PKWare zip format [67] and enables archival of various components of Java applications (class, image, and sound files). For this study we consider compression of only class files.

PACK [71] is a jar file compression tool from the University of Maryland. This utility defines a compact representation of class file information and substantially reduces redundancy by exploiting the Java class file representation, and by sharing information between class files. The compression ratios achieved by this tool are far greater than any other compression utility for Java applications. However, pack utility has very high decompression times since the class files must be reconstituted from this complex format.

Gzip is a standard compression utility, commonly used on UNIX operating system platforms. Gzip does not consider domain specific information and uses a simple, bit-wise algorithm to compress files. As such, gzip has very fast decompression times but does not achieve the compression ratios of pack. The TGZ format refers to files that are first combined (archived) using the UNIX tape archive (tar), then compressed with gzip. Tar combines a collection of class files, uncompressed and separated by headers, into a single file in a standardized format.

Decompression Time

To load a mobile Java program or class file, the JVM class loader reads a stream of bytes from which class files are constructed and placed into memory. The stream of bytes can be generated from any source (files stored locally on disk or obtained from a remote site over a network, etc.) using Java library routines. To incorporate decompression into the class

Table IV.4: Compression characteristics of the benchmarks using PACK, JAR, and TGZ. The three columns of data are for each compression technique. For each benchmark and compression format, the decompression rate is shown with the compressed application size in parenthesis.

Compression Format & Decompression Characteristics Sizes are in kilobytes and times are in seconds										
		PACK			JAR			TGZ		
		Compression		Dec	Compression		Dec	Compression		Dec
Program	Size	Size	Time	Time	Size	Time	Time	Size	Time	Time
Antlr	418	58	16.53	3.66	222	2.19	0.30	172.30	0.31	0.03
Bit	152	18	6.40	1.25	85	1.07	0.14	57.00	1.07	0.03
Jasmine	404	34	8.68	2.69	219	2.93	0.32	127.70	2.93	0.03
Javac	548	49	14.99	3.32	276	2.93	0.34	179.20	2.93	0.03
Jess	387	23	4.32	1.83	185	2.37	0.34	164.80	2.37	0.03
Jlex	85	14	5.95	1.04	48	0.56	0.20	37.80	0.56	0.04
Average	332	33	9.48	2.30	172	2.01	0.27	123.13	1.70	0.03

loading process, a decompression library is needed to convert a compressed stream of bytes (an application or class file) into the corresponding, decompressed byte stream for class file construction.

Java (v1.1.x and above) provides these libraries for the zip (JAR) and gzip formats. A public domain, Java library for the tar format is provided by [39] and is used in coordination with the gzip library for TGZ decompression. For these formats (JAR, TGZ), we time the decompression process for each compressed benchmark, with a user-defined class loader using these libraries. Since PACK at the time of this study, did not provide a mechanism for its incorporation into a class loader, we made off-line timings using the command-line interface. We simply decompressed the benchmarks 100 times on the same dedicated system as used for JAR and TGZ timings and took the average. Table IV.4 provides the characteristics on the benchmarks we consider in the compression study. We repeat the static, decompressed application size from Table IV.2 for the first column of data. For each compression format, we present three columns of data: the compression time (in seconds), the compressed size (in kilobytes), and the decompression time (in seconds). The average across all benchmarks is shown in the bottom row in the table.

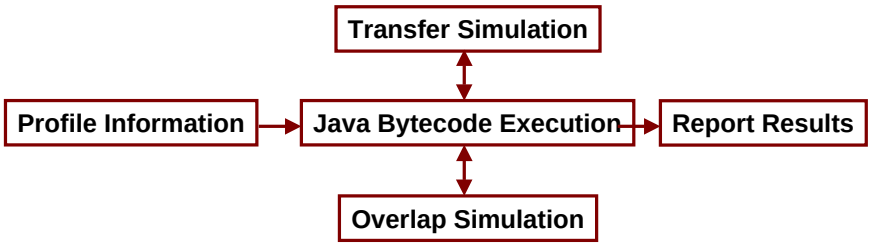


Figure IV.1: General depiction of our result generation model.
Profile information from off-line execution is passed to the execution environment. This information is used to guide the optimization. Once the optimization has been performed the program is executed. Transfer time is simulated using real network trace data and the amount of overlap is computed by determining the execution time of each basic block. The simulation results for each of our transfer delay optimizations are then reported.

IV.B.2 Simulation Model

Many of the techniques presented in the research chapters that follow (V- VIII) attack transfer delay. The amount and reduction of transfer delay is dependent upon the substratal network. We therefore must incorporate network performance measurement into our studies to evaluate our techniques. As such, we incorporate a range of representative values taken from traces of performance characteristics form actual Internetwork technologies. Since we use trace values (for repeatability) as opposed to real-time measurements, our result model is one of simulation.

In addition to simulation of transfer, we also simulate overlap of transfer with execution in some of our transfer delay reduction techniques. A depiction of our result generation environment is shown in Figure IV.1. Profile (if needed), transfer, and overlap information is used during execution to measure the effect of our transfer optimizations. We first detail the simulation of transfer then of overlap. We then discuss the assumptions we make and the implications resulting from this simulation model on Java bytecode verification.

Table IV.5: Description of the networks used in this study.
Each network is represented by a bandwidth value. These values are obtained from actual trace data collected using the connection. UTK indicates the University of Tennessee, Knoxville and UCSD is the University of California, San Diego. The residence used is located in Knoxville, TN.

Name:	From:	To:	Bandwidth:
MODEM	Residence (East)	UTK (East)	0.03 Mb/s
ISDN	Residence (East)	UTK (East)	0.128 Mb/s
INET	UCSD (West)	UTK (East) (Internet)	0.28 Mb/s
INET	UCSD (West)	UTK (East) (Internet)	0.50 Mb/s
INET	UCSD (West)	UTK (East)	0.75 Mb/s
LAN	UCSD (West)	LAN (10Mb/s Ethernet)	1.00 Mb/s

Network Transfer

We gather network performance traces using the JavaNws [51], a Java library ported from a subset of the Network Weather Service (NWS) [89] toolkit[1]. The JavaNws provides users or utilities with measurements of current network performance and accurate predictions of short-term future performance deliverable to an application or download. To measure network performance, the JavaNws conducts in a series of communication *probes* between itself and the server machine of interest. During each probe, measurements are taken of round-trip time and bandwidth. Other tools are available for similar trace generation, e.g., netperf [43], TTCP [76]. However, we chose to use the JavaNws since it is written in Java and provides network performance prediction. JavaNws prediction is detailed further in Chapter VIII.

We examine the effect of our transfer delay techniques for a variety of networks. Since it is difficult to characterize a network by a single bandwidth value, we selected a range of representative bandwidth values from 24-hour trace data. Table IV.5 shows the bandwidth values used in this study and the corresponding networks. UTK indicates the University of Tennessee, Knoxville and UCSD is the University of California, San Diego. The residence used is located in Knoxville, TN. The network performance available from these traces include a 28.8 baud modem (MODEM), an integrated services digital network link (ISDN), a series of common cross-country, common-carrier, Internet connections, and a 10Mb/s local area Ethernet connection(LAN).

We assume that the time to request a non-local file from its source is the time for one round-trip of the network. In addition, we assume that this time is 100ms (based our

[1]The NWS also performs measurement and prediction services for other resources (CPU, memory, etc). The JavaNws implements the subset of the NWS that provides these services for the network resource only.

cross-country Internet measurement); in addition, this value is commonly assumed in similar studies [35]. We compute the time to transfer a requested file by multiplying the size of the transferred file by the network bandwidth.

Overlap of Transfer and Execution

Some of our techniques enable overlap with execution with transfer. To do this we use simulated execution of programs instrumented using BIT. Currently, Java execution environments disallow execution to occur concurrently with transfer. To model this type of execution we need to measure the transfer delay imposed. To do this we intercept execution at each basic block and determine if it is the first block to execute from the class file it is contained in. If so, we determine, given the size of the class file and the underlying network performance, what the transfer delay (wait time in seconds) that would be imposed on a program that is executed without simulation.

To enable simulated overlap, we extend this model. Figure IV.2 exemplifies our simulation procedure. We model the transfer delay as described above and in addition, we compute the execution time of the basic block. It is this execution time that can be overlapped. We simply reduce the transfer delay by this overlap measurement. More specifically, instrumentation causes execution to be interrupted and a simulator method to be invoked at the start of each basic block. This method (simulation(...) in the figure) first determines if the class file containing the basic block has transferred (indicated by the boundary comparison in the figure). If an insufficient amount has transferred, the execution is stalled (the simulator increments the wait time) while the transfer completes. To compute the overlap (in seconds), the simulator next computes the execution time (in seconds) of the block. This computation is performed by multiplying the number of bytecode instructions in the block by the average bytecode cycles per instruction (BCPI) of the program. This number is then divided by the megahertz rate of the assumed CPU (300Mhz in our studies). The BCPI is computed by executing (interpreting) the benchmark off-line 100 times using a dedicated processor. Once we know the execution time of the block we can compute, based in the network bandwidth (actual or traced), the amount of transfer delay that can be overlapped by multiplying the seconds executed by the number of bytes per second that can be transferred (given the bandwidth).

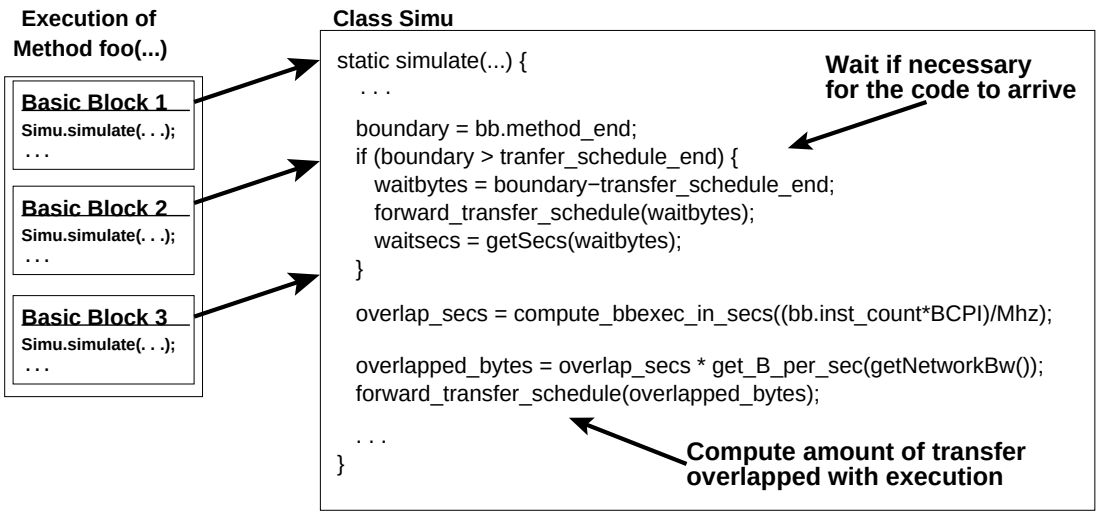


Figure IV.2: Example of transfer delay and overlap simulation

Each basic block of an application is instrumented so that our simulation method is invoked just prior to it during execution. The simulator determines whether or not enough transfer has completed for the method in which the basic block is contained can execute. If it has not, then the execution must stall (we increment the wait time in this case). If overlap is enabled then the simulator computes the number of seconds the basic block will executed and from that computes the amount of data that can transfer during that time.

IV.B.3 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. For our results using verification, we modeled the verification mechanism in JDK 1.2. This process is clarified in the Background chapter (Chapter II). We refer to results using no verification as *trusted transfer* (verification can be turned off using the `-noverify` runtime flag if desired).

We incorporate verification information by profiling the order in which class files are loaded for verification and first used during execution. Verified loading is made explicit by the -

verify and *-verbose* runtime flags. Such profiling enables construction of verification dependency chains for the class files used during execution. During simulated generation of our verified transfer results we account for any and all dependencies each time a class file is first used (and transferred). Partial output from the sample program presented in Figure II.1 of the Background chapter (Chapter II) is shown below; first using an input of 5 to the X class file (java -verify -verbose X 5) and then using an input of 5 (java -verify -verbose X 6).

```
myhost > java -verify -verbose X 5
. . .
[Loaded ./X.class]
[Loaded ./Z.class]
[Loaded ./ZSuper.class]
[Loaded ./ZSuperSuper.class]
[Loaded /users/ckrintz/java/classes/nonstrict/NS_Sim.class]
[Loaded java/io/Reader.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/io/FileReader.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/io/InputStreamReader.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/util/Vector.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/lang/ArrayIndexOutOfBoundsException.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/util/NoSuchElementException.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/lang/FloatingDecimal.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/lang/Double.class from /usr/lib/jdk1.1/lib/classes.zip]
CLASS: X FIRST EXECUTED AT INSTRUCTION: 6.0
myhost >
myhost > java -noverify -verbose X 6
. . .
[Loaded ./X.class]
[Loaded ./Z.class]
[Loaded ./ZSuper.class]
[Loaded ./ZSuperSuper.class]
[Loaded /users/ckrintz/java/classes/nonstrict/NS_Sim.class]
[Loaded java/io/Reader.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/io/FileReader.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/io/InputStreamReader.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/util/Vector.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/lang/ArrayIndexOutOfBoundsException.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/util/NoSuchElementException.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/lang/FloatingDecimal.class from /usr/lib/jdk1.1/lib/classes.zip]
[Loaded java/lang/Double.class from /usr/lib/jdk1.1/lib/classes.zip]
CLASS: X FIRST EXECUTED AT INSTRUCTION: 6.0
CLASS: ZSuper FIRST EXECUTED AT INSTRUCTION: 18.0
```

```
CLASS: ZSuperSuper FIRST EXECUTED AT INSTRUCTION: 20.0
answer: 10
myhost >
```

To generate the sample output, we instrumented the program so that for each basic block executed, the class name and time of execution is printed out. The time of execution is given as the cumulative bytecode instruction count so far. If a class name has already been printed (and hence transferred), it is not printed again. The output when an input of 5 is used, shows that even though class files are not accessed during execution, they still must be transferred for verification. The dependencies on class X for this first case are class Z, class ZSuper, and class ZSuperSuper. Therefore, during transfer simulation, when X is first used we accumulate the total transfer time for X, Z, ZSuper, and ZSuperSuper. Then when Z (or any of these classes) are ever used by the executing program we do not incur the transfer delay required for them since they have already transferred to verify X. Each class that the simulation transfers is marked as local so that it is only accounted for once.

IV.B.4 Transfer Delay Optimization Metrics

In this section, we summarize the various metrics we use to model and measure the efficacy of our transfer delay techniques. These metrics allow us to empirically evaluate the differences between existing technology and the advances that our runtime and compiler optimizations enable.

$$Request\_time = number\_of\_classfiles\_requested * 100ms$$

Request time is the round trip time required for a non-local class file to be requested from the server at which it is stored to the client for use during execution. Since class files are dynamically loaded individually they are requested and transferred (if compressed archives are not used for execution) as needed by the application at the client. We assume a round trip time of 100ms. This is a measured average value of a cross-country link between the University of Tennessee, Knoxville and the University of California, San Diego.

$$Transfer\_Delay = Request\_time + (wait\_in\_bytes * bytes\_per\_second(NetworkBW^*))$$

At any time during simulated execution when the total amount transferred so far is insufficient to continue execution, program progress must halt until the necessary transfer has occurred. We accumulate the number of bytes that is waited on during execution to compute

wait_in_bytes in the above expression. Each time stalling occurs, the currently available network bandwidth value is obtained (hence, the asterisk). For the work in this dissertation, we use a single value each time bandwidth is requested. However we perform the simulation for multiple bandwidth values. Our system is easily extended to use trace data and real network data. We use single bandwidth "snapshots" for repeatability purposes.

$$BCPI = ET\_secs * (1/bytecode\_instruction\_count) * Mhz$$

BCPI stands for average bytecode cycles per instruction. To compute this value we time the execution of a program 100 times on a 300Mhz dedicated processor to get the execution time in seconds (ET_secs). Instrumented execution is used to compute the number of bytecode instructions are executed by the program for a given input. The execution time is multiplied by the inverse of the number of bytecode instructions and the megahertz rate of the processor (300Mhz in this case).

$$bblock\_ET\_secs = bb.bytecode\_instruction\_count * BCPI$$

For our transfer and overlap simulations, we use the number of seconds a basic block will execute for (bblock_ET_secs). To compute this we take the BCPI and multiply it by the bytecode instruction count of the basic block.

$$bytes\_per\_second(NetworkBW) = Network\_megabits\_per\_second * (2^{20}/8)$$

To determine the number of bytes that can transfer per second given a certain network bandwidth we divide the bandwidth (Network_megabits_per_second) by eight to get megabytes per second and then multiply this by 2^{20} to get bytes per second. The bandwidth value itself can be obtained from trace or live network data.

$$olapped\_bytes = bblock\_ET\_secs * bytes\_per\_second(NetworkBW)$$

To compute the number of bytes that can be overlapped during execution of a program we compute the execution time of each basic block (bblock_ET_secs) and then multiply that by the number of bytes per second that can transfer given a specific network bandwidth value (NetworkBW).

$$wait\_secs = wait\_bytes/bytes\_per\_second(NetworkBW)$$

To compute the amount of wait time a program experiences due to transfer (wait_secs) we must first compute the number of bytes which are required for execution to proceed. To

compute the latter, simply determine the number of additional bytes that needed for execution to continue. We then divide this number by the number of bytes per second that the underlying network can currently support.

$$Decompression\_Rate = compressed\_size / average\_seconds\_to\_decompress$$

Some of our techniques assume compressed transfer. For these, we consider the time required for decompression. We compute this time by repeatedly (100 times) decompressing a benchmark on a 300Mhz dedicated processor. For our studies we use an average decompression rate (different for each benchmark). We compute this by taking the compressed application size and dividing it by the average decompression time.

$$Total\_Delay = Transfer\_Delay + (Decompression\_Rate * compressed\_size)$$

In addition to decompression rate, we compute total delay for the techniques for which we consider compression. To compute this we take the average decompression rate and multiply it by the size of the compressed file. We then add this to the transfer delay. In some cases, we also include compression time into this total delay figure. When we do so, we make it explicit in the text. Compression time is computed much like decompression: the application is compressed 100 times and timed on a 300Mhz dedicated processor.

IV.C Compilation Delay Optimization Methodology

The second source of load delay the overhead of dynamic compilation. We measure and implement optimizations that reduce the effect of dynamic compilation for two compilation environments. Many of the optimizations are profile-based: off-line statistics of execution behavior are collected and used during optimization. For compilation delay, we gather profiles of method compilation. To do this, we have extended each of our compilation environments to log the compilation that takes place and the time required for it during program execution. Like for transfer delay optimizations we also use profiles of invocation counts. For these, we use the same profiles generated for the transfer delay optimizations using the Bytecode Instrumentation Tool (BIT) [54, 55] as described above in Section IV.B.

IV.C.1 Compilation Environments

All of the techniques for transfer delay reduction use and assume JDK 1.1.8 execution environment using interpretation. For the techniques that attack compilation delay we use two

different compilation environments. Both infrastructures are designed for adaptive compilation. As such, they include multiple compilers (optimizing and fast, non-optimizing) to adapt to program runtime behavior. We do not use the adaptive framework with our compilation optimizations but do exploit the inclusion of multiple compilers in such systems. The first compilation environment we use is Jalapeño; the second is the Open Runtime Platform (ORP).

Jalapeño Virtual Machine

In one of the studies we present for compilation delay reduction (Chapter X), we attempt to reduce compilation delay incurred by the use of the Jalapeño Virtual Machine, a JVM being developed at the IBM T. J. Watson Research Center [2, 3]. Jalapeño is written in Java and designed to address the special requirements of SMP servers: performance and scalability. Extensive runtime services such as parallel allocation and garbage collection, thread management, dynamic compilation, synchronization, and exception handling are provided by Jalapeño.

Jalapeño uses a compile–only execution strategy, i.e., there is no interpretation of Java programs. Currently there are two fully–functional compilers in Jalapeño, a fast *baseline* compiler and the *optimizing* compiler. The baseline compiler provides a near-direct translation of Java class files thereby compiling very quickly and producing code with execution speeds similar to that of interpreted code. Jalapeño, using the baseline compiler, performs in much the same way as an interpreted system.

The second compiler is the optimizing compiler and builds upon extensive compiler technology to perform various levels of optimization [12]. The compilation time using the optimizing compiler is 98% slower on average for the programs studied than the baseline, but produces code that executes 71% faster. To warrant its use, compilation overhead must be recovered by the overall performance of the programs. All results are generated using a December, 1999 build of the Jalapeño infrastructure. We report results for both the baseline and optimizing compilers. The optimization levels we use in the latter include many simple transformations, inlining[2], scalar replacement, static single assignment optimizations, global value numbering, and null check elimination.

Jalapeño is invoked using a boot image [2]. A subset of the runtime and compiler classes are fully optimized prior to Jalapeño startup and placed into the boot image; these class files are not dynamically loaded during execution. Including a class in the boot image, requires

[2]The optimizing compiler performs both unguarded inlining of static and final methods and guarded inlining of non-final virtual methods.

that the class file does not change between boot-image creation and Jalapeño startup. This is a reasonable assumption for Jalapeño core classes. This idea can be extended with mechanisms to detect if a class file has changed since it is statically compiled to enable arbitrary application classes to be pre-compiled. This topic is further described in [73]. Infrequently used, specialized, and supportive library and Jalapeño class files are excluded from the boot image to reduce the size of the JVM memory footprint and to take advantage of dynamic class file loading. When a Jalapeño compiler encounters an unresolved reference, i.e., an access to a field or method from an unloaded class file, it emits code that when executed invokes Jalapeño runtime services to dynamically load the class file. This process consists of loading, resolution, compilation, and initialization of the class. If, during execution, Jalapeño requires additional Jalapeño system or compiler classes not found in the boot image, then they are dynamically loaded: there is no differentiation in this context between Jalapeño classes and application classes once execution begins. To ensure that our results are repeatable in other infrastructures, we isolate the impact of our approaches to just the benchmark applications by placing all of the Jalapeño class files required for execution into the boot image.

The results we present using Jalapeño are gathered by repeatedly executing applications on a dedicated, 166MhzX4–processor PowerPC-based machine running AIX v4.3. Table IV.6 shows various compilation characteristics of subset of benchmarks we used in this study. Compilation time (CT) and execution time (ET), in seconds, using the Jalapeño optimizing and the fast baseline compilers are shown for each input. The compilation time includes the time to compile only the class files that are used.

Open Runtime Platform (ORP)

For our compilation delay reduction techniques, we also consider the Open Runtime Platform (ORP), an open-source, dual-compiler system [65] which was recently released by the Intel Corporation [17]. The first compiler (O1) provides very fast translation of Java programs [1] and incorporates a few very basic bytecode optimizations that improve execution performance. The second (O3) compiler performs a small number of commonly used optimizations on bytecode and an intermediate form to produce improved code quality and execution time. O3 optimization algorithms are implemented with compilation overhead in mind, hence only very efficient algorithms are used [16]. The optimizations implemented include many simple transformations, inlining of small methods, copy and constant propagation, common subexpression elimination and simple loop optimizations. The compilation overhead, total time, and

Table IV.6: Jalapeño compilation statistics.
This data CANNOT be compared to that from other tables and figures since the measurements were made on a 166MhzX4 PowerPC processor during an internship at IBM Research during which Jalapeño was made available to us. The first four columns of data contain the execution (ET) and compile (CT) times when the Jalapeño optimizing compiler is used. The last four columns are the execution and compile times when the Jalapeño baseline compiler is used. Times for both inputs are given.

	Optimized Time (Secs) (Used Classes)				Baseline–Compiled Time (Secs) (Used Classes)			
	Train		Ref		Train		Ref	
Benchmark	ET	CT	ET	CT	ET	CT	ET	CT
Compress	7.4	8.2	84.0	8.1	47.0	0.1	525.1	0.1
DB	1.9	8.2	102.7	8.0	2.9	0.3	162.6	0.3
Jack	9.9	16.0	84.3	16.0	10.9	0.4	93.2	0.4
Javac	2.0	38.6	66.3	38.5	3.0	0.6	103.5	0.6
Jess	2.5	27.2	45.2	27.6	6.4	0.3	109.8	0.3
Mpeg	7.3	15.9	71.3	15.9	47.6	0.4	452.1	0.4
Avg	5.2	19.0	75.6	19.0	19.6	0.4	241.1	0.4

the number of methods compiled by the ORP compilers is shown in Table IV.6. Total time consists of both compile and execution time. For comparison, O3 execution time is 8% faster than O1 execution time on average and the compilation time of the O3 compiler is 89% slower than that for O1 on average in the programs studied.

The results we present using ORP are gathered by repeatedly executing applications on a dedicated, 300Mhz x86 machine running Linux version 2.2.15. Table IV.7 shows various compilation characteristics of subset of benchmarks we used in the ORP study. Compilation time (CT) and execution time (ET), in seconds, using the ORP O3 and O1 compilers are shown for each input. The compilation time includes the time to compile only the class files that are used.

IV.C.2 Compilation Delay Optimization Metrics

In this section, we summarize the various metrics we use to model and measure the efficacy of our compilation delay techniques. These metrics allow us to empirically evaluate the differences between existing technology and the advances that our runtime and compiler optimizations enable.

$$Compilation\_secs = average\_time\_for\_compilation$$

Table IV.7: Compilation characteristics using the Open Runtime Platform.
The first four columns of data contain the execution (ET) and compile (CT) times when the ORP O3 optimizing compiler is used. The last four columns are the execution and compile times when the ORP O1 compiler is used. Times for both inputs are given.

	O3 (Optimized) Time (Secs) (Used Classes)				O1-Compiled Time (Secs) (Used Classes)			
	Train		Ref		Train		Ref	
Benchmark	ET	CT	ET	CT	ET	CT	ET	CT
Jack	4.9	2.9	38.7	2.8	5.5	0.3	41.9	0.3
JavaCup	6.4	3.3	44.5	3.2	6.7	0.3	48.6	0.3
Jess	1.3	2.6	40.4	2.6	1.5	0.3	41.8	0.3
Jsrc	16.9	3.0	48.2	3.0	17.9	0.3	49.6	0.3
Mpeg	3.2	2.4	30.8	2.4	4.0	0.3	37.5	0.3
Soot	1.0	1.8	5.2	1.7	1.0	0.3	6.7	0.3
Avg	5.6	2.7	34.6	2.6	6.1	0.3	37.7	0.3

Compilation time using either infrastructure is computed in similar ways. The compilation system is instrumented so that each time a compiler is invoked a time is started. Upon completion, the time is stopped and the compilation time is recorded. This metric is accumulated throughout program execution. We repeatedly execute the programs, measuring the compilation time, 100 times using a dedicated processor.

$$Execution\_secs = average\_total\_time - Compilation\_secs$$

Average total time (compilation plus execution) is computed by repeatedly timing (100 times) application execution using uninstrumented compilation environments. We refer to this value as *Total Time* in the chapters on compilation delay reduction (Chapter IX- XI). The compilation time (Compilation_secs) is then subtracted from this value (average_total_time) to determine the execution time of the program without compilation.

Chapter V

General Solutions for Reducing Transfer Delay

The performance of Internet-computing applications that use remote execution (*mobile* programs) is dictated by the speed of the processor on which it executes as well as by the underlying, dynamically changing, network characteristics. As the gap between processor and network speeds continues to widen, mechanisms to compensate for transfer time are required to maintain acceptable performance of mobile programs. We next present five different techniques that reduce the effect of transfer delay and substantially improve mobile program performance.

General solutions to the problem of transfer delay work in one of two ways: By *overlap*ping transfer with useful work and by reducing the amount that is transferred i.e., *avoid*ing the delay. We first present a methodology that does both: *Non-strict Execution* (NSE). Non-strict execution enables transfer delay overlap and avoidance through JVM modification. We propose an extension to the existing JVM transfer and execution model in which the granularity with which mobile programs transfer and execute is changed from the class file to the method. In addition, the transfer model is further modified to implement a model in which the server *push*es the necessary code and data to the destination for execution. This is in contrast to the existing model in which the destination requests class files as required by the execution. We refer to this existing implementation as the *request* model and our modification of and extension to it as the *push* model.

Non-strict execution requires changes to existing Java Virtual Machine technology to achieve its performance benefits. In Chapter VII, we introduce two techniques that use avoidance and overlap to improve mobile program performance without JVM modification. We first present *Class File Splitting*, a technique that avoids delay by transferring only the code

and data that will be executed, i.e. the *hot* sections. The technique partitions a class file into separate hot and cold class files, so that only the code predicted as used during execution is transferred.

To enable overlap of transfer with useful work without JVM modification, we then present *Class File Prefetching*. This technique enables premature access, and thus transfer, of class files to occur in the background. Using a separate thread of execution to perform prefetching, transfer occurs concurrently with executing application thread(s). When the application thread accesses a (prefetched) class for the first time, the class has partially or completely transferred so that transfer delay is reduced.

In Chapter VIII, we then consider the effect of existing compression techniques for reducing transfer delay in mobile programs. We present *Dynamic Compression Format Selection* (DCFS), a technique that exploits the trade-off made by all compression algorithms: To achieve high compression ratios requires that there be expensive (in terms of time) decompression algorithms. With DCFS, on-line measurement of dynamic network performance is used to guide selection of the compression format the minimizes the total delay required for transfer and decompression. The application is stored at the server in multiple compression formats; DCFS uses the underlying resource performance characteristics to determine which format the application should be transferred in to achieve the least delay (from transfer and decompression).

In this same chapter, we also present a technique that archives and compresses together only those files that will be used during execution. This profile-guided technique, called *Selective Compression* reduces the amount that transfers in a compressed archive and thus transfer delay. We detail each of these techniques for the reduction of transfer delay, the first source of load delay overhead, in the three chapters that follow. Within each chapter, we describe the implementation of each technique and use empirical data to evaluate the extent to which each reduces transfer delay.

Chapter VI

Transfer Delay Avoidance and Overlap: Non-strict Execution

Network transfer delays can result in significant startup time and substantially degrade 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 [27] 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 program verification. Unfortunately, strictness prevents overlap of execution with network transfer, and little can be done to reduce the cost of transfer latency.

In this chapter we investigate the efficacy of *non-strict execution (NSE)* in which methods execute at the remote site before transfer has completed for the class the methods are contained in. This small change enables an abundance of optimizations to be implemented for the reduction in transfer delay through overlap and avoidance. Overlap is enabled when transfer is able to continue in the background concurrently with execution. Since execution can begin once method code and data is available, transfer of the remaining class file can be performed while the method is executing.

Transfer delay avoidance is enabled through file restructuring. Since methods can begin executing earlier than with strict execution, transfer delay can be minimized when methods are transferred in the order they will execute. This order is predicted through the use of off-line profiling techniques. File restructuring allows transfer of unused code and data to be avoided.

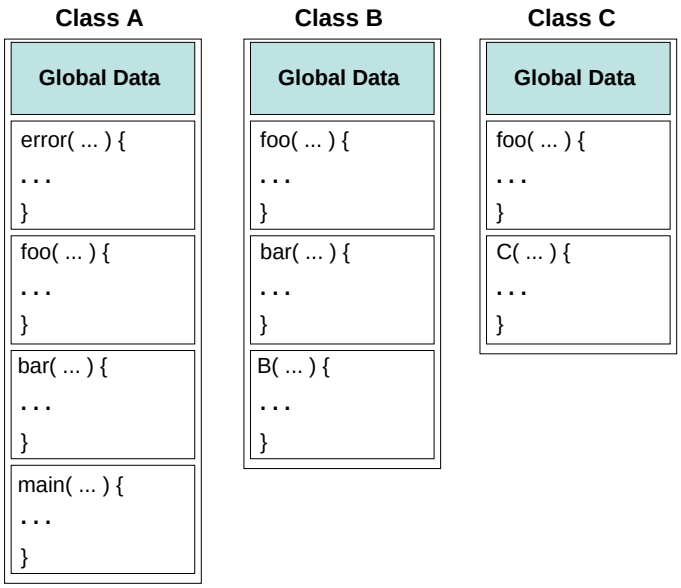


Figure VI.1: Example Java Application

In addition, since method execution order is determined using this technique, there is no need for the client to request program pieces as required by the execution. Using this *push* model as a replacement for the existing *request* model, transfer time required for requests is also avoided.

VI.A Design and Implementation

Using existing implementations of the JVM, each Java class is contained a separate file. Figure VI.1 provides a visual of the class files from an arbitrary application. There are three classes, A, B and C, containing four, three, and two methods, respectively. The classes also contain global data (as denoted). The order of the methods in the class file is equivalent to that in the Java source file from which the bytecode is generated.

Each time a class file is accessed by the executing program or JVM (for verification purposes) it is loaded into memory. Non-local class files are transferred at this time. During this dynamic class file loading, execution is stalled and does not proceed until the all necessary class files are completely loaded. We call this constraint *Strict Execution*. Strict execution of classes imposes a major performance limitation on Internet-computing programs. Given existing network transfer delays, the startup time and overall transfer delay using strict execution can be significant.

To decrease transfer delay, we propose a *Non-strict* execution and transfer model in which execution and transfer both occur at the method-level. *Method-level execution* (MLE)

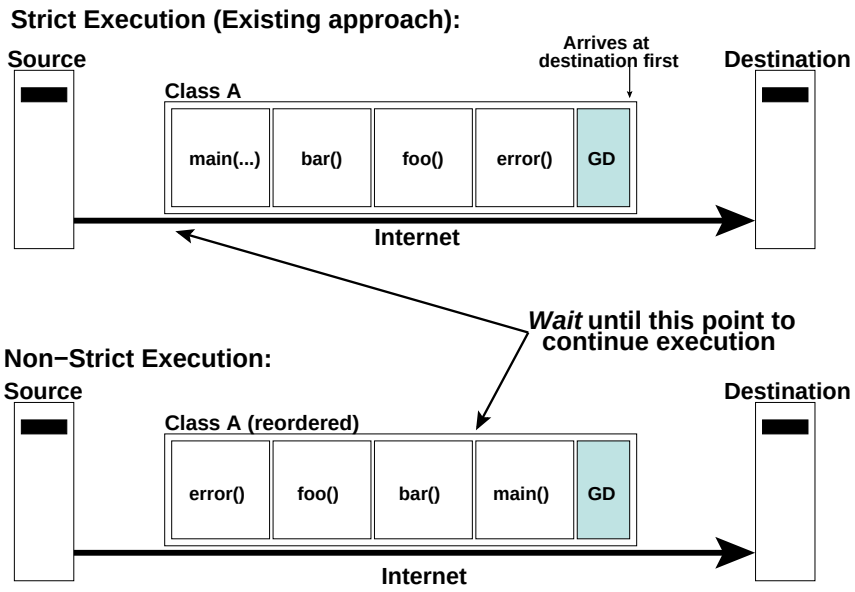


Figure VI.2: Strict v/s Non-Strict Execution

is implemented so that method execution takes place when the necessary code and data are in memory. This precludes the requirement that the entire class file be available to invoke a method within the class file. We address the implications of MLE on the Java verification mechanism in Section VI.A.3. *Method-level transfer* (MLT) is necessary so that method code and related data can be identified in the bytecode stream and placed in memory for execution. MLT is implemented by the inclusion of method delimiters in the bytecode stream. We describe the implementation of MLT and delimiters in Section VI.A.1 on program restructuring.

Both strict and non-strict execution are depicted in Figure VI.2. Strict execution is shown in the top half of the figure. A class file (Class A from our example above in this case) must transfer to completion before any method or field within the class is invoked or accessed, respectively. With non-strict execution (bottom half of the figure), method execution begins once the necessary code and data are loaded. That is, even if the class file containing the invoked method has not completed transfer, the method can execute. This enables execution to proceed earlier since it is not required to stall waiting for transfer and loading of the remaining portion of the class. Method-level transfer enables identification of method code and data boundaries and method-level execution allows execution to proceed once the code and data are in memory.

We propose method-level execution and transfer due to the modularity that methods provide. Non-strict execution can also be performed at the basic block level; however, preliminary experiments show that checking for a delimiter at the conclusion of each basic block

incurs substantial overhead with little added benefit. In addition, 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 methods can still benefit by using the compiler to break the method up into smaller methods. We do not perform any method splitting since the methods in our test programs are of reasonable size.

If the methods in the class file are in the order they will execute (as in the Class A (Reordered) in the bottom half of the figure), then transfer delay can be substantially reduced since only the code and data required for continuation of execution is waited for. If no reordering is performed (as in the top half of the figure) then the transfer delay incurred using non-strict execution can be equivalent to that of strict execution. We use program restructuring, an optimization in which code and data within a program is reordered, to exploit the non-strict execution model and reduce transfer delay.

VI.A.1 Transfer Schedules

To restructure programs for use in a non-strict execution environment, we first break each application into pieces called *transfer units*. Transfer units consist of method code and the global data required to execute the method. From the transfer units we construct a *transfer schedule* which contains the transfer-unit representation of the program. The transfer schedule is shipped from the source to the destination for remote execution of the program.

Transfer units are placed into the transfer schedule in the order in which the methods contained within them will execute at the destination. Since such an ordering is input-dependent and future input use is not known, we must *predict* future execution order. Unlike prior code reordering research, this ordering is the *First-Use* ordering of methods. That is, our goal is to place transfer units (methods) into the transfer schedule in the order they are first used by the executing program. We examine the performance of two first-use prediction techniques. The first approach uses static program estimation and the second uses off-line profiling to predict the first-use method order.

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 method 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 methods are first encountered during static traversal of the flow graph determines the first-use transfer order for the methods. When processing conditional branches inside of a loop, the first-use traversal traverses all the basic blocks inside the loop searching for method 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 first-use method order discovered by the static traversal is output.

Profile-Guided, First-Use Estimation

A first-use profile is generated by logging the order in which methods are invoked during a program's execution using a particular input. Any unexecuted methods are given a first-use ordering using static estimation described in the previous section. To evaluate the impact of profile-driven prediction, we measure the reduction in transfer delay when the same input is used to construct the transfer schedule as is executed with at the destination as well as when a different input is used for transfer schedule construction. We refer to the former as Ref-Ref results and the latter as Ref-Train results.

VI.A.2 Program Restructuring

Both static estimation and profile-base first-use ordering algorithms output the methods as a first-use call graph like the one depicted in Figure VI.4 for our sample application

```
class A {
  public B varB;
  A() { . . . }
  main( . . . ) {
    bar();
    varB = new B();
    foo();
    varB.foo();
  }
  foo() { . . . }
  error() { . . . }
}
```

```
class B {
  public C varC = null;
  public int var1;
  private int var2;
  protected int var3;
  B() {
    var1 = var2 = var3 = −1;
  }
   bar () {
     (varC = new C()).foo();
     var2 = 0;
   }
}
```

```
class C {
  C() { . . . }
  foo() { . . . }
}
```

Figure VI.3: Example application code

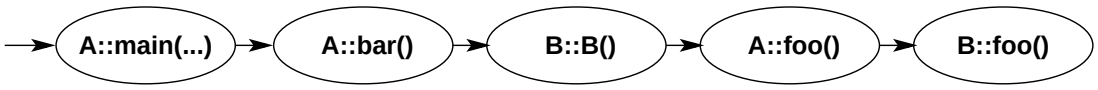


Figure VI.4: An example of a first-use call graph
The first use call graph is generated using the static first use estimation or the profile. It is then used to generate the transfer schedule for remote execution.

introduced in Figure VI.3. Each node in the graph identifies a class and method, the first node is the first method to execute and the final node is the last method to execute.

There are many ways to construct a transfer schedule in the predicted method execution order. For example, we can reorder just class files and leave the methods within them in the default order. We can reorder within classes but not across them. We can reorder across classes and put all of the global data up front to ensure that all data is available when used. Or we can distribute both methods and global data across all classes of an application in the first-used order. We describe each of these constructions and measure its effect on transfer delay.

The first type of program restructuring is to reorder class files within an application without modifying the code and data within them. This is the simplest of transfer schedule construction algorithms. Complete class files are placed in their entirety into the transfer schedule in the order the class files are first accessed. The methods within the class files are in the order in which programmer coded them (default ordering). Such a reordering and schedule is shown in Figure VI.5. Methods begin executing (as for all transfer schedules) as soon as the necessary code and data becomes available. We refer to this type of transfer schedule construction in our results as MLE (method-level execution).

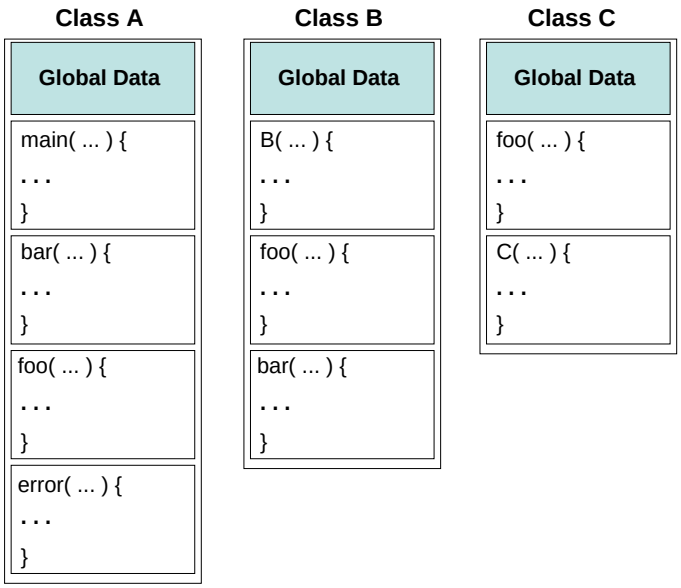


Figure VI.5: Restructured class files
The example application is reordered according to the first-use static call graph pictured in Figure VI.4. Classes are restructured so that methods appear in the order each is first invoked.

For the next transfer schedule, we reorder the methods within the classes (not across classes) as well as the classes themselves prior to schedule placement. The resulting transfer schedule for the example program is shown in Figure VI.7. We refer to this type of schedule as MLE plus *Intra-Class Reordering (CR)* (MLE + CR). Alternately, we can construct a transfer schedule by *interleaving* methods across classes as shown in Figure VI.8. For our results, we refer to this MLE plus *Global Method Reordering (MR)* (MLE + MR).

In each of these types of transfer schedule, the global data in each class file is placed just prior to the method placed earliest in the transfer schedule. An alternative placement would be to also distribute the global data throughout the transfer schedule according to its use by methods (determined by a static scan of the bytecode). Figure VI.9 depicts this scenario in the final transfer schedule construction algorithm. The global data required for execution of a method is placed just prior to the earliest method that uses that global data. We are able to determine which methods use which global data using static compiler analysis of the class files. We refer to this MLE plus MR plus *Global Data Reordering (GDR)* (MLE + MR + GDR) in our results. Any global data (like methods) that are unreachable or accessed only by methods predicted as unused during first-use call graph construction are placed at the end of the transfer schedule in the order determined by the modified-DFS of unused methods.

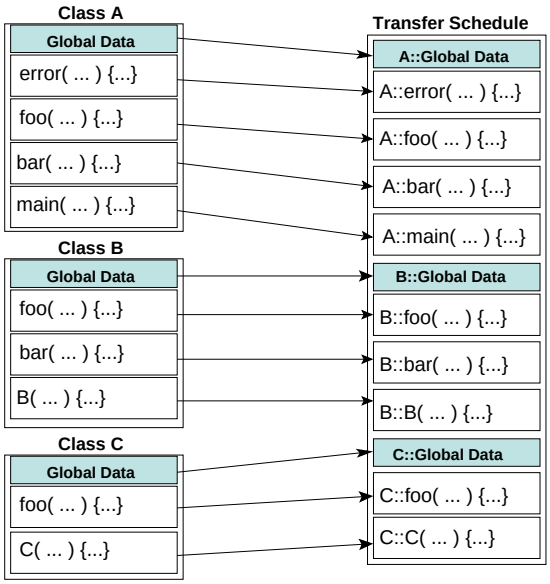


Figure VI.6: NSE transfer schedule for method-level execution.
This transfer schedule is a combination of all of the class files in an application. Class files are inserted into the schedule in the order they are predicted to execute. However, no reordering within class files is performed. A method can execute at the destination once the necessary code and data becomes locally accessible. We refer to this in our results as method-level execution (MLE).

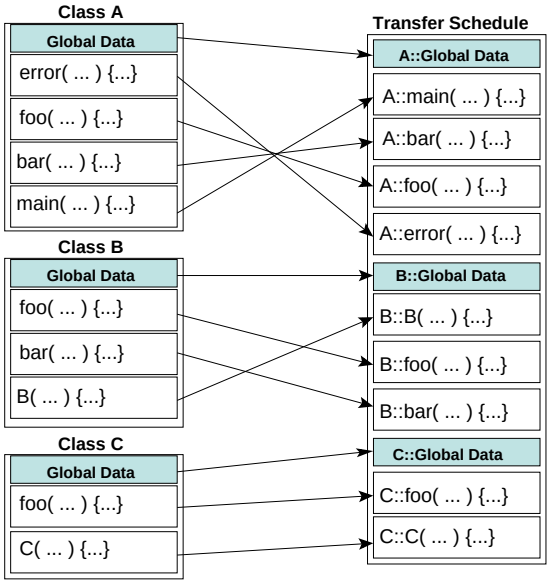


Figure VI.7: NSE transfer schedule for MLE and intra-class reordering.
Methods are first reordered within class files and then the class files are inserted into the transfer schedule. There is no reordering performed across classes. We refer to this in our results as method-level execution plus intra-class reordering (MLE + CR).

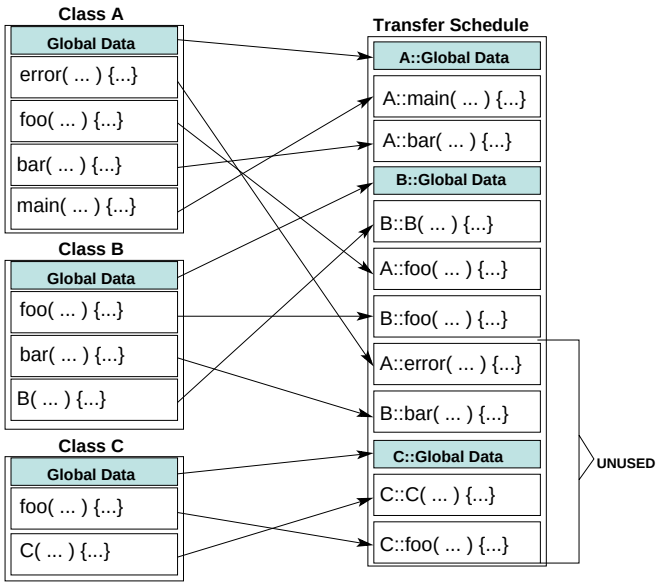


Figure VI.8: NSE transfer schedule for MLE and global method reordering.
For this transfer schedule, methods are reordered across all class files so that they can be interleaved into the schedule in the order they are predicted to execute in. We refer to this in our results as method-level execution plus global method reordering (MLE + MR).

Once the transfer schedule is constructed, it is transferred to completion from the source to the destination when the program is invoked. If the program exits before the transfer schedule has completed transfer, transfer is ceased. When this occurs (and it is the common case with the programs we studied), non-strict execution reduces the amount that transfers (over dynamic class file loading) thereby avoiding transfer delay.

If the code (or data) has not arrived when needed by the execution, the program stalls (as in the existing JVM class loading mechanism) until it arrives at the destination (according to the schedule). Our results indicate that even when predicted first-use order is incorrect and the program must stall to correct the misprediction, transfer delay is still reduced just not to the same degree as when prediction is correct. If this model proves to increase transfer delay (degrading strict execution performance) then we propose to use a *hybrid* transfer model which combines the push and request models. Using this model, the transfer schedule contains *only* those methods predicted as used during execution. If additional methods are invoked that were not predicted as used, they are requested on-demand by the destination. This incurs the overhead of round-trip time for the request but may require less transfer delay than if waiting for the invoked method in the transfer schedule. However, for the programs we studied in this thesis, misprediction has not proven to cause performance degradation.

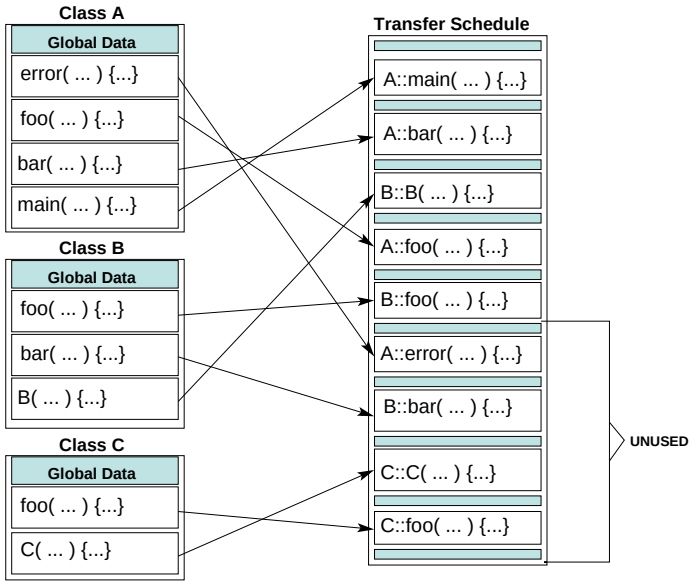


Figure VI.9: NSE transfer schedule for MLE, MR, and global data reordering.
For this transfer schedule, methods are reordered across all class files so that they can be interleaved into the schedule in the order they are predicted to execute in. In addition, instead of placing all of the global data prior to the first method transferred from each class, we distribute it throughout the transfer schedule. Global data used by a method is placed in the transfer schedule just prior to the first method in the program that accesses that global data. We refer to this in our results as method-level execution plus global method reordering plus global data reordering (MLE + MR + GDR).

Notice that since the transfer schedule dictates the order in which methods and global data are transferred (and thus the order of class files first-access), there is no need for the destination to dynamically request parts of the program via the JVM dynamic class file loading mechanism. We refer to this existing model as the *request* model. Our measurements indicate that requesting class files over a cross-country Internet link requires approximately 100ms; adding to the total transfer delay. By eliminating this overhead imposed by the request model through the use of transfer schedules, we are able to further reduce the transfer delay imposed by existing technology. We refer to the incorporation of transfer schedules for remote execution as the *push* model.

VI.A.3 Implications on JVM Verification

For our non-strict execution approach to be viable, we need to address its effect on Java class file verification. Verification enables security checks to be performed by the JVM to ensure a set of constraints are met. These constraints are described in the Methodology Chapter and detailed in [59]. In this subsection, we provide a high level overview of the the JVM verification changes that are necessary to support non-strict execution. Verification is, by default, performed on all non-local class files but can be performed on all class files or turned off completely.

To convert the Java bytecode class representation to the internal JVM representation for execution, the JVM performs (1) *verification*, (2) *preparation*, and (3) *resolution* on the bytecode stream when the class is first loaded. Verification is the process of checking the structure of a Java class file to ensure that it is safe to execute. Preparation involves allocation of static storage and any internal JVM data structures needed for execution of the loaded class file. 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 methods are invoked.

With non-strict execution, verification and preparation must be modified to perform lazily (as needed) as well. Since preparation, is performed using the global data in the class file (any non-code information about the class file structure). As global data becomes available, static allocation (preparation) can be performed. Modification to the verification mechanism is more complicated.

The JVM performs five steps during verification of a class file as described in [59]. Step

1 ensures that the class file format is correct, Step 2 ensures that static constraints are met on the constant pool and that field and method references are well formed. Step 3 verifies that the method code in the class meets specific constraints (this includes checks for consistent/correct usage of types, operand stack, method arguments, etc.). Step 4 checks that a newly loaded class is used consistently and correctly by the accessing instruction. Steps 3 and 4 are performed lazily, i.e. only when the method is invoked for the first time or a new class is accessed is step 3 or 4 performed, respectively. As such, only steps 1 and 2 need to be modified for use with non-strict execution.

Since both step 1 and 2 perform checks on the global data (class file structure) and do not access method code, we propose to change each step in a similar manner. Each time a class file is accessed for the first time, a "skeleton" of the class file is laid out in memory. As global data becomes available, the class file skeleton is filled in. During this incremental process, checks in steps 1 and 2 are performed on the newly added portions of the class file. When other classes are needed for cross-class dependency resolution, we use this same mechanism to perform the necessary checks incrementally. For example, to verify that a subclass implements the correct parent type, the global data from the parent class is transferred (using the non-strict execution model) and implemented into a class file skeleton and verified. This enables execution to proceed without waiting for the entire parent class to transfer since only the information required for verification is needed. When the parent class eventually transfers, the corresponding class file skeleton is filled in and incrementally verified just as all other class files are.

Other ways to ensure that a Internet-computing program can be trusted include the use of digital signatures [10] or software fault isolation [86]. We do not address the changes necessary for such verification systems to work with non-strict execution. We do however, measure the impact of our techniques both with and without verification.

VI.B Results: Non-strict Execution

To evaluate the impact of non-strict execution and program restructuring we present simulation results. Our simulation model is described in detail in Chapter IV (Methodology). In brief, we intercept program execution at each basic block and compute the amount of overlap that can occur during the execution of the block. In addition, we compute the amount of transfer (of the transfer schedule) that can occur during this overlap. Each time a method is

invoked or global data is accessed we check that the transfer schedule has transferred enough to make the code and data available. If it is not available, we determine the delay incurred by the necessary transfer. Once available, execution proceeds until the next basic block is executed.

We first present our results for trusted transfer in which no verification is used e.g., by setting the Java -noverify flag. Following this, we present results for verified transfer (in which verification of class files is performed).

VI.B.1 Trusted Transfer

Figures VI.10, VI.11, and VI.12 show the transfer delay (in seconds) that is experienced by the application with and without non-strict execution. A separate graph is shown for each benchmark, each containing both same-input (Ref-Ref) and cross-input (Ref-Train) results. The x-axis is bandwidth values for various network links ranging from a modem link (0.03Mb/s) to a T1 link (1Mb/s). For each network bandwidth there are eight bars. The first bar (far left) is the base case (strict execution) transfer delay. The second bar from the left for each bandwidth shows the effect of method-level execution (MLE) without any reordering; the transfer schedule for this contains the class file containing the "main" method first followed by all other class files in alphabetical order. The methods within the class files are in the default order. The next three pairs of bars (in the set of eight) show the results for profile-guided transfer schedule construction for each type of schedule previously described; for each type, Ref-Train and Ref-Ref results are shown. Ref-Train results again are those for which a different input was used to generate the profile that guides reordering than was used to generate the results. Ref-Ref results are those for which the same input was used for profile and result generation. The different types of transfer schedules can be summarized as follows:

- MLE + CR: method-level execution with method reordering within class files (intra-class reordering). Execution proceeds at the method-level and no interleaving of the class files is performed during transfer schedule construction. The transfer schedule is like that shown in Figure VI.7. Results are shown for both Ref-Train and Ref-Ref configurations.
- MLE + MR: method-level execution with interleaved class files. Method reordering is performed globally across all class files in an application. The transfer schedule is like that shown in Figure VI.8. Results are shown for both Ref-Train and Ref-Ref configurations.
- MLE + MR + GDR: method-level execution with global class, method, and data reordering in an interleaved transfer schedule. The transfer schedule is like that shown in Figure VI.9. Results are shown for both Ref-Train and Ref-Ref configurations.

The average strict-execution transfer delay for these benchmarks is 56 seconds. The average number of classes requested using the base case request model is 70, therefore 7 seconds

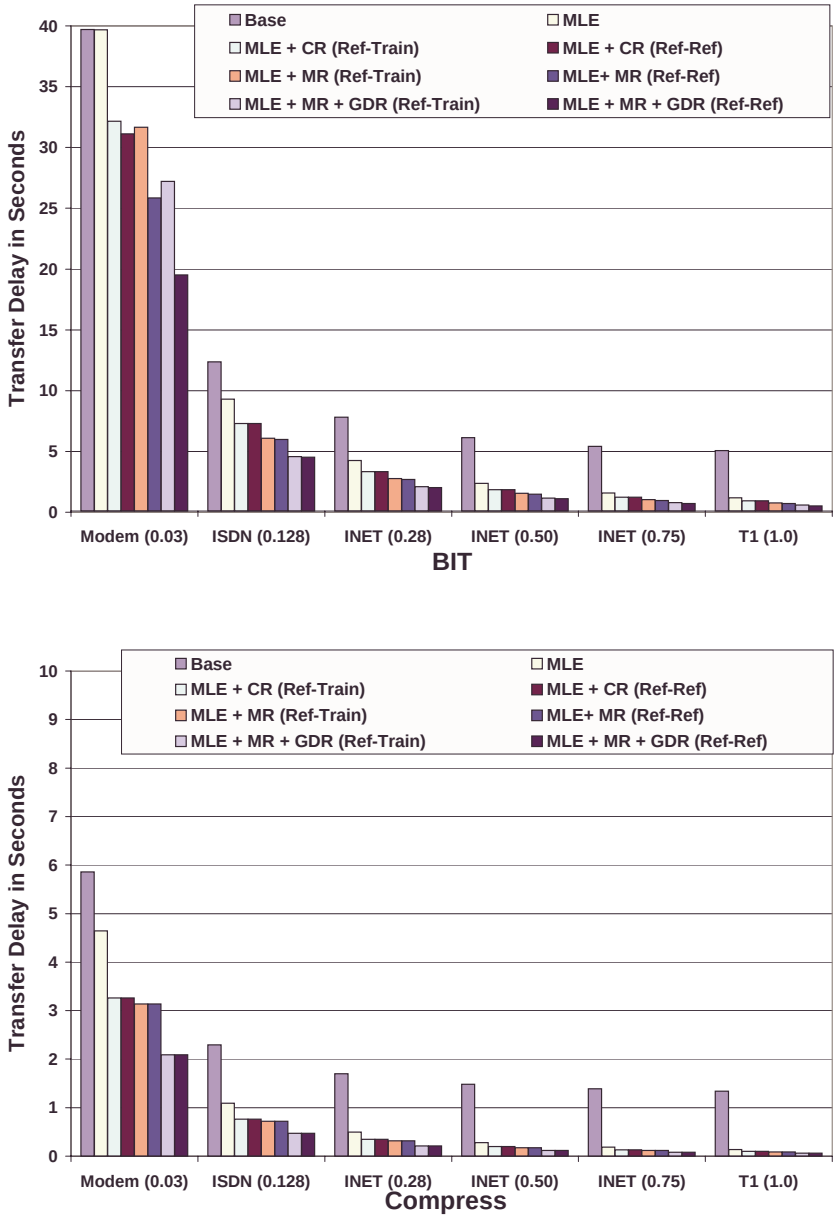


Figure VI.10: Resulting non-strict transfer delay for benchmarks Bit and Compress
Each graph provides a set of bars for each network bandwidth (x-axis). The y-axis is seconds due to transfer delay. From left to right, the eight bars in a set represent the total transfer delay that results from strict execution (Base), and non-strict execution using method-level execution (MLE) alone, (MLE) plus intra-class reordering (CR) (Ref-Train and Ref-Ref), MLE plus global method reordering (MR) (Ref-Train and Ref-Ref), and MLE plus MR and global data reordering (GDR) (Ref-Train and Ref-Ref).

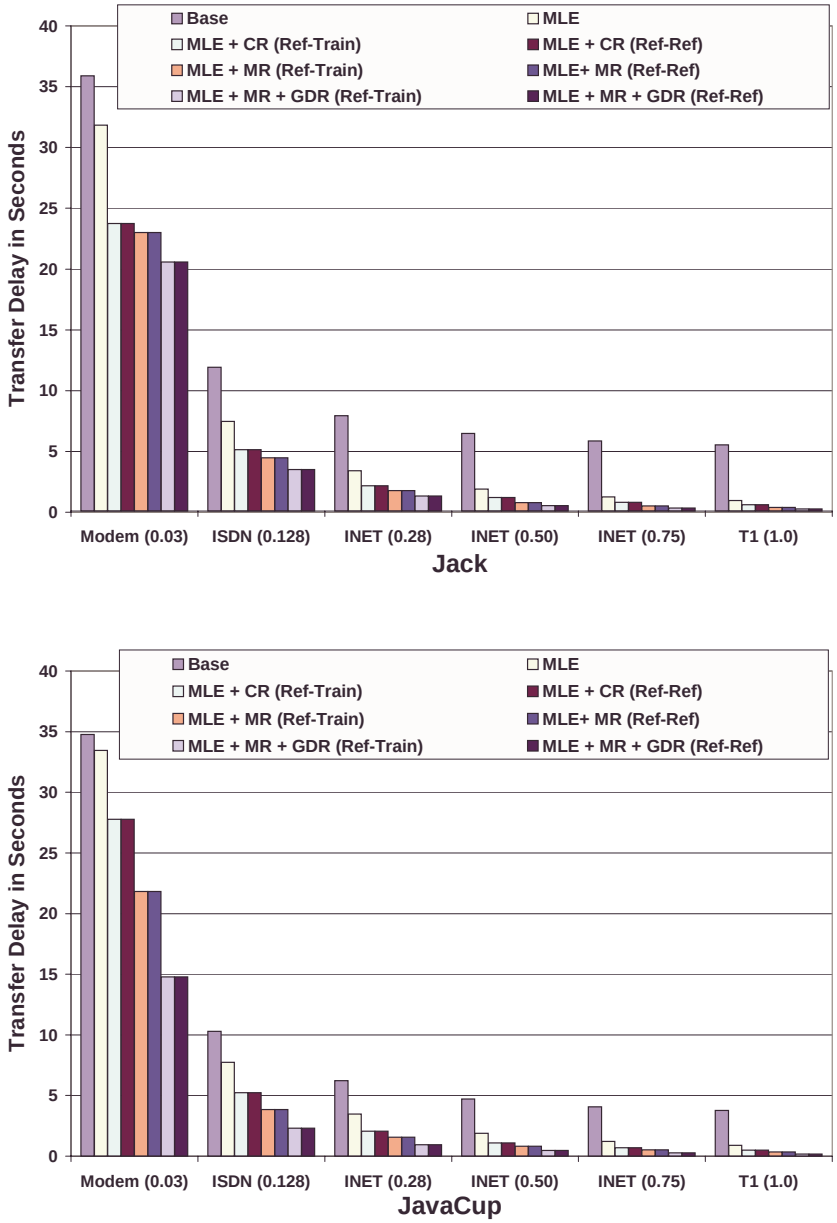


Figure VI.11: Resulting non-strict transfer delay for benchmarks Jack and JavaCup
Each graph provides a set of bars for each network bandwidth (x-axis).The y-axis is seconds due to transfer delay. From left to right, the eight bars in a set represent the total transfer delay that results from strict execution (Base), and non-strict execution using method-level execution (MLE) alone, (MLE) plus intra-class reordering (CR) (Ref-Train and Ref-Ref), MLE plus global method reordering (MR) (Ref-Train and Ref-Ref), and MLE plus MR and global data reordering (GDR) (Ref-Train and Ref-Ref).

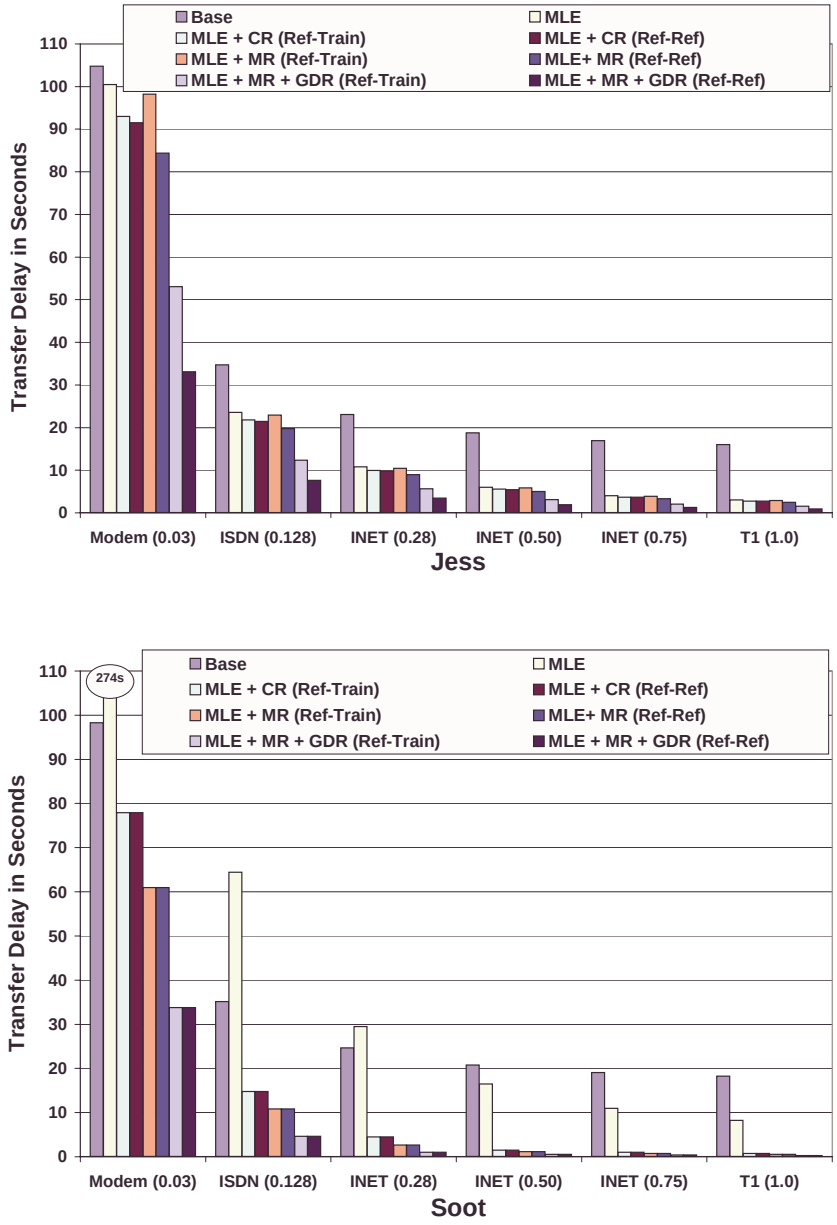


Figure VI.12: Resulting non-strict transfer delay for benchmarks Jess and Soot
Each graph provides a set of bars for each network bandwidth (x-axis). The y-axis is seconds due to transfer delay. From left to right, the eight bars in a set represent the total transfer delay that results from strict execution (Base), and non-strict execution using method-level execution (MLE) alone, (MLE) plus intra-class reordering (CR) (Ref-Train and Ref-Ref), MLE plus global method reordering (MR) (Ref-Train and Ref-Ref), and MLE plus MR and global data reordering (GDR) (Ref-Train and Ref-Ref).

of transfer delay in the average base case is due to round-trip request time. Non-strict execution uses the push model and incurs no such overhead. In our discussion of results, we refer to only the cross-input results (Ref-Train) since they are more representative of real world non-strict execution performance.

Using MLE (method-level execution alone without reordering), 0 to 5 seconds of transfer delay can be reduced on the modem link and 3 to 10 seconds for the T1 link for all but the Soot benchmark. Notice though, that for Soot, MLE alone actually increases transfer delay. This is because no file restructuring is performed and the default ordering includes many unused methods and data and hence mispredicts them across inputs. This increase emphasizes the need for file restructuring to be combined with non-strict execution.

Using reordering within classes but not across them (MLE + CR), transfer time reduction for the modem link ranges from 2.5 to 20 seconds and a T1 link, 1.1 to 16 seconds. Global method reordering (MLE + MR) results show 2.6 to 38 seconds of reduction for modem and 1.1 to 16 for T1. Global data reordering (MLE + MR + GDR) reduces transfer time for the modem by 3.8 to 62 seconds and 1.1 to 16 seconds for the T1 link on average. In every case, the performance improvement due to restructuring and non-strict execution is substantial. When the raw reduction time is small, it is still a substantial percentage of that which results from strict execution (the base case). That is, for fast links, non-strict execution reduces almost all transfer delay. However, the base case delay for such links is small to begin with. Most of the transfer delay improvement for the fast links results from the elimination of the dynamic class file loading requests (7 seconds on average).

All of the results presented in the previous figure were generated from transfer schedules that are constructed from profile-guided, first-use estimation. We can instead use static estimation, as described in Section VI.A.1, to determine the first-use ordering (using a modified-DFS of the control flow graph of the program). The benefit such estimation offers is that no off-line execution of the instrumented program is required. Figures VI.13, VI.14, and VI.15 show the effect of using a static call graph (SCG in the graphs) to guide our transfer schedule construction for non-strict execution. The graphs are the same as those in the previous figures (Figures VI.10, VI.11, and VI.12) and present the transfer delay in seconds (one graph for each benchmark). However, the same-input (Ref-Ref) results have been removed and bars for static call graph estimation have been added (denoted as SCG). Results for the various types of transfer schedule construction are shown for both SCG results and profile-based, cross-input results (method level execution (MLE) with intra-class reordering (MLE + CR), with global

method reordering (MLE + MR), and with global method and data reordering (MLE + MR + GDR)).

The results in this figure indicate that static estimation for transfer schedule construction reduces transfer delay, i.e., bars 2, 4, and 6 are all less than the base case (bar 1). For these benchmarks, profile-guided estimation, however reduces transfer delay an additional 250-640ms for 1Mb/s bandwidth (T1) and 6-16 seconds for 0.03Mb/s bandwidth (modem) over static estimation on average. However for these benchmarks, when off-line profiles are unavailable, non-strict execution can be used with static transfer schedules to reduce delay. On average across benchmarks, static estimation reduces delay for the T1 link 2-38 seconds and 5-6 seconds for the modem link.

We have purposely left out Soot benchmark results from this group. A graph for this benchmark is shown in Figure VI.16. Soot proves to be an anomaly in the performance of benchmarks using non-strict execution and static estimation to guide transfer schedule construction. For all links with bandwidths less than 0.5Mb/s, such use of static estimation degrades performance. For 0.5Mb/s bandwidths and greater, it reduces transfer delay however to a much lesser degree than the profile-guided techniques. This results since the predicted first-use method order that is generated using our modified-DFS algorithm is substantially different from the actual first-use order and misprediction proves costly since there are so many methods and class files in application.

We next show the effect of NSE on program startup in Figures VI.17, VI.18, and VI.19 for the modem link (0.03Mb/s) and in Figures VI.20, VI.21, and VI.22 for the T1 link (1Mb/s). Two cumulative distribution functions (CDF) are given in each graphs (one for each benchmark). Each function indicates the cumulative transfer delay (y-axis) in seconds at particular point during execution of the programs (shown as percentage of program execution completed on the x-axis). A CDF is shown for strict execution and for non-strict execution cross-input (Ref-Train) results. The non-strict results shown are those for method-level execution with global method and data reordering. For every benchmark, the reduction in transfer delay translates to substantial progress made at program startup. Much less transfer delay is incurred in the first 10% of program execution. The average execution time of these programs is 49 seconds. Non-strict execution with global method and data reordering reduce the transfer delay that is required during the first 10% (5 seconds) of program execution by 21 seconds for the modem link and 5 seconds for the T1 link.

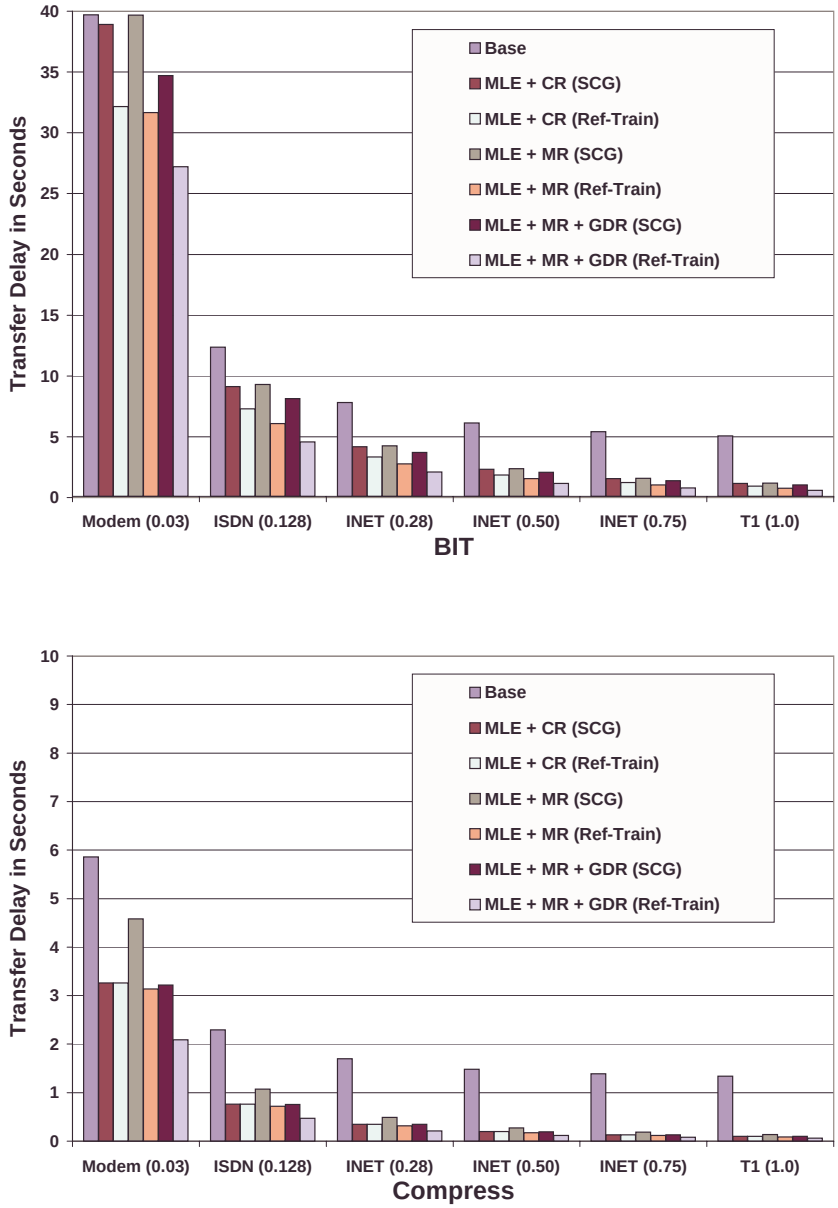


Figure VI.13: SCG transfer schedule construction for benchmarks Bit and Compress. This figure is the same as those shown in Figures VI.10, VI.11, and VI.12 without the Ref-Ref results. In addition, results showing the effect of using a static call graph (SCG) to determine first-use estimation and hence, construct non-strict transfer schedules have been added (denoted by SCG). Use of an SCG precludes the need for off-line profiling. Results for the various types of transfer schedule construction are shown for both SCG results and profile-based, cross-input results (method level execution (MLE) with intra-class reordering (MLE + CR), with global method reordering (MLE + MR), and with global method and data reordering (MLE + MR + GDR)).

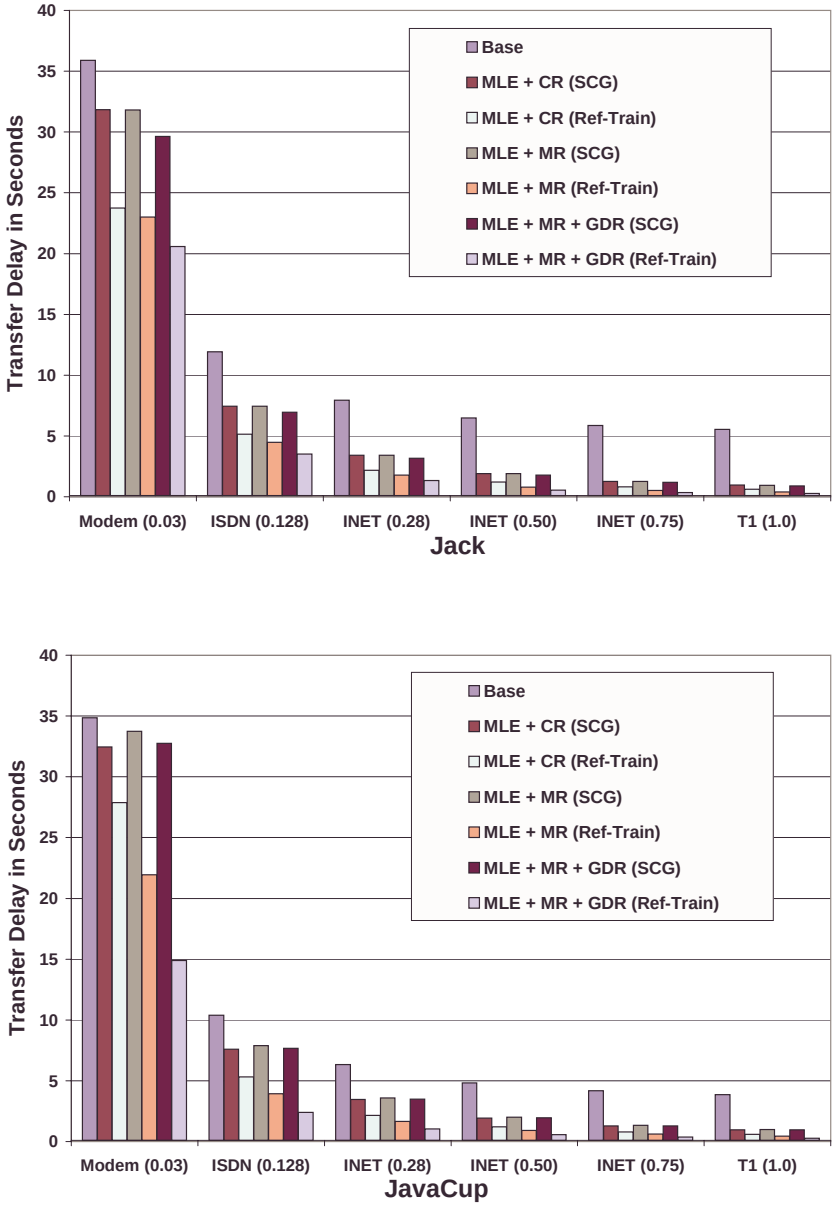


Figure VI.14: SCG transfer schedule construction for benchmarks Jack and JavaCup. This figure is the same as those shown in Figure VI.11 without the Ref-Ref results. In addition, results showing the effect of using a static call graph (SCG) to determine first-use estimation and hence, construct non-strict transfer schedules have been added (denoted by SCG). Use of an SCG precludes the need for off-line profiling. Results for the various types of transfer schedule construction are shown for both SCG results and profile-based, cross-input results (method level execution (MLE) with intra-class reordering (MLE + CR), with global method reordering (MLE + MR), and with global method and data reordering (MLE + MR + GDR)).

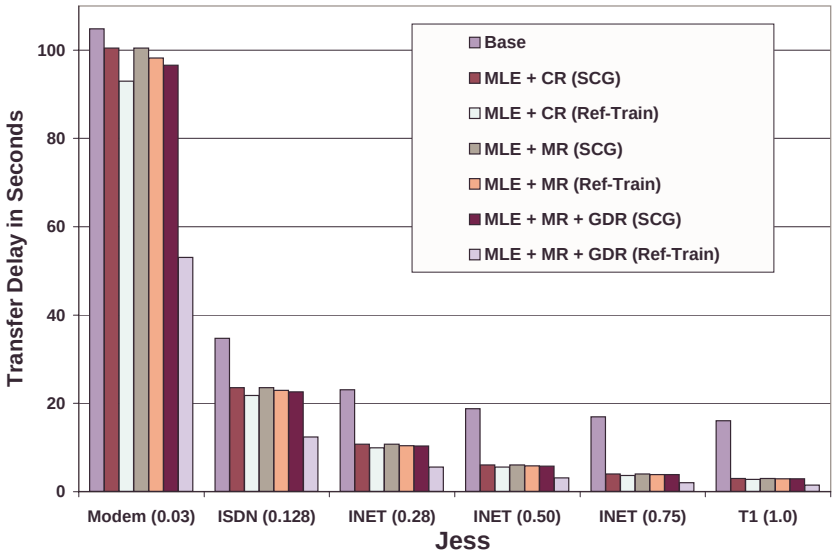


Figure VI.15: SCG transfer schedule construction for the Jess benchmark.
This figure is the same as that for Jess in Figure VI.12 without the Ref-Ref results. In addition, results showing the effect of using a static call graph (SCG) to determine first-use estimation and hence, construct non-strict transfer schedules have been added (denoted by SCG). Use of an SCG precludes the need for off-line profiling. Results for the various types of transfer schedule construction are shown for both SCG results and profile-based, cross-input results (method level execution (MLE) with intra-class reordering (MLE + CR), with global method reordering (MLE + MR), and with global method and data reordering (MLE + MR + GDR)).

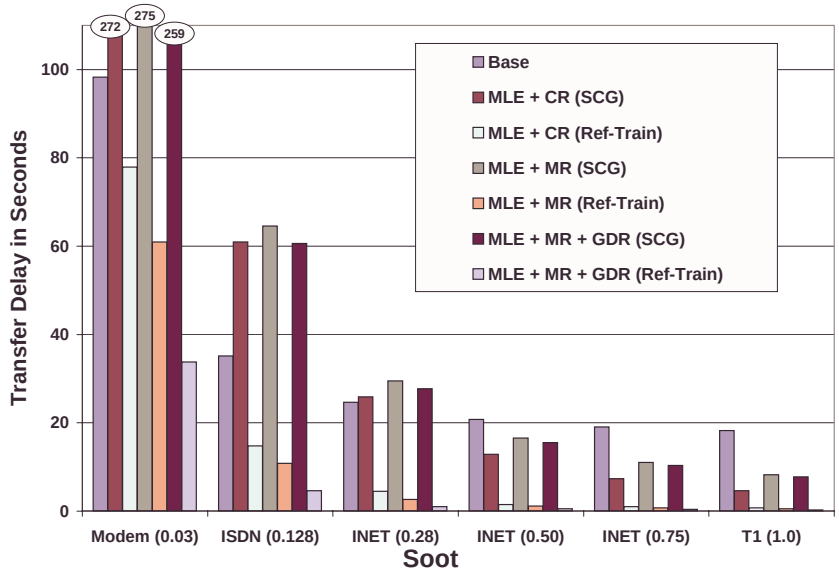
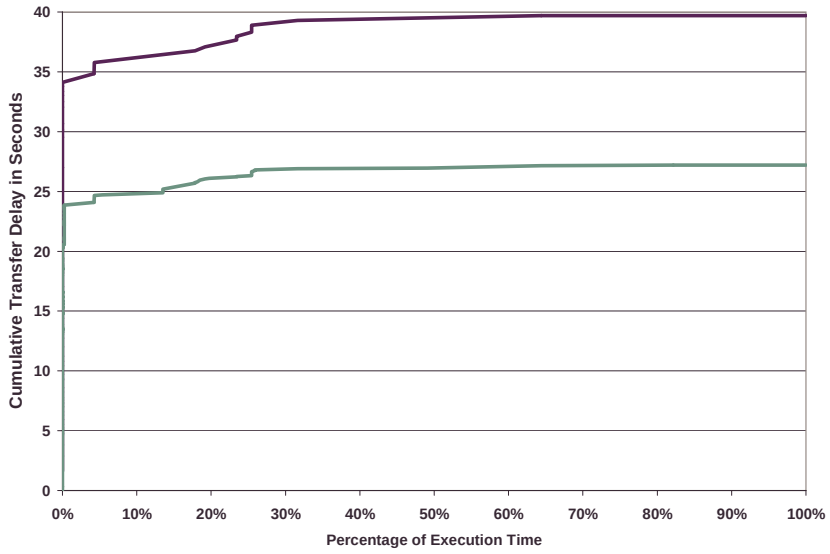
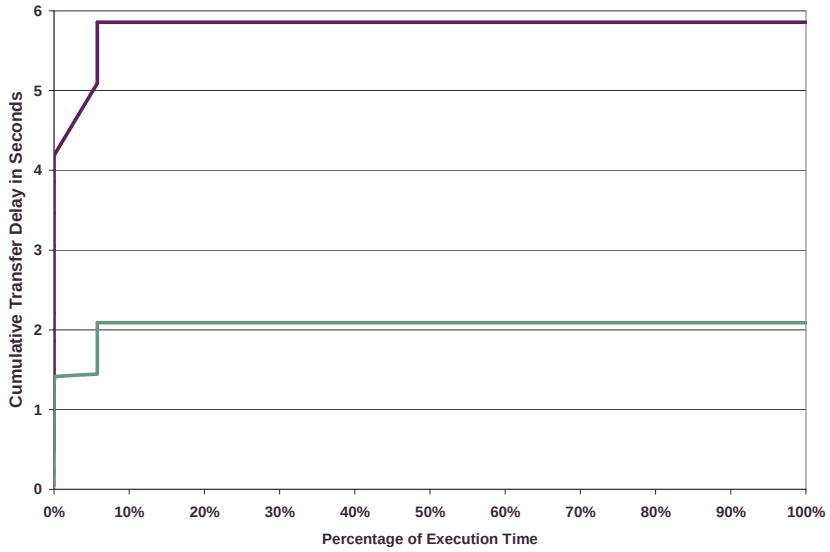


Figure VI.16: Performance degradation for the Soot benchmark using static estimation. This graph is the same as those presented in Figures VI.13, VI.14, and VI.15 but is for the Soot benchmark. It indicates the transfer delay in seconds required without non-strict execution (BASE) and with non-strict execution. Profile-guided transfer schedule techniques (MLE + CR,MLE + MR, and MLE + MR + GDR) are repeated from those in Figures VI.10, VI.11, and VI.12. Three bars have been added for each of the techniques for which static estimation has been used guide transfer schedule construction. For all links with bandwidths less than 0.5Mb/s, such use of static estimation degrades performance. For 0.5Mb/s bandwidths and greater, it reduces transfer delay however to a much lesser degree than the profile-guided techniques. Soot is the only benchmark for which this happens.

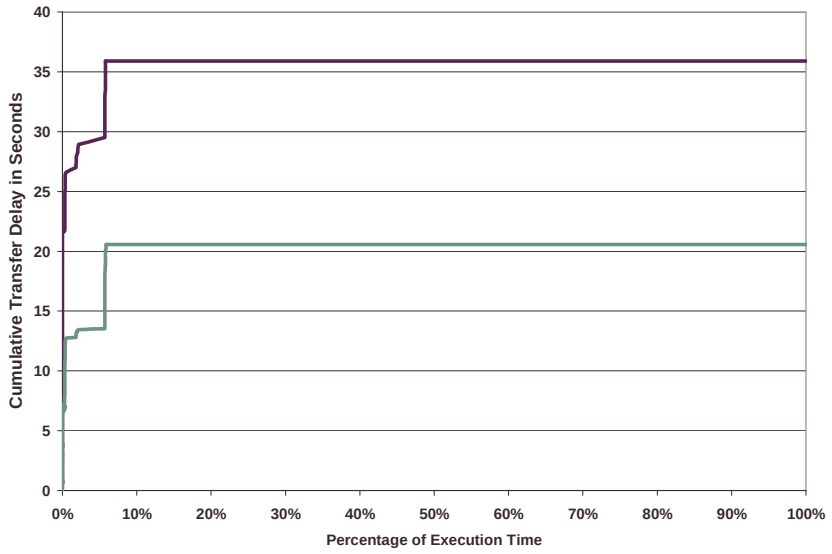


Bit

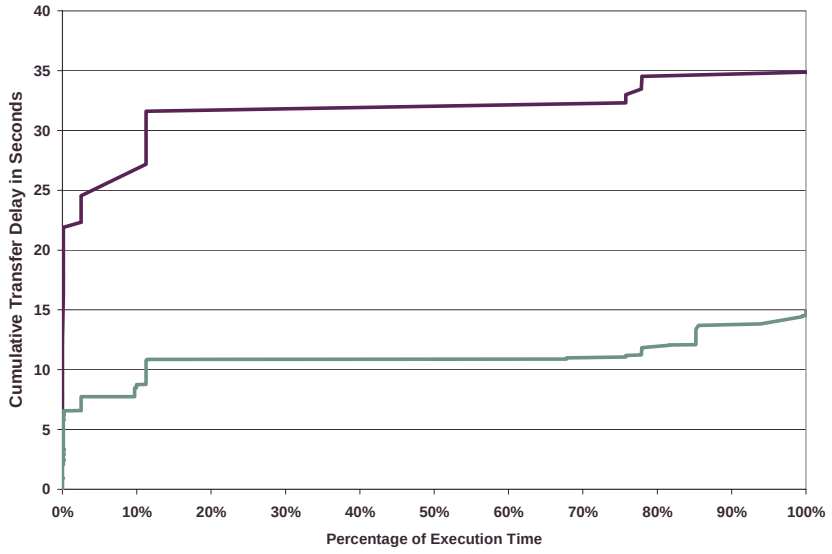


Compress

Figure VI.17: The effect of NSE on program startup (Bit & Compress) and modem link. Each of these graphs (one for each benchmark) shows the cumulative distribution of bytes transferred during program execution time. The top function is strict execution. The lower is non-strict execution with method-level execution (MLE) and restructuring with global data distribution using imperfect information (Ref-Train).

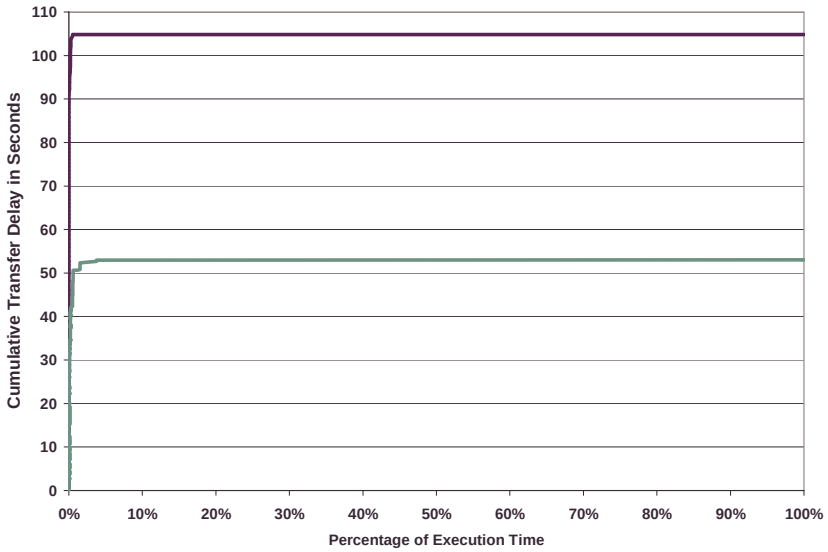


Jack

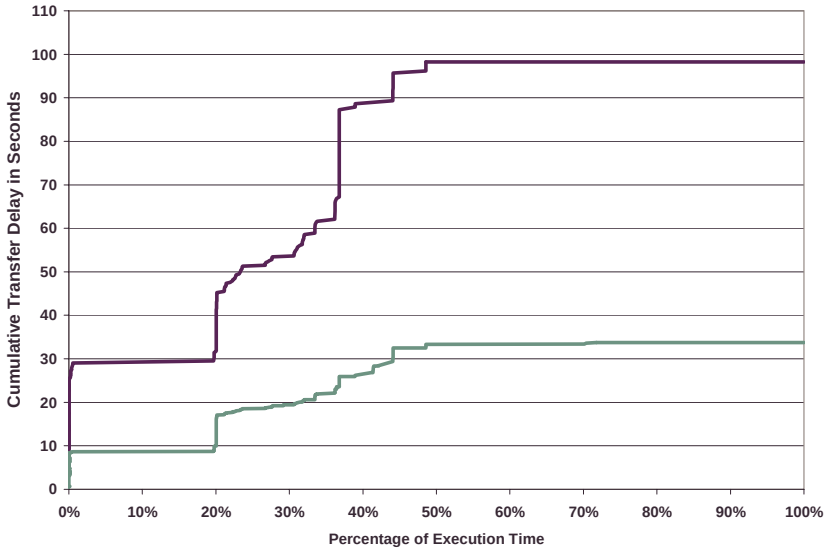


JavaCup

Figure VI.18: The effect of NSE on program startup (Jack & JavaCup) and modem link. Each of these graphs (one for each benchmark) shows the cumulative distribution of bytes transferred during program execution time. The top function is strict execution. The lower is non-strict execution with method-level execution (MLE) and restructuring with global data distribution using imperfect information (Ref-Train).

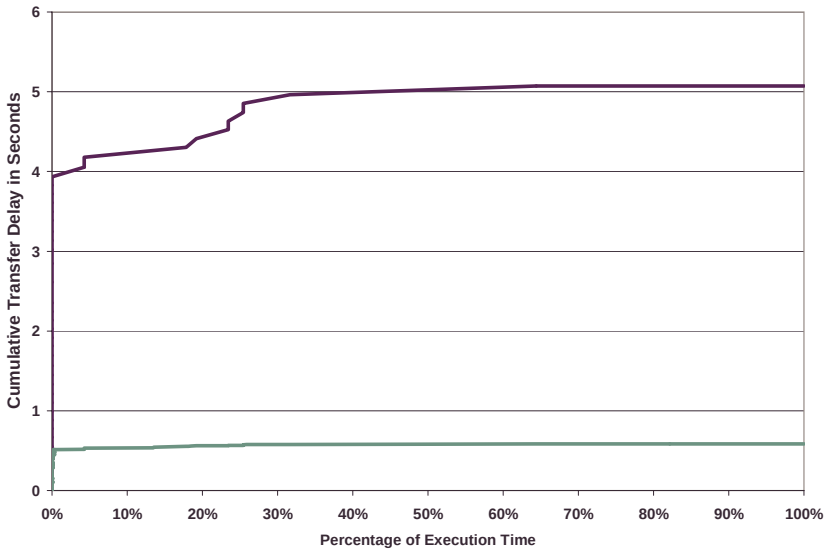


Jess

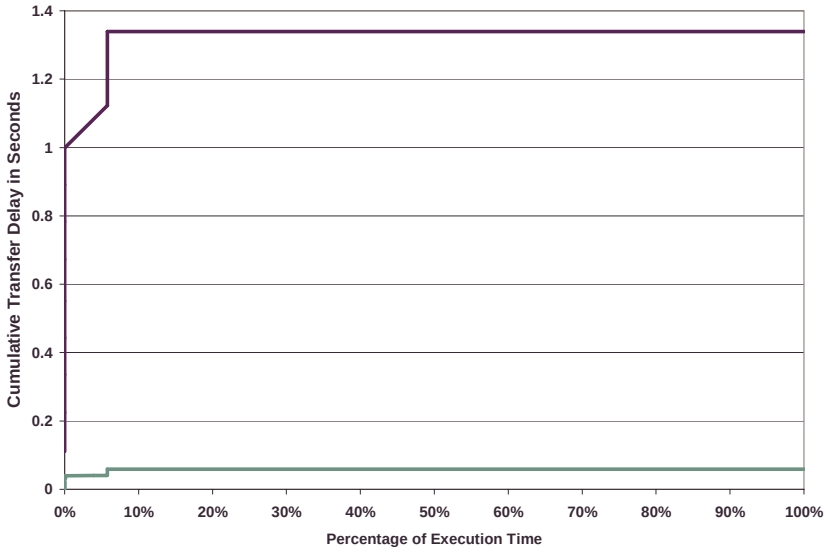


Soot

Figure VI.19: The effect of NSE on program startup (Jess & Soot) and modem link. Each of these graphs (one for each benchmark) shows the cumulative distribution of bytes transferred during program execution time. The top function is strict execution. The lower is non-strict execution with method-level execution (MLE) and restructuring with global data distribution using imperfect information (Ref-Train).

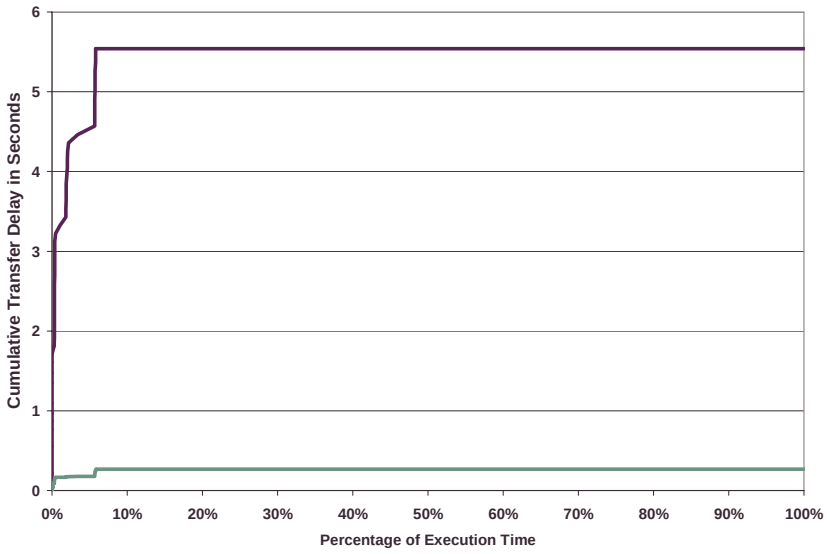


Bit

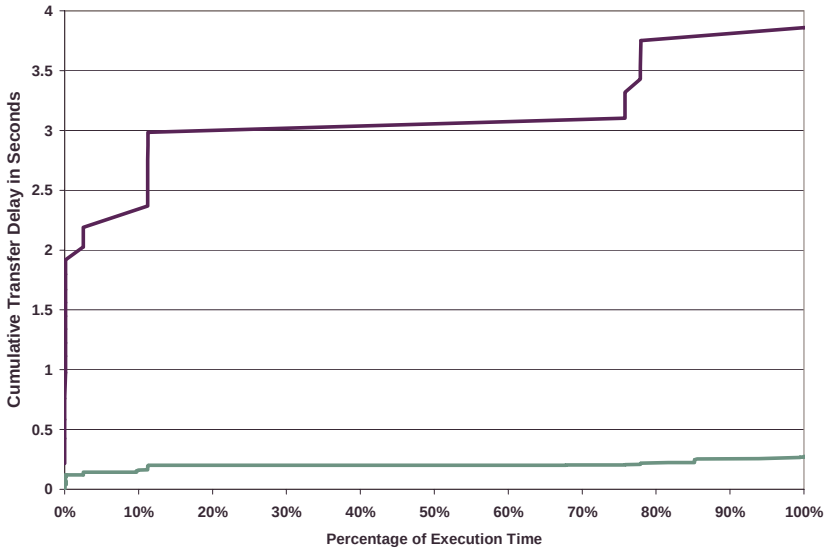


Compress

Figure VI.20: The effect of NSE on program startup (Bit & Compress) using a T1 link. Each of these graphs (one for each benchmark) shows the cumulative distribution of bytes transferred during program execution time. The top function is strict execution. The lower is non-strict execution with method-level execution (MLE) and restructuring with global data distribution using imperfect information (Ref-Train).

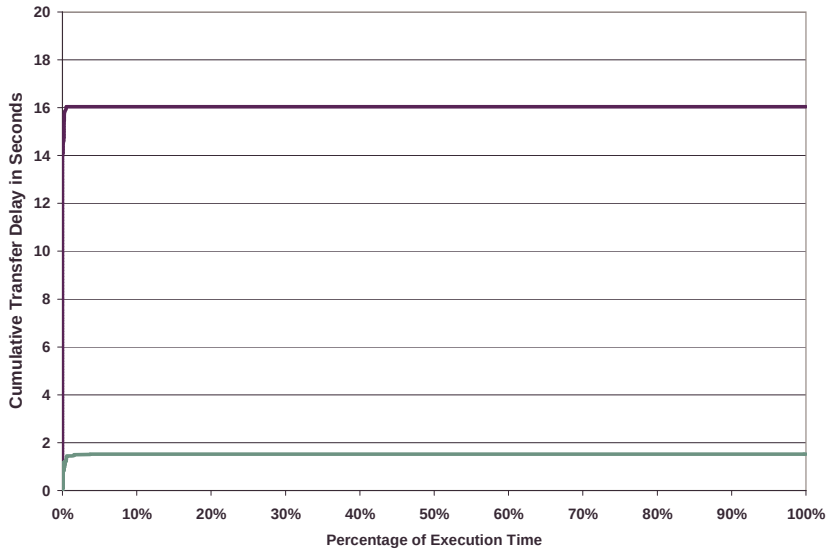


Jack

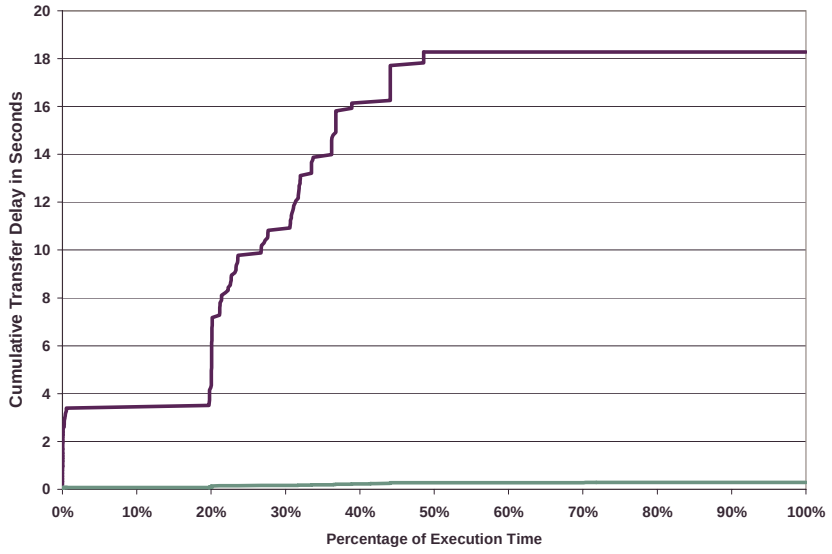


JavaCup

Figure VI.21: The effect of NSE on program startup (Jack & JavaCup) using a T1 link. Each of these graphs (one for each benchmark) shows the cumulative distribution of bytes transferred during program execution time. The top function is strict execution. The lower is non-strict execution with method-level execution (MLE) and restructuring with global data distribution using imperfect information (Ref-Train).



Jess



Soot

Figure VI.22: The effect of NSE on program startup (Jess & Soot) using a T1 link. Each of these graphs (one for each benchmark) shows the cumulative distribution of bytes transferred during program execution time. The top function is strict execution. The lower is non-strict execution with method-level execution (MLE) and restructuring with global data distribution using imperfect information (Ref-Train).

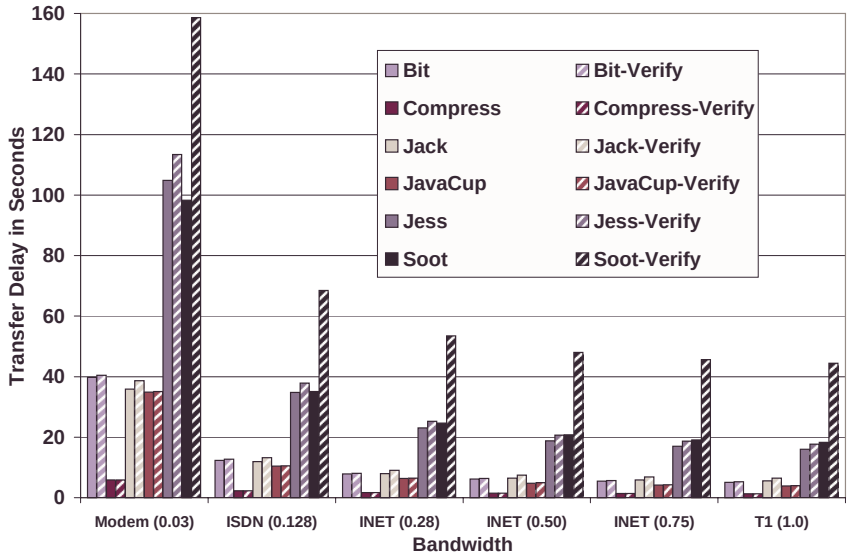


Figure VI.23: Difference in transfer delay for trusted and verified execution.
For each benchmark, there are two bars presented. The first of each pair is the transfer delay for trusted transfer and the second (striped) is for verified transfer. For the latter, all application class files (non-library) required to verify the program according to the JVM specification must transfer regardless of whether or not they are used.

VI.B.2 Verified Transfer

Verification is commonly used to ensure expected behavior of Java programs. This mechanism checks that the program is well-formed and type-safe, among other things. The process must occur at runtime just prior to execution of untrusted programs. In this section we consider the effect of verified-execution with and without non-strict execution. As with all of our results, we only consider the effect of our optimizations (and in this case verification) on application code (not local library files).

Five of the six benchmarks presented in the previous sections have different class file loading characteristics when verification is turned on (Compress accesses the same files with or without verification so we omit repeating results for it). Figure VI.23 shows the difference in transfer delay for each of these benchmarks with and without verification. Transfer delay with verification is shown by striped bars. Verification has a significant effect on Jess and Soot benchmarks for which it increases transfer delay 2s, 26s, respectively, for the T1 link and 9s, 60s, respectively, for the modem link. The others account for increases of 100ms to 1s for the T1 link and 300ms to 3 seconds for the modem link.

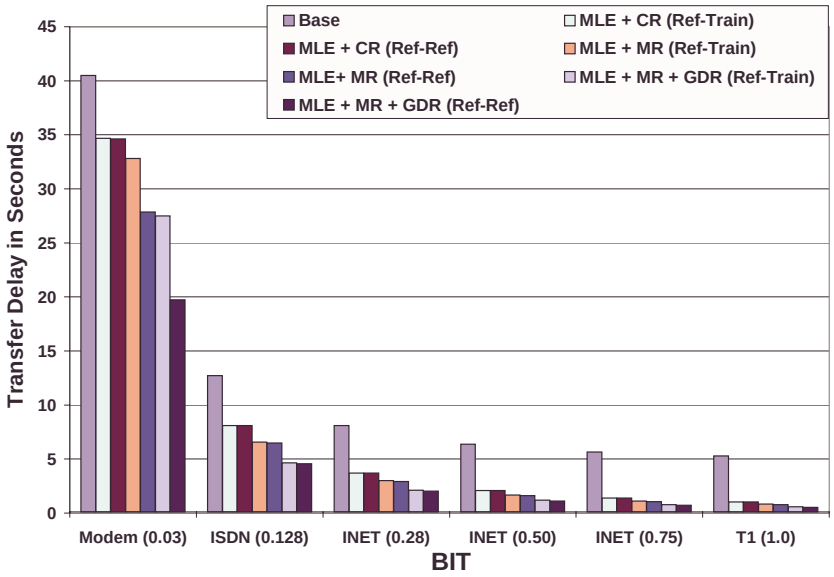


Figure VI.24: Resulting verified transfer delay for the Bit benchmark.
The graph shows a set of bars for each network bandwidth (x-axis). The y-axis is seconds. From left to right, the eight bars in a set represent the total transfer delay that results from strict execution (Base), and non-strict execution using method-level execution (MLE) alone, (MLE) plus intra-class reordering (CR) (Ref-Train and Ref-Ref), MLE plus global method reordering (MR) (Ref-Train and Ref-Ref), and MLE plus MR and global data reordering (GDR) (Ref-Train and Ref-Ref). The compress benchmark class loading characteristics are the same with and without verification so we have omitted it from this figure.

Figures VI.24, VI.25, and VI.26 show the effect of non-strict execution on verified-transfer delay. Again we present results for the various transfer schedule construction techniques. Relative to the trusted transfer results, the percent reduction in transfer delay is very similar when using verified transfer. As with trusted transfer, method and global data restructuring are most effective for transfer delay reduction.

VI.C Summary

In this chapter, we present a non-strict model for transferring and executing programs for Internet computing. We present new techniques for restructuring code and data of Java programs for more efficient non-strict execution. A summary of results is presented in Figure VI.27 in terms of transfer delay (in seconds). Five bars are shown for each network bandwidth and the values of each bar is an average over all benchmarks. The first bar (far left) is the base case transfer delay. The top graph shows the results for trusted transfer and the bottom graph

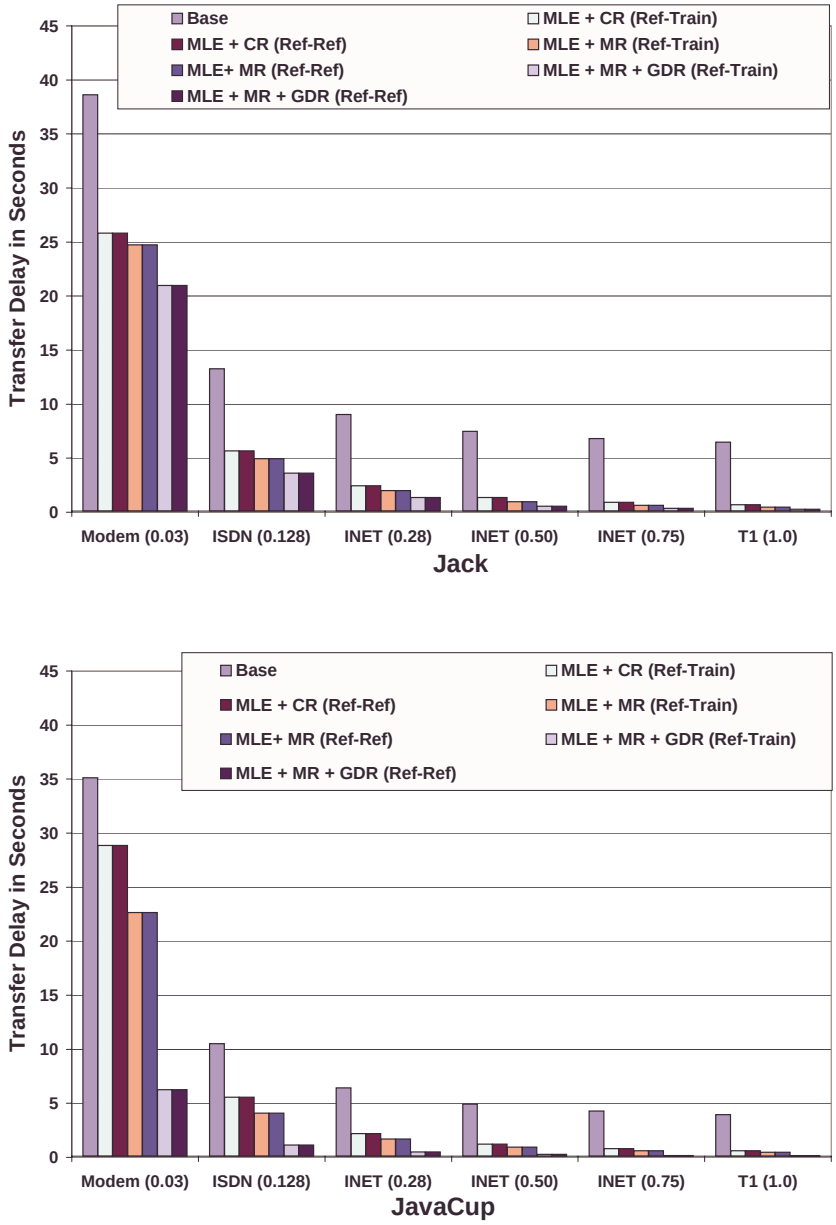


Figure VI.25: Resulting verified transfer delay for benchmarks Jack and JavaCup. Each graph provides a set of bars for each network bandwidth (x-axis). The y-axis is seconds. From left to right, the eight bars in a set represent the total transfer delay that results from strict execution (Base), and non-strict execution using method-level execution (MLE) alone, (MLE) plus intra-class reordering (CR) (Ref-Train and Ref-Ref), MLE plus global method reordering (MR) (Ref-Train and Ref-Ref), and MLE plus MR and global data reordering (GDR) (Ref-Train and Ref-Ref).

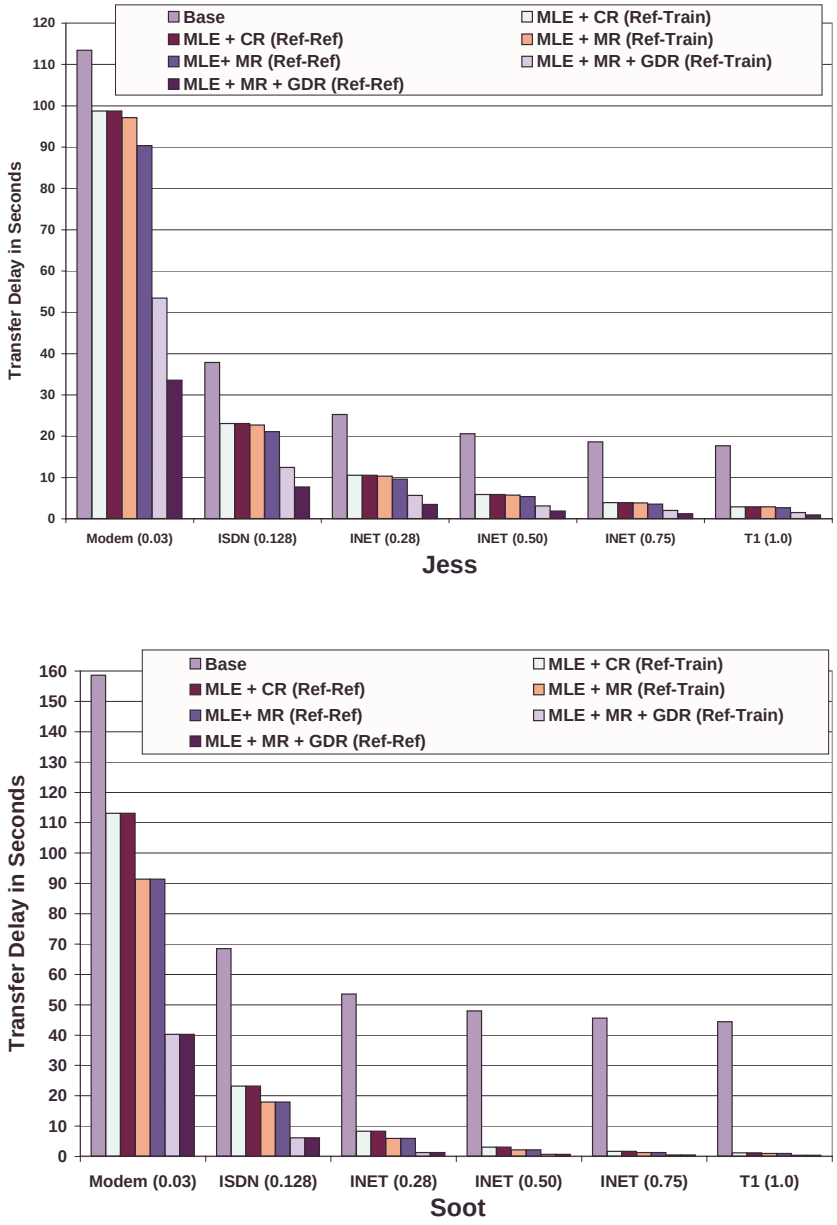


Figure VI.26: Resulting verified transfer delay for benchmarks Jess and Soot.
Each graph provides a set of bars for each network bandwidth (x-axis). The y-axis is seconds. From left to right, the eight bars in a set represent the total transfer delay that results from strict execution (Base), and non-strict execution using method-level execution (MLE) alone, (MLE) plus intra-class reordering (CR) (Ref-Train and Ref-Ref), MLE plus global method reordering (MR) (Ref-Train and Ref-Ref), and MLE plus MR and global data reordering (GDR) (Ref-Train and Ref-Ref).

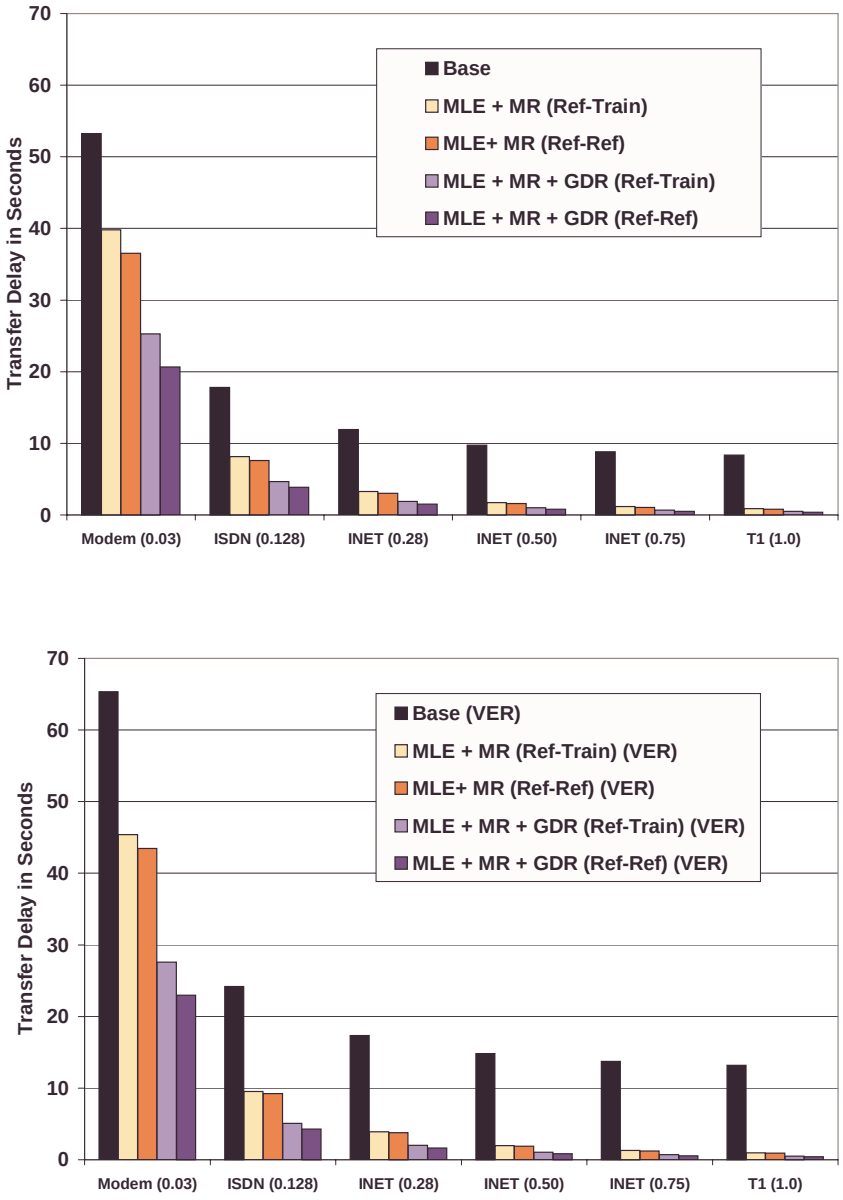


Figure VI.27: Average transfer delay (in seconds) using non-strict execution.
Five bars are shown for each network bandwidth and the values of each bar is an average over all benchmarks. The first bar (far left) is the base case transfer delay. The top graph shows the results for trusted transfer and the bottom graph shows the same for verified transfer. Results for global method reordering alone and global method and data reordering are shown. Both cross-input (Ref-Train) and same-input (Ref-Ref) results are given for each type of restructuring.

shows the same for verified transfer. Using existing technology, transfer delay costs 53 and 65 seconds on average for trusted and verified transfer, respectively, when using a modem link. Over a T1 link the cost is 8 and 12 seconds, respectively on average. Non-strict execution using method-level execution with method reordering globally across class files eliminates 13 and 20 seconds of delay for trusted and verified transfer, respectively across inputs on average over a modem link. Global data reordering reduces this same delay (modem) by 28 and 38 seconds, respectively for trusted and verified transfer. For the T1 link, delay is reduced 8 and 12 seconds for trusted and verified transfer, respectively by global data reordering. In addition, the substantial reduction in trusted-transfer delay equates to improved progress at program startup since most class files in an application transfer in the first 10% (5 seconds) of program execution. On average across inputs, global method and data reordering eliminates 21 seconds of transfer delay in the first 10% of program execution for the modem link and 5 seconds for the T1 link.

The text of this chapter is in part a reprint of the material as it appears in the 1998 conference proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). The dissertation author was the primary researcher and author and the co-authors listed on this publication directed and supervised the research which forms the basis for this chapter.

Chapter VII

Transfer Delay Avoidance and Overlap: Class File Prefetching And Splitting

In the previous chapter we presented non-strict execution, a JVM modification that overlaps transfer with useful work and avoids unnecessary transfer. Changes made to the JVM enabled method-level transfer and execution and transfer to occur concurrently with execution. In this chapter, we present two complementary techniques that also use overlap and avoidance to improve mobile program performance using *existing* JVM technology. They are *Class File Prefetching and Splitting.*

For class file prefetching, we insert prefetch commands into the bytecode instruction stream that cause as-yet-unaccessed class files to be accessed and transferred prematurely. Since the request is made on a thread separate from the application thread, transfer is performed in the background. The goal of this optimization is to prefetch the class file far enough in advance to remove part or all of the transfer delay associated with the first access of the class file by the application thread. Since prefetching modifies only class files, no changes to existing JVM technology are needed to enable the performance improvements.

We next propose class file splitting to partition a class file into separate sections that contain frequently used and infrequently used (or unused) code. We refer to the frequently used code sections as "hot" and the unused as "cold". When only hot sections are used by the executing program, less is transferred so transfer delay is reduced. If a cold class is ever accessed, the existing dynamic class file loading mechanism in the JVM initiates their transfer. Like prefetching, splitting is implemented using existing JVM technology.

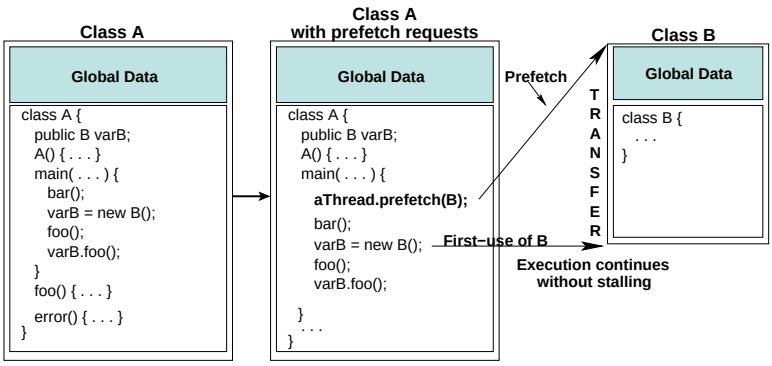


Figure VII.1: The potential of class file prefetching.
A prefetch to class file B is inserted into class file A. The full transfer delay will be masked if class file B has fully transferred by the time the command *new B()* is executed.

VII.A Design And Implementation

We first describe the implementation of class file prefetching, a technique that enables overlap of transfer with useful work. Following this, we detail class file splitting for transfer delay avoidance. Both techniques reduce the effect of transfer delay without JVM modification.

VII.A.1 Class File Prefetching

Figure VII.1 shows the potential benefit of prefetching on a example application. The first class to be transferred is class *A*. 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*. Alternately, 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*.

In the optimal case, the overlap enabled by class file prefetching 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.

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.

Finding First-Use References

We use program analysis to find each point in the program where *first-use* references are made. This is the same technique used to create the first-use call graph using static estimation for non-strict execution presented in the previous chapter. 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 depth-first search of the basic block control flow graph (CFG) using a few simple heuristics to guide the search.

First, 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 methods are first encountered during static traversal of the flow graph determines the first-use transfer order for the methods. When processing conditional branches inside of a loop, the first-use traversal traverses all the basic blocks inside the loop searching for method 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 methods determines their first-use order, and the methods are reordered within each class file to match this ordering.

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 basic block order is the sequential ordering of blocks (and thus first-use references in those basic blocks) based on the first time each basic block is first executed. Figure VII.2 shows the first-use execution order of the class files in a sample application. Since we cannot predict program execution exactly, we need to estimate the cycle in which each basic block is first executed and thus the first-use order of class files. To do this we generate profiles to determine this *first-execution* order of class files and cycle of execution at which they occur.

For profile generation, we log the order of procedure invocations and basic block executions during program execution for a particular input. The order of the first-use references

Figure VII.2: First-use execution order of class files in a sample application.
The first class accessed is A. Next is class B followed by class C. We couple such a first-use ordering with a temporal ordering of basic blocks to determine the point during program execution at which to insert of prefetch requests.

during the profile run determines the order in which we place prefetch requests into the class files. We also account for class files that are required for verification purposes. 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 described above. For example in Figure VII.2, class C is included in the graph but may not have been placed in the position it is in by profiled-execution. It is possible for it to be placed at the end of the first-use list (if unused) according to the modified-DFS ordering.

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 VII.3 is the algorithm we use for this step. We clarify it with the example shown in Figure VII.4. In the example, we wish to insert two prefetches for the first-use references to class B and class C. Figure VII.4 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 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.

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

```
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 end of basic block parent */
            insert_prefetch_at_end_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 VII.3: Algorithm for finding the basic block to place the prefetch.

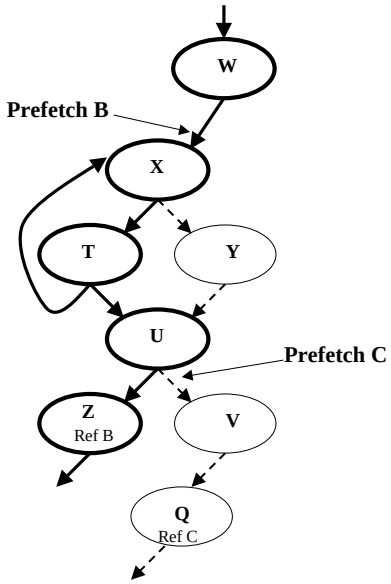


Figure VII.4: Prefetch insertion example.
In this figure, nodes represent basic blocks in the control flow graph. Solid edges represent the basic blocks executed on the first traversal through the CFG. The dashed edges represent a later traversal through the CFG. Class B is first referenced in basic block Z, and class C is first referenced in basic block Q.

effective, the prefetch mechanism must have low-overhead and must not cause the main thread of execution to stall and 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 IV.B.3. 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.

VII.A.2 Class File Splitting

Class file prefetching enables overlap of communication and computation just as with non-strict execution but without JVM modification. A complementary technique that avoids unnecessary transfer (also like non-strict execution) is class file splitting. Using this technique, a class file is split into two: a *hot* class containing *used* fields and methods; and a *cold* class containing unused, or infrequently used, fields and methods.

Like prefetching, splitting requires no changes to the JVM. When class files are accessed, regardless of whether they are hot or cold, loading, transfer (if non-local), and possibly verification, occur using existing class file loading mechanisms. Splitting reduces the amount that transfers when cold classes go unused as predicted.

Splitting Algorithm

Class file splitting is applied to Java bytecode as depicted in Figure VII.5. 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. 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 required for the represent ion of an empty class file is approximately 200 bytes. In this section, we explain the primary steps for class file splitting using Figure VII.6 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
5. Update variable usages in hot and cold class code to access relocated fields/methods via the new reference

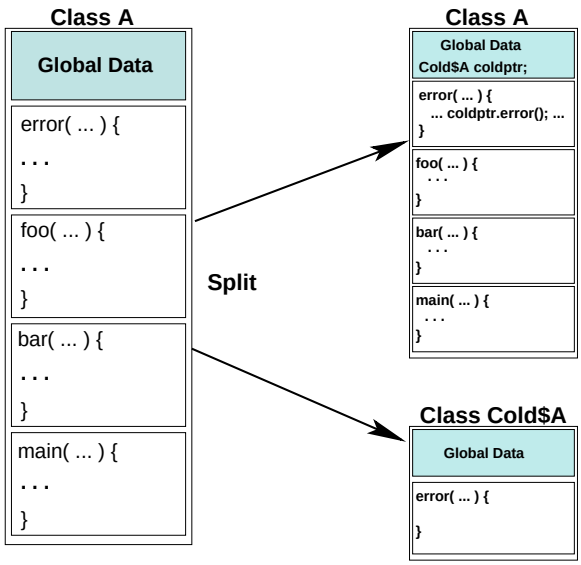


Figure VII.5: Class file splitting example.
Using class file splitting, infrequently used or unused methods in a class file are split out into a cold class (in this example, error() is split out into a cold class). If error() is never called then the transfer of the cold class is avoided. If it is called then the existing dynamic class file loading mechanism is used to initiate transfer of the cold class.

The original code, shown in Figure VII.6(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 VII.6(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 and protected 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 VII.6(b).

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.

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. If a member is defined as protected, only descendents (subclasses) of this class can access the member. To retain the semantics of the original program during splitting, hot class members must be able to access cold class members

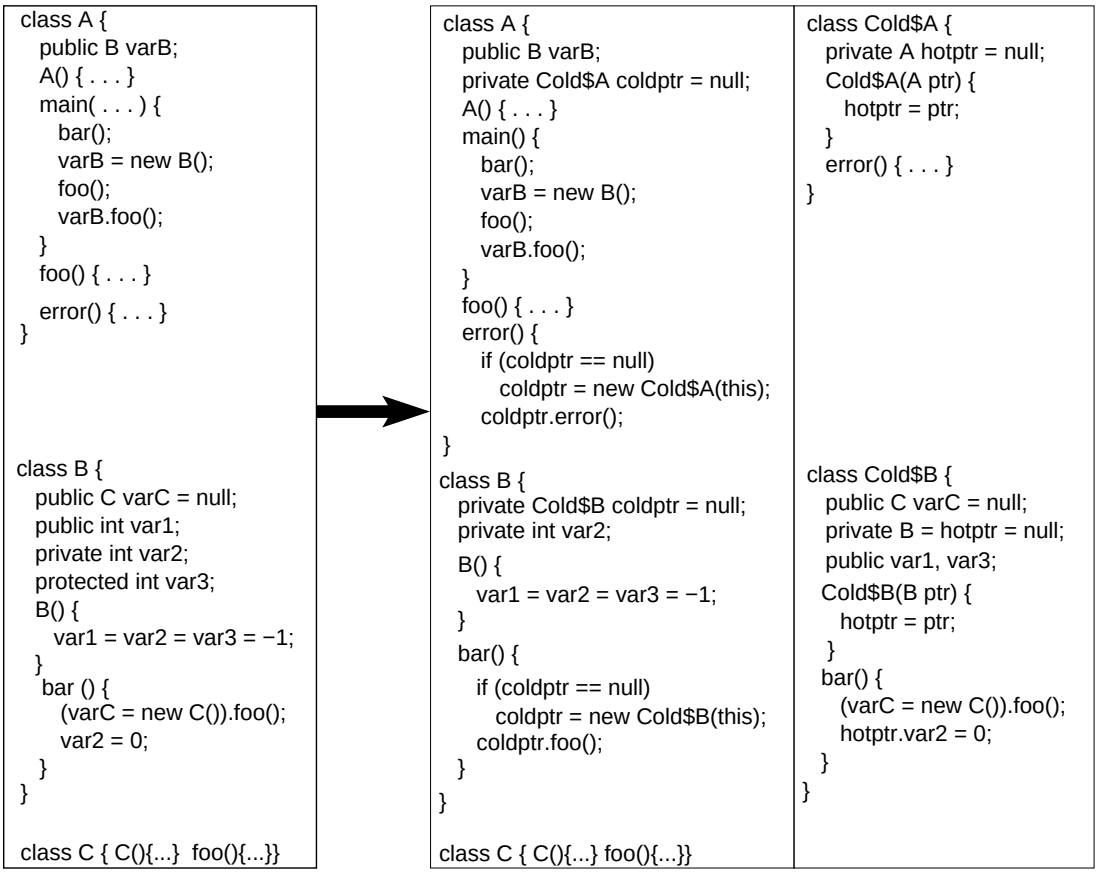


Figure VII.6: Code splitting example.

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 VII.6 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 [55]. 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.

VII.B Results: Class File Prefetching And Splitting

We first present results for class file prefetching alone. In Figure VII.7, we show the percentage of execution time that is available for overlap given various network bandwidths. The number of seconds an average program executes (without load delay) is 49 seconds. Two bars are shown for each network bandwidth. The left bar is the Ref-Train, or cross-input results; the right is the results using the Ref-Ref input (perfect information). On average, just under 2 seconds can be overlapped by prefetching for the modem link and 400 milliseconds for the T1 link. The amount available for overlap is small due to the transfer of a majority of the application at program startup. We articulate the effect of our techniques on program startup later in this section.

The percentage overlap differs across network performance since such performance determines the number of bytes that can transfer. Assume, for example, that an application only uses two class files during execution and the first one has transferred and is executing. In the background, the second class file transfers during execution. With a very fast link, the

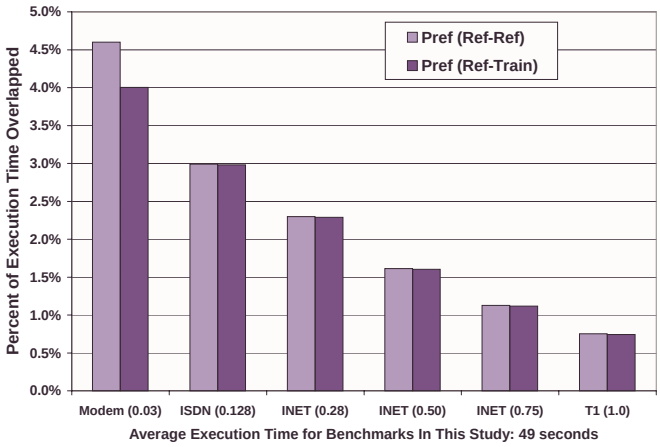


Figure VII.7: Percentage of execution time overlapped with transfer.
This graph shows the amount of execution time that can be overlapped by prefetched execution on average across all benchmarks. On average the the benchmarks executed for 42 seconds. Two bars are shown for each network bandwidth. The left bar is the Ref-Train, or cross-input results; the right is the results using the Ref-Ref input (perfect information). On average, just under 4 seconds can be overlapped by prefetching for the modem link and 2 seconds for the T1 link.

class may complete transfer before execution reaches the point at which the class file is first used. In this case, the percentage of execution time that is overlapped is the transfer time of the second class file. For a slow link, execution may reach the first access before the class has completed transfer, stalling the application thread. In this case, the overlapped execution time is the time from the start of the prefetch to the first access. In this example the fast link will have a smaller percentage of execution time overlapped by transfer.

We next evaluate the effect of class file splitting and the combined effect of prefetching and splitting. To do this, we present simulation results both in terms of trusted transfer, in which no bytecode verification is performed, and of verified transfer.

VII.B.1 Trusted Transfer

We first present the percent reduction in transfer size due to class file splitting. Figure VII.8. illustrates this reduction for each benchmark. The average application size across these benchmarks is 178KB. The top graph shows the impact of having perfect profile information (Ref-Ref); the bottom graph shows results using imperfect profile information (Ref-Train). In both graphs we consider two ways of performing the splitting. Using the first, called *Sin-*

gleSplit we move all of the cold methods from a class into a single cold class. For the second, called *MultiSplit* we move the methods each into their own cold class.

For results using the same input for profile and result generation(Ref-Ref), SingleSplit does slightly better in reducing the amount transferred. This occurs since more global data must be inserted into a MultiSplit hot class so that it can reference each of the MultipSplit cold classes. However, as the bottom graph indicates, the cost of misprediction is much higher for SingleSplit classes. Using SingleSplit, when a cold class is accessed (mispredicted) it is transferred on-demand. Since it can contain many cold methods that are possibly unused and hence, unnecessary transferred, it can degrade performance. Using MultiSplit, no such degradation occurs. On average across benchmarks (far right bars in each graph), class file splitting (MultiSplit) avoids 36% of the transfer using perfect information (Ref-Ref) and 30% across inputs (Ref-Train). For the remainder of this chapter we use only the MultiSplit technique and refer to it as simply Split.

We next present the transfer delay (in seconds) required for non-local class request and transfer with and without class file prefetching and splitting (Figures VII.9, VII.10, and VII.11). A graph is presented for each benchmark. For each network bandwidth, a set of seven bars is shown. The first bar (Base) depicts the base-case transfer delay (dynamic class file loading without prefetching and splitting). The second two bars (Pref (Ref-Train) and Pref (Ref-Ref)) show the Ref-Train and Ref-Ref profile results for prefetching alone. The next two bars (Split (Ref-Train) and Split (Ref-Ref)) show the same for splitting alone. The final two bars (Pref + Split (Ref-Train) and Pref + Split (Ref-Ref)) depict the transfer delay that results from the combination of class file prefetching and splitting (and each profile input).

On average, across-inputs (Ref-Train), class file prefetching reduces transfer delay by 2 seconds for the modem link and 300 milliseconds for the T1 link. Class file splitting reduces transfer delay 19 seconds for the modem link across inputs on average and 200 milliseconds for the T1 link. When splitting is combined with prefetching, transfer delay is reduced by 20 seconds for the modem link and 600ms on average across inputs. Combined results can be better than the sum of the two individual optimizations since splitting may expose additional opportunity for overlap.

We next show the effect of class file prefetching and splitting on program startup in Figures VII.12 through VII.17. Two cumulative distribution functions (CDF) are given in each graphs (one for each benchmark). Each function indicates the cumulative transfer delay (y-axis) at particular point during execution of the programs (shown as percentage of program execution

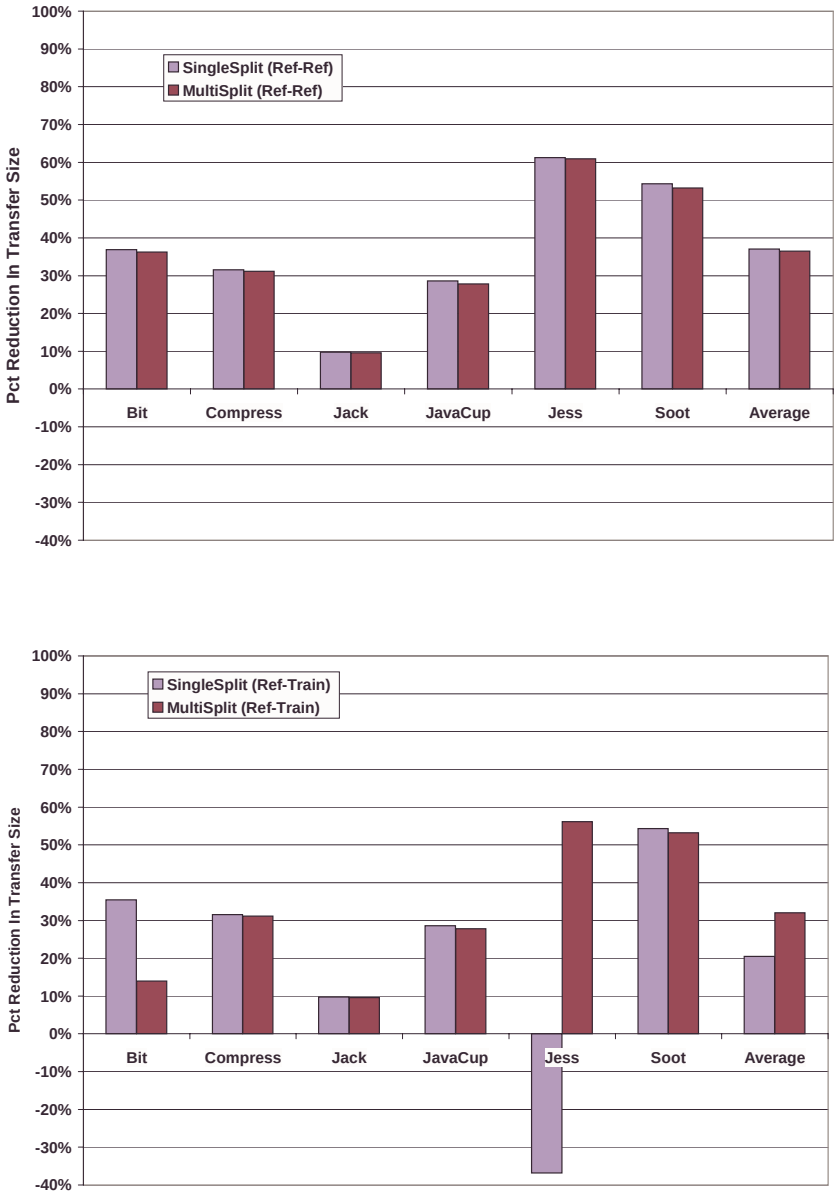


Figure VII.8: Percent reduction in transfer size.
These two graphs depict the percent reduction in the number of bytes transferred. The left graph shows the perfect information results (Ref-Ref) and the left graphs shows the cross-input (Ref-Train) results. The left bar of each pair shows the SingleSplit effect in which all of the cold methods from a class are split into a single cold class. The right bar depicts the MultiSplit effect in which each cold method is contained in its own cold class. Since more global data are necessary to represent the multiple cold classes in MultiSplit, SingleSplit Ref-Ref results show greater reduction in transfer size. Across inputs (Ref-Train) however, SingleSplit degrades performance since the use of a mispredicted cold class requires that all cold methods in a class be transferred. For the remainder of this chapter we use and assume MultiSplit and refer to it as simply Split.

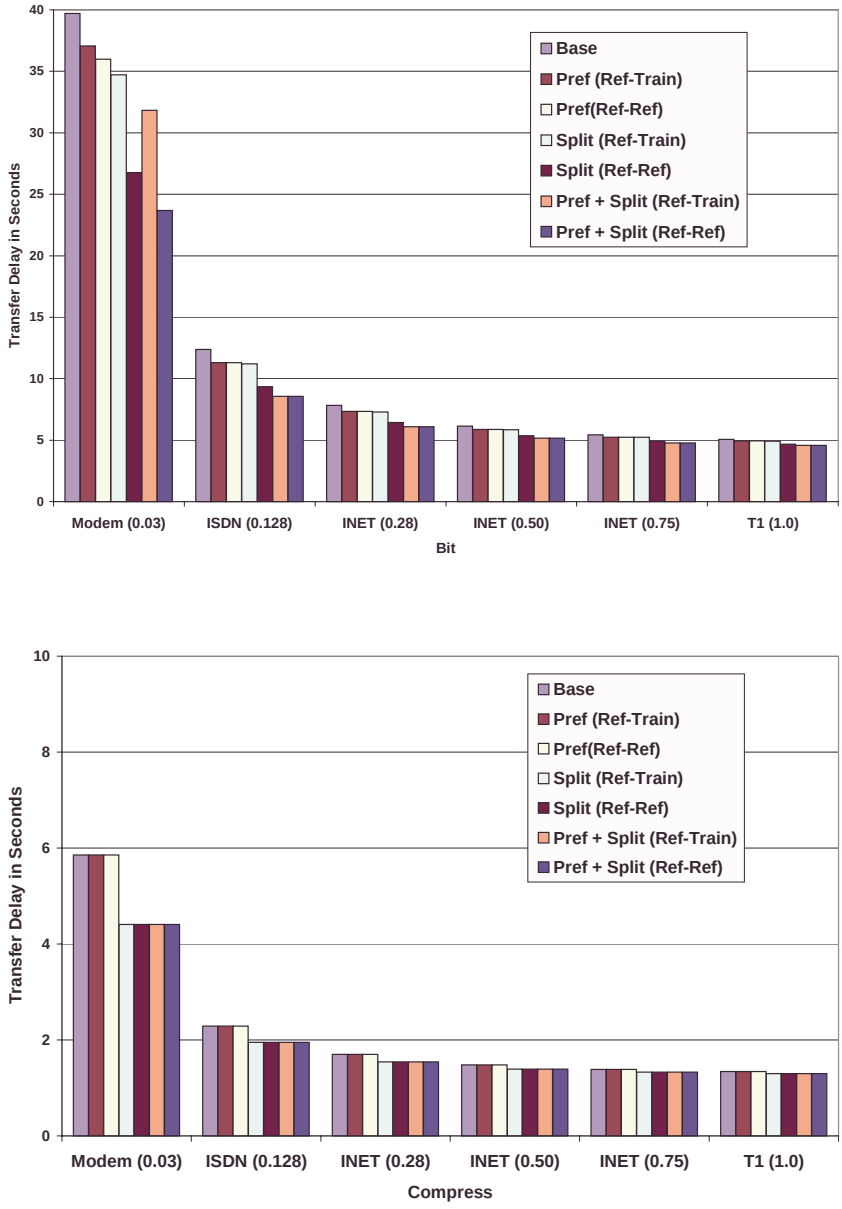


Figure VII.9: Transfer delay for Bit & Compress using prefetching and splitting.
Each graph provides a set of bars for each network bandwidth (x-axis). From left to right, the seven bars in a set represent the total transfer delay that results from strict execution (Base) and from strict execution with prefetching alone (Ref-Train and Ref-Ref), with splitting alone (Ref-Train and Ref-Ref), and prefetching and splitting combined (Ref-Train and Ref-Ref). in the bottom row of graphs.

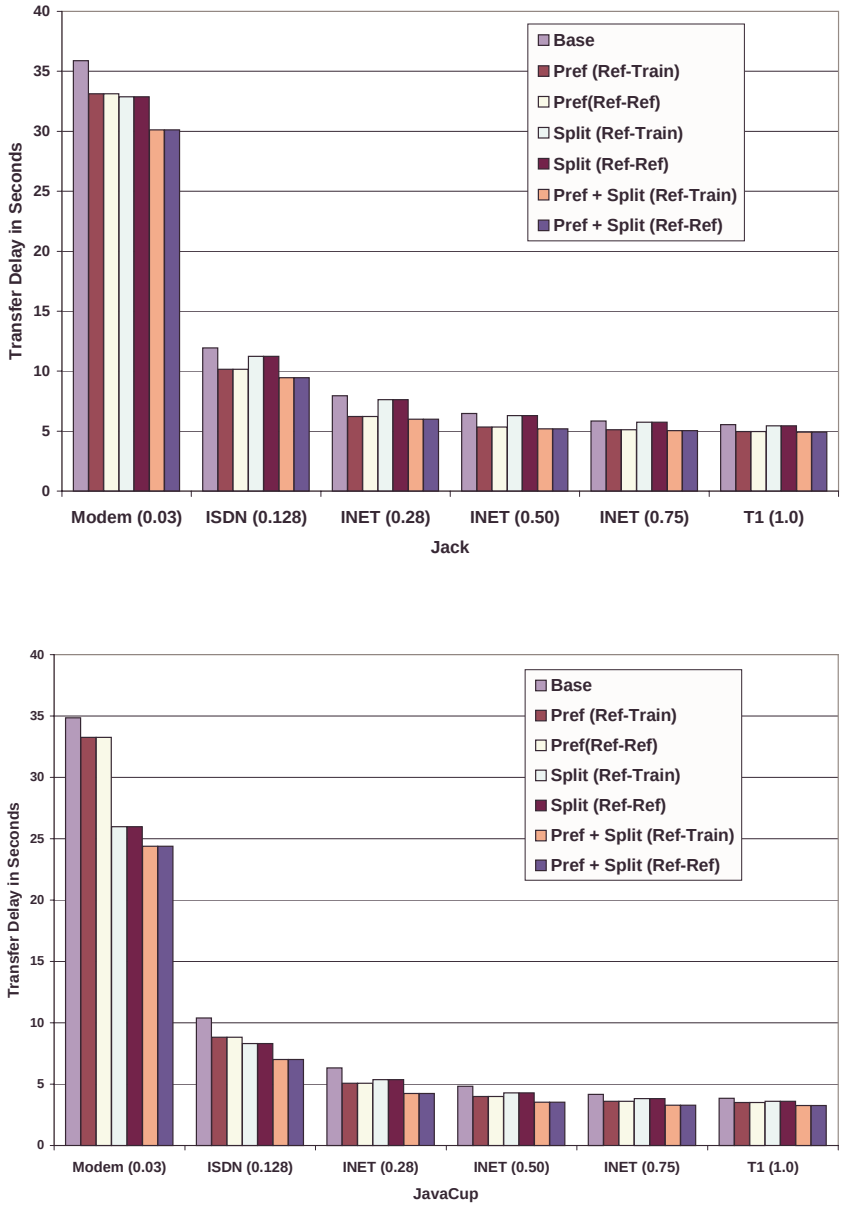


Figure VII.10: Transfer delay for Jack & JavaCup using prefetching and splitting. Each graph provides a set of bars for each network bandwidth (x-axis). From left to right, the seven bars in a set represent the total transfer delay that results from strict execution (Base) and from strict execution with prefetching alone (Ref-Train and Ref-Ref), with splitting alone (Ref-Train and Ref-Ref), and prefetching and splitting combined (Ref-Train and Ref-Ref). in the bottom row of graphs.

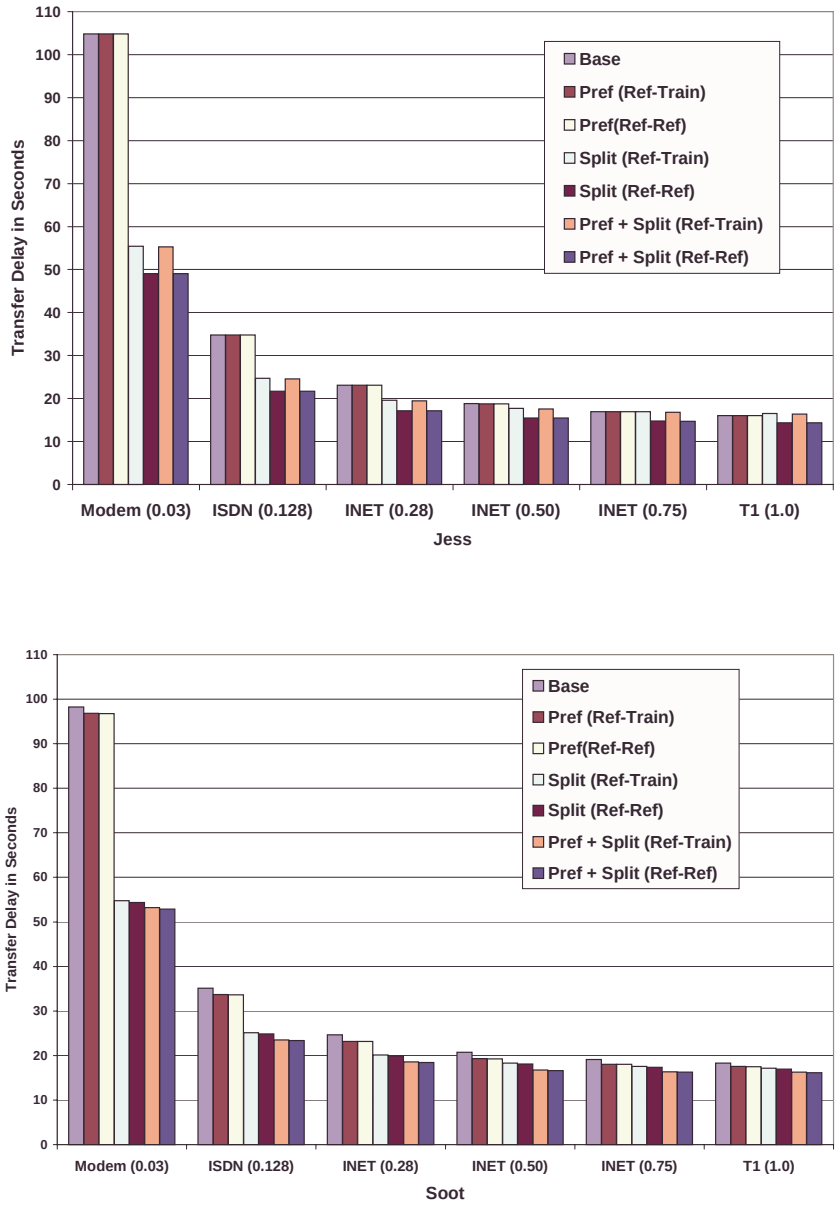


Figure VII.11: Transfer delay for Jess & Soot using prefetching and splitting.
Each graph provides a set of bars for each network bandwidth (x-axis). From left to right, the seven bars in a set represent the total transfer delay that results from strict execution (Base) and from strict execution with prefetching alone (Ref-Train and Ref-Ref), with splitting alone (Ref-Train and Ref-Ref), and prefetching and splitting combined (Ref-Train and Ref-Ref). in the bottom row of graphs.

completed on the x-axis). The average execution time for the programs is 49 seconds. CDFs are shown for unoptimized transfer and execution, as well as that with both prefetching and splitting across inputs (Ref-Train). Figures VII.12, VII.13, and VII.14 show the startup CDFs for transfer delay resulting from the use of a modem link. Figures VII.15, VII.16, and VII.17 show the same for th T1 link. On average, 14 seconds of the required transfer delay that is incurred during the first 10% (5 seconds) of program execution is eliminated for the modem link by splitting and prefetching (200ms for the T1 link).

VII.B.2 Verified Transfer

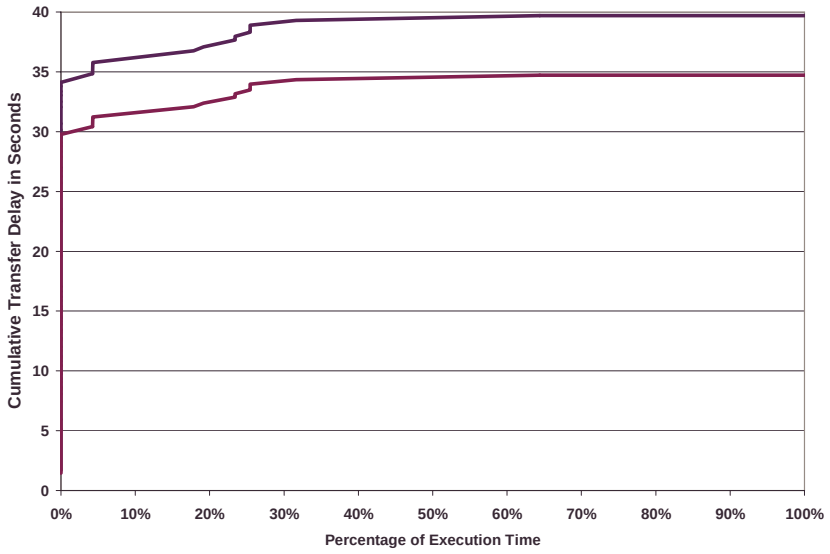
Verification is commonly used to ensure expected behavior of Java programs. This mechanism checks that the program is well-formed and type-safe, among other things. The process must occur at runtime just prior to execution of untrusted programs. In this section we consider the effect of verified-execution with and without non-strict execution. We only consider the effect of verification for application code (not local library files).

Five of the six benchmarks presented in the previous sections have different class file loading characteristics when verification is turned on. Figure VII.18 shows the difference in transfer delay for each of these benchmarks with and without verification. Verification has a significant effect on the Jess and Soot benchmarks for which it increases transfer delay 2s, 26s, respectively, for the T1 link and 9s, 60s, respectively, for the modem link. The others account for increases of 100ms to 1s for the T1 link and 300ms to 3 seconds for the modem link.

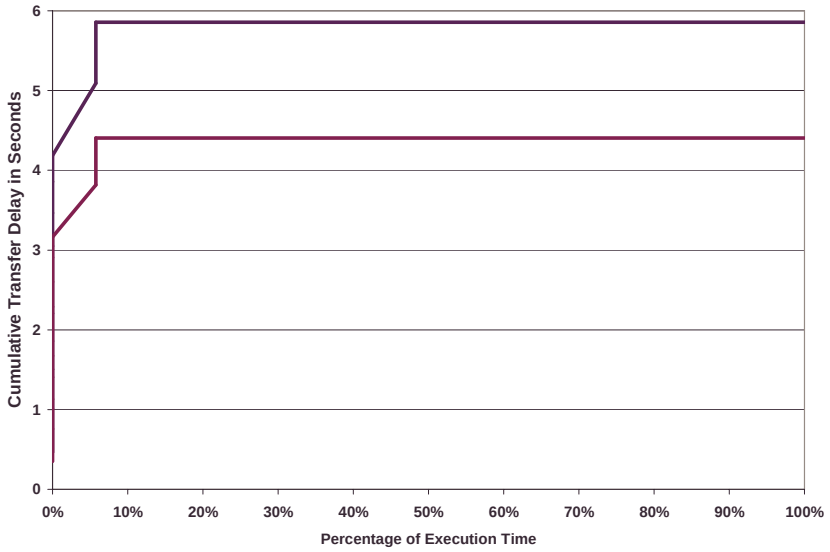
Figures VII.19, VII.20, and VII.21 show the effect of class file splitting and prefetching on verified-transfer delay. Again we present results for prefetching alone, splitting alone, and prefetching and splitting together. For each of these we present both cross-input (Ref-Train) and same-input (Ref-Ref) results. Relative to the trusted transfer results, the percent reduction in transfer delay is very similar for verified transfer results. As with trusted transfer, using splitting and prefetching together results in the greatest reduction in transfer delay.

VII.C Summary

In this chapter, we present two techniques that use existing JVM technology to reduce transfer delay. The first is a latency-hiding technique in which Java class files are prefetched prior to the first reference to the class by an application. Prefetching enables overlap of execution cycles with the transfer of class files. However, since most of the transfer delay occurs

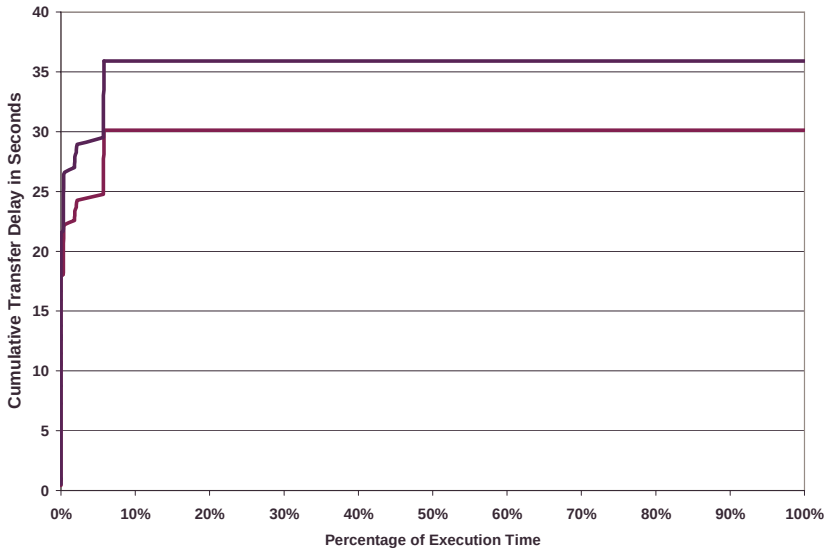


Bit

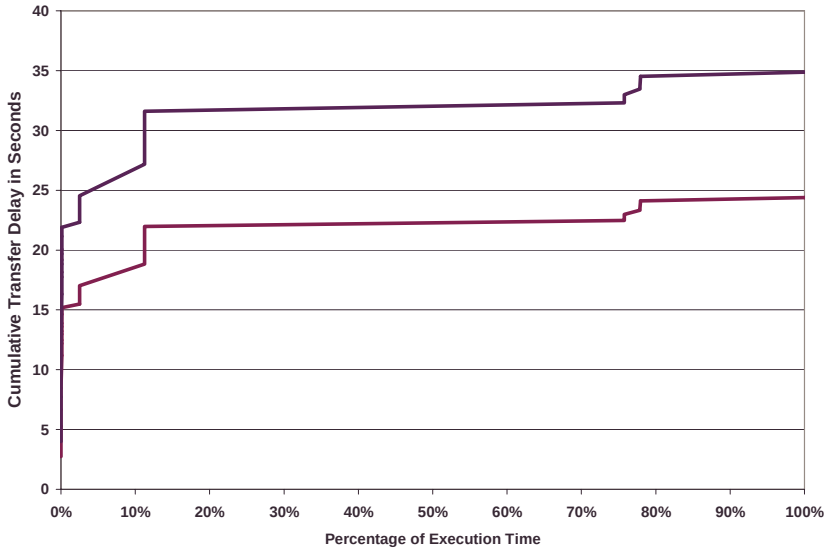


Compress

Figure VII.12: Program startup (Bit and Compress) using a modem link.
Each of these graphs show (one for each benchmark) the cumulative distribution of transfer delay (over a Modem link) during program execution (as a percentage on x-axis). The top function is unoptimized transfer and execution. The lower is the effect of both class file prefetching and splitting across inputs (Ref-Train).

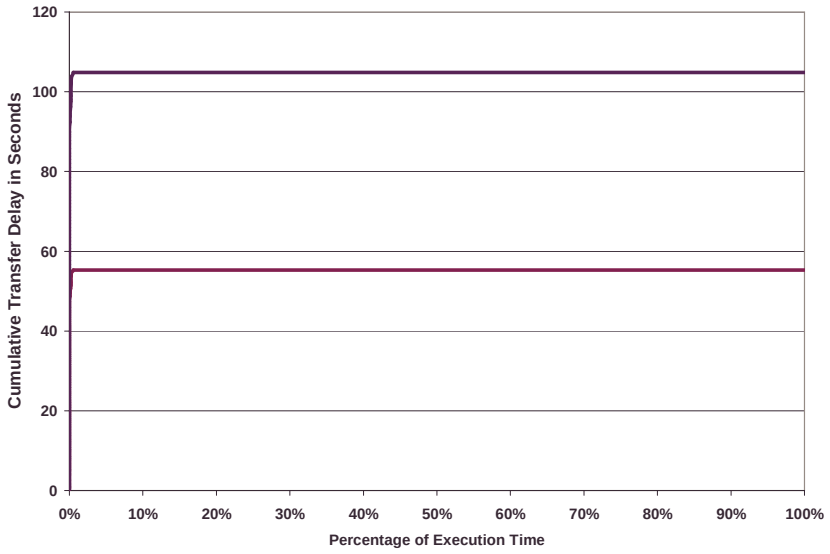


Jack

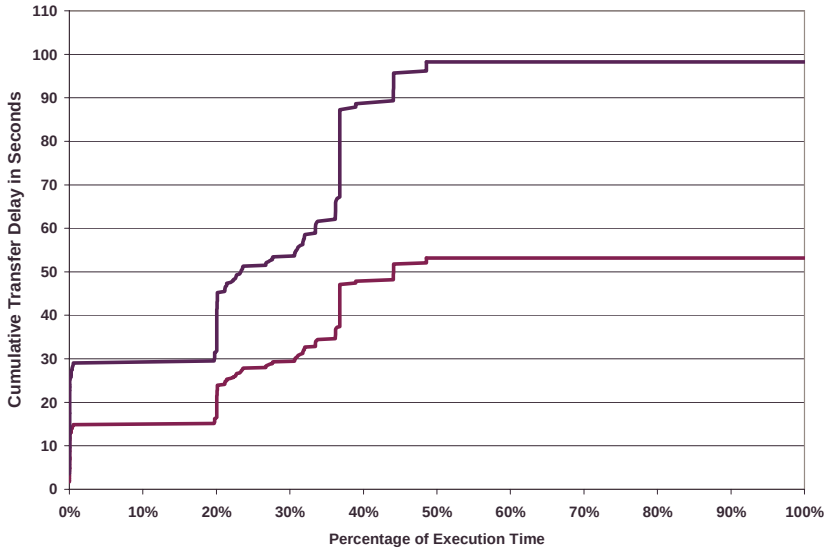


JavaCup

Figure VII.13: Program startup (Jack and JavaCup) using a modem link.
Each of these graphs show (one for each benchmark) the cumulative distribution of transfer delay (over a Modem link) during program execution (as a percentage on x-axis). The top function is unoptimized transfer and execution. The lower is the effect of both class file prefetching and splitting across inputs (Ref-Train).

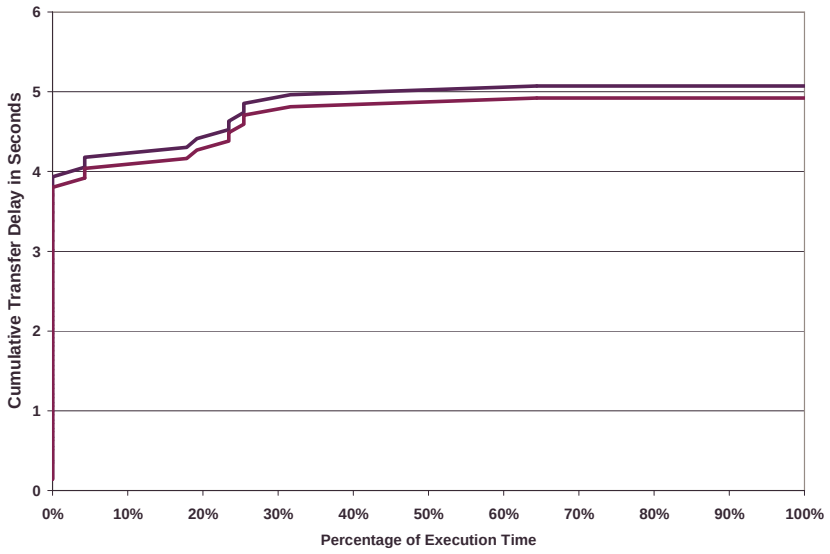


Jess

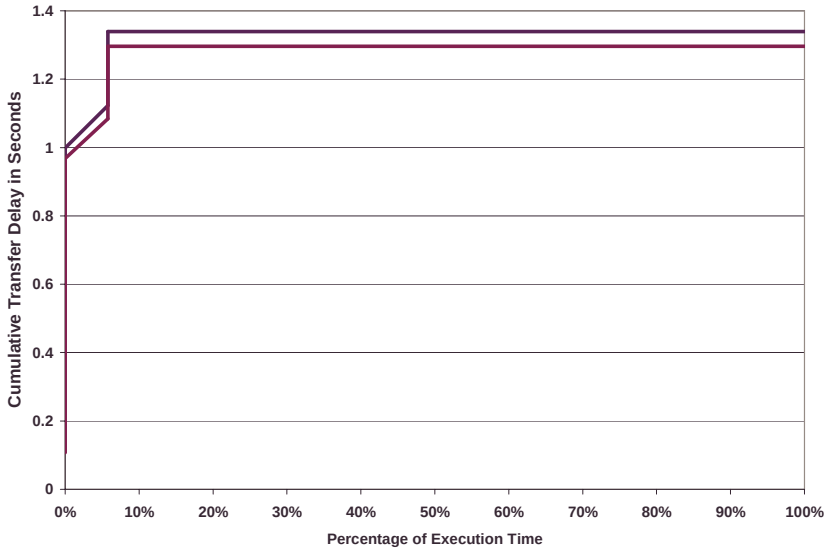


Soot

Figure VII.14: Program startup (Jess and Soot) using a modem link.
Each of these graphs show (one for each benchmark) the cumulative distribution of transfer delay (over a Modem link) during program execution (as a percentage on x-axis). The top function is unoptimized transfer and execution. The lower is the effect of both class file prefetching and splitting across inputs (Ref-Train).

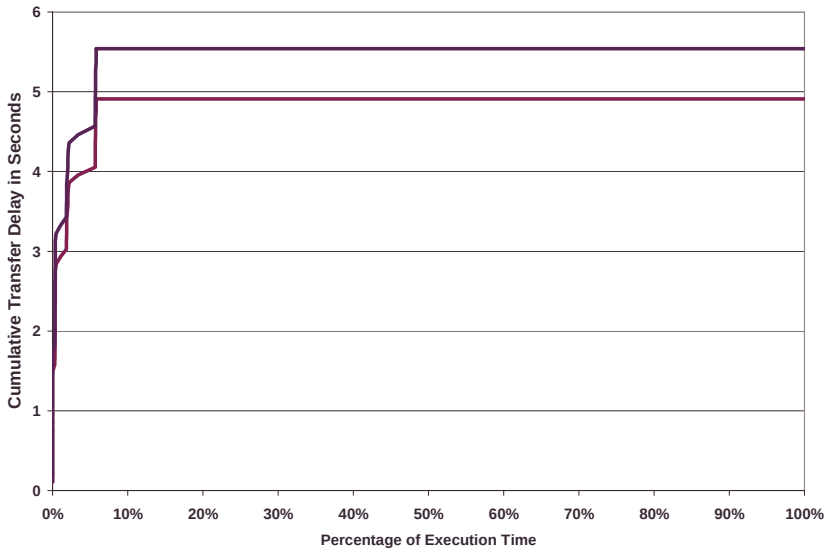


Bit

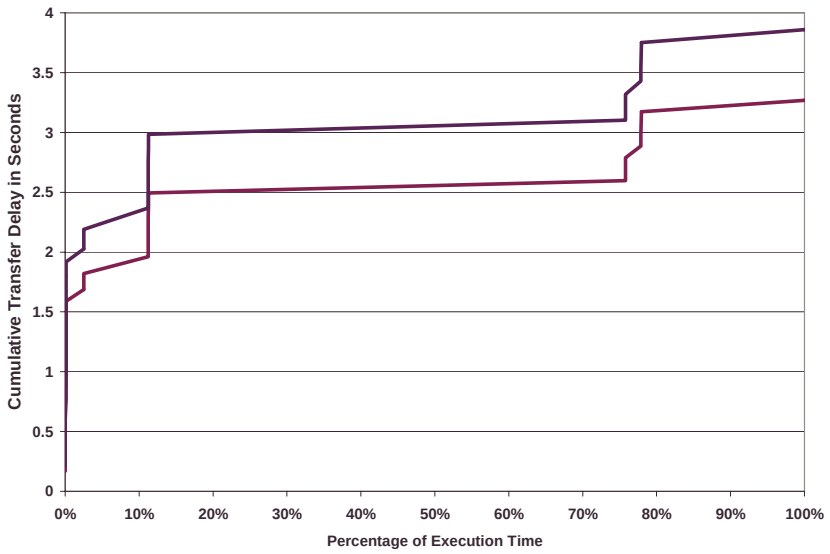


Compress

Figure VII.15: Program startup (Bit and Compress) using a T1 link.
Each of these graphs show (one for each benchmark) the cumulative distribution of transfer delay (over a T1 link) during program execution (as a percentage on x-axis). The top function is unoptimized transfer and execution. The lower is the effect of both class file prefetching and splitting across inputs (Ref-Train).

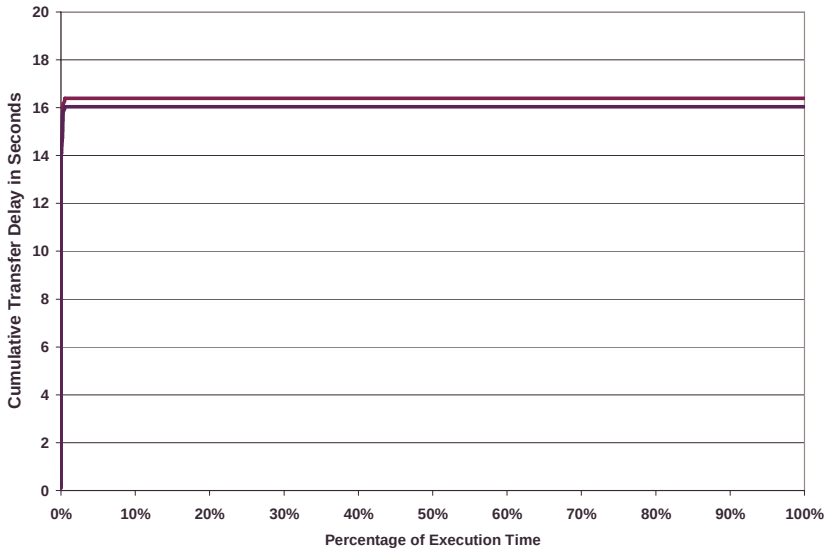


Jack

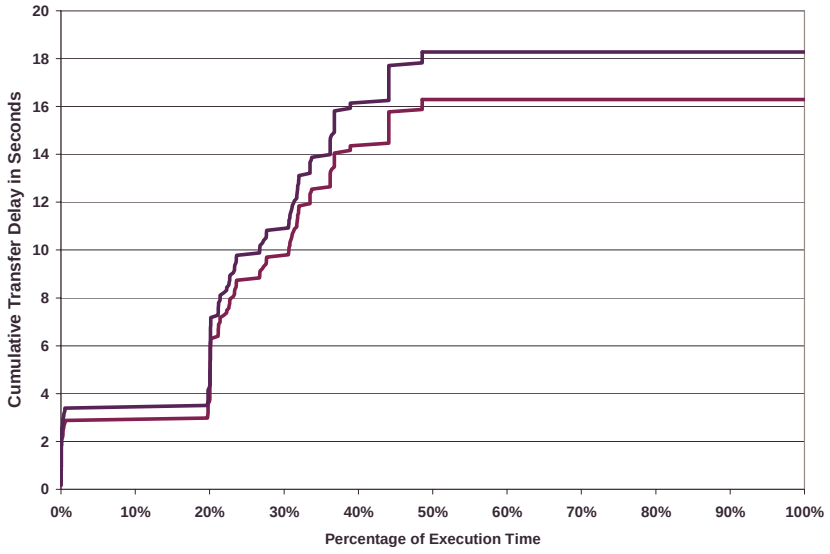


JavaCup

Figure VII.16: Program startup (Jack and JavaCup) using a T1 link.
Each of these graphs show (one for each benchmark) the cumulative distribution of transfer delay (over a T1 link) during program execution (as a percentage on x-axis). The top function is unoptimized transfer and execution. The lower is the effect of both class file prefetching and splitting across inputs (Ref-Train).



Jess



Soot

Figure VII.17: Program startup (Jess and Soot) using a T1 link.
Each of these graphs show (one for each benchmark) the cumulative distribution of transfer delay (over a T1 link) during program execution (as a percentage on x-axis). The top function is unoptimized transfer and execution. The lower is the effect of both class file prefetching and splitting across inputs (Ref-Train).

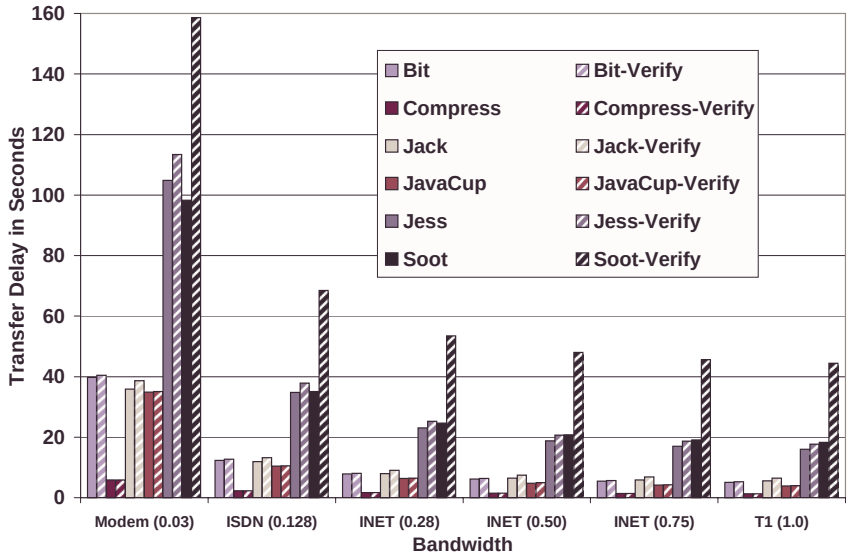
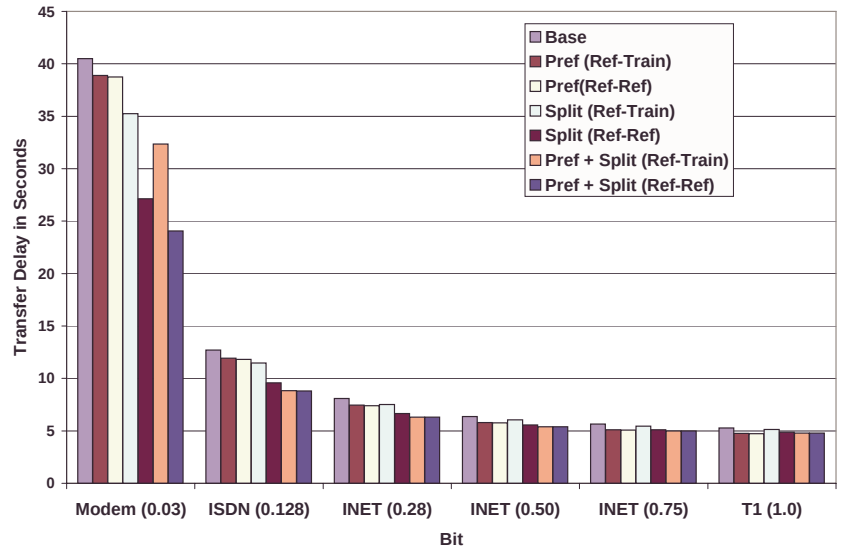


Figure VII.18: Difference in transfer delay for trusted and verified execution.
For each benchmark, there are two bars presented. The first of each pair is the transfer delay for trusted transfer and the second is for verified transfer. For the latter, all application class files (non-library) required to verify the program according to the JVM specification must transfer regardless of whether or not they are used.

in the first 10% of program execution, only a small amount of execution time can be overlapped (less than 4% which equates to 2 seconds on average). To compensate for this, we also present an optimization to split Java class files to reduce the size of class files transferred thereby avoiding transfer. On average, the total amount transferred is reduced by 36%. Neither technique (unlike non-strict execution) requires modification to the JVM. The optimizations use compile-time analysis and heuristics with profiles to guide selection of classes to split and when to prefetch. Once the class files are modified, Java applications execute with improved performance and the same semantics of the original execution without optimization.

A summary of results is presented in Figure VII.22 in terms of transfer delay (in seconds). Seven bars are shown for each network bandwidth and the values of each bar is an average over all benchmarks. The first bar (far left) is the base case transfer delay. The remaining 3 pairs of bars show the transfer delay that results from prefetching alone, splitting alone, and prefetching and splitting together. The first bar in each pair is the cross-input results (Ref-Train) and the second bar is the same-input results (Ref-Ref). The right graph shows the results for trusted transfer and the left graph shows the same for verified transfer. Without prefetching and splitting, transfer delay costs 53 and 65 seconds on average for trusted and



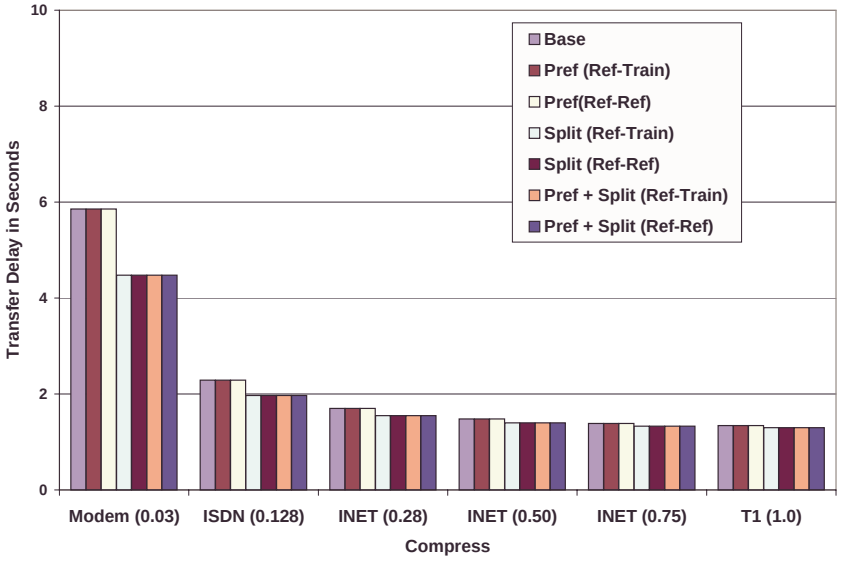


Figure VII.19: Verified transfer delay (Bit and Compress) using prefetching and splitting. Each graph provides a set of bars for each network bandwidth (x-axis). From left to right, the seven bars in a set represent the total transfer delay that results from strict execution (Base) and from strict execution with prefetching alone (Ref-Train and Ref-Ref), with splitting alone (Ref-Train and Ref-Ref), and prefetching and splitting combined (Ref-Train and Ref-Ref).

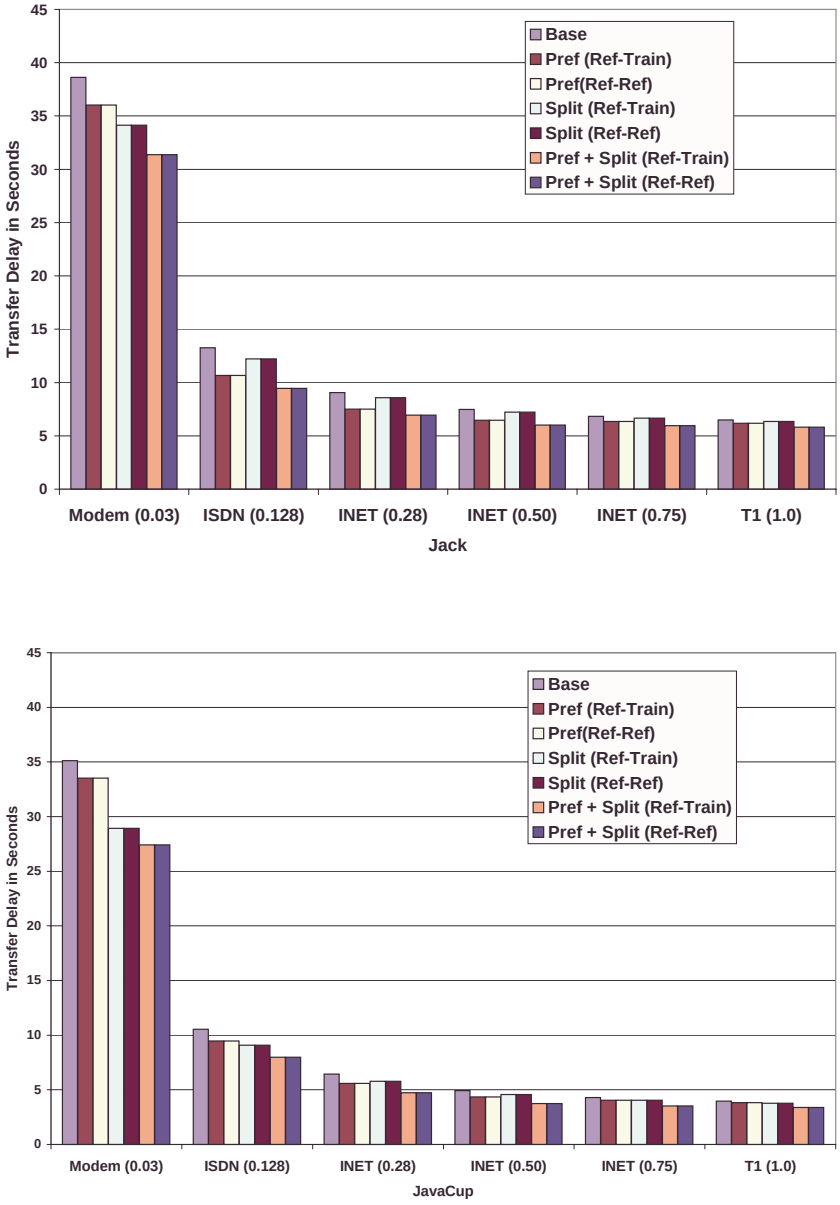


Figure VII.20: Verified transfer delay (Jack and JavaCup) using prefetching and splitting. Each graph provides a set of bars for each network bandwidth (x-axis). From left to right, the seven bars in a set represent the total transfer delay that results from strict execution (Base) and from strict execution with prefetching alone (Ref-Train and Ref-Ref), with splitting alone (Ref-Train and Ref-Ref), and prefetching and splitting combined (Ref-Train and Ref-Ref).

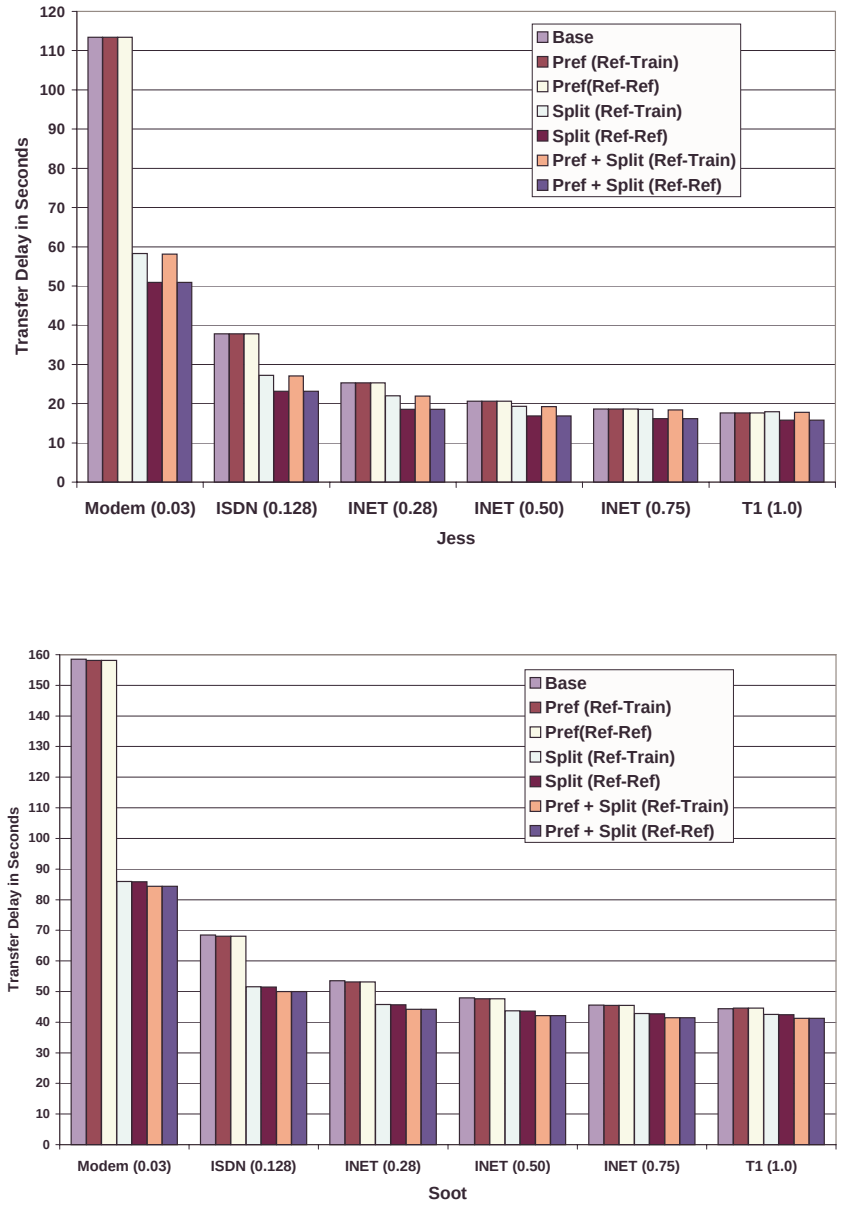


Figure VII.21: Verified transfer delay (Jess and Soot) using prefetching and splitting. Each graph provides a set of bars for each network bandwidth (x-axis). From left to right, the seven bars in a set represent the total transfer delay that results from strict execution (Base) and from strict execution with prefetching alone (Ref-Train and Ref-Ref), with splitting alone (Ref-Train and Ref-Ref), and prefetching and splitting combined (Ref-Train and Ref-Ref).

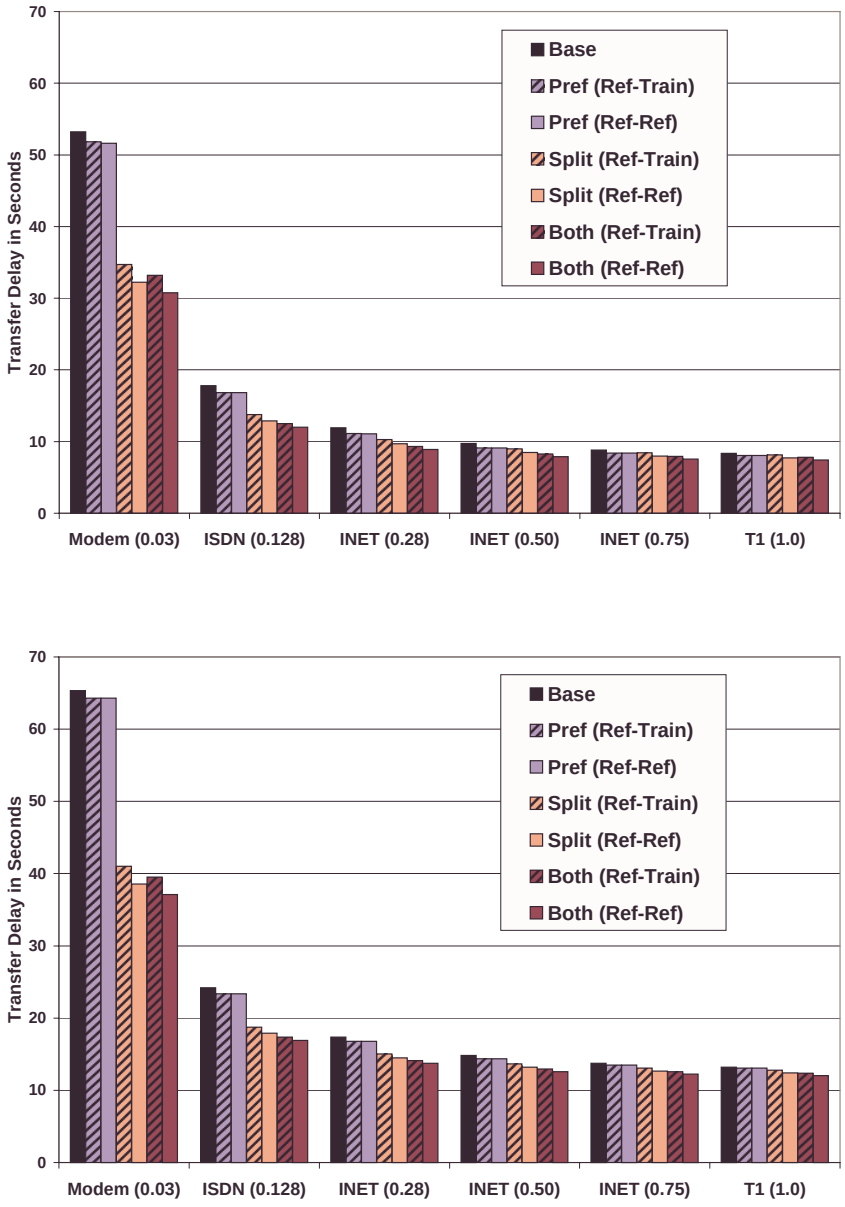


Figure VII.22: Average transfer delay using class file prefetching and splitting. Seven bars are shown for each network bandwidth and the values of each bar is an average over all benchmarks. The first bar (far left) is the base case transfer delay. The right graph shows the results for trusted transfer and the left graph shows the same for verified transfer. The remaining 3 pairs of bars show the transfer delay that results from prefetching alone, splitting alone, and prefetching and splitting together. The first bar in each pair is the cross-input results (Ref-Train) and the second bar is the same-input results (Ref-Ref). On average, class file splitting and prefetching reduce trusted-transfer delay by 20 seconds for the modem link and 1 second for the T1 link. For verified transfer, splitting and prefetching together reduce transfer delay by 25 seconds and 300 ms.

verified transfer, respectively, when using a modem link. Over a T1 link the cost is 8 and 12 seconds, respectively on average. Class file prefetching and splitting together, across inputs, reduces this cost by 20 seconds for the modem link and 1 second for the T1 link. This translates to a reduction in startup time: 14 seconds for the modem link and 200 milliseconds for the T1 link during the first 10% of program execution (5 seconds). For verified transfer, prefetching and splitting reduces 25 seconds and 300ms of the transfer delay for the modem and T1 link, respectively, on average.

The text of this chapter is in part a reprint of the material as it appears in the 1999 conference proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). The dissertation author was the primary researcher and author and the co-authors listed on this publication directed and supervised the research which forms the basis for this chapter.

Chapter VIII

Transfer Delay Avoidance: Dynamic Selection of Compression Formats and Selective Compression

Compression is used to reduce transfer delay by decreasing the number of bytes transferred through the use of compact file encoding. The resulting size of compressed files (compression ratio) is dependent upon the complexity of the encoding algorithm. Similar complexity is also required for decompression of the file prior to its use. That is, techniques with a high compression ratio are necessarily time consuming to decompress. Alternately, techniques with fast decompression rates are unable to achieve aggressive compression ratios and thus the transfer times that such encodings enable.

Compression is commonly used to improve the performance of applications that transfer over the Internet for remote execution, i.e. *mobile* programs. The overhead imposed by this compression-based execution is similar to mobile execution without compression: it includes the time for mobile code requests (program invocation and dynamic loading) and for transfer. However, since compression techniques must trade off compression ratio for decompression time, the latter must also be considered a source of delay since it occurs on-line while the program is executing. We refer to the combined overhead due to file request, transfer, and decompression as **Total Delay**.

To minimize total delay, a compression technique should be selected based on the underlying resource performance (network, CPU, etc). Moreover, since such performance is highly variable [18, 88], selection of the "best" compression algorithm should be able to change

dynamically for the *same* link. Such adaptive ability is important since the selection of a non-optimal format may result in substantial total delay (25-44 seconds for some cases in the programs studied) at startup or intermittently throughout mobile program execution. Much prior research has shown that even a few seconds of interruption substantially effects the user's perception of program performance [21].

To address this selection problem, we introduce **Dynamic Compression Format Selection** (DCFS), a methodology for automatic and dynamic selection of competitive, compression formats. Using DCFS, mobile programs are stored at the server in multiple compression formats. DCFS is used to predict the compression format that will result in the least delay given the bandwidth predicted to be available when transfer occurs. We use the Java execution environment for our DCFS implementation since it is the most common language for this computational paradigm (remote execution). We incorporate an extension to Network Weather Service (NWS) [88] for network performance prediction.

As a result of investigation of the problems addressed by this thesis, we discovered that it is common for only a small subset of class files in an application to be accessed during execution. However, compressed archives of Internet-computing applications typically contain all of the files that make up the application. We exploit this characteristic to further reduce the size of a compressed archive (and hence transfer delay) with a profile-directed, compiler optimization, called **Selective Compression**.

Selective compression is a technique that excludes unused class files from the archive. Profiles are used to ascertain which class files are accessed during execution and to construct a selectively compressed archive. If class files, not included in the archive, are used, they are transferred via existing dynamic class loading mechanisms. Selective compression enables further reduction transfer and decompression time.

VIII.A Design and Implementation

We first describe the implementation of DCFS, a technique that reduces the transfer delay that remains despite compressed transfer. Following this, we describe selective compression and provide results of the effect of each technique individually and collectively.

VIII.A.1 Dynamic Compression Format Selection

The compression techniques we incorporate into this study are described in the methodology in Chapter IV. Characteristics of each format are shown in Table IV.4 for the benchmarks

used in the empirical evaluation of the techniques presented in this chapter. The inherent trade-off made by compression techniques (compression ratio for decompression time) is exhibited in the final three columns of the table. The first number in each column is the decompression rate (KB/sec) for the application. The second, parenthesized number shows the compressed size (in KB) of the application from which the compression ratio can be computed (original application size (column 3) over compressed size). The total time for transfer and decompression for various networks is shown in Table VIII.1.

The data in Table IV.4 shows, for example, that the PACK format requires over 2.3 seconds to decompress the applications on average (JAR and GZIP require 89% and 98% less time, respectively), yet it enables a compressed file size that is 81% and 74% smaller than JAR and GZIP archives, respectively. This indicates that for slow networks PACK should be used due to its compression ratio, and for fast links TGZ should be used since it is inexpensive to decompress. *No* single utility enables the least total delay (request, transfer, decompression time) for all network performance characteristics and applications. In addition, each format is able to offer substantial benefit under certain circumstances. The choice of compression format should therefore be made dynamically, depending upon such circumstances to enable the best performance of mobile programs. To do this, we introduce **Dynamic Compression Format Selection** (DCFS), a technique that automatically and dynamically selects the format that results in the least total delay.

Figure VIII.1 exemplifies our DCFS model. The client-side Java Virtual Machine (JVM) incorporates a DCFS class loader. When an executing program accesses a class file for the first time (or the program itself is invoked), the request made to the JVM is forwarded to the DCFS class loader. Concurrently, a network performance measurement and prediction tool (called JavaNws) monitors the network connection between the client and the server at which the application is stored. The DCFS class loader acquires the network (as well as, possibly the CPU) performance value from the JavaNws and forwards the value(s) with the request to the server. With the initial server request, the DCFS class loader also includes the compression formats for which the client machine has decompression utilities.

At the server, applications are stored in multiple compression formats. When a server receives a request for an application or file, it uses the information sent (predicted resource performance value(s) and available compression formats) to calculate the potential total delay for each format. That is, given the predicted performance of the network to and the CPU at the client, the server determines the format that results in the least total delay. Total delay,

Table VIII.1: Total delay in seconds for the network bandwidths studied.
Raw data for the three wire-transfer formats is shown. Total delay is the time for transfer and decompression given each network technology. In parentheses is included the percentage of total delay due to decompression time.

		Total Delay in Seconds (Pct. of Delay due to Decompression)					
Program	Network	PACK		JAR		TGZ	
Antlr	MODEM (0.03)	20.9	(17.5)	66.3	(0.5)	51.4	(0.1)
	ISDN (0.128)	7.7	(47.5)	15.7	(1.9)	12.0	(0.3)
	INET (0.28)	4.6	(80.0)	3.6	(8.3)	2.6	(1.3)
	INET (0.50)	4.4	(83.5)	2.8	(10.7)	2.0	(1.7)
	T1 (1.00)	4.1	(89.6)	2.1	(14.3)	1.4	(2.4)
Bit	MODEM (0.03)	6.7	(18.6)	25.3	(0.5)	17.1	(0.2)
	ISDN (0.128)	2.6	(49.0)	6.0	(2.3)	4.0	(0.8)
	INET (0.28)	1.6	(78.5)	1.4	(9.5)	0.9	(3.4)
	INET (0.50)	1.5	(81.2)	1.1	(11.9)	0.7	(4.3)
	T1 (1.00)	1.4	(87.0)	0.8	(16.4)	0.5	(6.0)
Jasmine	MODEM (0.03)	12.9	(20.9)	65.5	(0.5)	38.1	(0.1)
	ISDN (0.128)	5.1	(52.8)	15.5	(2.0)	8.9	(0.4)
	INET (0.28)	3.3	(82.5)	3.6	(8.8)	2.0	(1.9)
	INET (0.50)	3.2	(85.3)	2.8	(11.3)	1.5	(2.4)
	T1 (1.00)	2.9	(94.1)	2.0	(15.8)	1.0	(3.6)
Javac	MODEM (0.03)	18.0	(18.5)	82.4	(0.4)	53.4	(0.1)
	ISDN (0.128)	6.8	(49.1)	19.5	(1.7)	12.5	(0.3)
	INET (0.28)	4.1	(80.8)	4.4	(7.7)	2.7	(1.2)
	INET (0.50)	4.0	(84.1)	3.4	(9.9)	2.1	(1.6)
	T1 (1.00)	3.7	(90.9)	2.5	(13.5)	1.5	(2.2)
Jess	MODEM (0.03)	8.7	(21.0)	55.6	(0.6)	49.1	(0.1)
	ISDN (0.128)	3.4	(53.0)	13.2	(2.5)	11.5	(0.3)
	INET (0.28)	2.2	(81.7)	3.1	(10.8)	2.5	(1.4)
	INET (0.50)	2.2	(84.3)	2.4	(13.7)	1.9	(1.8)
	T1 (1.00)	2.0	(92.7)	1.8	(18.3)	1.3	(2.6)
Jlex	MODEM (0.03)	5.2	(19.7)	14.4	(0.7)	11.4	(0.4)
	ISDN (0.128)	2.0	(50.7)	3.4	(2.8)	2.7	(1.5)
	INET (0.28)	1.3	(78.7)	0.9	(11.0)	0.7	(6.2)
	INET (0.50)	1.3	(80.9)	0.7	(13.4)	0.5	(7.6)
	T1 (1.00)	1.1	(95.6)	0.5	(18.8)	0.4	(9.5)
Avg	MODEM (0.03)	12.1	(19.0)	51.6	(0.5)	36.7	(0.1)
	ISDN (0.128)	4.6	(49.9)	12.2	(2.1)	8.6	(0.4)
	INET (0.28)	2.9	(80.6)	2.8	(9.0)	1.9	(1.9)
	INET (0.50)	2.7	(83.7)	2.2	(11.4)	1.5	(2.4)
	T1 (1.00)	2.5	(90.9)	1.6	(16.1)	1.0	(4.6)

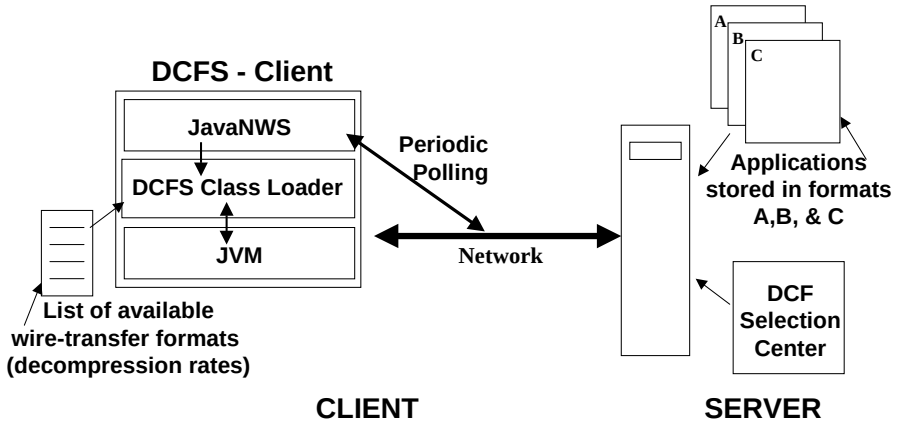


Figure VIII.1: The Dynamic Compression Format Selection (DCFS) Model.
The client requests an application from the server. It supplies the server with a list of the compression formats for which it has a decompression utility. It also gives the server a prediction of the bandwidth that is available between the client and the server. This prediction is obtained by the JavaNws. The server uses this information to determine the compression format that will result in the least total delay (request,transfer, and decompression time).

again, consists of the transfer plus decompression time. The selected format is the one in which the application or class file is sent to the client.

The JavaNws [51] utility at the client is an extension of the Network Weather Service (NWS), a resource monitoring and prediction tool. The JavaNws makes periodic measurements of the network performance between the client and the server which are used by a set of forecasting techniques to make short-term predictions of bandwidth and round-trip time. The forecasting techniques are further described in [88]. The NWS can also measure non-network resources such as CPU and memory. As part of future work, we will incorporate this functionality into the JavaNws and thus into DCFS. The current implementation of the DCFS uses network performance prediction only.

In the course of our DCFS study (presented in the previous chapter) and prior research, we discovered that often, many class files in an application are not used. For example, Table IV.3 shows that on average only 17 of the 104 classes are used. However, all of the class files are archived, compressed, and transferred to the destination for remote execution.

VIII.A.2 Selective Compression

To further reduce transfer delay we propose to combine and compress only those class files that are used by the application. Since we are unable (as yet) to predict the future and

know precisely which class files will be used by every execution of an application, we use profile-directed techniques to predict this set. This Selective Compression optimization uses profiles of previous executions to determine the classes used by a given input. The class usage pattern is then used to combine and compress the used class files. When an application is initially invoked and requested from a server, the compressed file of used classes is sent for execution. When prediction is incorrect and a class is accessed that is not contained in the used set, it is requested by the class loader and transferred alone without compression.

Selective compression mechanism is easily incorporated into the DCFS infrastructure. Selectively compressed applications are stored at the source and selected between using DCFS. When an application is initially requested storage site, the selectively compressed archive is transferred for execution. If a class file is accessed by the executing program that was not transferred as part of the selectively compressed archive, it is transferred (uncompressed) via dynamic class loading.

VIII.B Results: DCFS and Selective Compression

We first present the empirical effect of DCFS. Then we present results for selective compression both with and without DCFS.

VIII.B.1 Dynamic Compression Format Selection

To evaluate DCFS, we implemented both of the DCFS modules (DCFS client and server). However, to enable repeatability of results, we execute the modules on the same machine and simulate different networks between them. Instead of using JavaNws prediction, we use bandwidth and round-trip time averages from network traces. This enables the presentation of the upper-bound potential of DCFS. We provide a discussion of the impact of incorporating prediction in the next section.

Table IV.5 shows the bandwidth and round-trip time measurements from 24-hour, JavaNws trace data for each network used in this study. To compute total delay using this execution environment, upon client program invocation the server computes the sum of the average round-trip time (for the request), the transfer time (the average bandwidth value multiplied by the size of the compressed application), and the decompression time (the decompression rate multiplied by the size of the compressed application). The decompression rate is supplied by the client as part of the initial request. The total delay is calculated by the server for each of

the compression formats, TGZ, JAR, and PACK, and the minimum is selected.

To illustrate the performance potential of dynamic format selection, we first present the percent reduction in total delay due to DCFS in Figures VIII.2, VIII.3, and VIII.4. The base case is the use of PACK compression (or JAR or TGZ, respectively for each bar of data presented) for each type of network without dynamic selection. The percent reduction is defined as $(TD\_Base - TD\_DCFS)/TD\_Base$ for each network, where TD_Base is the total delay for the base case and TD_DCFS is the total delay using DCFS. Notice that the percent reduction can be zero when DCFS selects the base case and the base case results in the minimum total delay for that network. That is, when the base-case format is the optimal one, DCFS selects it and no additional improvement can be gained. Each bar in the figure represents a base case (compression format: PACK, JAR, or TGZ) for each network bandwidth and indicates the performance improvement a user would experience if DCFS were used instead of the base case. For example, if a user that consistently invokes a jar file for execution of the Jasmine program instead uses DCFS, he or she will experience an 80% reduction in total delay on a modem link, 45% using an Internet connection, and 50% on a local area network. The overall benefit from DCFS does not simply result from using a different compression utility, it results from selecting the **best** compression utility given the underlying network performance. The benefits achieved using DCFS are quite substantial for every benchmark and network performance rate.

We next present the total delay in seconds (log scale) in Figures VIII.5, VIII.6, and VIII.7. For each network, the number of seconds required for request, transfer and decompression is shown for each compression technique (PACK, JAR, TGZ). The fourth (far right, striped) bar of each set shows the DCFS total delay. The DCFS bar is always equivalent to the minimum of the prior three bars since it is the "best" performing format. The bar that is equal to the DCFS bar in each graph is the zero-valued bar in the the respective graph in Figures VIII.2 through VIII.4. Averaged across all networks, DCFS reduces total delay 0.3 to 1.6 seconds over PACK, 2.1 to 16.1 seconds over JAR, and 1.4 to 9.7 seconds over TGZ.

A summary of the results is shown in Figure VIII.8. The format of this graph is the same as Figures VIII.2 through VIII.4 (percent reduction) with the average reduction across all benchmarks given instead of that for a specific benchmark. If PACK is used for non-MODEM bandwidths, e.g., LAN, then DCFS reduces total delay over PACK by almost 90% (2 seconds) on average across all benchmarks. For the LAN bandwidth, the optimal format is TGZ. However for MODEM and INET bandwidths, DCFS provides 67% and 46% average reduction (24 and 4 seconds), respectively, over TGZ. On average across all networks, 34% (1.7 second), 52% (7.2

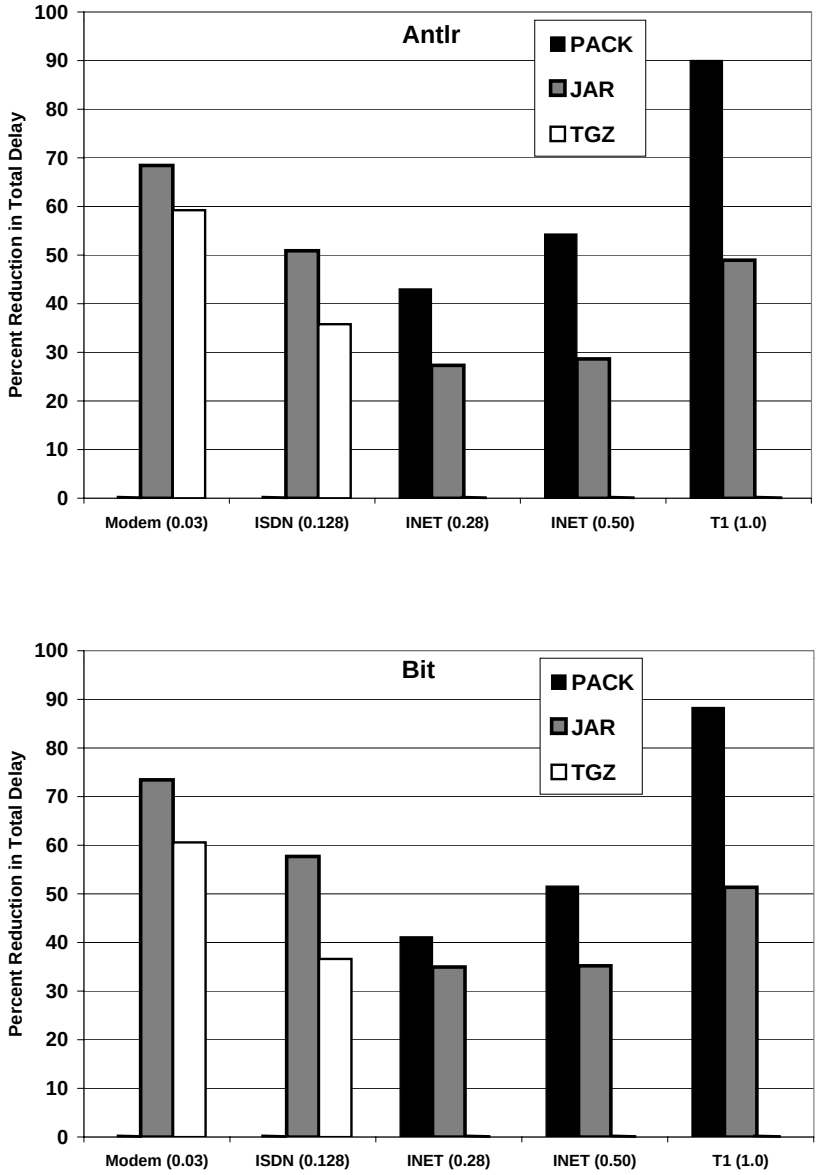


Figure VIII.2: Pct. reduction in total delay due to DCFS for Antlr and Bit.
Each graph shows data for a different benchmark. Each bar is a different compression format (base case) and represents the percent reduction in total delay (y-axis) when DCFS is used instead of that format alone. The choice made by DCFS for a given network (x-axis) is represented as zero-valued (missing) bars, i.e., when the format chosen is the base case, DCFS enables no further reduction since it has selected the optimal format given the ones available. In every case, DCFS correctly determines and uses the format that requires the minimum total delay and significantly reduces it.

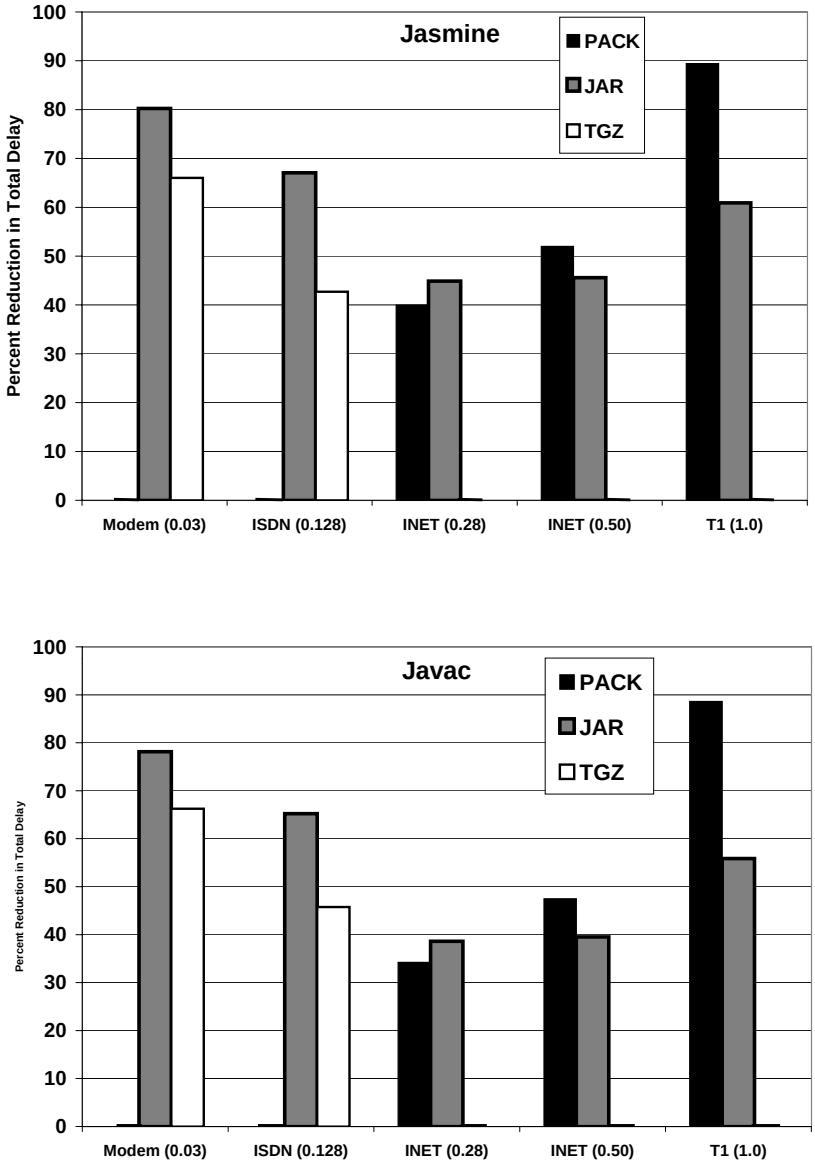


Figure VIII.3: Pct. reduction in total delay due to DCFS for Jasmine and Javac. Each graph shows data for a different benchmark. Each bar is a different compression format (base case) and represents the percent reduction in total delay (y-axis) when DCFS is used instead of that format alone. The choice made by DCFS for a given network (x-axis) is represented as zero-valued (missing) bars, i.e., when the format chosen is the base case, DCFS enables no further reduction since it has selected the optimal format given the ones available. In every case, DCFS correctly determines and uses the format that requires the minimum total delay and significantly reduces it.

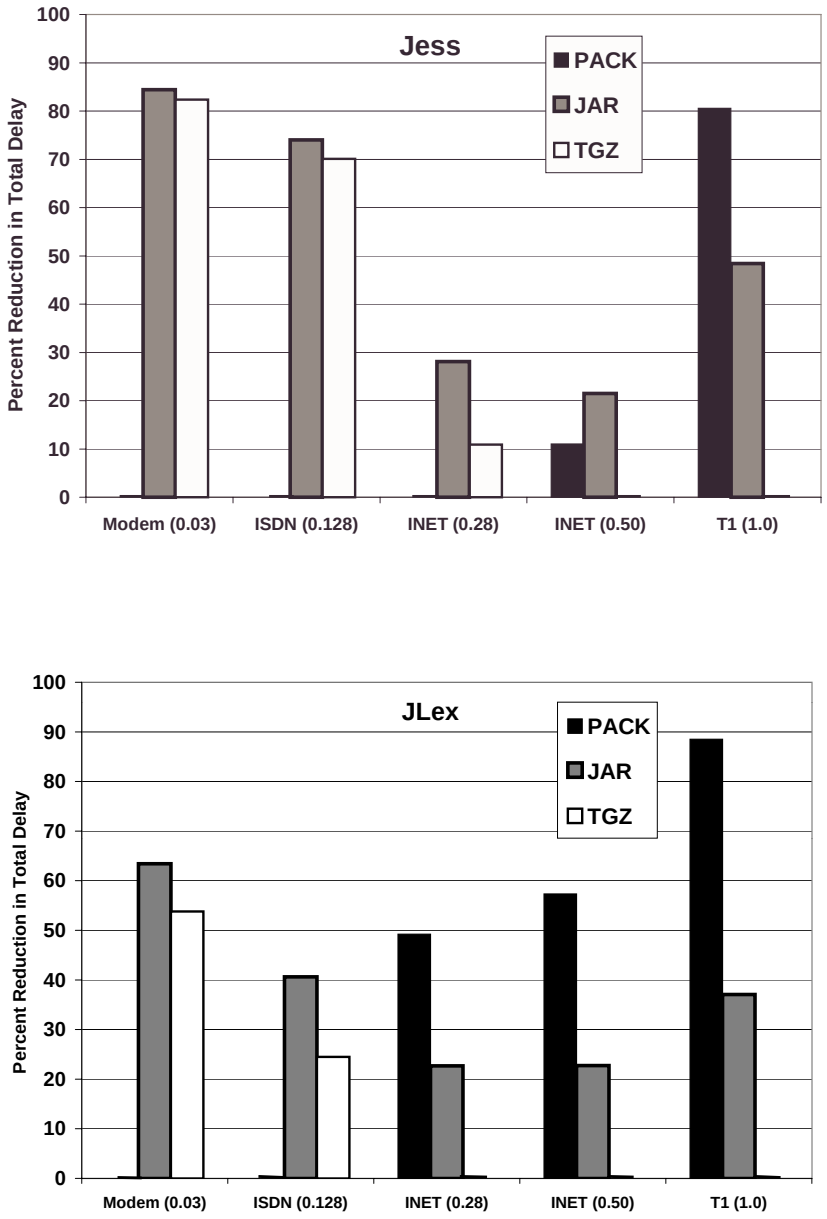


Figure VIII.4: Pct. reduction in total delay due to DCFS for Jess and Jlex.
Each graph shows data for a different benchmark. Each bar is a different compression format (base case) and represents the percent reduction in total delay (y-axis) when DCFS is used instead of that format alone. The choice made by DCFS for a given network (x-axis) is represented as zero-valued (missing) bars, i.e., when the format chosen is the base case, DCFS enables no further reduction since it has selected the optimal format given the ones available. In every case, DCFS correctly determines and uses the format that requires the minimum total delay and significantly reduces it.

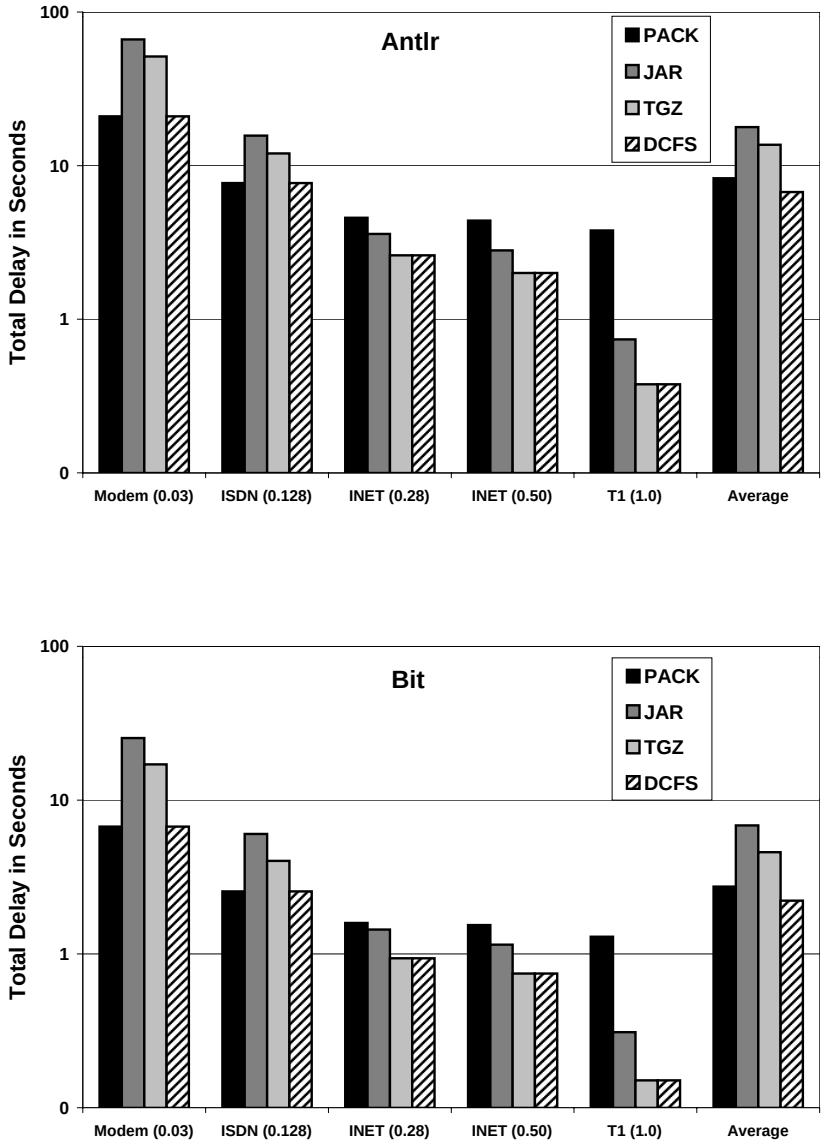


Figure VIII.5: Total delay in (log) seconds using DCFS for Antlr and Bit.
This set of graphs shows (for each benchmark) the total number seconds required for request, transfer, and decompression using each compression technique (bar), PACK, JAR, TGZ. The fourth (striped) bar of each set is the total delay when using DCFS. DCFS selects the format that results in the minimum total delay.

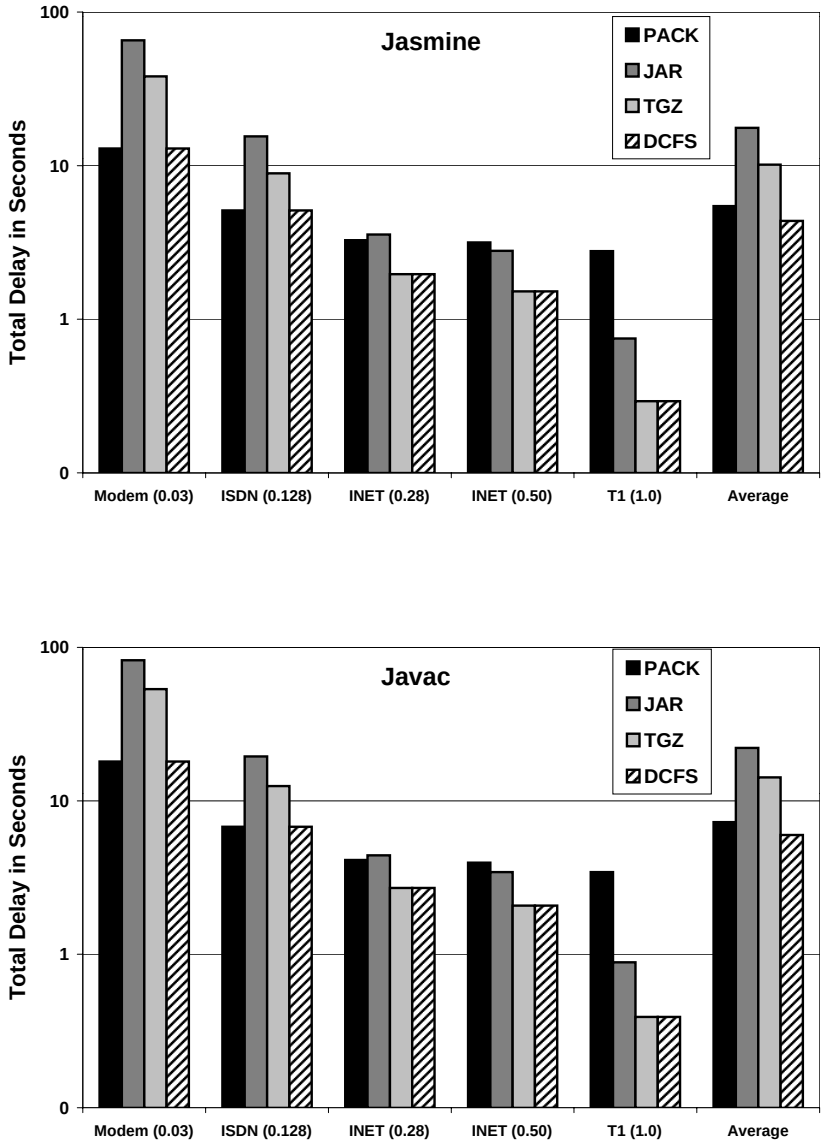


Figure VIII.6: Total delay in (log) seconds using DCFS for Jasmine and Javac.
This set of graphs shows (for each benchmark) the total number seconds required for request, transfer, and decompression using each compression technique (bar), PACK, JAR, TGZ. The fourth (striped) bar of each set is the total delay when using DCFS. DCFS selects the format that results in the minimum total delay.

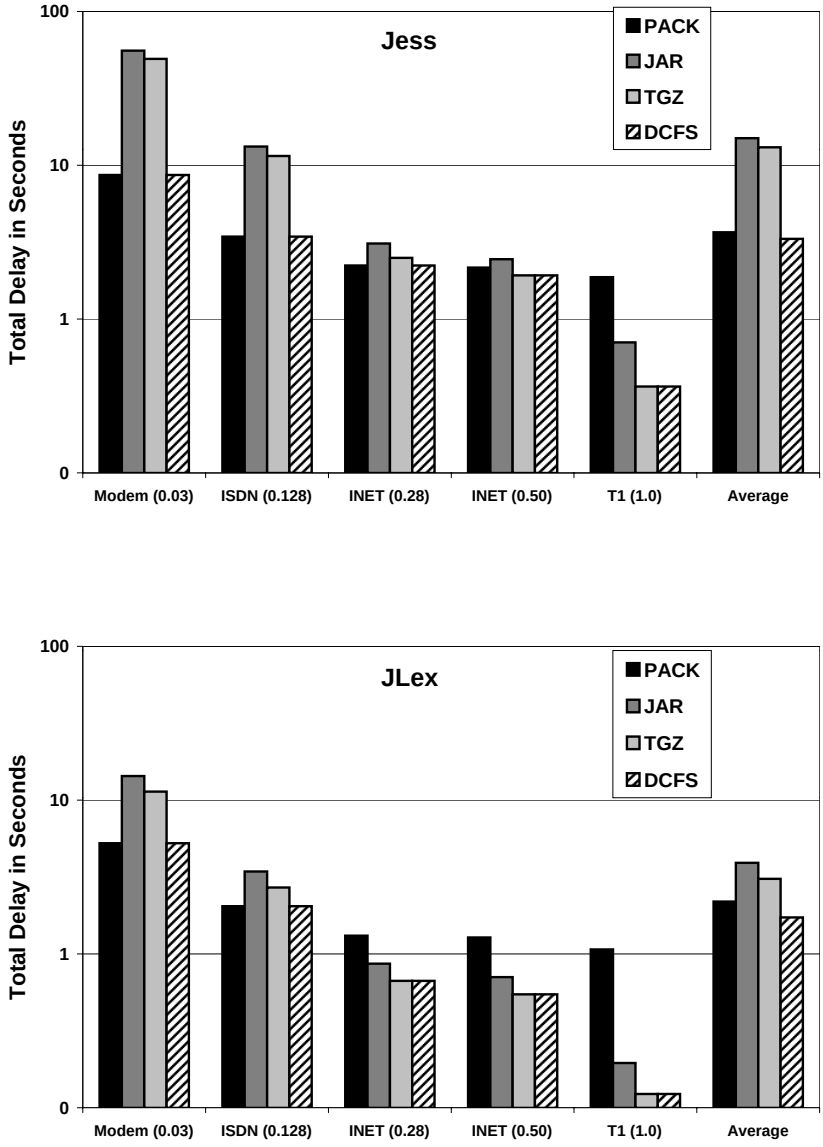


Figure VIII.7: Total delay in (log) seconds using DCFS for Jess and Jlex.
This set of graphs shows (for each benchmark) the total number seconds required for request, transfer, and decompression using each compression technique (bar), PACK, JAR, TGZ. The fourth (striped) bar of each set is the total delay when using DCFS. DCFS selects the format that results in the minimum total delay.

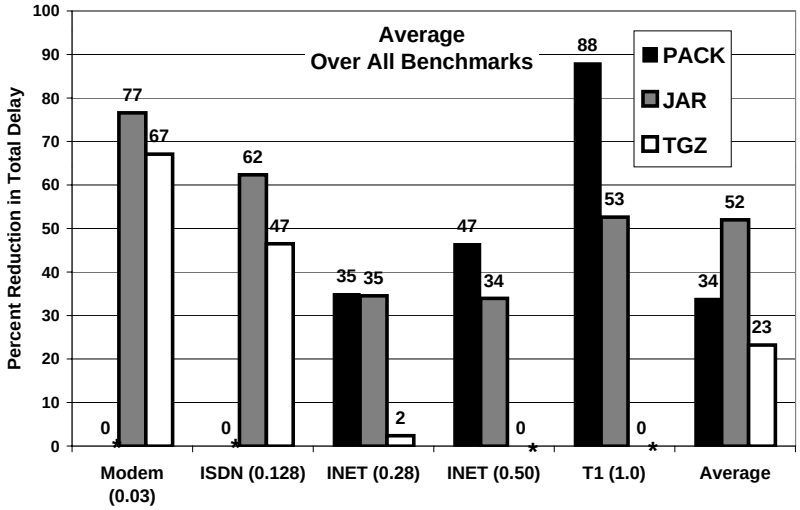


Figure VIII.8: Average reduction in transfer delay enabled by DCFS.
Each bar shows the average percent reduction in total delay across all benchmarks for each of the three compression formats. Bars with zero values indicate that the base case was the format selected by DCFS, i.e., the base case was optimal and no additional benefits are possible. In every case, DCFS correctly determines and uses the format that requires the minimum total delay and significantly reduces it. The average reduction in total delay over all network types is shown by the rightmost three bars.

seconds), and 23% (2.3 seconds) of the delay can be eliminated over PACK, JAR, and TGZ, respectively, if selection of the compression format is made dynamically.

Interestingly, JAR is never selected by DCFS (there are no zero-valued JAR bars in Figures VIII.2 through VIII.4) using the bandwidths examined. This implies that using DCFS with only two compression formats can improve (substantially, in many cases) the performance of programs compressed using jar, given any network technology. This also implies that only two formats need to be stored at the server for application download, given current compression technology, to achieve the substantial reductions in total delay presented here. As compression utilities change, however, providing additional DCFS choices enables additional opportunity for improved performance. On average, across all benchmarks and bandwidths, DCFS reduces total delay imposed by jar compression, the most commonly used Java application compression technique, by more than half.

VIII.B.2 Selective Compression

We next presents results for selective compression. To evaluate the effectiveness of selective compression, we present results (without DCFS) in terms of the percent reduction in total delay. The graphs in Figures VIII.9 through VIII.14 show the percent reduction using the PACK, JAR, and TGZ formats; for each benchmark we show a pair (row) of graphs. The x-axis is network bandwidth and the y-axis is the percent reduction in total delay (transfer plus decompression) for each compression format due to selective compression. The top graph of each pair shows the test result: the train input is used for both profile and result generation; the bottom graph shows the cross-input, test results. On average, reduction in total delay for all compression formats is 14% with perfect information and 10% across inputs.

In Figures VIII.13 and VIII.14 there are two cross-input (bottom) graphs with negative bars. For these benchmarks, `BIT` and `Jess`, selective compression degrades performance. For these benchmarks, misprediction degrades performance. Misprediction occurs when class files are used that were not predicted as used and thus were not included in the selectively compressed archive. A small number of mispredicted class files does not increase transfer delay in most cases, however it is possible for selective compression to degrade performance when the class usage patterns across inputs differ greatly.

To combat this, we modified DCFS to check the difference in the sizes of the completely compressed and the selectively compressed application. For some programs, there is little difference between the transfer time required for an entire application (compressed) and the

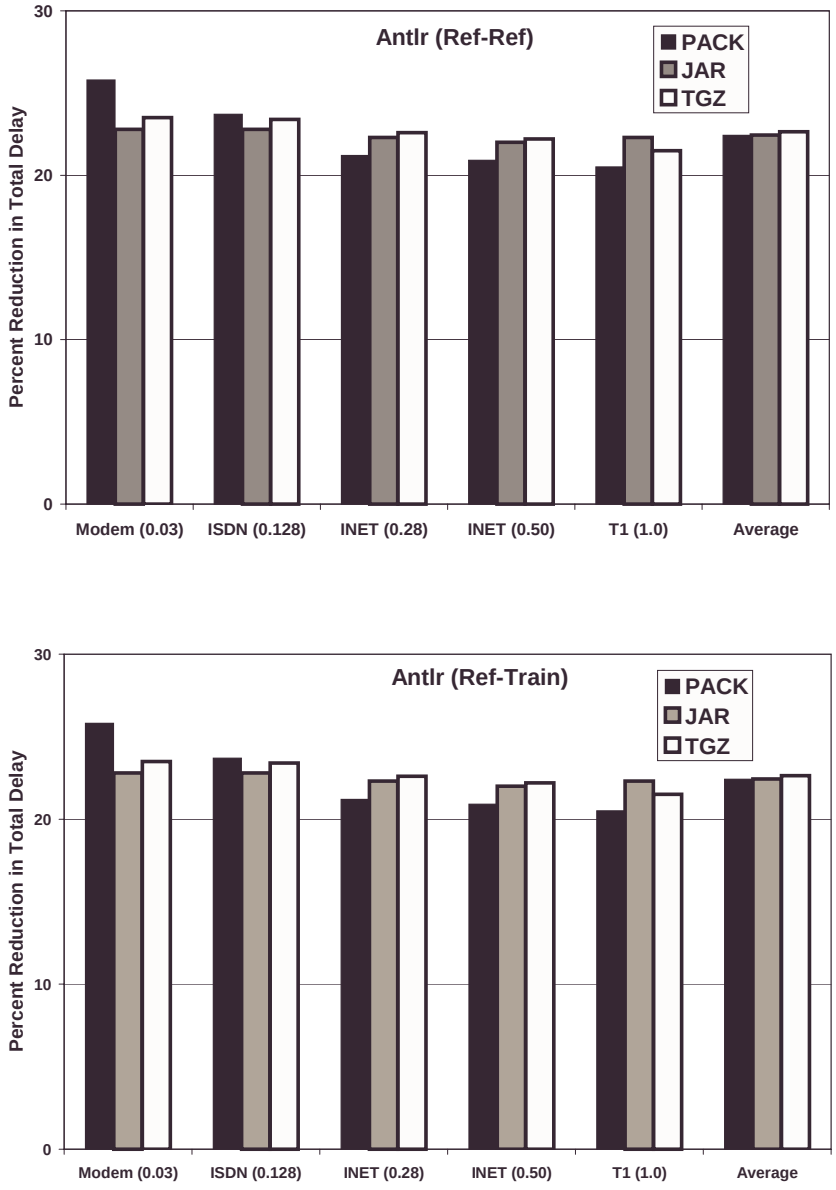


Figure VIII.9: Pct. reduction in total delay due to selective compression (Antlr). The top graph is data collected by using the same (test) input for both profiling and result gathering (Ref-Ref); the bottom uses a training input to generate the profile (Ref-Train). The y-axis shows the percent reduction in total delay. The average reduction in total delay is shown in the rightmost three bars.

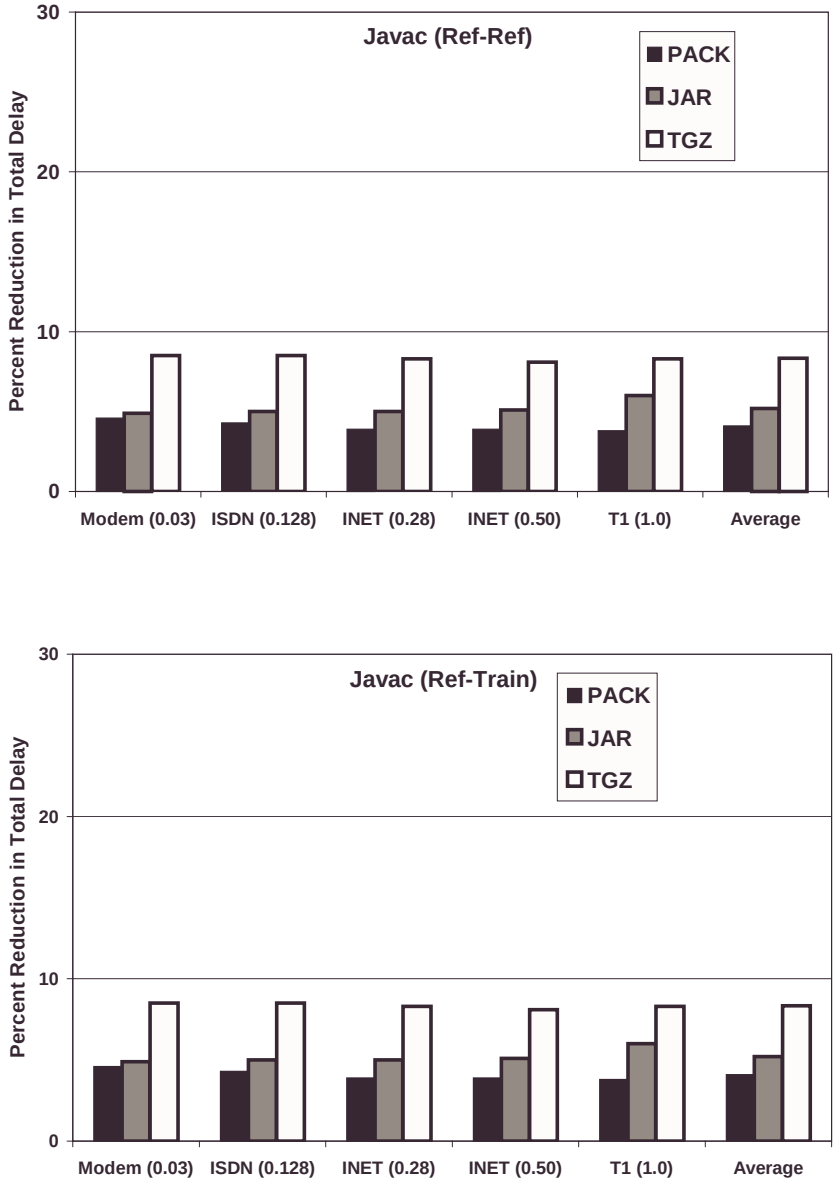


Figure VIII.10: Pct. reduction in total delay due to selective compression (Javac). The top graph is data collected by using the same (test) input for both profiling and result gathering (Ref-Ref); the bottom uses a training input to generate the profile (Ref-Train). The y-axis shows the percent reduction in total delay. The average reduction in total delay is shown in the rightmost three bars.

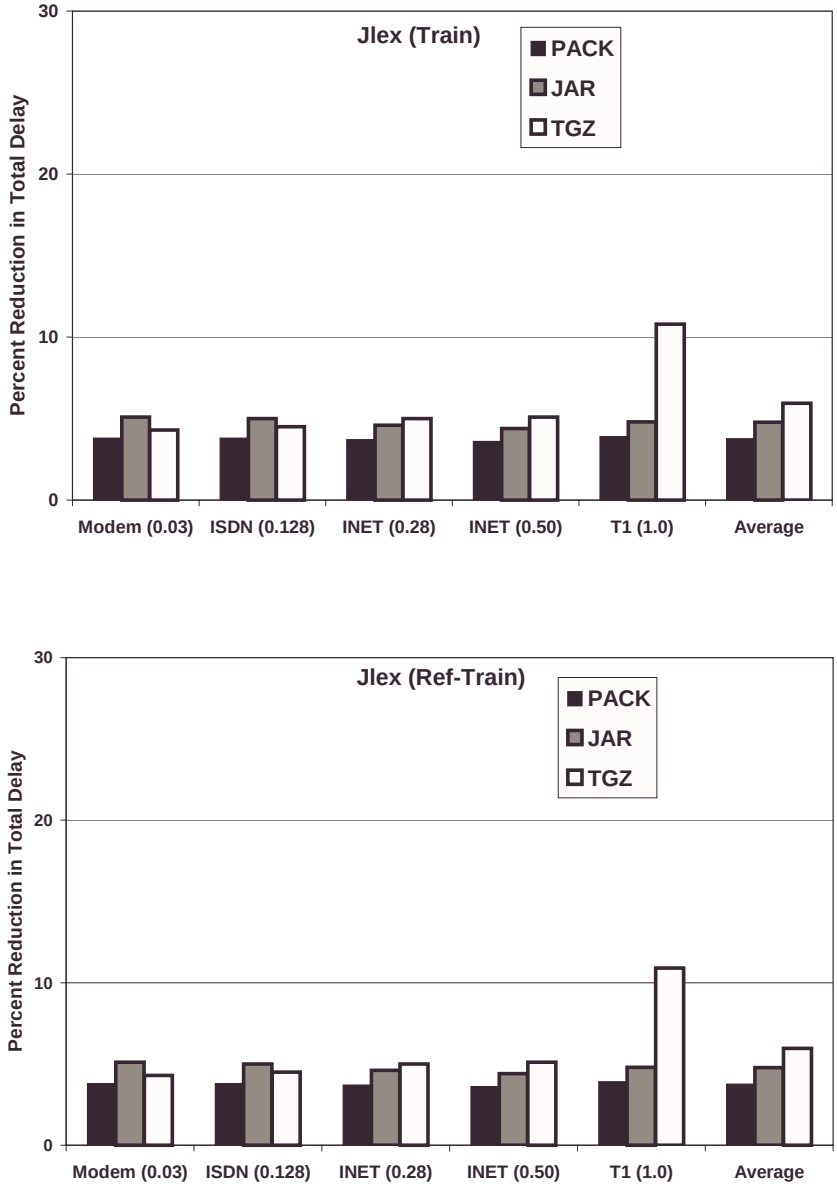


Figure VIII.11: Pct. reduction in total delay due to selective compression (Jlex). The top graph is data collected by using the same (test) input for both profiling and result gathering (Ref-Ref); the bottom uses a training input to generate the profile (Ref-Train). The y-axis shows the percent reduction in total delay. The average reduction in total delay is shown in the rightmost three bars.

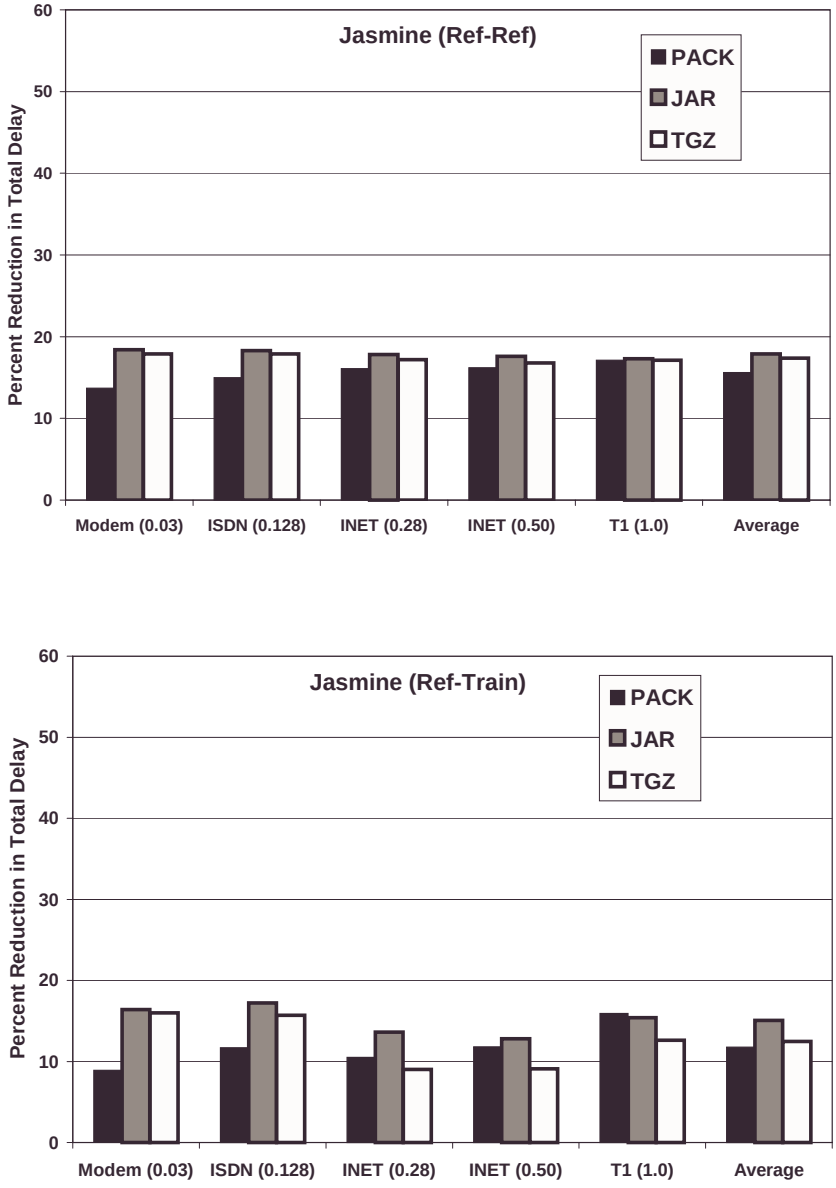


Figure VIII.12: Pct. reduction in total delay due to selective compression (Jasmine). The top graph is data collected by using the same (test) input for both profiling and result gathering (Ref-Ref); the bottom uses a training input to generate the profile (Ref-Train). The y-axis shows the percent reduction in total delay. The average reduction in total delay is shown in the rightmost three bars.

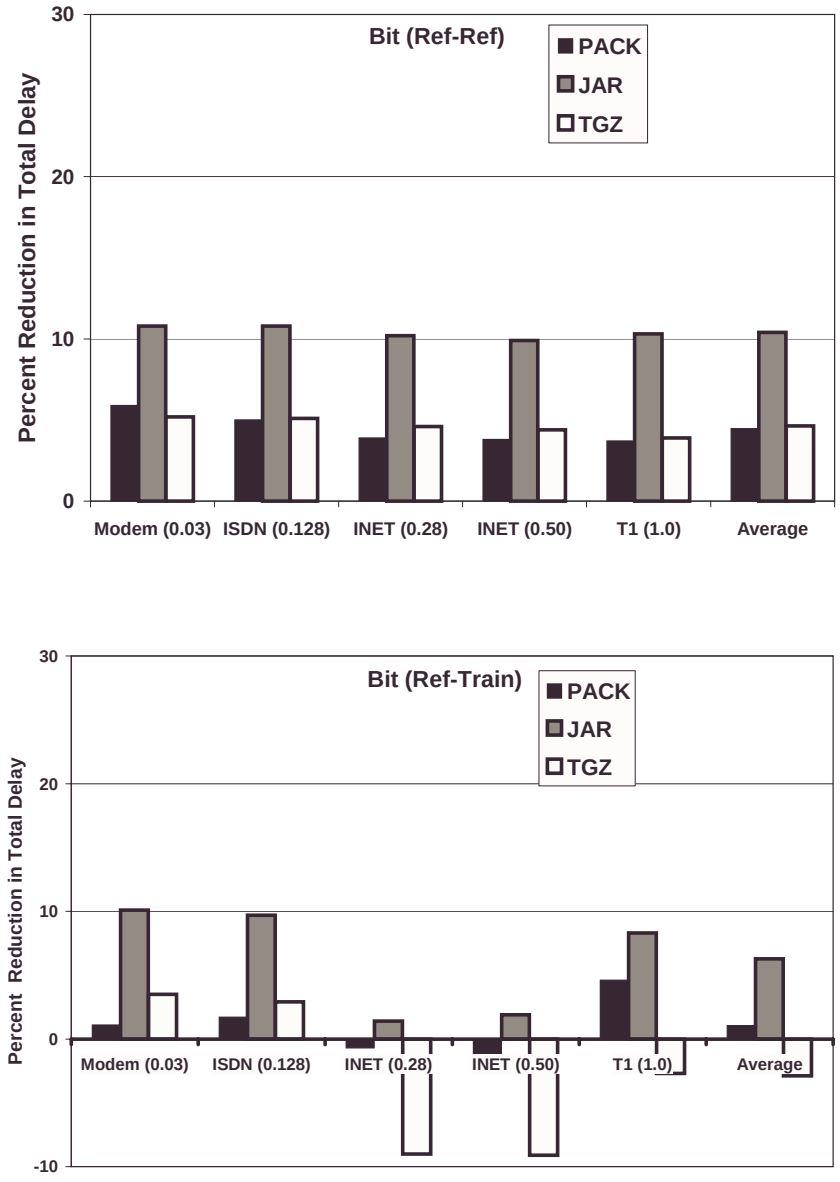


Figure VIII.13: Pct. reduction in total delay due to selective compression (Bit). The top graph is data collected by using the same (test) input for both profiling and result gathering (Ref-Ref); the bottom uses a training input to generate the profile (Ref-Train). The y-axis shows the percent reduction in total delay. The average reduction in total delay is shown in the rightmost three bars. Degradation in performance for cross-input (bottom graph) results, this is corrected by incorporating selective compression into DCFS.

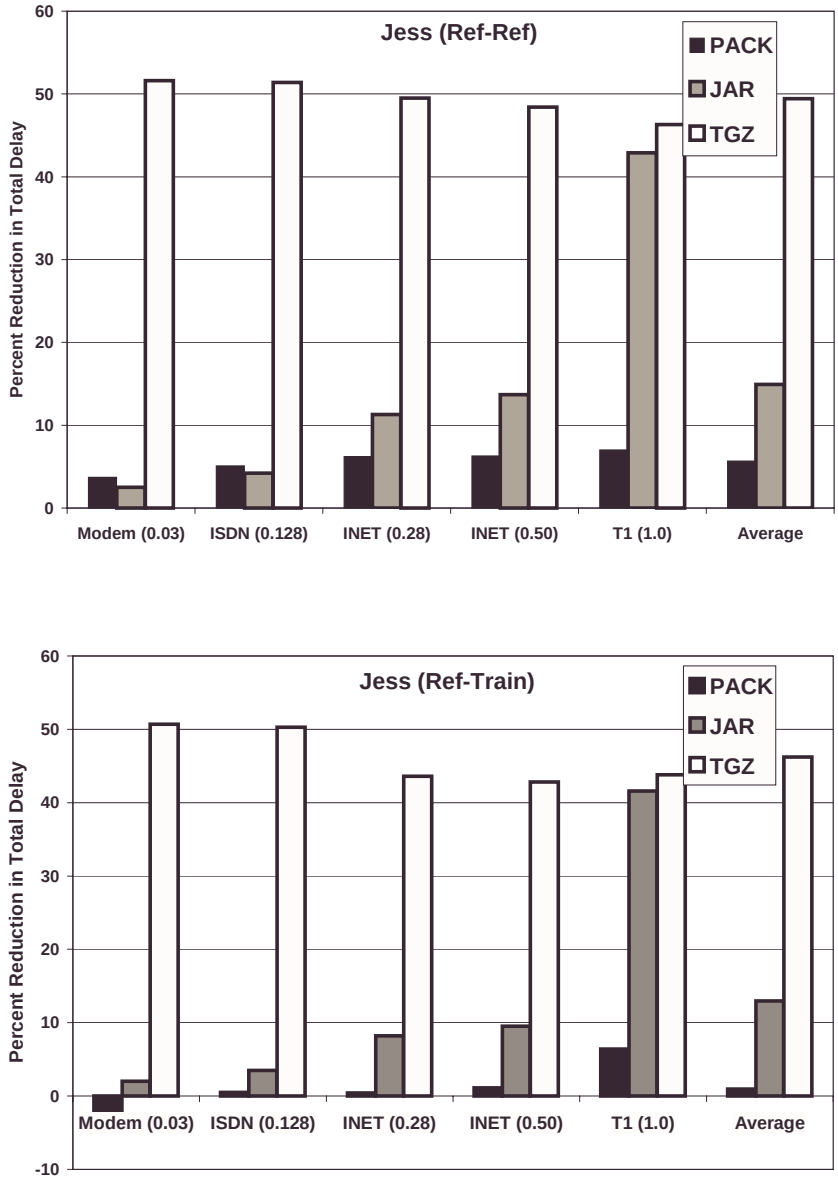


Figure VIII.14: Pct. reduction in total delay due to selective compression (Jess)
. The top graph is data collected by using the same (test) input for both profiling and result gathering (Ref-Ref); the bottom uses a training input to generate the profile (Ref-Train). The y-axis shows the percent reduction in total delay. The average reduction in total delay is shown in the rightmost three bars. Degradation in performance for cross-input (bottom graph) results, this is corrected by incorporating selective compression into DCFS.

Table VIII.2: Pct. difference in sizes of complete and selective compression.
The table includes the sizes of completely compressed applications and selectively compressed versions of each benchmark that differs across inputs. Data for both inputs (train and test) are shown for each compression technique. We use this information to determine when it is feasible to use selective compression. If the size of the selectively compressed application is more than 5% greater than that of the complete application, then we use selective compression. We request the complete application for BIT using PACK and TGZ and for Jess using PACK since this size criteria is not met.

	Percent Difference in Size Of Entire Compressed Application and Selectively Compressed Application					
	PACK		JAR		TGZ	
Program	Train	Test	Train	Test	Train	Test
Bit	4.9	4.8	10.9	9.4	5.0	4.9
Jasmine	12.8	12.7	18.4	17.8	18.0	17.5
Jess	2.8	3.2	5.5	5.8	51.7	52.1

selectively compressed version, given various network speeds. This occurs if the sizes of the two versions are very similar. When class files are used during execution, which have not been transferred as part of the selectively compressed archive, they transfer alone, on-demand. When the delay incurred by this additional transfer is larger than the reduction of total delay due to selective compression, performance is degraded. Therefore we use the difference between the completely compressed and the selectively compressed files. If the potential transfer delay reduced by this difference is less than the transfer delay required for transfer of an additional class file, then the complete application is requested in its compressed format. We use the average class file size in each application for this computation.

Table VIII.2 shows the percent difference between the size of each compressed application and the size of the selectively compressed version for three benchmarks that have different class usage patterns across inputs. When the size of the selectively compressed application is within 5% of the entirely compressed application, the benefit from selective compression is small and the risk of performance degradation due to incorrect prediction increases. To ensure that selective compression does not degrade performance, we use a size heuristic performed on the server when selectively compressed files are created. If the selectively compressed size is within 5%, selective compression is not used. In this case, the compressed file on the server will contain all of the class files. Figures VIII.15 and VIII.16 show the effect of this DCFS modification to incorporate selective compression.

Figures VIII.17, VIII.18, and VIII.19 show the potential improvement when selective

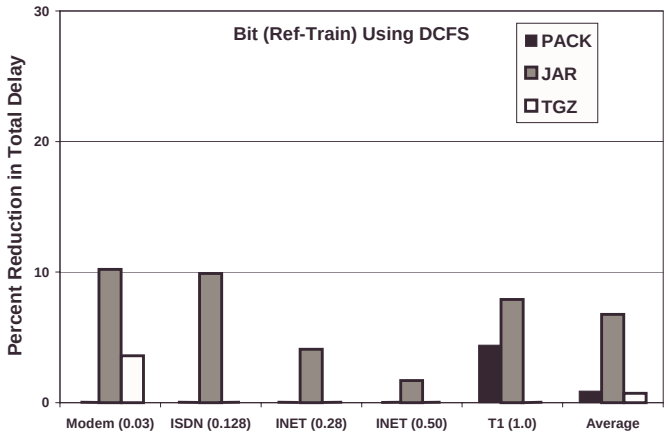


Figure VIII.15: Pct. reduction in total delay (across inputs) for the Bit benchmark. The graph shows the effect of selective compression across inputs for the Bit benchmark when selective compression is incorporated into DCFS in which the decision whether or not to request the selectively compressed application is made dynamically.

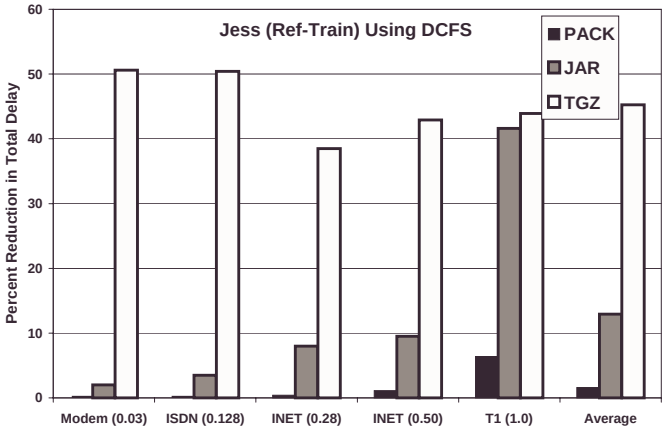


Figure VIII.16: Pct. reduction in total delay (across inputs) for the Jess benchmark. The graph shows the effect of selective compression across inputs for the Jess benchmark when selective compression is incorporated into DCFS in which the decision whether or not to request the selectively compressed application is made dynamically.

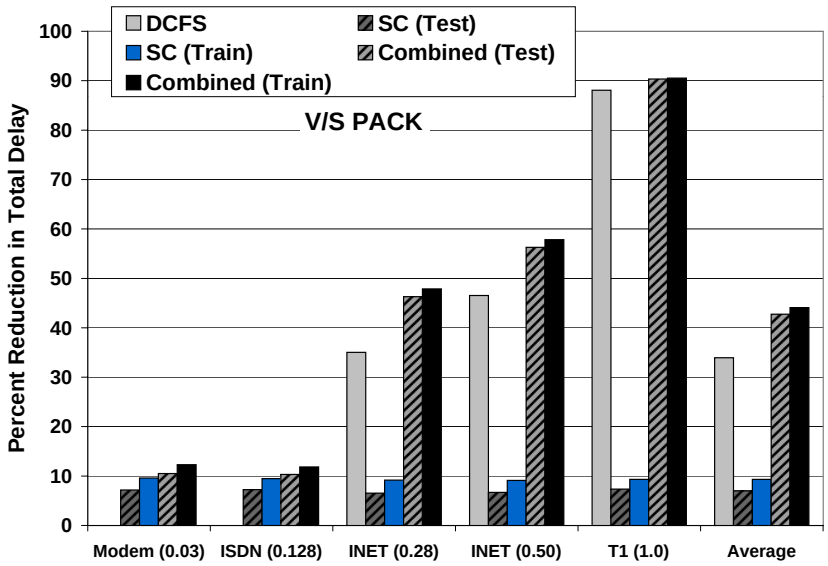


Figure VIII.17: Summary of results using PACK compression as base case.
A series of bars is shown for each network bandwidth. From left to right, the five bars represent the percent reduction in total delay due to DCFS alone, selective compression alone (Ref-Train and Ref-Ref), and DCFS and selective compression combined (Ref-Train and Ref-Ref). The average across the range of network bandwidth is given by the final set of five bars.

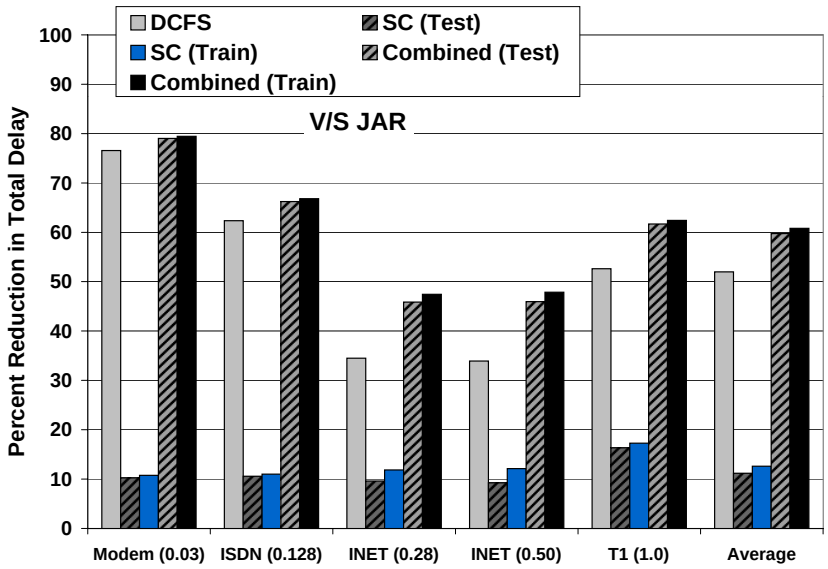


Figure VIII.18: Summary of results using JAR compression as base case.
A series of bars is shown for each network bandwidth. From left to right, the five bars represent the percent reduction in total delay due to DCFS alone, selective compression alone (Ref-Train and Ref-Ref), and DCFS and selective compression combined (Ref-Train and Ref-Ref). The average across the range of network bandwidth is given by the final set of five bars.

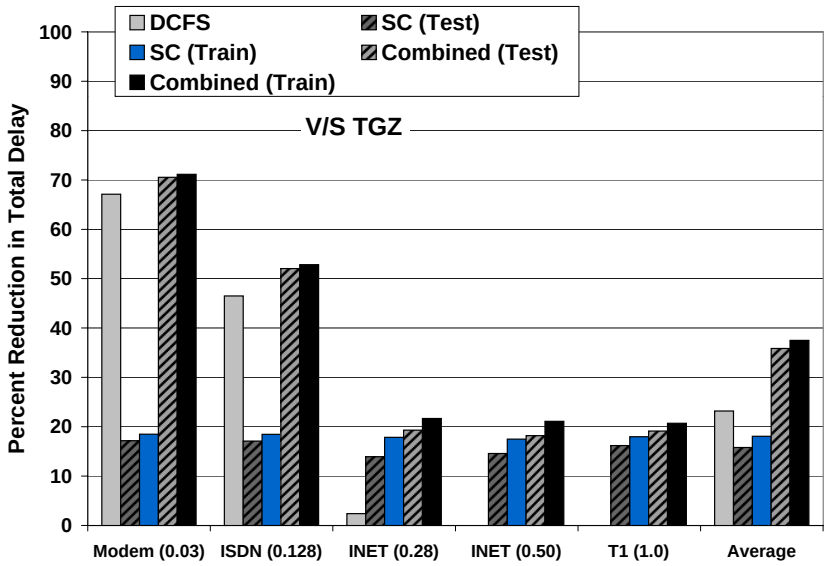


Figure VIII.19: Summary of results using TGZ compression as base case.
A series of bars is shown for each network bandwidth. From left to right, the five bars represent the percent reduction in total delay due to DCFS alone, selective compression alone (Ref-Train and Ref-Ref), and DCFS and selective compression combined (Ref-Train and Ref-Ref). The average across the range of network bandwidth is given by the final set of five bars.

compression is combined with DCFS. The graphs present the results as percent reduction in transfer delay. A separate graph is shown each for a different base case, PACK (Figure VIII.17), JAR (Figure VIII.18), and TGZ (Figure VIII.19). The graphs indicate the improvement in total time (request, transfer, and decompress) an application would experience if DCFS were used over always using Pack, JAR, or TGZ compression. A set of five bars are included for each of the network bandwidths which show the average performance across benchmarks. The first bar in the set are the results due to DCFS alone as presented previously. The second and third bars indicate the performance benefit from selective compression using different profile inputs (Ref-Train, Ref-Ref). The final two bars show the combined effect of selective compression and DCFS using the different inputs (for selective compression) for profile and result generation (Ref-Train), and using the same input, respectively. The cross-input (Ref-Train) results show that on average across all benchmarks, selective compression alone reduces transfer delay 8% for the modem link (1.0 seconds) and 8% for the T1 link (0.2 seconds) over always using the PACK utility. When combined with DCFS, for dynamic selection as well as selective compression, delay is reduced 10% for the modem (1.2 seconds) and 90% for the T1 link (2.2 seconds). Average improvements over always using JAR compression are 10% for the modem (5.2 seconds) and 16% for T1 (0.1 seconds) using selective compression alone and 80% for the modem (41.3 seconds) and 61% for T1 (0.4 seconds) when combined with DCFS. Improvements over TGZ are, on average, 18% for the modem (6.6 seconds) and 18% for T1 (0.1 seconds) using selective compression alone and 71% for the modem (26.1 seconds) and 19% for T1 (0.1 seconds) with DCFS.

VIII.C Discussion

In the previous sections, we articulated the DCFS design and reported results to indicate the potential of dynamic selection of compression formats to improve mobile program performance. Our results use average performance values from real network traces. In this section, we discuss practical implementations of DCFS and the implications of incorporating predictions of future network bandwidth.

Our results showed that below a certain bandwidth value (0.19Mb/s in our study), the PACK utility is always selected. For networks for which bandwidth is always less than a given threshold, e.g. 0.03Mb/s (MODEM), we propose that DCFS be used to calibrate the JVM to request applications in the most commonly selected format, if available. This

calibration can be performed at JVM installation or when compression utilities are added and removed, eliminating any application startup overhead introduced by the DCFS. The DCFS can be used in this setting until the underlying network changes. For connections capable of bandwidth values above this threshold (the Internet), no single compression format enables the best performance for all bandwidths. Thus, dynamic selection of compression techniques is needed for networks with variable performance to reduce delay.

VIII.C.1 DCFS for Variable Bandwidth Connections

The results in the previous section show that DCFS is able to select the appropriate compression format that results in the minimum delay. For example, for modem links, DCFS commonly chooses PACK; similarly for LAN, DCFS chooses TGZ. However, the affect of variance is not represented by these results since we use a single network bandwidth value (the trace average). Such network variance can cause DCFS to change the selection for a single link. For example, Figure VIII.20 shows the bandwidth for two Internet connections. The first row of graphs is the data trace from which the INET bandwidth average was obtained. The second row provides data for a different Internet connection between the University of Tennessee and the University of California, San Diego. The left graph of each pair is the raw bandwidth measurement taken; the right is the cumulative distribution function (CDF) over all bandwidth values. Measurements were taken at just under one minute intervals over a 24 hour period that began at approximately 8PM.

In the right, CDF graphs, we have incorporated a vertical and horizontal line. The vertical line indicates the average bandwidth value 0.32 Mb/s at which the DCFS selection changes from PACK to TGZ over all of the benchmarks studied. For less than 42% of the values, PACK is chosen by DCFS for the link represented by the top pair of graphs; the remainder of time TGZ is chosen. For the link represented by the bottom pair, over 50% of the values on average, causes DCFS to select PACK. These results show that it is unclear as to which compression technique to use for this network. Hence, dynamic selection should be used to achieve the least total delay.

VIII.C.2 Prediction of Network Characteristics

In the real-world implementation of DCFS, the future performance of the network (at the time the compressed file transfers) is unknown. We must predict this value to determine which compression technique results in the least total delay. The results in the previous

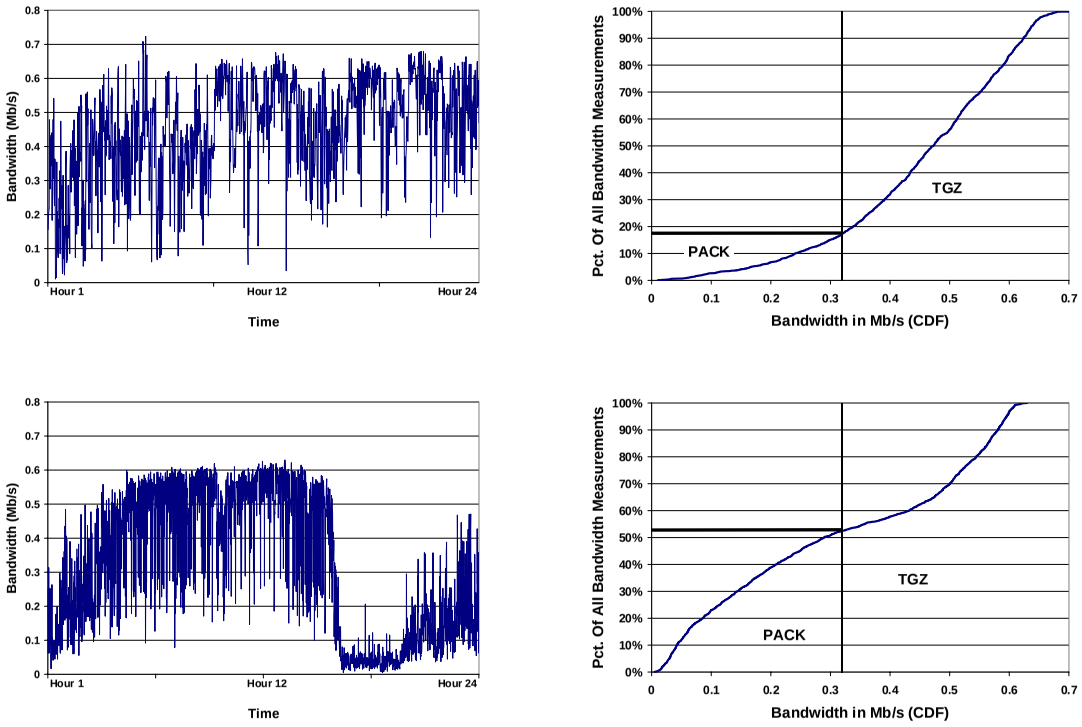


Figure VIII.20: Raw data (left) and cumulative distribution functions (CDF) (right). The top pair is data from the INET trace used throughout this study. The bottom pair in also Internet data, however, between two different hosts. In the left graphs, the y-axis is bandwidth, and the x-axis is time. Both traces are of a single 24-hour period starting approximately at 8PM at night. The data indicates that these connections are highly variable and hence different DCFS choices can be made for a single link. The right graphs indicate (given the average bandwidth value, 0.32Mb/s indicated by the vertical line, at which DCFS chose a different format over all benchmarks studied) that in the top pair, PACK is chosen less than 19% of the time. In the bottom pair, the number of times PACK is chosen by DCFS is about the same number as that for TGZ.

section use a known bandwidth value to compute the total delay; this is the performance (bandwidth) that the application experiences during the transfer. Thus, the results indicate the best performance achievable by the DCFS for that value.

For the use of DCFS to be practical, we must show that this performance potential is not substantially degraded by the use of prediction; which may occur if the predictions made are inaccurate. To determine the accuracy of a predicted value, it is common to examine the error value: the difference between the predicted value and the actual value when it occurs. DCFS incorporates prediction using known techniques and previously implemented tools. Performance prediction is a well studied area beyond the scope of this thesis and is not a contribution made by the techniques we present. We refer the reader to [58, 88, 30, 18] for chapter from a small subset of this research area. The contribution of DCFS is to extend the use existing forecasting techniques for bandwidth prediction to the prediction of total delay and dynamic compression format selection.

To determine the accuracy of predicted values, it is common to examine the error value: the difference between the predicted value and the actual value when it occurs. Prediction errors impact the performance of DCFS only when they cause non-optimal format selection. In the remainder of this section, we empirically evaluate the effect of prediction on the DCFS performance potential. We consider two techniques for bandwidth prediction: last bandwidth prediction via probes for simplicity and network weather service (NWS) prediction for its dynamic choice between multiple predictive algorithms.

Last Bandwidth Prediction Via Probes

One way to predict the bandwidth when a transfer occurs, is to probe the bandwidth immediately prior to transfer. Using this approach, we predict that the bandwidth at a transfer time in the near future, will be equal to the current bandwidth. This is called last bandwidth prediction. Last bandwidth prediction can be incorporated for DCFS bandwidth prediction using any network probe utility available, e.g., ping, netperf [43], TTCP [76], JavaNws [51], etc. In addition, simple probing socket routines can easily be written from scratch.

The accuracy of last bandwidth prediction is demonstrated by its error values. For example, in the Internet connection data in Figure VIII.20 in the top pair of graphs, the average error using last bandwidth prediction is 11.3KB/s. This value is obtained by taking the bandwidth values and subtracting them from the previous value in the trace, and taking the average of this difference over all measurements. On average, the difference between the

last bandwidth value and the bandwidth when the application transfers is 11.3KB/s. However, since DCFS is making a binary decision (1 for PACK and 0 for TGZ in this case) based on whether the prediction is above or below a given threshold, this error will only effect predictions that are $+-$ 11.3KB/s of this threshold. To determine the extent to which this error limits the overall improvement by DCFS, we selected 100 random bandwidth values from both sets of INET trace data presented in the figure and computed the total delay required by each of the wire-transfer formats, as well as by DCFS. We found total delay reduction from DCFS using last bandwidth prediction to be within 4% of that from DCFS using the actual trace values, i.e. perfect information.

NWS Prediction

An alternate approach to last bandwidth prediction is to use the Java implementation of the Network Weather Service [88] prediction utilities in the JavaNws. This tool treats measurement values from network probes as time series. It applies a set of very fast[1], adaptive, statistical forecasting techniques to these time series to produce accurate, short-term predictions of available network performance [88].

The average JavaNws prediction error from the top Internet connection data in Figure VIII.20 is 7.5KB/s (for an average bandwidth value of 0.5 (INET), this is 111ms). Smaller average error improves the potential for correct selection (and improved performance) by the DCFS. Since one of the forecasters used by JavaNws is a last bandwidth predictor, JavaNws will always enables equal or better accuracy than a last bandwidth predictor alone. Using 100 randomly selected bandwidth values from both sets of INET trace data presented in the figure, we achieve total delay reduction from DCFS using JavaNws prediction within 2% of that from DCFS using the actual trace values . For these links, DCFS with JavaNws prediction enables an additional 2% reduction in total delay than last bandwidth prediction alone.

VIII.D DCFS Extensions

An alternative to requiring that the server store applications in a number of different wire-transfer formats, we consider simply storing class files. Then, when a request is made to a server for application download, the server is instructed to compress the application prior to transfer. The format is chosen by DCFS and is sent to the server upon application request.

[1]The JavaNws forecasting techniques require approximately 0.25ms to produce a single prediction, on the processor used in this study.

In this section, we articulate some preliminary results of future work in which we include the compression process with class file load time.

In addition to decompression rates (Kb/s) and compression ratios, wire-transfer format plug-ins to the DCFS will include compression rates. Despite not being optimized for compression time, existing compression techniques can still be incorporated into DCFS to give insight into feasibility of including compression with dynamic format selection at class load time. As utilities change, improve, or are optimized for compression rates, the DCFS can incorporate and select them; those for which compression times prove impractical will not be selected. Total delay, when on-demand compression is performed, consists of time to request a class, to compress the collection of files, to transfer the collection, and to decompress the required class. Our results are shown in Table VIII.3. Using DCFS with compression reduces total delay by 50% over using jar files *without* compression (using the wire-transfer formats and networks from this study). That is, it is faster to compress, transfer, and decompress applications using dynamically selected wire-transfer formats than it is to simply transfer and decompress jar files.

VIII.E Summary

Despite benefits provided by compression, transfer time continues to impede the performance of mobile programs. With this work we exploit the trade-off that is made by compression techniques (compression ratio for decompression time). Since no one technique is best (in terms of transfer and decompression time) for every level of network performance (and such performance is highly variable for a single link), we introduce *Dynamic Compression Format Selection* (DCFS). By dynamically selecting the compression technique based on the underlying, available resource performance, we ensure that for any network bandwidth, the format resulting in the least total delay is used. We show that DCFS reduces total delay on average across the networks and benchmarks studied: 52% (7s) over jar compression, the most commonly used format for mobile Java programs. DCFS reduces total delay for fast links (T1) 90% (2s) over PACK compression on average for the benchmarks studied. For slow links (modem) it can reduce total delay on average 67% (24s) over TGZ (tar and gzip) compression.

We also introduce a technique called selective compression in which only those class files predicted as used during execution are included in the compressed archive. We use off-line profiling to determine which class files to exclude. When combined with the dynamic selection

Table VIII.3: Compression-on-demand with DCFS.
Data is presented for the range of network bandwidths for each benchmark. The first column of data shows the cumulative time for request, decompression, and transfer for jar file remote execution. Jar files are the most common transfer format for Java applications. The second column of data shows the cumulative time for DCFS compression-on-demand (request, compression, transfer, and decompression). The final column shows the percent time reduction enabled by DCFS compression-on-demand. The final set of data shows the average over all benchmarks for each network bandwidth.

		Total Delay In Seconds		
		JAR Transfer and Decompression	DCFS Compression, Transfer, and Decompression	
Program	Network	Time ONLY	Time	% Rdctn
Antlr	MODEM (0.03)	66.3	37.4	43.6
	ISDN (0.128)	15.7	12.3	21.3
	INET (0.28)	3.6	2.9	17.9
	INET (0.50)	2.8	2.3	16.6
	T1 (1.00)	0.7	0.7	3.2
Bit	MODEM (0.03)	25.3	13.1	48.3
	ISDN (0.128)	6.0	4.1	31.4
	INET (0.28)	1.4	1.0	27.3
	INET (0.50)	1.1	0.9	25.6
	T1 (1.00)	0.3	0.3	15.6
Jasmine	MODEM (0.03)	65.5	21.6	67.1
	ISDN (0.128)	15.5	9.2	40.6
	INET (0.28)	3.6	2.3	36.8
	INET (0.50)	2.8	1.8	35.3
	T1 (1.00)	0.7	0.6	22.5
Javac	MODEM (0.03)	82.4	33.0	60.0
	ISDN (0.128)	19.5	12.9	34.0
	INET (0.28)	4.4	3.1	30.1
	INET (0.50)	3.4	2.5	28.6
	T1 (1.00)	0.9	0.8	13.5
Jess	MODEM (0.03)	55.6	14.9	73.0
	ISDN (0.128)	13.2	4.4	66.4
	INET (0.28)	3.1	1.1	63.9
	INET (0.50)	2.4	0.9	62.7
	T1 (1.00)	0.7	0.2	60.5
Jlex	MODEM (0.03)	14.4	11.1	22.3
	ISDN (0.128)	3.4	2.8	19.6
	INET (0.28)	0.9	0.7	15.6
	INET (0.50)	0.7	0.6	14.1
	T1 (1.00)	0.2	0.2	5.8
Avg	MODEM (0.03)	51.6	21.8	52.4
	ISDN (0.128)	12.2	7.6	35.6
	INET (0.28)	2.8	1.8	31.9
	INET (0.50)	2.2	1.5	30.5
	T1 (1.00)	0.6	0.5	20.2

of DCFS, we are able to reduce delay, on average, 90% (2s), 61% (400ms), and 19% (100ms) over always using either PACK, JAR, or TGZ over a network link with 1Mb/s bandwidth. For a modem link (0.03Mb/s), we reduce delay 10% (1s), 80% (41s), and 71% (26s) over PACK, JAR, and TGZ on average.

To reduce total delay, the DCFS implementation requires that applications be stored in various formats at the server and that the server compute minimum load delay. Currently, servers supply users with mirror sites to improve download times. In addition, companies that manage servers are motivated by competition and continuously improve sites to ensure the satisfaction of customers/users. We believe that our results motivate the need for compression format selection and as such, storage of applications in additional formats is a reasonable tradeoff.

The text of this chapter is in part a reprint of the material that has been submitted to the 2001 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC). The dissertation author was the primary researcher and author and the co-authors listed on this publication directed and supervised the research which forms the basis for this chapter.

Chapter IX

General Overview on Reducing Compilation Delay

The execution model for remotely executed, Java programs once at the destination, is one of either interpretation or dynamic compilation. With interpretation, bytecodes (the format of Java programs) are executed instruction-by-instruction. It is a very simple mechanism and enables immediate progress to be made by the application since this translation of an individual instruction is very fast. However, interpretation imposes severe performance limitations on mobile program execution. Since the process only considers a single bytecode instruction at a time, the quality of the resulting native code is very poor. In addition, traditionally, there is no reuse of interpreted code, i.e. multiple executions of the same instruction are repeatedly interpreted.

In an effort to overcome the performance limitations of interpretation the next generation of Java execution systems [79, 3, 65, 34] employ dynamic, or just-in-time, compilation. These new JVMs dynamically compile the bytecode stream (on a method-by-method basis) into machine code before executing it. The resulting execution performance is substantially higher than for interpreted bytecodes, but execution must pause each time a method is initially invoked so that it may be compiled. We refer to this intermittent, pause time as *compilation* delay.

Compilation also exposes optimization opportunities unavailable to interpretation. Optimization can theoretically reduce execution time of mobile program to near that of a similar C program. Dynamic compilation offers the potential for better performance than can be achieved by static compilation since runtime information can be exploited for optimization and specialization. Several dynamic, optimizing compiler systems have been built in industry

and academia [3, 8, 29, 34, 44, 45, 56, 79]. Despite its potential benefits, optimization increases compilation delay since it is performed while the program executes.

Most systems attempt to reduce compilation delay introduced by the optimization in one of two ways: they incorporate multiple compilers [12, 16, 84], or they use an interpreter in coordination with an optimizing compiler [34, 68]. The dual-compiler systems use one very fast, non-optimizing compiler and one slow, optimizing compiler. Typically, both system types use the fast compiler or interpreter when methods execute for the first time. Then, on-line measurements are made (using instrumented execution), to determine when program execution characteristics warrant optimization. When a threshold for a method is met, the optimizing compiler re-compiles it using various levels of optimization (or just a single level in some systems). Such systems are called adaptive compilation systems since they use optimization to enable program performance to adapt as program execution behavior changes.

In the following two chapters, we present techniques that propose alternate uses of two adaptive compilation systems to reduce compilation delay. We use the general techniques of *overlap* and *avoidance* as in the previous chapters on the reduction of transfer delay. We first consider the Jalapeño virtual machine from IBM T. J. Watson Research Center [3] in Chapter X. In this chapter, we first empirically evaluate the effectiveness of method-level, or lazy, compilation in contrast to class-level, or eager, compilation. We then present *Background Compilation*, in which off-line execution profile information is used to selectively optimize the program (thereby avoiding unnecessary optimization). In addition, we use a background processor to perform all optimization so that it is overlapped with useful work.

In Chapter XI, we present *Annotation-guided Compilation* using the Open Runtime Platform (ORP) [16] from Intel Corporation. For this study, we perform as much analysis as possible off-line and communicate the results in the bytecode stream of the application. At runtime, the results (annotations) are used by the compilation system to "shortcut" optimization decisions and reduce compilation time. In addition, the information we annotate also includes profile information. This enables the compilation system to avoid optimization as guided by the profile data to further reduce compilation delay.

The results and measurements made in these two chapters cannot be compared due to the difference in architectures upon which the available execution environments ran at the time these studies were performed. The Jalapeño Virtual Machine executes on a PowerPC (a 166Mhz dual processor machine was used). ORP is an x86 based tool which we ran on 300Mhz single processor hardware. The techniques we present are general, however; in fact, annotation-

guided compilation extends the techniques and results presented for background compilation. All of the techniques substantially reduce compilation overhead to improve the performance of mobile programs.

Chapter X

Compilation Delay Avoidance and Overlap: Background Compilation

The execution model for mobile programs consists of code and data first being transferred to a remote destination and then executed. Typically, an architecture-independent program representation (e.g., bytecodes for the Java language) is shipped to the execution site and interpreted by a virtual machine. However, to overcome the performance limitations interpretation usually imposes, these systems now employ just-in-time compilation [79, 3, 65, 34]. These new virtual machines dynamically compile the bytecode stream (on a method-by-method basis) into machine code before executing it. The resulting execution time is lower than for interpreted bytecodes, but execution must pause each time a method is initially invoked so that it may be compiled. The tradeoff imposed by dynamic compilation for the improved execution time is compilation overhead. Since compilation and optimization occurs at runtime, execution must stall until compilation completes.

The goal of this chapter is to develop techniques that reduce the effect of compilation delay while maintaining optimized execution performance. To better understand dynamic compilation overhead, we first evaluate and quantitatively compare the tradeoffs between eager, or class–level, and lazy, or method–level, compilation. Lazy compilation is used in all existing JIT compilation environments but there is no study, to our knowledge, that empirically evaluates the differences between lazy and eager compilation. Since more optimization can be performed across methods within a class file, an aggressive optimizing compiler using eager compilation may be able to produce more efficient code than one which only optimizes at the method-level (lazily). However, such optimization may too costly to perform dynamically; lazy compilation

guarantees that only those methods executed are optimized. Our studies using a specific lazy and eager compilation implementation show that lazy compilation outperforms eager. We detail our experiences with this implementation and empirical evaluation.

We then use the remainder of the chapter to introduce *Background Compilation*. Background compilation is a technique in which a dedicated processor on an SMP machine is used for optimization. This enables overlap with compilation with application execution so that optimization overhead is masked. In addition, we use profiles to guide the selection of methods to optimize thereby avoiding unnecessary optimization. Our results show that background compilation achieves optimized execution time with very little optimization overhead.

The infrastructure used to perform our measurements of compilation delay is Jalapeño, a new JVM (Java Virtual Machine) built at the IBM T. J. Watson Research Center. Jalapeño [3] is a multiple–compiler, compile–only JVM (no interpreter is used). Therefore, it is important to consider compilation delay in the overall performance of the applications executed. Prior to the work reported in this chapter, the default compilation mode in Jalapeño was eager compilation. After the results reported in this chapter were obtained, the default compilation mode for Jalapeño was changed to lazy compilation.

X.A Design And Implementation

Dynamic class loading in Java loads class files as they are required by the execution on demand. Using Just-In-Time (JIT) compilation, each method is compiled upon initial invocation. We refer to this method-level approach as **Lazy Compilation**. Lazy compilation is used in most dynamic compilation systems [34, 45, 52, 81].

An alternative approach is *Eager Compilation*. Instead of compiling a single method at a time, an entire class file is compiled when it is first accessed. Prior to this study, the Jalapeño virtual machine only used eager compilation. In this section, we describe our experiences with, and the implementation of, lazy compilation in Jalapeño. As a result of this work, both eager and lazy compilation were made available in Jalapeño; lazy compilation has become the default. More importantly, with this study we empirically quantify the performance differences between eager and lazy compilation.

We implemented eager compilation in Jalapeño for its reduced complexity and potential benefits. First, eager compilation reduces the overhead caused by switching between execution and compilation. Switching may decrease application memory performance by polluting the cache during compiler operation. If all of the methods in a class file are used during

execution, eager compilation results in compilation of the same methods and substantially less switching overhead. Second, eager compilation can also potentially improve execution performance since it simplifies interprocedural analysis and optimization by ensuring that all methods of a class are analyzed before any of them are compiled.

However, eager compilation increases the time required by class file loading since the entire class file is compiled before execution continues. This delay is experienced the first time each class is referenced. In some cases, it may take seconds to compile a class if high optimization levels are used, affecting a user's perception of the application performance. In addition, for some applications, many methods may be compiled and optimized but never invoked, leading to unnecessary compilation time and code bloat. It is unclear whether lazy or eager compilation results in the best overall performance. This study empirically determines the answer. To our knowledge, no such study has yet been performed.

X.A.1 Lazy Compilation

As part of loading a class file in Jalapeño, entries for each method declared by the class are created in the class' virtual function table and/or a static method table. These entries are the code addresses that should be jumped to when one of the methods is invoked. In eager compilation, these addresses are simply the first instruction of the machine code produced by compiling each method. To implement lazy compilation, we instead initialize all virtual function table and static method table entries for the class to refer to a single, globally shared stub[1]. When invoked, the stub will identify the method the caller is actually trying to invoke, initiate compilation of the target method as necessary[2], update the table through which the stub was invoked to refer to the real compiled method, and finally, resume execution by invoking the target method. Our implementation of lazy compilation is somewhat similar to the backpatching done by the Jalapeño baseline compiler to implement dynamic linking [4] and shares some of the same low-level implementation mechanisms (notably, special compilation of "dynamic bridge" methods to ensure that both volatile and non-volatile registers are saved by the callee). After the stub method execution completes, all future invocations of the same class

[1] Note that using a single globally shared stub complicates the implementation of the "method test" used by the optimizing compiler to perform guarded inlinings of non-final virtual methods. This test relies on the invariant that pointer equality of target instructions implies that the source-level target methods are equal. Therefore, when the method test is being used for guarded inlining, the virtual function tables are initialized with unique trampolines that jump to the globally shared stub.

[2] Because we lazily update virtual function tables on a per-class basis, it is possible that the target method has already been compiled but that some virtual function tables have not yet been updated to remove the stub method.

and method pair will jump directly to the actual, compiled method.

X.A.2 The Effect of Lazy Compilation

To gather our results using this lazy approach, we time the compilation using internal Jalapeño performance timers. Whenever a compiler is invoked, the timer is started; the timer is stopped once compilation completes. To measure the execution time of the program, we use the time reported by a wrapper program called SpecApplication.class distributed with the Spec JVM98 programs [77]. Programs are executed repeatedly (10 times) in succession, and timings of the execution are made separately.

To analyze the effectiveness of lazy compilation we first compare the total number of methods compiled with and without lazy compilation. Figure X.1 depicts the percent reduction in the number of methods compiled using the Ref input. The numbers are very similar for the Train input since the total number of methods used is similar in both inputs. Above each bar is the number of methods compiled lazily, shown to the left of the slash, and eagerly, shown to the right of slash. On average, lazy compilation compiles 57% fewer methods than eager compilation.

To understand the impact of lazy compilation in terms of reduction in compilation overhead, we measured compilation time in Jalapeño with and without lazy compilation. Figure X.2 shows the percent reduction in compilation time due to lazy compilation in relationship to eager compilation for both the optimizing compiler, shown in the top graph, and baseline compiler, shown in the bottom graph, for the Ref input. The data shows that lazy compilation substantially reduces compilation time for either compiler. On average, for the optimizing compiler, 29% of the compilation overhead is eliminated. Using the baseline compiler, on average 50% is eliminated. Since methods require varying amounts of time for optimization (depending upon method size and complexity), the relationship between the reduction in number of methods compiled and compilation time is not proportional.

Table X.1 provides the raw execution and compilation times with and without lazy compilation using the optimizing compiler for both inputs. The data in this table includes compilation times used in Figure X.2 as well as execution times. Data for the baseline compiler is not shown because compilation overhead is a very small percentage of total execution time, and thus the 50% reduction in compilation time only results in a 1% reduction in total time. Columns 2 through 6 are for the Train input and 7 through 11 are for the Ref input. The sixth and eleventh columns, labeled "Ideal" contain the execution time alone for batch–compiled

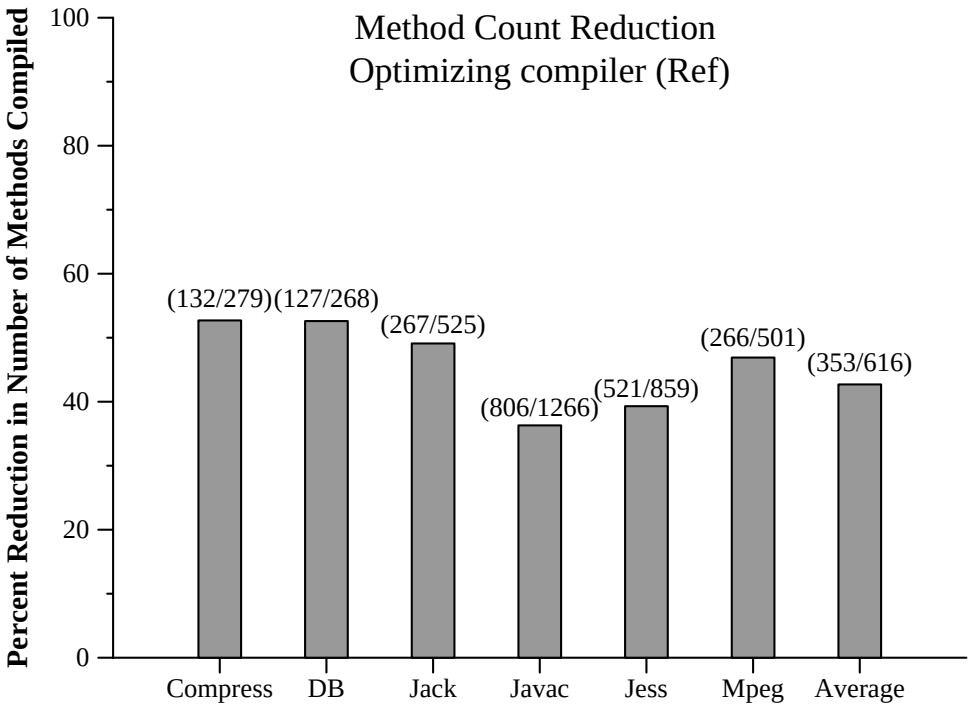


Figure X.1: Percent reduction in methods compiled.
This graph shows the reduction in compiled methods when lazy compilation is used over eager. Above the bars, we include the the number of methods compiled over the total number of methods. We only include data for the Ref input since the number of used methods is similar across inputs for the Spec JVM98 benchmarks. In addition these numbers are typical regardless of which compiler,optimizing or baseline, is used.

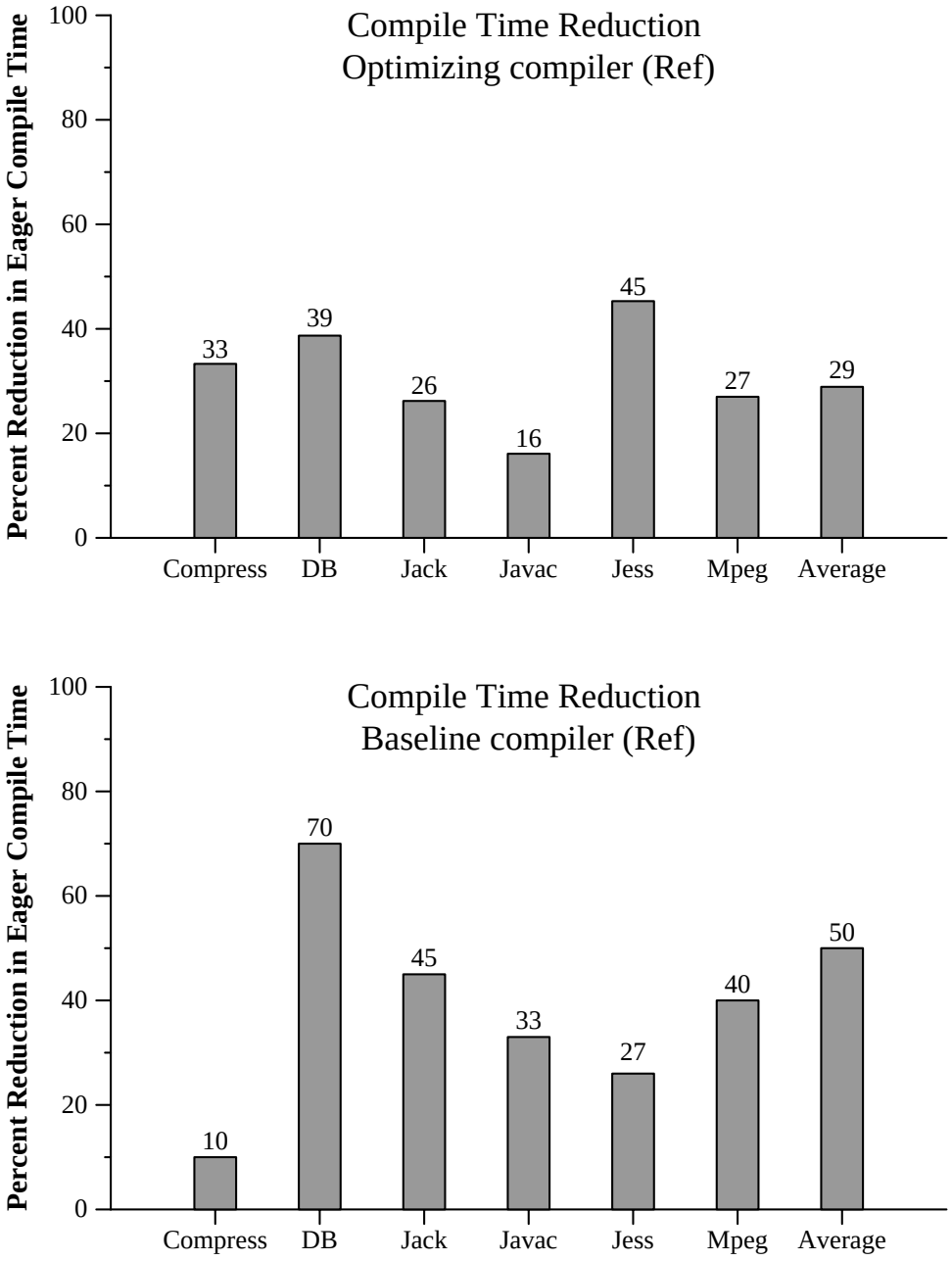


Figure X.2: Reduction in compilation time due to lazy compilation.
The percent reduction in compilation delay is given above each bar explicitly. The top graph shows the reduction in compilation time over eager compilation for the optimizing compiler and the bottom graph shows the reduction for the baseline compiler. Since the results are the same for both inputs so we include only the data data for the Ref input.

Table X.1: Raw execution time data.
This table shows the execution (ET) and compile (CT) times (in seconds) with and without lazy compilation using the optimizing compiler. The sixth and eleventh columns contains the benchmark execution time when the application is batch compiled off-line. Batch compilation (Ideal) eliminates dynamic linking code from the compiled application and enables more effective inlining. Columns 2 through 6 are execution and compile times for the Train input and columns 7 through 11 are for the Ref input. For each input, times for both the eager and lazy approaches are given.

	Train (in seconds)					Ref (in seconds)				
	Eager		Lazy		Ideal	Eager		Lazy		Ideal
Benchmark	ET	CT	ET	CT	ET	ET	CT	ET	CT	ET
Compress	7.4	8.2	5.3	5.4	5.3	84.0	8.1	58.3	5.4	58.3
DB	1.9	8.2	1.9	5.0	1.7	102.7	8.0	98.8	4.9	98.8
Jack	9.9	16.0	9.4	11.6	9.1	84.3	16.0	80.1	11.8	77.6
Javac	2.0	38.6	2.0	31.2	1.9	66.3	38.5	68.1	32.3	62.6
Jess	2.5	27.2	1.8	14.7	1.8	45.2	27.6	38.4	15.1	37.9
Mpeg	7.3	15.9	6.7	11.7	5.4	71.3	15.9	61.7	11.6	51.3
Avg	5.2	19.0	4.5	13.3	4.2	75.6	19.0	67.6	13.5	64.4

applications. *Batch Compilation* is off–line compilation of applications in their entirety. We include this number as a reference to a lower–bound on the execution time of programs given the current implementation of the Jalapeño optimizing compiler. Batch compilation is not restricted by the semantics of dynamic class file loading; information about the entire program can be exploited at compile time. In particular all methods are available for inlining and all offsets are known at compile time.

Columns 2 and 3, and 7 and 8, are the respective execution and compile times for eager compilation. Columns 4 and 5, and 9 and 10, show the same for the lazy approach. In addition to reducing compilation overhead, the data shows that lazy compilation also significantly reduces execution time when compared to eager compilation. This reduction in execution time was caused by the direct and indirect costs of dynamic linking. In the following section, we provide background on dynamic linking and explain the unexpected improvement in optimized execution time enabled by lazy compilation.

The Impact of Dynamic Linking

Generating the compiled code sequences for certain Java bytecodes, e.g., *put-field* or *invokevirtual*, requires that certain key constants, such as the offset of a method in the virtual function table or the offset of a field in an object, be available at compile time. However, due to

dynamic class loading, these constants may be unknown at compile time: this occurs when the method being compiled refers to a method or field of a class that has not yet been loaded. When this happens, the compiler is forced to emit code that when executed, performs any necessary class loading thus making the needed offsets available, and then performs the desired method invocation or field access. Furthermore, if a call site is dynamically linked because the callee method belongs to an unloaded class, optimizations such as inlining cannot be performed. In some cases, this indirect cost of missed optimization opportunities can be quite substantial.

Dynamic linking can also directly impact program performance. A well-known approach for dynamic linking [9, 19] is to introduce a level of indirection by using lookup tables to maintain offset information. This table-based approach is used by the Jalapeño optimizing compiler. When it compiles a dynamically linked site, the optimizing compiler emits a code sequence that, when executed, loads the missing offset from a table maintained by the Jalapeño class loader.[3] The loaded offset is checked for validity; if it is valid it can be used to index into the virtual function table or object to complete the desired operation. If the offset is invalid, then a runtime system routine is invoked to perform the required class loading updating the offset table in the process, and execution resumes at the beginning of the dynamically linked site by re-loading the offset value from the table. The original compiled code is never modified. This scheme is very simple and, perhaps more importantly, avoids the need for self-modifying code that entails complex and expensive synchronization sequences on SMPs with relaxed memory models such as the PowerPC machine used in our experiments. The tradeoff of simplicity is the cost of validity checking: subsequent executions of dynamically linked sites incur a four-instruction overhead. [4]

If dynamically linked sites are expected to be very frequently executed, then this per-execution overhead may be unacceptable. Therefore, an alternative approach based on backpatching, or self-modifying code, can be used [4]. In this scheme, the compiler emits a code sequence that when executed invokes a runtime system routine that performs any necessary class loading, overwrites the dynamically linked sites with the machine code the compiler would have originally emitted if the offsets had been available, and resumes execution with the first instruction of the backpatched, or overwritten, code. With backpatching, there is an extremely high cost (aggravated by the synchronization and memory barriers required on the PowerPC) the first time each dynamically linked site is executed, but the second and all subsequent executions of the site incur no overhead.

[3] All entries in the table are initialized to 0, since in Jalapeño all valid offsets will be non-zero

[4] The four additional instructions executed are two dependent loads, a compare, and a branch.

The Jalapeño optimizing compiler used in this chapter uses the table-based approach. This design decision was mainly driven by the need to support type-accurate garbage collection (GC). As in other systems that support type-accurate GC, compilers must produce mapping information at each *GC-safe point* detailing which registers and stack-frame offsets contain pointers. By definition, all program points at which an allocation may occur, either directly or indirectly, must be GC-safe points, since the allocation may trigger a GC. Because allocation will occur during class loading, all dynamically linked sites must also be GC-safe points. If the optimizing compiler used backpatching, it would actually need to generate two GC-maps for each dynamically linked site: one that described the initial code sequence and one that described the backpatched code. Although the two maps would contain very similar information, both are needed since the GC-safe point in the initial and backpatched code sequences are at different offsets in the machine code array. In practice, it turned out to be burdensome to modify the optimizing compiler's GC-map generation module to produce multiple maps for a single intermediate language instruction, so the issue was avoided by using the table-based approach which only requires one GC-map for a dynamically linked site.

Since class files are not changed once loaded, we are able to increase the probability that an accessed class will be resolved at the time the referring method is compiled with the delayed compilation of the lazy approach. Table X.2 shows the number of times dynamically linked sites are executed with eager and lazy compilation. On average, code compiled lazily executes through dynamically linked sites 92% fewer times than eager compilation for the Train input and 99% fewer times for the Ref input. Although the reduction in direct dynamic linking overhead can be quite substantial, e.g. roughly 25 million executed instructions on `compress` with the Ref input, the missed inlining opportunities are even more important. For example, more than 99% of the executed dynamically linked sites in the eager version of `compress` are calls to very small methods that are inlined in the lazy version. Thus, the bulk of the 25 second reduction in `compress` execution time as shown in Table X.1 is due to the direct and indirect benefits of inlining, and not only to the elimination of the direct dynamic linking overhead. Similar inlining benefits also occur in `mpegaudio`.

The effect of lazy compilation on total time is summarized in Figure X.3. The graph shows the relative effect by lazy compilation both on execution time as well as compilation time using the optimizing compiler. The top graph is for the Train input and the bottom graph is for the Ref input. The top, dark-colored portion of each bar represents compilation time, the bottom light-colored portion represents execution time. A pair of bi-colored bars is given for

Table X.2: Dynamic execution count of dynamically linked sites.
Columns 2–4 are for the Train input and 5–7 are for the Ref input. Columns 2 and 5 give the counts in 100,000's of executed sites that were dynamically linked using the optimizing compiler. Columns 3 and 6 are the counts when lazy compilation is used and Columns 4 and 7 show the percent reduction.

	Train			Ref		
	x 100,000		Percent	x 100,000		Percent
Benchmark	Eager	Lazy	Reduced	Eager	Lazy	Reduced
Compress	492	3	99	6202	3	100
DB	12	3	75	455	4	99
Jack	32	28	13	71	51	28
Javac	27	17	37	480	33	93
Jess	64	7	89	790	8	99
Mpeg	133	5	96	1547	6	100
Avg	127	11	92	1591	18	99

each benchmark. The first bar of the pair results from using the eager approach; the second bar from lazy compilation. Lazy compilation reduces both compilation and execution time significantly when compared to eager compilation. On average, lazy compilation reduces total time by 26% for the Train input and 14% for the Ref input. Execution time alone is reduced by 13% and 11% on average for each input, respectively, since lazy compilation greatly reduces both indirect and direct costs of dynamic linking.

X.A.3 Background Compilation

In this section, we describe background compilation, a technique that reduces compilation delay by overlapping compilation with computation. With lazy compilation, each method is compiled upon initial invocation. However, the execution characteristics of the method may not warrant its, possibly expensive, optimization. In addition, this on–demand compilation in an interactive environment may lead to inefficiency. In environments characterized by user interaction, the CPU often remains idle waiting for user input. Furthermore, the future availability of systems built using single-chip SMPs makes it even more likely that idle CPU cycles will intermittently be available. The goal of background compilation is to extend lazy compilation to further mask compilation delay by using idle cycles to perform optimization.

Background compilation consists of two parts. The first occurs during application execution: when a method is first invoked, it is lazily compiled using a fast, non–optimizing compiler or the method is interpreted. This allows the method to begin executing as soon as

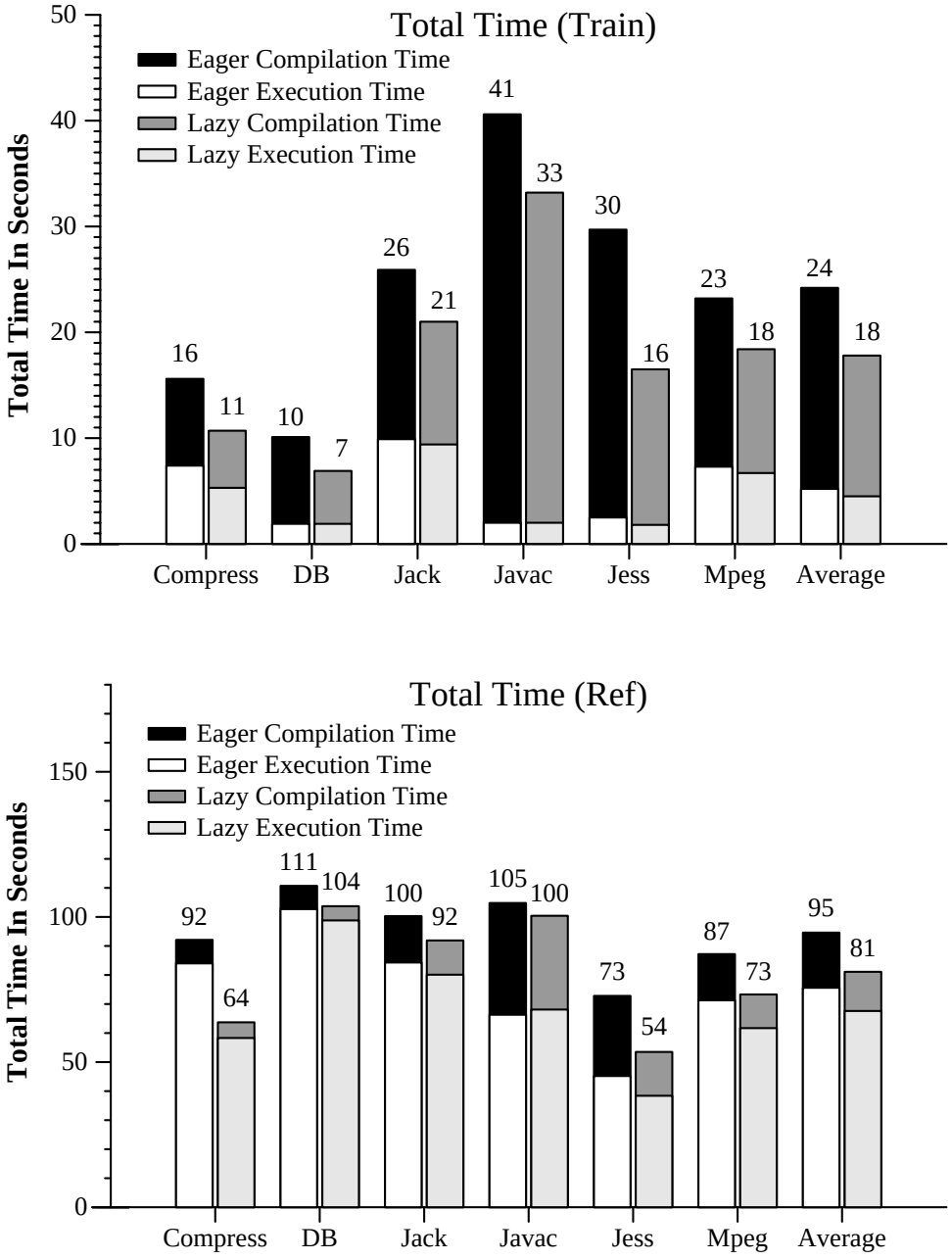


Figure X.3: Overall impact of lazy compilation application performance.
The top graph is for the Train input and the bottom graph is for the Ref input. The left bar of each pair results from using eager compilation, the right bar lazy. The top, dark colored, portion of each bar is compilation time, the bottom, light-colored execution. The number above each bar is the total time in seconds required for both execution and compilation time. Lazy compilation reduces both execution time as well as compilation time.

possible. However, since this type of compilation can result in poor execution performance, methods which are vital to overall application performance should be optimized as soon as possible.

This is achieved with the second part of the background compilation by using an *Optimizing Compiler Thread* (OCT). At startup we initiate a system thread that is used solely for optimizing methods. The OCT is presented with a list of methods that are predicted to be the most important methods to optimize. The OCT processes one method at a time, checking whether or not the class in which it is defined has been loaded. If it has, then the method is optimized. Once compiled, the code returned from the optimizing compiler is used to replace the baseline compiled code or the lazy compilation stub if the method has not yet been baseline compiled. Future invocations of this method will then use the optimized version. If existing stack frames reference previously compiled code, then this code will be used until the referenced invocation returns.

To predict which methods should be optimized by the OCT, we use profiles of the execution time spent in a method. To generate these profiles for our experimental results, we execute the application off-line and accumulate measurements of the amount of time spent in a method, each time it is executed. We gather this timing data for executions using two inputs as described in Chapter IV.

At JVM startup, the profiled list of methods and the time spent in each is read into memory and processed. Each method is assigned a global priority ranking with respect to all other methods executed by the application. We then record the global priorities with the methods in each class. As each class is loaded, any methods of the class that have been prioritized are inserted into the OCTs priority queue for eventual optimization. If the priority queue becomes empty, the OCT sleeps until class loading causes new methods to be added.

This model extends to a dynamic, mobile environment in which class files may be uploaded into the Jalapeño server from different sources. At each source, profiles are generated and methods within each class are prioritized prior to transfer, as described above. Often, execution of an application accesses library files that are not transferred as part of the application but are dynamically loaded from the machine on which the program is executed. Information about high priority methods in these classes, are sent to the destination with the application in the form of annotations. Annotation is a mechanism for including additional information in a class file. For background compilation, each application method contains an annotation consisting of its priority as well as the name and priority of any library, or other

non-transferred methods. When a class is loaded by a Jalapeño server, the annotated priority for important methods guides insertion into the global priority queue of the OCT. Bytecode annotations of method priorities are inserted into the bytecode as method attributes using a bytecode re-writing tool.

We currently use a single OCT that synchronously compiles prioritized methods. When uncompiled methods are invoked, they are compiled using the baseline compiler. We may be able to gain additional performance benefits through the use of multiple OCTs once the Jalapeño optimizing compiler is made re-entrant. In this case, the baseline compiler will still be used to compile newly-invoked methods so that optimization decisions are made solely by the OCT system. We estimate that having multiple OCTs will provide additional performance benefits in certain cases. For example, currently the OCT uses one processor separate from the one used by the application thread. If there are additional processors or idle cycles, more compilation can be performed using multiple OCTs. The system should be adaptive however, so that the application is not starved for resources. That is, when the application is in need of processing cycles, the OCT activity should be reduced so as to maintain acceptable application performance while continuing to compile high-priority methods, even if it must scale back to a single thread. This implementation and the associated analysis to achieve a beneficial performance balance is part of future work.

X.B Results: Lazy and Background Compilation

In this section *total time* refers to the combination of compilation, execution, and all other overheads. The total time associated with background compilation includes:

- Baseline, or fast, compilation time of executed methods
- Execution time from methods with baseline-compiled code
- Execution time from methods invoked following code replacement by the optimizing background thread, and
- Thread management overhead

The examples in Figure X.4 illustrate the components that must be measured as part of total time for a different scenarios involving a method, Method1. In the first scenario, Method1 is invoked, baseline compiled, and executed. Following its initial execution the OCT encounters Method1 in its list and optimizes it. By the time it is able to replace Method1's baseline compiled code, Method1 has executed a second time. For the third invocation, however,

the OCT has replaced the baseline compiled code and Method1 executes using optimized code. Total time for this scenario includes baseline compilation time of Method1 and execution time for two Method1 invocations using baseline compiled code and one using optimized code.

In the second scenario, the OCT encounters, optimizes, and replaces Method1 before it is first invoked. This implies that the class containing Method1 has been loaded prior to OCT optimization of Method1. The OCT replaces a stub that is in place for Method1 with the optimized code. When this occurs the use of background compilation can also reduce the memory footprint of the Jalapeño VM and the executing program since baseline code is not kept in memory. All executions of Method1 use the optimized code. Total time for this scenario includes only the execution time for three invocations of Method1 using optimized code.

To measure the effectiveness of background compilation, we provide results for the total time required for execution and compilation using this approach. Figures X.5 and X.6 compare total time with background compilation to total time for the eager, lazy, and ideal configurations results from Table X.1 (Ref and Train, respectively). Four bars (with absolute total time in seconds above each bar) represent the total time required for each approach for a given benchmark. The first bar shows results from eager and the second bar from the lazy approach. The third bar is the total time using background compilation and the fourth bar is "ideal" execution time alone. Ideal execution time results from a batch-compiled application (complete information about the application enables more effective optimization and removes all dynamic linking, and there is no compilation cost).

The summary figures show that background compilation eliminates the effect of almost all of the compilation delay that remains when using the lazy approach. On average, background compilation provides an additional 71% average reduction in total time over lazy compilation for the Train input (14% for the Ref input). On average there are 151 fewer methods optimized by the OCT over lazy compilation. In comparison with eager compilation, background compilation reduces the total time (execution plus compilation) by 79% and 26% for the Train and Ref input, respectively. The percentage of total time due to compilation is 79% and 20%; hence background compilation reduces total time by more than just the compilation delay. This occurs since background compilation extends lazy compilation and thereby enables additional optimization and avoids the dynamic linking effects (as discussed in the lazy compilation section). That is, when the OCT optimizes each method, most required symbols are resolved.

Most importantly, however, are the similarities between background and "ideal" execution time. Total time using the background approach is within 21% and 8% (on average for

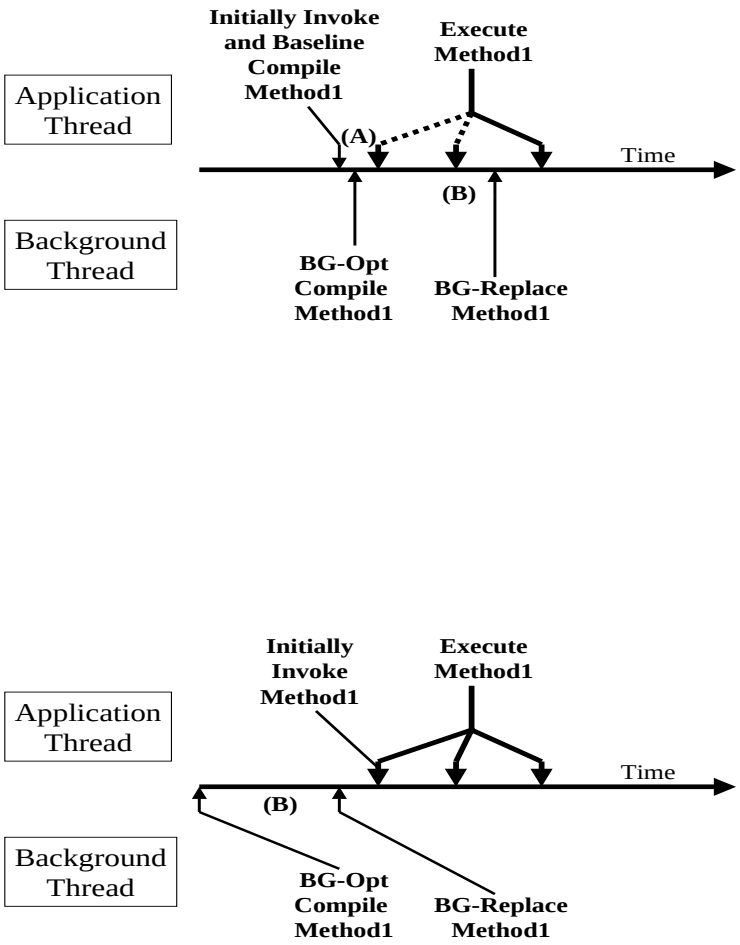


Figure X.4: Example scenarios of background compilation.
In the first scenario, upon initial invocation of Method1, execution suspends and Method1 is baseline-compiled. "(A)" in each figure represents the time required for baseline compilation. When Method1 is executed the code invoked is the baseline-compiled version. This is represented in all figures by the dotted arrow. Next, in the background, as indicated below each timeline, the optimizing compilation thread (OCT) optimizes Method1. "(B)" in each figure represents the time required for optimization. Due to the time required for Method1 optimization, Method1 is invoked and executed a second time with the baseline-compiled code before the OCT replaces the baseline-compiled code with the optimized version. Once replaced, Method1 executes using the optimized version of the code. This is represented by a solid line in the figure. In the second scenario, the OCT is able to compile and replace Method1 before any invocations of Method1 occur; therefore, all executions use the optimized code.

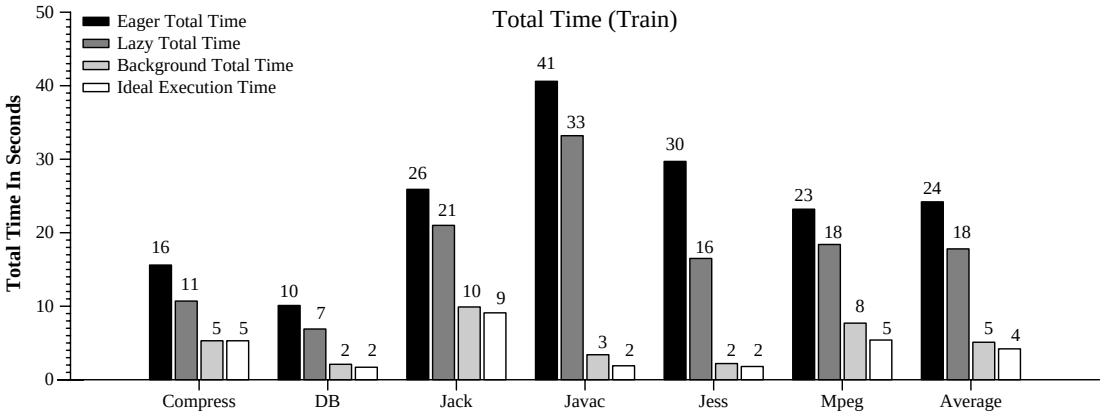


Figure X.5: Summary of total time (in seconds) for the Train input.
Times for all of the presented approaches including background compilation are shown for the Train input. Total time includes both compilation and execution time. Four bars are given for each input. The first three bars show total time using eager compilation, lazy compilation, and background compilation, respectively. The fourth bar shows "ideal" execution time alone (from execution of off-line compiled benchmarks). Absolute total time in seconds appears above each bar.

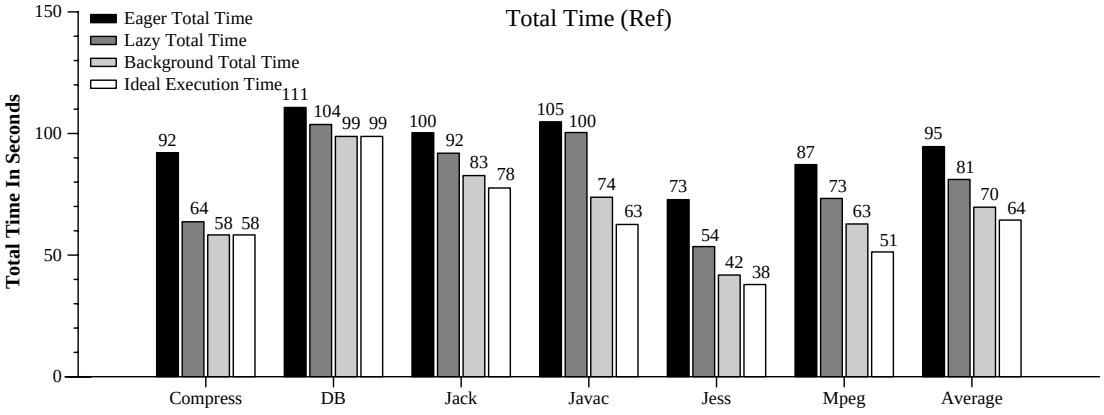


Figure X.6: Summary of total time (in seconds) for the Ref input.
Times for all approaches including background compilation are shown for the Ref input. Total time includes both compilation and execution time. Four bars are given for each input. The first three bars show total time using eager compilation, lazy compilation, and background compilation, respectively. The fourth bar shows "ideal" execution time alone (from execution of off-line compiled benchmarks). Absolute total time in seconds appears above each bar.

the Train and Ref inputs, respectively) of the ideal execution time. Our background compilation approach therefore, correctly identifies performance-critical methods and achieves highly optimized execution times while masking almost all compilation delay.

X.C Summary

The infrastructure we use to examine the impact of our compilation strategies introduced in this chapter is the Jalapeño Virtual Machine, a compile–only execution environment being developed at IBM T. J. Watson Research Center. Currently in Jalapeño, two compilers are used, the fast baseline compiler that produces code with execution speeds of interpreted versions, and the optimizing compiler, a slow but highly optimizing compiler that produces code with execution speeds two to eight times faster than the code produced by the baseline compiler. Our goal was to design and implement optimizations that enable compilation times of the baseline compiler and execution speeds of optimized code.

We first empirically quantify the effect of lazy compilation on both compilation time and execution time. We show that lazy compilation requires 57% fewer methods be compiled on average than eager compilation for each input of the benchmarks studied. In terms of compilation time, this equates to approximately 30% reduction on average for either input, since the number of methods used between inputs is relatively the same. In addition to reducing compilation delay, lazy compilation also improves execution time by greatly reducing the number of dynamically linked sites, thus avoiding both the direct costs of dynamic linking and the indirect costs of missed optimization opportunities. Lazy compilation reduces optimized execution time 13% and 10% on average for the Train and Ref input, respectively. In terms of total time, lazy compilation enables a 26% and 14% reduction over eager compilation using the optimizing compiler. Jalapeño, as a result of this work, uses lazy compilation by default.

We also present a compilation approach that extends lazy compilation. Background compilation masks the delay incurred by compilation by overlapping it with useful work. With this optimization, we use the Jalapeño optimizing compiler on a background thread to compile only those methods we predict as important for optimization. On the primary thread(s) of execution, the Jalapeño baseline compiler is used so that methods can begin executing much earlier than if they are optimized. The background thread then replaces the baseline compiled method with an optimized version so that future invocations of the method call the optimized version. Our results show that background compilation achieves execution times of optimized

code with compilation delay of baseline compilation. On average, background compilation effectively reduces total time by 79% and 26% for the Train and Ref input, respectively. When compared to lazy compilation, the background optimization reduces total time of 71% for the Train input and 14% for the Ref input. We also show that background compilation achieves the runtime performance of applications that are batch compiled, i.e. off–line optimization of the entire application at once.

The text of this chapter is in part a reprint of the material as it appears in the Journal of Software: Practice and Experience, Software: Practice and Experience, Volume 31, Issue 8, pp. 717-738, Dec. 2000. The dissertation author was the primary researcher and author and the co-authors listed on this publication directed and supervised the research which forms the basis for this chapter.

Chapter XI

Compilation Delay Avoidance and Overlap: Annotation-guided Compilation

As articulated in the previous two chapters, dynamic optimization of a program can cause significant delays during execution. Most systems attempt to reduce this delay by incorporating multiple compilers [12, 16] or a compiler and interpreter [34, 68] into a single execution environment. Using such systems, a program is compiled first with the fast compiler, and then frequently executed methods are compiled later with the optimizing compiler based on dynamic information gathered during execution. This ensures that compile time is expended on frequently executed, or "hot", methods only and compilation overhead is reduced.

In the previous chapter, extend one such dual-compiler system called the Jalapeño virtual machine [3]. We introduce a technique called *background compilation* which uses off-line profile information to guide hot method selection. This eliminates the need for on-line profiling and instrumentation. However, we provide no automatic mechanism for the communication of this information to compilation system. In this chapter, we present such a mechanism that introduces annotation into the bytecode stream to communicate compilation analysis as well as off-line profile information. The goal of our research is to minimize the overhead introduced by dynamic compilation while achieving optimized execution speeds.

Existing annotation-based techniques annotate Java bytecode with analysis information that is time-consuming to collect to guide dynamic compilation [6, 42, 69, 29]. The goal of this prior work was to make costly optimizations feasible in dynamic compilation settings. In this chapter, we extend annotation-based compilation and optimization (1) to provide a general annotation representation to guide dynamic compilation, (2) to examine the effects

of using annotations to reduce the startup delay and intermittent interruption caused by dynamic optimization, (3) to examine new profile-based annotations to guide optimization, and (4) to generate annotations that do not increase the application transfer size. The latter is very important if annotated-execution is to be used in a mobile environment. If the size of the annotations are not very small, they can introduce significant transfer delay which can negate any benefit achieved through the use of the annotations. Since we intend for our annotation optimizations to be used in a mobile environment, we ensure that they not only improve performance at runtime but do not introduce transfer overhead. A primary contribution of this work is the implementation of annotations that increase the size of annotated applications by less than 0.05% on average.

Another contribution the work in this chapter makes is the reduction in program startup time. We have found that, like transfer delay, most of the dynamic compilation for Java programs occurs at program startup. In the programs studied, 77% of of the compilation overhead occurs in the first 4 seconds (initial 10%) of program execution on average. The application of our techniques reduces startup delay by more than 2 seconds in many cases which enables significantly more progress to be made by the programs. Startup delay has been the focus of much past research since it substantially effects a user's productivity and perception of program performance [21, 78]. Using annotations extends and compliments these and many other efforts [74, 53] to substantially reduce the startup time of mobile programs.

XI.A Design and Implementation

A compiler annotation is additional information attached to program code and data to help guide optimization. Annotations have been widely used on program source code in various languages to exploit parallelization and optimization opportunities in parallel and distributed codes. More recently, annotation-based techniques have focused on communicating information that aids optimization, but is too time-consuming to collect on-line [6, 42, 69, 29]. The goal of these efforts has been to make costly optimizations feasible in dynamic compilation settings.

We extend annotation-based compilation and optimization to provide a general annotation representation that minimizes the number of bytes used to represent the annotation. To this end, we incorporate compression into our framework and ensure that all annotations implemented impose very little space overhead in the program bytecode stream. In addition, we examine using new static and profile-based optimization techniques to guide dynamic compilation.

For this research, we use an open-source, dual-compiler system called the Open Runtime Platform (ORP), which was recently released by the Intel Corporation [65]. The first compiler (O1) provides very fast translation of Java programs [1] and incorporates a few very basic bytecode optimizations that improve execution performance. The second (O3) compiler performs a small number of commonly used optimizations on bytecode and an intermediate form to produce improved code quality and execution time. O3 optimization algorithms were implemented with compilation overhead in mind, hence only very efficient algorithms are used [16]. The execution and compilation times for a number of benchmarks compiled using the ORP compilers is shown in Table IV.7 in the methodology chapter (Chapter IV). For comparison, O3 execution time is 8% faster than O1 execution time on average and the compilation time of the O3 compiler is 89% slower than that for O1 on average for the programs studied.

We next consider where ORP optimization time is spent in the O3 compiler. Figure XI.1 gives a breakdown of where time is spent during the different O3 compilation phases. We use these results along with the speedups resulting from the different optimizations and their combinations in order to determine which optimizations might benefit from the use of annotation-guided optimization. The y-axis is time in seconds; the average O3 compile time for the benchmarks is approximately 2.7 seconds. The bar for each application is broken down into eight pieces. *Other* denotes memory allocation of data structures and any other code transformation costing less that 100 milliseconds. *Const-prop* is constant and copy propagation. *Global-reg* is global register allocation, i.e., the time to allocate physical registers to the local variables of a method. *Build-ir* is the intermediate form translation time; the bytecode of each method is converted to a lower-level form for further optimization. *DCE* is dead code elimination. *Local-reg* is local register allocation, i.e., the time to allocate physical registers to temporary variables required by the translation. *Fg-create* is the time for flow-graph construction, and *loop-opts* is the time for loop optimization.

XI.A.1 Framework

Our framework incorporates a bytecode rewriting tool called BIT [55] with which we insert annotations into a Java program. Annotations are included in class files and are transferred as part of the bytecode stream when remotely executed. Annotations are stored in a bytecode data structure called an *attribute* as defined in the Java language specification [28]. An attribute data structure is defined for class files as well as for the methods and fields contained in the class. For example, a `Code` attribute is defined in the Java language specification as

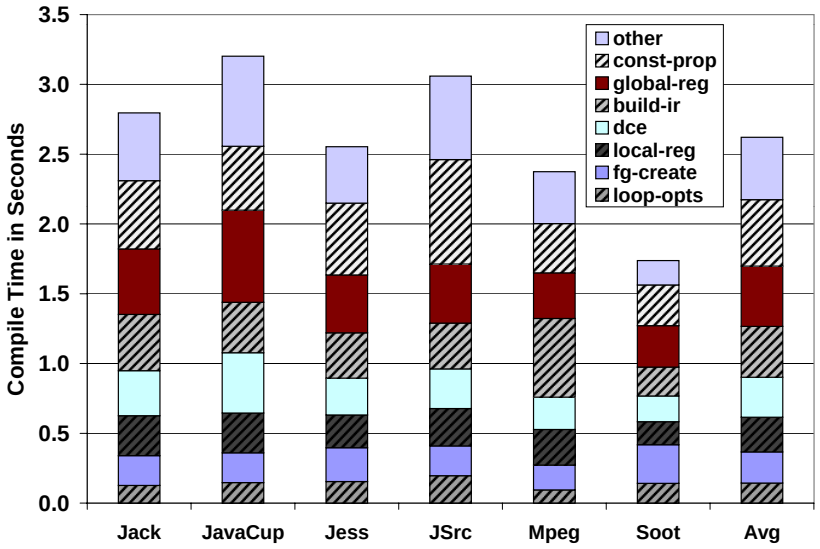


Figure XI.1: ORP O3 (Optimizing) Compilation Time Breakdown.
The y-axis is time in seconds. *Other* denotes memory allocation of data structures and any other code transformation costing less that 100 milliseconds. *Const-prop* is constant and copy propagation. *Global-reg* is global register allocation, i.e., the time to allocate physical registers to the local variables of a method. *Build-ir* is the intermediate form translation time; the bytecode of each method is converted to a lower-level form for further optimization. *DCE* is dead code elimination. *Local-reg* is local register allocation, i.e., the time to allocate physical registers to temporary variables required by the translation. *Fg-create* is the time for flow-graph construction, and *loop-opts* is the time for loop optimization.

a method-level attribute and contains the actual bytecode. The *name* of each attribute at any level (class, method, field), is included in the constant pool of the class file and is used to distinguish, parse, and make use of the attribute. When a virtual machine encounters an undefined attribute it is required to ignore it. This makes attributes ideal for the storage of annotations since it allows annotated class files to remain compatible with all JVMs that are not annotation-aware.

In this work, we add a *single*, user-defined, class attribute for annotations. We combine multiple annotations into the same attribute and use a single character of Unicode [28] to represent the name of the attribute in the constant pool. These two design decisions minimize the size increase of a class file required for annotations. To further reduce annotation size, annotations within each attribute are compressed using gzip compression. Gzip is a standard compression utility, commonly used on UNIX operating system platforms. These decisions also distinguish our framework from prior research in this area (see section III).

In our design, annotations of variable length appear sequentially in a given attribute. The encoding we chose for our annotation language is very similar to an instruction set architecture (ISA) format for a variable length ISA. The general format of an annotation attribute is a series of triples of the form: $< opcode, size, data >$. The opcode tells the compiler how to parse and make use of the annotation. In addition, since there are possibly many annotations in a single attribute, the compiler must be able to determine where one starts and the next begins. This is done by including the size of the attribute after the opcode. The annotation data then appears right after the opcode and size. The elements of each annotation are summarized as:

- *opcode*: (2 bytes) The identifier of this annotation that tells the compilation system how to parse and make use of the following annotation. The end of the attribute section, and thus all annotations, is indicated by a 0 opcode.
- *size*: (2 bytes) The number of bytes for the annotation data that follows. The maximum size of an annotation is 64KB.
- *data*: (variable number of bytes) The annotated information.

To incorporate the annotations, the compiler decompresses the annotations contained in the attribute (using the gzip compression library) and reads the opcode and size elements of the first triple (6 bytes total). It then looks up the opcode to determine the use of the annotation. The annotation is parsed and placed in the appropriate data structure in memory or processed directly. For annotations that are specific to a method within the class, the first two bytes of data contain a method identifier. Annotations can also be used across all methods in a class file; for these no method identifier is needed. This parsing procedure is repeated for

the next annotation until the end of the attribute is reached. We use a 1-byte opcode of 0 to delimit the end of an annotation stream.

XI.A.2 Annotation Optimizations

A goal of this research is to reduce compilation overhead while maintaining optimized execution time. The annotations we describe next are meant to achieve this goal for the ORP compiler. As shown previously in Figure XI.1, the dynamic optimization time in ORP is spread across multiple operations. To reduce time spent overall, we consider annotations that effect those phases that are the largest contributors to total compilation overhead.

We examine using annotations of both static and profile-based analysis to enable efficient, dynamic optimization of methods. Static information is structural and syntactic information explicitly available in Java bytecode and class files. Profile-based information consists of runtime program characteristics and is collected by instrumenting and executing the programs off-line. We present four types of annotations and describe an implementation of each: those that provide static analysis information, those that enable optimization reuse, those that enable selective optimization, and those that enable optimization filtering. The name of each specific annotation we will provide results for is shown in parentheses at the start of each paragraph describing the annotation.

Provision of Static Analysis Information

All compilers collect static information about the code they are compiling to perform translation, transformations, and optimization. For example, information about local variables, control flow, exception handling, etc., may be collected for an optimization by scanning the code. If the collection of data analysis can be performed independent of its use, the analysis and acquisition of it can be performed off-line. We first present annotations that communicate such analysis information to the compilation system in an efficient format. Since the analysis is performed off-line, dynamic compilation overhead is reduced.

Global Register Allocation Annotation (global-reg). The Java bytecode format is base upon a stack architecture and hence, it is difficult to achieve acceptable performance of Java programs on register-based architectures without complex analysis and algorithms for register allocation. Many commonly used allocation routines prioritize variables in order to apply more advanced algorithms for the assignment of registers, e.g., prioritized graph-coloring. In ORP,

priorities are determined by static counts of local variable uses in the bytecode. Counting is performed by walking through each method. To avoid this bytecode scan, we use annotations to indicate the priorities. In addition, more advanced prioritization (via profiling or static heuristics) can be used to improve register allocation in ORP, but this is left for future work.

For this global register allocation annotation, we make static counts of local variable usage just like those made in ORP and communicate this information via the annotation. The data element of the annotation triple consists of two bytes to indicate the method and one byte for each local variable. The bytes are arranged in the same order as are the local variables in the local variable array [28]. In the programs studied, the maximum number of local variables used by a single program is 1719 (JSrc); the maximum for any single method is 31 (Mpeg). The average number of locals per method is 2.5. These numbers include methods in the system class libraries also. For non-local class files, the maximum number of local variables used by a program is 1247 (JSrc) with an average method use of 2.6 variables.

Flow Graph Generation Annotation (fg-create). A flow graph is a data structure commonly used by compilers to identify changes in program control flow for effective and correct optimization. Most Java compilers generate a flow graph for every method to find basic block boundaries and other pertinent control flow information [12, 16, 44]. This construction requires multiple passes of the Java bytecode. To reduce the time required for such passes we implemented a flow graph generation annotation using our framework. We characterize the control flow structure of each method and use an annotation to present it to the compiler for single pass flow graph construction.

We construct this annotation for each method to enable automatic generation of the flow graph without the prepass operation. As with all other optimizations, fg-create is implemented with a single annotation per method (using a single opcode). The annotation is a list of the basic blocks in the method. The annotation begins with a two-byte method identifier (id) followed by the count of the total number of basic blocks that follow. Each basic block representation includes the block id, its (annotation) size, id numbers of the predecessor and successor blocks, start and ending bytecode indices, and other special information (loop header flag, exception handling block, etc), if any. The annotation enables construction of an ORP flow graph for a method with a single scan of the annotation. Across all benchmarks studied there are just under 14000 basic blocks and each method requires 4.2 blocks on average.

Optimization Memory Reuse

The goal behind the implementation of this next annotation is to enable reuse of analysis information required for multiple optimizations during execution. Most dynamic compilers regenerate information about a method compilation instead of storing it for possible reuse. Since a compiler is unable to predict which analysis information will be reused by future phases of optimization, it must store all of the information or repeatedly regenerate it. Regeneration is performed since storage of the analysis information can substantially increase the size of the memory footprint of the execution. Some compiler stages may also modify analysis information requiring additional copies to be stored for reuse. Annotations can be used to indicate which data it is cost effective for the compiler to store for reuse. The annotation optimization we implement is for the reuse of inlining information in ORP.

Inlining (inlining) Annotation. Inlining is a common optimization used by all compilers to reduce method or function call overhead. By inlining a method call, the call and return are removed, the inlined method code becomes part of the method which contained the call, and that code is optimized along with the rest of the code in the method during optimization. Commonly in dynamic compilation systems [12, 16], when a method is inlined into another, its (control) flow graph is generated, processed, and possibly optimized prior to insertion into the method it is inlined into. If this method is later inlined into a different method, the process of flow graph creation and optimization is repeated. For methods that are inlined many times, much redundant work is performed by the compiler. It is not desirable to keep all flow graphs in memory in case of reuse, since it can dramatically increase the size of the memory footprint unnecessarily. We therefore, analyzed off-line profiles to determine which executed methods might be inlined multiple times.

For this optimization, we include one annotation per method, the annotation contains the method identifier and a single bit of information as the data element. When this bit is set, it indicates to the compiler that the optimized flow graph should be stored in memory for reuse. When unset, the bit indicates that the flow graph should not be stored but generated each time.

Selective Optimization

We next present selective optimization annotations that determines if a function should be optimized or not, or selects among the existing optimizing compilers available on

a system. In ORP, the O1 fast compiler or the O3 optimizing compiler can be used. For this type of annotation we identify the most important methods to optimize and indicate to the compilation system that all other can be fast-compiled. This annotation can potentially reduce compilation overhead since methods that are fast-compiled are more prevalent. It also enables methods that are most frequently executed (and hence, important for the overall execution performance) to be optimized.

Method Priority Annotation (top25%). For this annotation, we use off-line profile data to predict the methods that should be optimized. This is similar to the function of an adaptive compilation system in which methods are first compiled with a very fast, non-optimizing compiler, then optimized when deemed *hot*. Hot-ness is identified using analysis of on-line profiles enabled by method instrumentation. With this annotation, we indicate whether a method is hot using off-line profiles obviating the need for on-line instrumentation and profiling. The annotation indicates whether a method should be compiled using the fast O1 compiler or optimized. To determine the percentage of methods that are important to optimize we gathered execution times for the histogram shown in Figure XI.2. The graph shows the total time (execution plus compilation). For each benchmark, each bar (within a set of nine) indicates the total time given optimization of some percentage of the most frequently executed methods. For example the 0% bar (left-most in each set) shows the total time when 0% of the methods are O3-compiled (optimized) and 100% of the methods are O1-compiled (fast). The 100% bar (right-most of each set) shows the total time when 100% of the methods are O3-compiled (optimized) and 0% of the methods are O1-compiled (fast). The remaining bars represent the total time for various percentages between 0 and 100.

When 0% of the of the methods are O3-compiled (all of the methods are O1-compiled), the total time (compilation plus execution) is dominated by the execution time. O1-compilation time is very small (0.3 seconds for the entire application on average) but since no optimizations are performed, execution time is slow (38 seconds on average). At the far right of the spectrum (right-hand bars of each set) in which all methods are O3-compiled, compilation time is very high (2.6 seconds on average) and optimization enables improved execution time (33.8 seconds on average). We generated this histogram to discover the balance between these two extremes; the point at which both execution and compilation time are at their minimum.

The figure shows that the top 25% of frequently executed methods should be optimized to achieve the minimum compilation time as well as execution time. We used profile data to

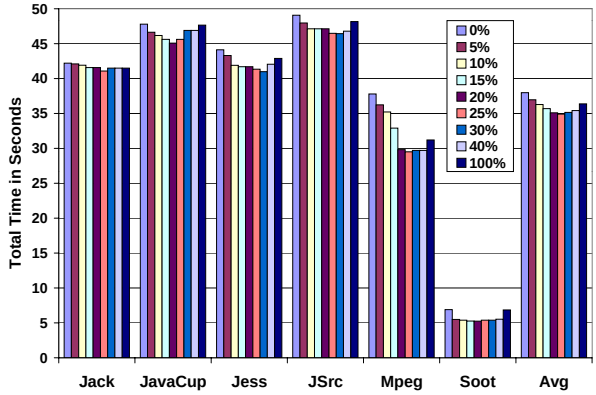


Figure XI.2: The histogram used to find the "Hot" methods important for optimization. Each bar shows the total time (compilation plus execution) for the program when different percentages of the most frequently executed methods are optimized. The remaining methods are compiled with the O1 compiler. The top 25% of most frequently executed methods should be optimized on average for the best performance.

determine which methods were contained in the top 25% in terms of invocation frequency. The annotation for selective optimization is similar to that for the reuse optimization: it contains a one-byte method id and sets a single bit for each method in the top 25% most frequently executed methods. The bit indicates to the compiler that the method should be optimized.

Annotations for Optimization Filtering

The final type of annotation we describe is used for optimization filtering. Currently an optimizing compiler performs all available optimizations on a method. To reduce compilation overhead, it may be beneficial to perform a subset of the optimizations when static or profile-based analysis of the method indicates that it is profitable to do so.

We construct an annotation for each method that consists of a two-byte method identifier and a 1-byte bit mask as part of the annotation data for each method that maps to a list of expensive optimizations. For example, the first bit might map to inlining, the second to register allocation, the third to constant propagation, and so on. Each bit that is set in the mask indicates to the compiler that the associated optimization should *not* be performed for that method. This annotation has the potential for reducing compilation overhead by filtering time-consuming optimizations that do not result in substantially improved execution performance on a per method level. We are able to determine which optimizations improve

performance of methods using profiling.

Constant Propagation Filtering (const-prop). In Figure XI.1, the ORP O3 compiler spends a substantial amount of time performing constant propagation. For some methods, the reduction in runtime performance after applying this optimization is not enough to warrant the use of the optimization. For those methods, we use the bit mask of this annotation to indicate that the copy propagation optimization should be excluded.

We gathered total times (execution plus compilation) for the benchmarks for which constant propagation was used on various percentages of the most frequently executed methods. We created a histogram of these values like we did for the selective optimization annotation and discovered that 70% of the methods benefit from using constant propagation on average. For all other methods, it is not cost effective in terms of execution time to warrant using constant propagation. For each of these latter methods, we include an optimization filtering annotation that has the constant propagation bit set. This indicates to the compilation system that constant propagation should be bypassed during optimization of the method.

XI.A.3 Security of Annotations

An important issue that must be addressed by annotation-based systems for mobile-computing environments is security. Annotations must be verified or guaranteed that their use will not corrupt the JVM or machine on which they are used. Use of most existing bytecode annotation systems poses serious security risks since the annotations implemented using these systems affect program semantics and no verification mechanism is provided. If the bytecode stream is intercepted and modified by an untrusted party, illegal and possibly destructive program behavior can result.

The annotations we present here with the exception of the flow-graph generation optimization, if modified with harmful intent, can only effect program performance. As part of the empirical evaluation of our techniques, we measure the effect of the optimizations in a mobile environment for which only remote (non-library) classes are annotated. We do not include the flow-graph optimization in that part of the study (Section XI.B.2) to guarantee that untrusted execution using our annotations are safe. Benefits from our secure annotations are two-fold: their modification does not effect program semantics and they do not require the additional runtime overhead of verification. The latter is significant for our work, since our goal is to eliminate (not introduce) as much runtime overhead as possible.

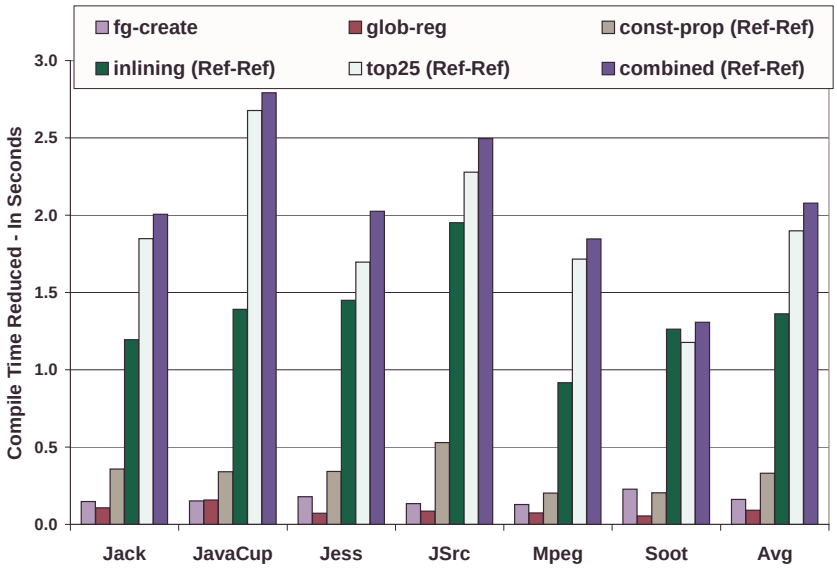
XI.B Results: Annotation-guided Compilation

Three of the optimizations we present (const_prop, inlining, and top25%) require profile information for annotation construction as described in the previous section. Since compilation overhead (method use) is dependent upon program inputs, we gather execution profiles using two different inputs, called Ref and Train, as described in the methodology chapter (Chapter IV). Ref is used to generate all of the results in this section as well as the compilation time statistics shown in the methodology chapter in Table IV.7.

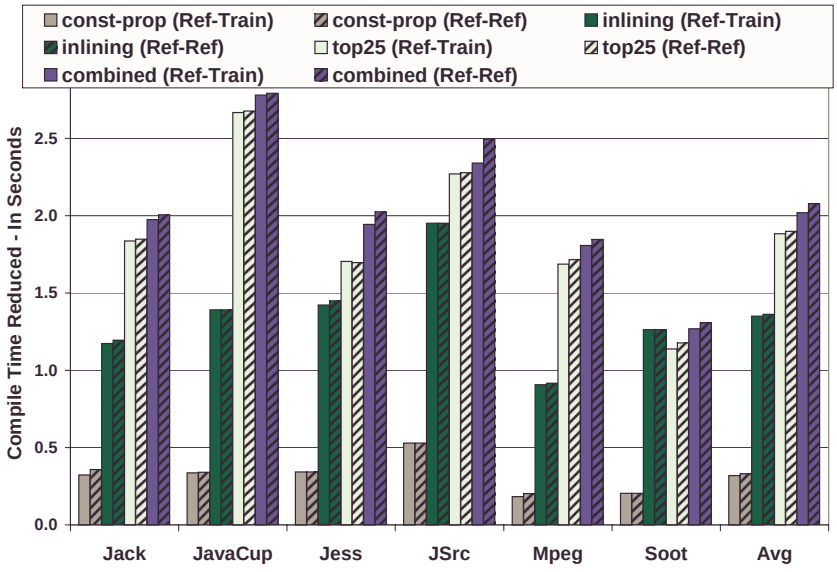
The overhead associated with annotations consists of class file size increase and the execution overhead needed to process and make use of the annotations. We detail the former in Section XI.B.3. Any execution overhead imposed by annotations is included in the overall results. We first present results in terms of the reduction in compilation time resulting from the use of our annotations. The number of seconds of compilation time reduced is shown in Figure XI.3. Each bar shows the reduction due to each of the individual optimizations. The two, right-most bars of each group is the number of seconds reduced using all of the annotations we examine in this chapter. *Const_prop, Inlining, and top25* use profile information to guide the optimization. We therefore show two bars of each of these optimizations (as well as for the combined results). The first bar of each pair (Ref-Train) shows the cross-input results (different inputs were used to generate the profile and the results). The second bar of each pair (Ref-Ref) shows the effect of using the same input for profile and result generation. On average, our annotation optimizations reduce compilation overhead by 1.9 seconds using imperfect information and over 2 seconds using perfect information (this equates to a 78.1% reduction for cross-input and a 79% reduction for same-input results).

Figure XI.4 shows the total compilation time required for optimized compilation (O3), annotated compilation (Annot), and unoptimized compilation (O1) for same-input and cross-input configurations. Annot results show the combined effect from all of the annotation optimizations we describe in this chapter. These results are the same as shown in Figure XI.3, however, they are in terms of resulting compile time instead of the number of seconds reduced and include O1-compilation for comparison. On average, annotated compilation time is approximately 250 milliseconds slower than O1 compile time and 2 seconds faster than O3 compile time.

The results in Figure XI.3 show that the majority of compile time reduction is due to two optimizations, inlining and top25%. This indicates that using these two optimizations alone is enough to achieve substantial performance benefit. As such, we also have collected



(a)



(b)

Figure XI.3: Seconds of compilation delay reduced.
Results are broken down by optimization. The two, right-most bars of each group is the number of seconds reduced using all of the annotations we examine in this chapter. *Const_prop, Inlining, and top25* use profile information to guide the optimization. Graph (a) shows the Ref-Ref results in which the same input is used for both profile and result generation. These results are repeated in graph (b) and the cross-input results (Ref-Train) are added for comparison.

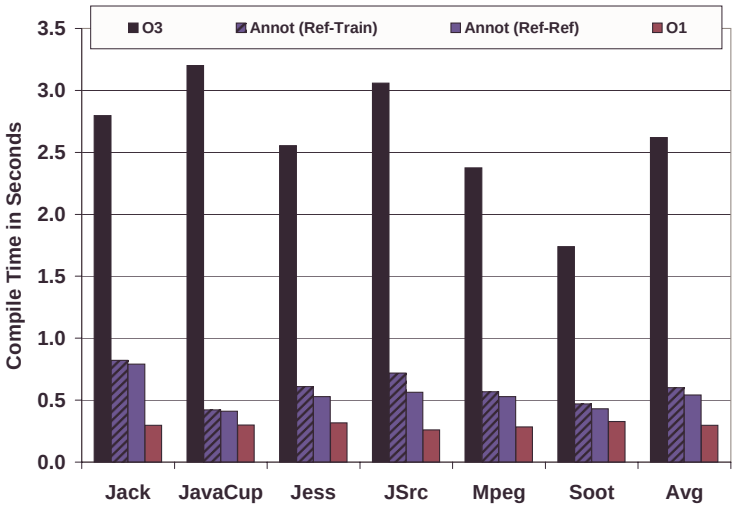


Figure XI.4: Total compilation time for O3, O1, & annotated compilation.
All of the annotation-guided optimizations presented in this chapter are included in the latter denoted by *Annot*.

results for the combination of these two optimizations alone. This data is shown by the *Annot-Inlining_top25* results in the following graphs. We next present the effect of our annotation optimizations on total time: compilation and execution time combined.

Figure XI.5 shows the speedup achieved through the use of annotation over O3 total time for both of our annotation schemes, Annot and Annot-Inlining_top25. The former are the results for all of the annotations and the latter are when only the inlining and selective optimization annotations are used. Results are shown for both input configurations (Ref-Ref and Ref-Train). The cross-input (Ref-Train) results are very similar to having perfect information (Ref-Ref results). The overall effect on total time is more dramatic the shorter the execution time, as expected. Our annotation optimizations achieve 2% to 23% speedup over O3 total time for the programs studied. However, a more important result from annotated-compilation is in its effect on startup time.

XI.B.1 The Effect on Startup Time

A significant contribution of this work is the reduction in startup time that is achieved. Startup time is arguably more important to an end user over a few percent speedup in execution time. Our studies revealed that almost all compilation in Java programs occurs at startup: In the programs studied, 77% of the compilation overhead occurs in the first 4 seconds (initial

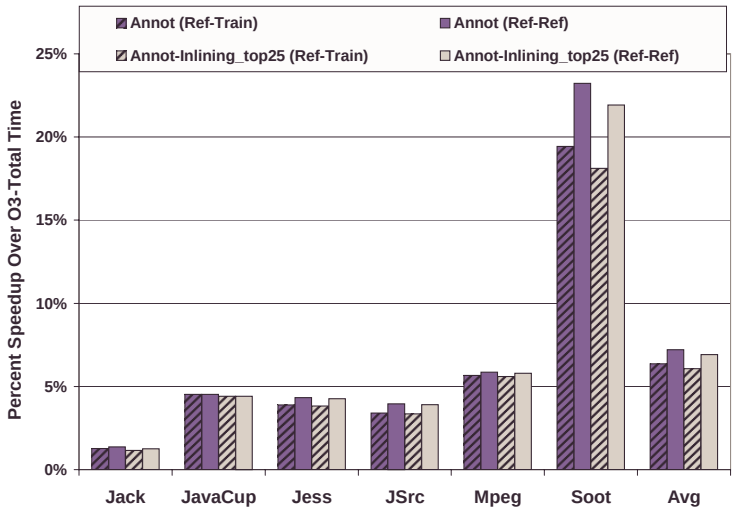
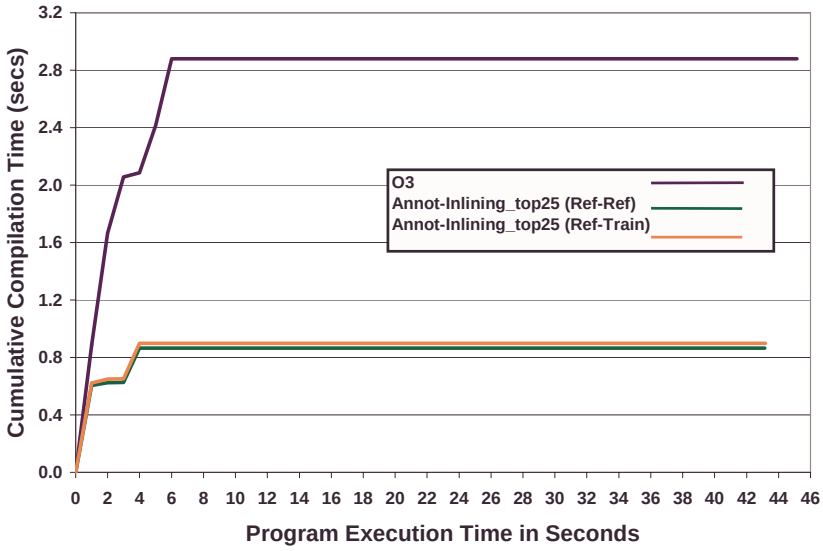


Figure XI.5: Speedup over optimized (ORP O3) total time due to annotated execution. Annot-Inlining_top25 denotes results that use the inlining and top 25% annotations alone to guide dynamic compilation.

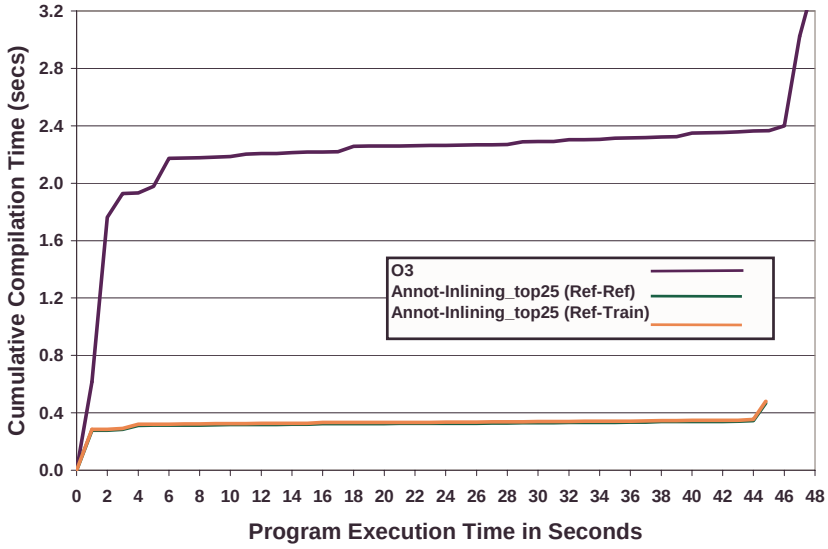
10%) of program execution on average. By reducing compilation overhead, annotated execution should substantially reduce this startup cost. Figures XI.6 through XI.8 confirm this with graphs of the cumulative distribution of compile time over program lifetime (in seconds on the y-axis). We show the cumulative time in seconds on the x-axis as opposed to the percentages commonly shown for a cumulative distribution function. The compilation overhead in seconds that has occurred since the start of the program's execution is expressed in this figure. The overhead for the O3 compiler is shown by the top (dark) line on each graph.

The bottom two lines in each graph indicate the effect of our two best-performing annotation optimizations (inlining and top25%). The results show that startup overhead is substantially reduced for every program. For example, for Mpeg, all compilation completes in 3 seconds. Using annotated-execution this point is reached in approximately one second. On average, 77% of the compilation overhead occurs in the first 4 seconds (10%) of program total time, e.g., for Jack, almost all of the 2.8 seconds of compilation overhead occur in the first 6 seconds. For all programs, startup compilation completes more than 2 seconds earlier using annotation.

Another interesting detail shown by these graphs is the compilation overhead that occurs at the end of execution for JavaCup and JSrc. The methods compiled during this period are those for I/O and clean up. We plan to use this characteristic to guide future annotation

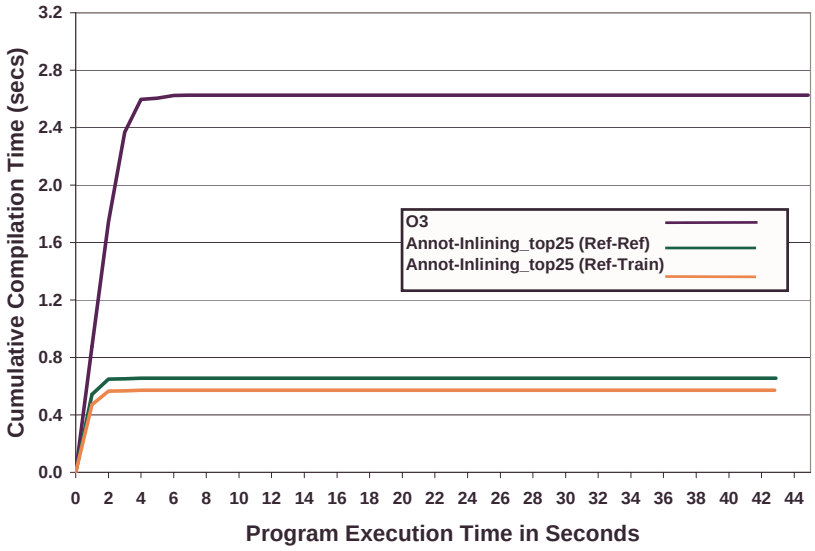


Jack

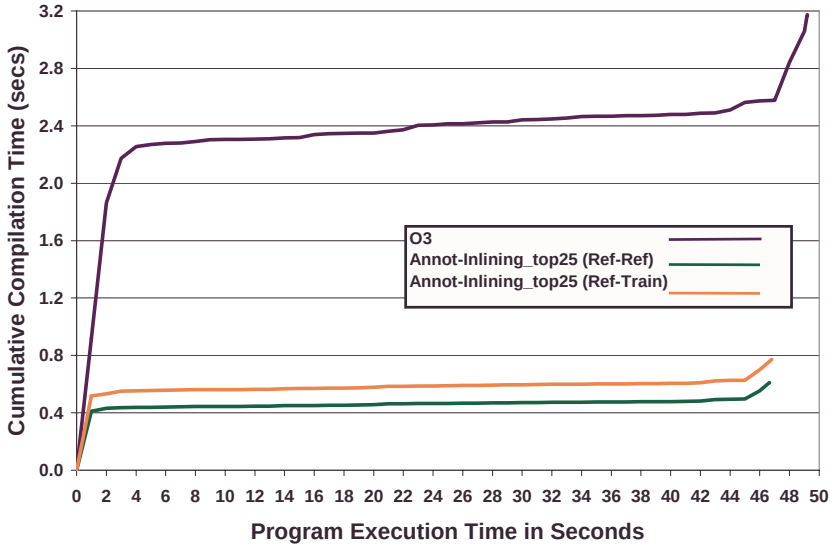


JavaCup

Figure XI.6: The effect of annotated execution on startup time (for Jack and JavaCup). Cumulative compilation time in seconds is shown over the the lifetime of each program (y-axis in seconds). These graphs show, throughout program execution, the number of seconds of compilation overhead that have occurred (x-axis) since the start of the program. The overhead for the O3 compiler is shown by the top dark line on each graph; the bottom lines indicate the effect of annotation using our inlining and top25% optimizations.

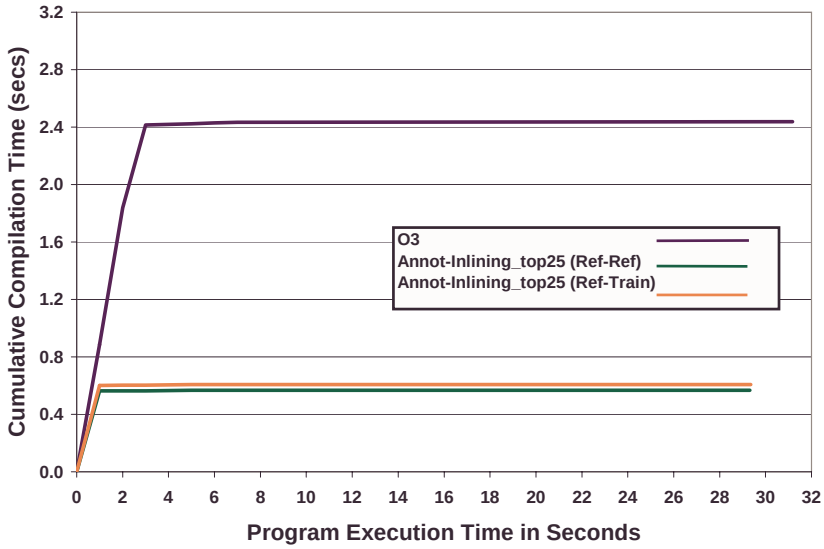


Jess

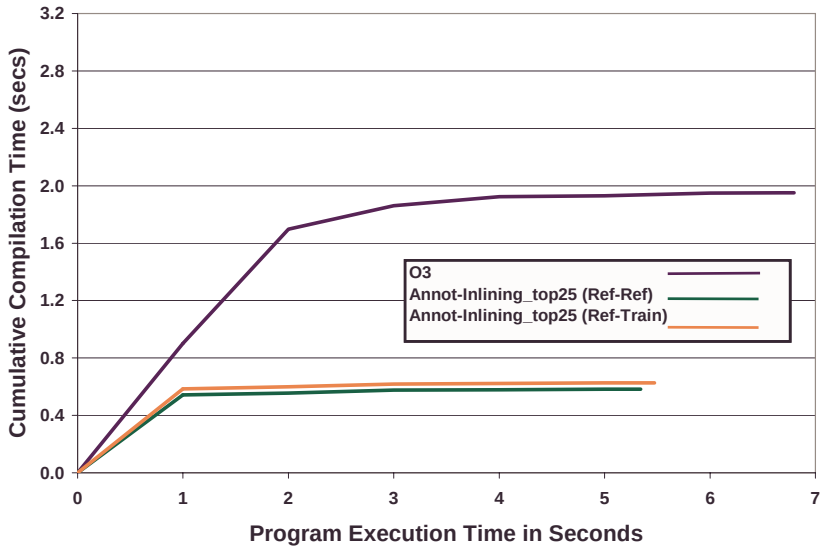


Jsrc

Figure XI.7: The effect of annotated execution on startup time (for Jess and Jsrc). Cumulative compilation time in seconds is shown over the the lifetime of each program (y-axis in seconds). These graphs show, throughout program execution, the number of seconds of compilation overhead that have occurred (x-axis) since the start of the program. The overhead for the O3 compiler is shown by the top dark line on each graph; the bottom lines indicate the effect of annotation using our inlining and top25% optimizations.



Mpeg



Soot

Figure XI.8: The effect of annotated execution on startup time (for Mpeg and Soot). Cumulative compilation time in seconds is shown over the the lifetime of each program (y-axis in seconds). These graphs show, throughout program execution, the number of seconds of compilation overhead that have occurred (x-axis) since the start of the program. The overhead for the O3 compiler is shown by the top dark line on each graph; the bottom lines indicate the effect of annotation using our inlining and top25% optimizations.

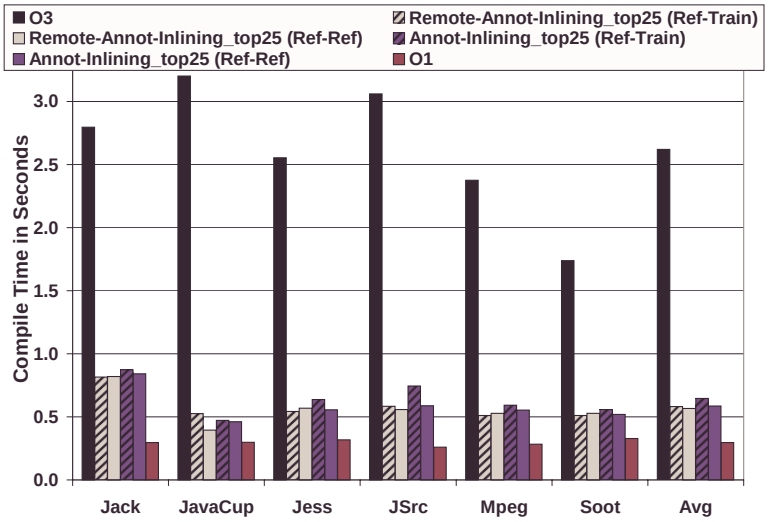


Figure XI.9: Total compilation overhead for O3, O1, and annotated compilation. Results are shown both for all class files and for remote class files only. Annot-Inlining_top25 denotes results that use the inlining and top 25% annotations alone to guide dynamic compilation. "Remote" indicates that only non-local class files use annotation-guided optimization. For this configuration, all other class files are compiled using the O1 compiler.

implementation. For example, optimization of such methods for an interactive program can be avoided since they are only used at the end of the program.

XI.B.2 Local v/s Remote Execution

Mobile Java programs are commonly transferred over a network for remote execution either through the use of dynamic class file loading of individual class files or by archiving and compressing the application as a single file, e.g. Java archive (jar). In addition, these programs commonly use Java class libraries during execution that are not part of the application itself, but are shared by all such programs. These library classes are not transferred for remote execution but are located at the destination for use by remotely executed Java programs. To this point, we have assumed that we are able to annotate both the application as well as the libraries it uses. However, this is not always the case and as such, we present results on the effect of annotating only non-local, or application, class files. Most of the prior work [6, 42] on bytecode annotations does not address the effect of limiting optimization to remote class files yet it is vital to the empirical evaluation of annotation-based techniques.

Figure XI.9 shows the reduction in compile time due to annotated-execution of only non-local class files. Class files that are non-local (or remote) include non-library files used

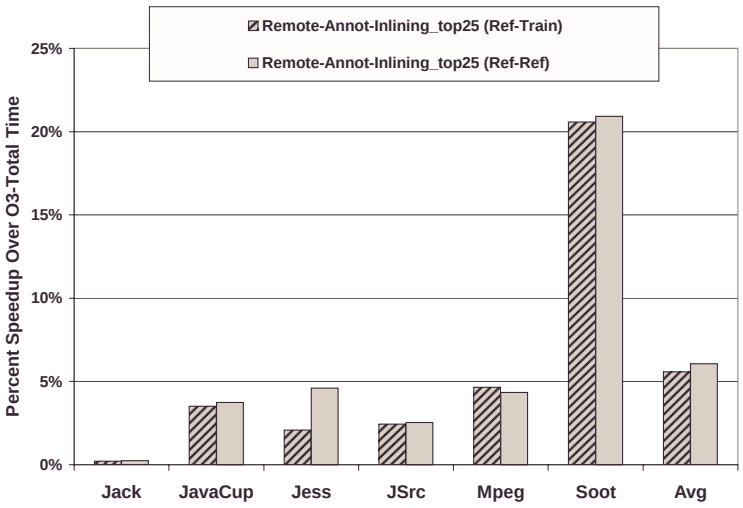


Figure XI.10: Speedup over optimized (ORP O3) total time (Remote classes only). This graph shows the speedup in O3 total time due to annotated execution of remote classes only. For these results, only the inlining and top25% annotations are used.

during execution of each program. Local (library) files are compiled using the O1 compiler (no-optimization). Realistically, library files should be optimized but the overhead associated with their compilation should not degrade startup or cause intermittent interruption. Because of this we collected results for O1-compilation of these files. Some execution environments store optimized versions of library files on disk and dynamically load them [3]. As part of future work we will incorporate such functionality into ORP.

The graph in this figure is the same as the one presented earlier in Figure XI.4 with two additional bars: one for cross-input remote-only compile time (second bar) and one for same-input remote-only (third bar) results. These results are achieved by using only the two best-performing annotations inlining and selective optimization (top25%). The two input configurations included (Ref-Ref and Ref-Train) again indicate that imperfect information has little effect on the overall performance of these annotations. Over 80% of the compilation delay is reduced by using annotation-guided optimization for non-local class files and O1-compiling all others. Figure XI.10 shows the percent speedup over O3-compilation due to annotations on non-local class files. Our inlining and top25% annotation optimizations on remote class files alone achieve 1% to 21% speedup over O3 total time for the programs studied. The average speedup across benchmarks is 6.1% and 5.6% for the same-input and cross-input configurations, respectively.

Table XI.1: The added size in kilobytes due to annotations.
Columns 2 through 5 contain the added size from each the flow graph creation, register allocation, constant propagation, reuse optimization for inlining, and selective optimization using the top 25% of frequently executed methods, respectively. Remote class file annotation alone is shown first in each column. In parenthesis, we show the annotation size across all class files in an application. The results show the percent increase in application size due to the annotation (in that column) on average across all benchmarks.

	Size of Annotation in KBytes for Non-Local Classes - Total for All Classes in Parens.				
Program	FG-Create	Regalloc	Const-prop	Inlining	SelOpt
Jack	12.55 (16.41)	0.74 (1.25)	0.28 (0.50)	0.04 (0.06)	0.04 (0.06)
JavaCup	7.02 (13.61)	0.47 (1.37)	0.20 (0.54)	0.03 (0.07)	0.03 (0.07)
Jess	9.74 (14.62)	1.02 (1.61)	0.39 (0.64)	0.05 (0.08)	0.05 (0.08)
JSrc	12.29 (15.39)	1.22 (1.68)	0.41 (0.62)	0.05 (0.08)	0.05 (0.08)
Mpeg	5.15 (8.35)	0.71 (1.17)	0.22 (0.40)	0.03 (0.05)	0.03 (0.05)
Soot	6.44 (9.66)	0.60 (1.09)	0.33 (0.54)	0.04 (0.07)	0.04 (0.07)
Avg	8.87 (13.01)	0.79 (1.36)	0.28 (0.54)	0.04 (0.07)	0.04 (0.07)
Incr	3.6% (5.2%)	0.3% (0.5%)	0.1% (0.3%)	0.0% (0.0%)	0.0% (0.0%)

As part of future work we will consider the effect of providing general annotations for local (library) class files that can be used to improve performance regardless of the invoking program. For example, it has been shown for C and Fortran that execution behavior of commonly used Unix libraries is similar across different programs and that this information can be used to guide optimization [13]. This implies that profile-based techniques for a subset of programs can potentially be used to optimize shared libraries. We are investigating such techniques as part of future work.

XI.B.3 Annotation Size

Since annotations are added to class files that transfer for remote execution, we must ensure that our framework and annotation implementations increase class file size minimally. Since it is this size that dictates the transfer time on a particular network, we present overhead as the number of kilobytes added for annotation. Table XI.1 shows this data for the different annotations implemented across all non-local class files. In parentheses is shown the number of kilobytes added for annotations in both the non-local and local class files combined. On average, the annotations add less than 0.05% to 3.4% to applications alone. By only incorporating the two best performing annotations (inlining and top25%) we increase application size by less than 0.05%. This is compared to the 7% to 97% increase in size by existing annotation

implementations for a single annotation optimization [36, 42, 69]

XI.C Summary

We have presented an annotation framework for Java programs that substantially reduces compilation overhead in the ORP dynamic optimizing compiler. The annotation language is highly extensible and represents an instruction set with opcode, size, and annotation data, and requires only one bytecode attribute for multiple annotations. The framework enables incorporation of highly-compressed, static and profile-based information into the Java bytecode stream for use in dynamic optimization. These *annotations* enable reduction in compilation overhead of 75% on average, while increasing class file size (and hence transfer delay) by less than 0.05%.

Compilation overhead in execution environments for mobile code is expensive since optimization, resulting in improved execution time, uses time-consuming analysis and processing even for very simple optimizations. However, the potential for execution speedup is large since runtime information can be used for program optimization and specialization. The annotation optimizations we present perform analysis off-line and communicate it to the optimizing compiler to obviate the need for its collection at runtime. In addition, we pass dynamic information from off-line profiles via annotations to the compilation system so that methods, predicted to be most important, are selectively optimized.

Annotation-guided optimization also reduces startup time. In the programs studied, 77% of of the compilation overhead occurs in the first 4 seconds (initial 10%) of program execution on average. Using our annotation-guided optimizations startup delay is reduced by more than two seconds on average, enabling substantial improvement in the initial progress made by program execution.

The text of this chapter is in part a reprint of the material as it is to appear in the 2001 conference proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). The dissertation author was the primary researcher and author and the co-authors listed on this publication directed and supervised the research which forms the basis for this chapter.

Chapter XII

Conclusions

Recently, there has been increased interest in the use of the Internet as a computational entity. A fundamental difficulty with such use is how to make efficient and effective use of the diverse resources that are made available by the connectivity of the Internet. One methodology that has been developed to solve this problem is remote execution in which programs first transfer over the Internet to a destination machine and then execute.

Inherently, remote execution imposes two, potentially significant, performance-limiting constraints: Transfer time must now be considered as part of total execution time, and the same program must be able to execute efficiently on a multitude of heterogeneous resources. Due to the widening gap between network and processing power and the large variation in network performance across the Internet, program transfer time using existing technology can be long and highly variable. In addition, programs that are remotely executed are commonly transferred in an architecture-independent format and an interpreter or compiler at the target sites converts it to native code. Like transfer, this translation time must be considered as part of overall program performance since it occurs while the program is executing. Also like for transfer delay, the compilation time required to enable efficient execution of these mobile programs can be substantial.

The Java programming language uses remote execution (via the Java applet execution model) to enable Internet-computing. Due to its wide acceptance and use, we use Java as our experimental language infrastructure in this dissertation. Both transfer and compilation delay severely restrict mobile Java program performance. Java attempts to reduce the effect of transfer delay through dynamic class file loading in which program files, called class files, are

transferred as needed by the executing program as opposed to all at once at program startup. However, the transfer time continues to impose substantial delay: For an average benchmark over a modem link (0.03Mb/s), transfer delay costs over 50 seconds (2 seconds using a T1 link (1Mb/s)). Our empirical measurements show that the performance of a cross-country Internet connection fluctuates between these values (0.03Mb/s and 1Mb/s) for Java programs.

Once at the destination, a class file (in the Java architecture-independent format called bytecode), must be converted to the native code format of the machine on which the program it is to be executed. As alluded to above, this process is commonly performed using compilation, a method-by-method translation of bytecode to native code. Compilation is used (over interpretation) since it enables more efficient code generation and it exposes opportunity for optimization. However, the optimization required for efficient code generation can be substantial and execution must again stall until compilation completes each time a method is initially invoked. We refer to the cumulative stall time required for transfer and compilation as "load delay".

In this thesis, we describe the execution model we assume for Internet computing, identify and detail the sources of load delay and articulate the degree to which load delay degrades program performance. We then present numerous compiler and runtime techniques that reduce the effect of load delay through *overlap* of delay with useful work and delay *avoidance*. We exemplify the effect of our optimizations with two graphs. In both, we summarize the results for three of our most effective optimizations for the reduction of load delay: Non-strict execution, class file splitting, and annotation-guided compilation (referred to in the graphs as annotated-execution).

Figure XII.1 depicts load delay (both transfer and compilation overhead) as a function of network bandwidth with and without our optimizations. This is the same graph as that presented in Figure I.1 in the introduction of this dissertation. Three additional functions have been added, however, which represent the effect of non-strict execution, class file splitting, and annotated-execution on load delay. Load delay measurements consist of the time for transmission of the (non-library) code and data, the time to request program files required for execution (for all but non-strict execution since for this optimization the request model is eliminated), and the time for optimization of executed methods. The average Java program accesses 70 non-local classes, totaling 178 transferred, and compiles 238 methods (totaling 2.6 seconds) using the Open Runtime Platform (ORP) [15] as the Java execution environment. The total delay for an average program is 55.8 seconds (transfer delay accounts for 53.2 seconds).

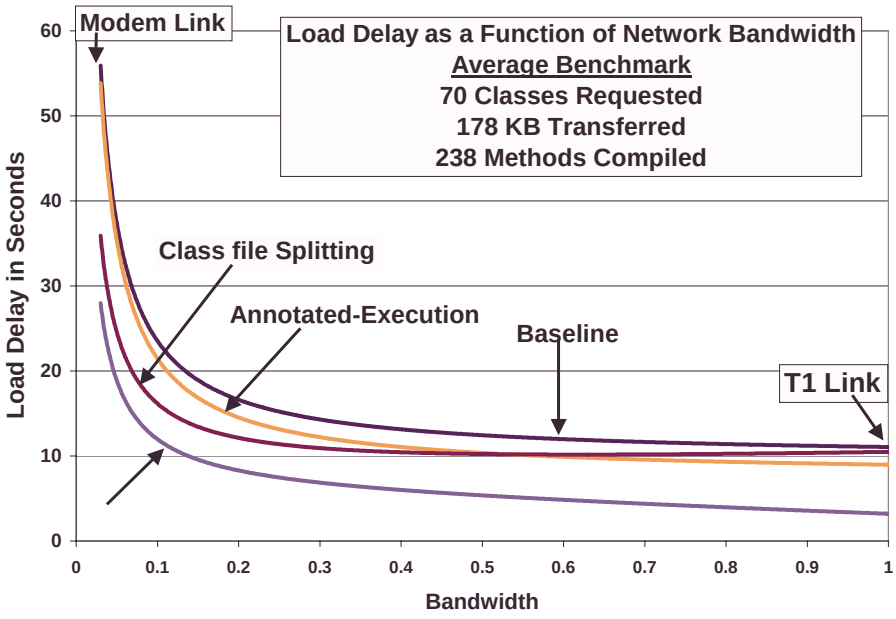


Figure XII.1: Summary of the effect of our optimizations on load delay.
The figure shows load delay in seconds (x-axis) for an average benchmark as a function of network bandwidth (y-axis) without optimization, with annotated-execution, with class file splitting, and with non-strict execution. Compilation delay (before optimization) accounts for 3 seconds of total load delay in this graph. The remainder is due to transfer delay (request and transmission of the program). Annotated-compilation reduces 77% of the compilation delay; this results in a 2 second decrease in load delay (regardless of the network bandwidth). Class file splitting reduces load delay by over 16 seconds for the modem link (0.03Mb/s) and 200 milliseconds for the T1 link. Non-strict execution reduces load delay by over 33 seconds for the modem link (0.03Mb/s) and 8 seconds for the T1 link.

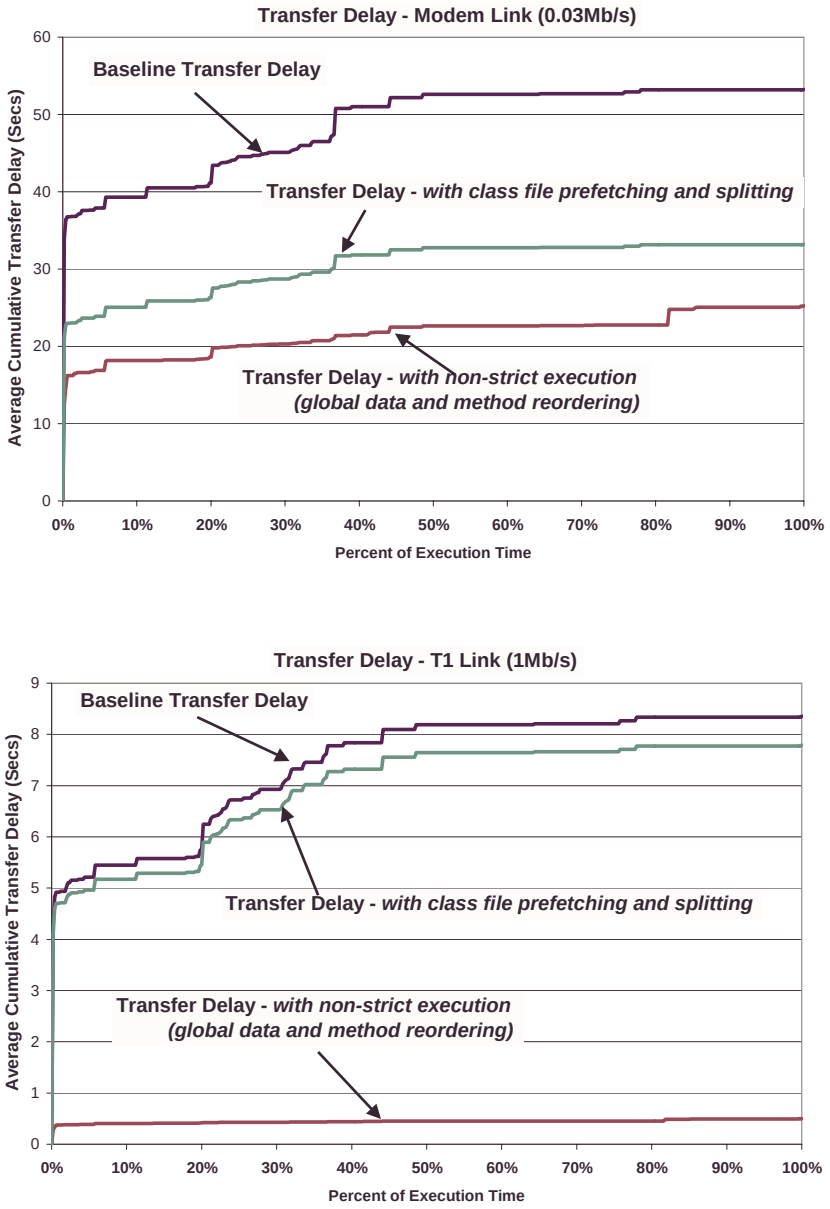


Figure XII.2: Summary of the effect of our transfer delay optimizations on startup time. The graphs show the average cumulative transfer delay (y-axis) that is experienced during program execution. The x-axis is the percent of execution time completed by the average program The average execution time for the programs used for this figure is 49 seconds. The top is for a modem link (0.03Mb/s bandwidth) and the bottom is for a T1 link (1Mb/s bandwidth). Transfer delay consists of time for request and transmission of class files from the source to the destination machine. Each graph is read by taking an (x,y) position on the function; y seconds of delay (transfer or compilation) occurs during the first x% of program execution. The baseline functions are shown in each graph. The graphs also include the results due to non-strict execution, and the combined effect of class file prefetching and splitting.

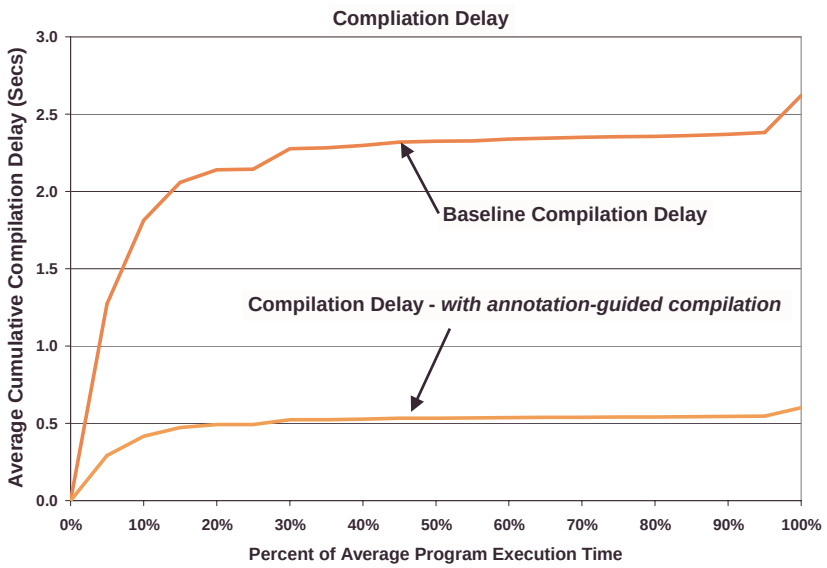


Figure XII.3: Summary of the effect of our compilation optimization on startup time. The graph shows the average cumulative compilation delay (y-axis) that is experienced during program execution. The x-axis is the percent of execution time completed by the average program The average execution time for the programs used for this figure is 49 seconds. Compilation delay is the time spent compiling and optimizing the programs. Each graph is read by taking an (x,y) position on the function; y seconds of delay (transfer or compilation) occurs during the first x% of program execution. The baseline functions are shown in each graph as well as the results due to our annotation-guided optimizations.

Non-strict execution is a technique in which JVM modification is used to enable method-level transfer, method-level execution, and overlap of execution with transfer. In addition, non-strict execution obviates the need for use of the *request* model as currently implemented by dynamic class file loading. Instead, a transfer schedule is *pushed* from source to destination in the order the code and data is predicted to be used during execution at the destination. Non-strict execution using method-level execution with method reordering across class files eliminates 13 seconds of transfer delay imposed by transfer over a modem link (8 seconds for the T1 link). Global data reordering reduces this same delay (modem) by 28 seconds (8 seconds for the T1 link).

Class file splitting and prefetching reduces transfer delay by 32%. It does this using existing JVM technology. Class files are modified to contain only the code and data that will be used during execution. Like non-strict execution, off-line profiles from instrumented execution are used to guide the splitting. Unused methods and data are split out into "cold" classes that if used are transferred using the existing class file loading mechanism. If unused, significant savings in transfer time are achieved. In addition to splitting, prefetching is performed using a background thread that prematurely accesses a class file causing it to transfer. Upon first-use by the executing program, the class is partially transferred since the delay was masked by execution. Class file splitting and prefetching together reduce 20 seconds of transfer delay for a modem link and 1 second for a T1 link.

Annotation-guided compilation avoids compilation overhead by performing program analysis required for optimization off-line. In addition, profile information is communicated to the compilation system so that optimization can be applied selectively to parts of the program for which it is most cost effective. Both types of information (static analysis and profile) are passed to the compilation system via compact bytecode annotations. Since we are interested in keeping load delay small, compact encoding of annotations is essential to minimize any increase in application size. On average, our annotations increase this size by less than 0.05% yet avoid 77% of the compilation overhead and enable speedups of 6% over optimized total time (compilation plus execution). In terms of load delay, annotated-execution reduces it by over 2 seconds.

Figures XII.2 and XII.3 depict the effect of our key techniques on program startup time. The graphs show the average cumulative delay (y-axis) that is experienced during program execution. The x-axis is the percent of execution time completed by the average program (no transfer or compilation delay is included in this value). The average execution time for the

programs used for this figure is 49 seconds. The first two graphs (Figure XII.2) are for transfer delay: the top is for a modem link (0.03Mb/s bandwidth) and the bottom is for a T1 link (1Mb/s bandwidth). This data assumes that a request for each class costs 100ms, a common (based on empirical data) cross-country round-trip time value. The graph in Figure XII.3 shows the average cumulative compilation delay, the second source of load delay. Each graph is read by taking an (x,y) position on the function; y seconds of delay (transfer or compilation) occurs during the first x% of program execution.

The startup time function for the modem link indicates that 39 of the 56 seconds of transfer delay occur in the first 10% (5 seconds) of execution time and 90% of all transfer delay (50 seconds) occurs in the first 40% (44 seconds) of program execution. Similarly for the T1 link, 5 of the 8 seconds of transfer delay is incurred during the first 10% of program execution and 90% (7 seconds) of the transfer delay occurs in the first 30% of program execution (14 seconds). Compilation overhead is also incurred at program startup as shown by Figure XII.3. This function indicates that 1.8 seconds of the 2.6 seconds (69%) of compilation delay occur during the first 10% of execution and 90% of it occurs in the first 30% of program execution. Almost 70% of *all* load delay occurs in the first 10% of program execution.

By reducing the effect of load delay, we improve the progress made at program startup as well as throughout overall execution. The functions denoted as non-strict execution and class file splitting on the graphs show the effect of our optimizations on startup delay caused by transfer. On average across inputs, non-strict execution using method and global data reordering eliminates 21 seconds of transfer delay in the first 10% of program execution for the modem link and 5 seconds for the T1 link. Class file splitting and prefetching reduces startup transfer delay (delay incurred during the first 10% of program execution (5 seconds)) by 14 seconds for the modem link and by 200 milliseconds for the T1 link. Annotated-execution results are included on the compilation delay graph and indicate a reduction in startup delay of almost 2 seconds. All of these techniques reduce startup time significantly (as well as overall execution time) and enable greater execution progress during the initial seconds of program execution. Faster program startup, response, and overall execution time achieved by our techniques can potentially improve user perception of program performance and productivity.

In this dissertation, we define load delay as the cumulative overhead imposed by transfer and compilation on remotely executed Java programs. We detail the limitations of the existing mobile Java execution model that cause load delay and then present a body of work in which we suggest changes to as well as techniques for the exploitation of mobile execution envi-

ronments. Our solutions are general and enable overlap of load delay with execution as well as load delay avoidance, regardless of its source (transfer or compilation), to substantially improve mobile program startup time as well as overall performance. Such performance improvements are vital to acceptance and wide-spread use of the Internet as a computational engine for the vast number and diversity of resources it connects.

Chapter XIII

Future Directions

As high-performance network connectivity becomes omnipresent, end-users have come, and will continue, to expect delivered application performance in an Internet-computing environment. Current trends indicate that the future of Internet computing will evolve into a combination of *peer-to-peer* (PTP) and *Computational Grid* computing to enable substantially improved program performance. PTP systems attempt to harness unused but ubiquitous computer capacity via the expanding internetwork of high-performance connectivity by aggregating the computing power and storage capacity available throughout the Internet. Similarly, the Computational Grid is a computing paradigm for the development of software systems that enable dynamic acquisition of resources from a heterogeneous and non-dedicated resource pool. These paradigms require that applications adapt to the dynamically changing systems on which they are executed as well as to variable resource performance. In addition, such systems require mechanisms that restrict access to resources and prohibit unauthorized and destructive behavior by programs as required by administrators. This decentralized approach to Internet computing motivates the need for mechanisms and optimizations that improve application performance while ensuring secure behavior of untrusted programs.

Our future work will focus on the design and implementation of compilation systems with optimization and specialization techniques that take advantage of this decentralized computational model to enable secure, yet high performance, application execution that is vital for widespread user acceptance of such computing paradigms. Three categories of optimization we plan to consider are:

- Adaptive Optimization guided by Real-time Resource Performance.

In future Internet-computing paradigms (such as PTP computing), programs will likely be transferred in a portable intermediate format, e.g., bytecode, .Net, or others. Such formats enable the efficient "write once, run anywhere" paradigm. These formats must be converted into instructions that are executable by the underlying architecture dynamically by a compiler. Efficient dynamic optimization and re-optimization are needed to facilitate high-performance execution of such programs. We plan to develop program specialization techniques and compilation strategies that aggressively target different architectures to produce high-performance executables from the most common intermediate program formats. In addition, we plan to use resource performance prediction tools to guide initial and adaptive compiler optimization. Using forecasts of dynamic resource performance and availability for the peer CPUs as well as for the network between them, we will automate cost function computation for use in guiding (re-)optimization decisions.

- Embedded Systems.

  Currently embedded systems and the Internet have been thought of as mutually exclusive execution environments. Current trends indicate that the distinction between the two is fading, however. As this continues, it will be important to have applications that can run anywhere e.g., on a cell phone, laptop, coffee pot, etc., using the same code (intermediate form). Dynamic compilation techniques can be used to enable such execution in resource-limited environments and enable high-performance (and possibly migrating), transparent execution of the same programs on very different devices. We will develop techniques that enable performance of this class of application in such execution environments in the future.

- Compiler-guided Security.

  Secure, yet efficient and scalable, execution is a challenging and open question in Internet-computing research. We plan to consider compilation techniques that enable multiple levels of security to be implemented, e.g., trusted, semi-trusted, and untrusted. In each case, it is desirable for the code to run as efficiently as possible and for static and run-time checks to implement necessary security levels while imposing minimal performance overhead. A compilation system can be used to adaptively optimize for and enable dynamically changing security requirements. We will research compiler-aided security systems for PTP systems as part of future work.

Bibliography

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. Fast,Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, October 2000.

[2] B. Alpern, C. Attanasio, J. Barton, A. Cocchi, S. Hummel, D. Lieber, T. Ngo, M. Mergen, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1999.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[4] B. Alpern, M. Charney, J. Choi, A. Cocchi, and D. Lieber. Dynamic linking on a shared-memory multiprocessor. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[5] A. Appel and E. Felten. Secure Internet Programming. `http://www.cs.princeton.edu/sip/index.php3`.

[6] A. Azevedo, A. Nicolau, and J. Hummel. Java Annotation-Aware Just-In-Time Compilation System. In *ACM Java Grande Conference*, June 1999.

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

[8] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report Technical Report HPL-1999-78, HP Laboratories, 1999.

[9] J.L. Bash, E.G. Benjafield, and M.L. Gandy. The Multics operating system – an overview of Multics as it is being developed. Technical report, Massechusettes Institute of Technology, 1967. Project MAC, MIT, Cambridge, Mass.

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

[11] Blackdown. Java Linux. `http://www.blackdown.org/`.

[12] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamically optimizing compiler for Java. In *ACM Java Grande Conference*, June 1999.

[13] B. Calder, D. Grunwald, and A. Srivastava. The Predictability of Branches in Libraries. In *28th International Symposium on Microarchitecture*, November 1995.

[14] 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.

[15] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[16] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, October 2000.

[17] Intel Corporation. Intel Corporation. `http://www.intel.com`.

[18] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. In *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1996.

[19] R.C. Daley and J.B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, 1968.

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

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

[22] B. Eckle. *Thinking in C++, Second Edition Volume One: Introduction to Standard C++.* Prentice Hall, 2000.

[23] 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.

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

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

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

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

[28] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[29] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Dyc: An expressive annotation–directed dynamic compiler for c. Technical Report Tech Report UW-CSE-97-03-03, University of Washington, 2000.

[30] N. Groschwitz and G. Polyzos. A time series model of long-term traffic on the nsfnet backbone. In *Proceedings of the IEEE International Conference on Communications (ICC'94)*, May 1994.

[31] A. Hashemi, D. Kaeli, and B. 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.

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

[33] U. Hölzle and D. Ungar. A third–generation self implementation: Reconciling responsiveness with performance. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1994.

[34] The Java Hotspot performance engine architecture.

[35] D. Hovemeyer and B. Pugh. More efficient network class loading through bundling. In *Proceedings of the USENIX JVM'01 Conference*, April 2001.

[36] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the Java Bytecodes in Support of Optimization. In *Journal Concurrency:Practice and Experience, Vol. 9(11)*, November 1997.

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

[38] Hypertext transfer protocol. `http://www.w3.org/Protocols/`.

[39] Ice Inc. the tar archive utiltity in Java (public domain). `http://www.gjt.org/javadoc/com/ice/tar/package-summary.html`.

[40] Microsoft Inc. Microsoft Explorer. `http://www.microsoft.com/net/`.

[41] Microsoft Inc. Microsoft Explorer. `http://www.microsoft.com/windows/ie/`.

[42] J. Jones and S. Kamin. Annotating Java Bytecodes in Support of Optimization. In *To appear in the Journal of Concurrency: Practice and Experience*, 2000.

[43] R. Jones. `http://www.cup.hp.com/netperf/netperfpage.html`. Netperf: a network performance monitoring tool.

[44] Kaffe – An opensource Java virtual machine.

[45] A. Krall and R. Grafl. Cacao - a 64 bit JavaVM just-in-time compiler. In *Concurrency: Practice and Experience*, volume 9 (11), pages 1017–1030, November 1997.

[46] C. Krintz and B. Calder. Using Annotation to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, October 1998.

[47] C. Krintz and B. Calder. Reducing Transfer Delay with Dyanamic Selection of Wire-Transfer Formats. Technical Report UCSD CS00-650, University of California, San Diego, April 2000.

[48] C. Krintz, B. Calder, and U. Hölzle. Reducing Transfer Delay Using Java Class File Splitting and Prefetching. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1999.

[49] 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.

[50] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the Overhead of Dynamic Compilation. *Software—Practice and Experience*, 31(8):717–738, 2001.

[51] C. Krintz and R. Wolski. JavaNws: The network weather service for the desktop. In *ACM JavaGrande 2000*, June 2000.

[52] Latte: A fast and efficient Java vm just-in-time compiler.

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

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

[55] 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.

[56] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[57] 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.

[58] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, February 1994.

[59] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[60] 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.

[61] Sun Microsystems. Java Appletviewer. `http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/appletviewer.html`.

[62] R. Morelli. *Java Java Java - Object-Oriented Problem Solving*. Prentice Hall, 2000.

[63] Netscape. Netscape. `http://www.netscape.com`.

[64] Scott Oaks. *Java Security*. O'Reilly and Associates, Inc., Sebastopol, CA, 1998.

[65] Open Runtime Platform (orp) from Intel Corporation. `http://intel.com/research/mrl/orp`.

[66] 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.

[67] Pkware inc. `http://www.pkware.com/`. PKZip format discription: `ftp://ftp.pkware.com/appnote.zip`.

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

[69] P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A Framework for Optimizing Java Using Attributes. In *Sable Technical Report No. 2000-2*, 2000.

[70] 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.

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

[72] W. Savitch. *Java - An Introduction to Computer Science and Programming.* Prentice Hall, 2001.

[73] Mauricio Serrano, Rajesh Bordawekar, Sam Midkiff, and Manish Gupta. Quasi-static Compilation in Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2000.

[74] 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.

[75] D. Smith. *The Concepts of Object-Oriented Programming.* McGraw-Hill, 1991.

[76] Chesapeake Network Solutions. Test TCP (TTCP). `http://www.ccci.com/product/network_mon/tnm31/ttcp.htm`.

[77] Spec jvm98 benchmarks.

[78] A. Srivastava. From communication on reducing startup delay in Microsoft Applications.

[79] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1), 2000.

[80] Sun Microsystems JIT Compiler.

[81] The Symantec Just–In–Time Compiler.

[82] F. Tip, C. Laffra, P. Sweeney, and D. Streeter. Practical experience with an application extractor for java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1999.

[83] 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.

[84] D. Ungar and R. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA '87*, pages 227–242, December 1987.

[85] J. Vitek and C. Jensen, editors. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. Number 1603 in Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, 1999.

[86] R. Wahbe, S. Lucco, T. Anderson, and S. 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.

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

[88] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998. also available from http://www.cs.ucsd.edu/users/rich/publications.html.

[89] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 1999. available from `http://www.cs.utk.edu/~rich/publications/nws-arch.ps.gz`.