Applying Dynamic Software Updates to Computationally-Intensive Applications

Dong Kwan Kim

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Advisory Committee:

Dr. Eli Tilevich, Chair

Dr. James D. Arthur
Dr. Shawn A. Bohner
Dr. Marc Fisher II
Dr. Denis Gracanin
Dr. Jung-Min Park
Dr. Calvin J. Ribbens

June 16 2009

Blacksburg, Virginia

Keywords: Dynamic Software Updates, Computationally-Intensive Applications, HPC,

Binary Rewriting, Java Virtual Machine, Proxy Pattern

Copyright 2009 ©, Dong Kwan Kim

Applying Dynamic Software Updates to Computationally-Intensive Applications

Dong Kwan Kim

(ABSTRACT)

Dynamic software updates change the code of a computer program while it runs, thus saving the programmer's time and using computing resources more productively. This dissertation establishes the value of and recommends practices for applying dynamic software updates to *computationally-intensive applications*—a computing domain characterized by long-running computations, expensive computing resources, and a tedious deployment process. This dissertation argues that updating computationally-intensive applications dynamically can reduce their *time-to-discovery* metrics—the total time it takes from posing a problem to arriving at a solution—and, as such, should become an intrinsic part of their software lifecycle. To support this claim, this dissertation presents the following technical contributions: (1) a distributed consistency algorithm for synchronizing dynamic software updates in a parallel HPC application, (2) an implementation of the Proxy design pattern that is more efficient than the existing implementations, and (3) a dynamic update approach for Java Virtual Machine (JVM)-based applications using the Proxy pattern to offer flexibility and efficiency advantages, making it suitable for computationally-intensive applications. The contributions of this dissertation are validated through performance benchmarks and case studies involving computationally-intensive applications from the bioinformatics and molecular dynamics simulation domains.

Dedication

To My Dear Wife, Eunju,

And

To My Loving Parents and Parents-in-law

Acknowledgments

Any accomplishments I have achieved on this long journey would not have been possible without the aid of many individuals, and I would like to sincerely thank all those who have helped and supported me with various aspects of my research.

I deeply thank Dr. Eli Tilevich, my advisor and research mentor, for his prompt, effective, and encouraging guidance. He always made himself available whenever I needed him, selflessly spending numerous hours guiding this research, which regularly included weekends and holidays. I find his dedication to research and work hours somewhat intimidating. His guidance encompassed such diverse activities as brainstorming new research ideas, designing and evaluating research prototypes, and editing research manuscripts. As a result of our interactions, I have learned the intricacies and challenges of conducting research. I do not think I will ever forget his special research elixir in the form of dark chocolate, hours and hours of pair programming and writing, and late night discussions on the essence of the research enterprise.

I thank the rest of my thesis committee for helping me to see this dissertation to completion. Dr. James D. Arthur encouraged me never to give up my studies and was instrumental in helping me complete this work in a timely manner. Dr. Calvin J. Ribbens generously contributed his deep expertise of high performance computing, which was invaluable in enabling me to validate some of the most important contributions of this dissertation. Dr. Marc Fisher II provided the much needed assistance in writing and proofreading this dissertation manuscript. Dr. Shawn A. Bohner shared with me his insightful comments about research and life in general. Drs. Denis Gracanin and Jung-Min Park provided inter-

esting comments that helped me clarify the contributions of this work. My special thanks go to Dr. Barbara G. Ryder, our department head, who took time from her extremely busy schedule to provide me with prompt and constructive help in resolving several very difficult predicaments that arose during the course of my studies.

My many Korean friends in the department, with Kibum and Seonho having graduated before me, shared all the difficulties, disappointments, and victories of my graduate studies. Their encouragement has gotten me through this difficult journey.

My employer and colleagues at ETRI generously allowed me to take a leave of absence to pursue my graduate studies abroad.

I would not have gotten that far without the support of my family. I always knew that my parents and parents-in-law prayed fervently for me, while patiently waiting for me to complete my studies. My wife Eunju was always there for me—I could always turn to her for solace and encouragement, and her tears and prayers for me had kept me going.

Finally, I gratefully thank God for leading me to this moment in my life and beyond. I always felt the presence of the living God during this difficult journey, arranging the most needed encounters with the people who helped me throughout my studies.

Table of Contents

1 Introduction **1**

1.1 Problem Formulation 6

1.2 Contributions 14

1.3 Outline of the Dissertation 16

2 Background **17**

2.1 Bytecode Transformation 18

2.2 Proxy Pattern 19

2.3 Dynamic Software Updating Systems 24

2.3.1 DSU System Properties 26

2.3.2 Dynamic Software Updates of Java Applications 32

2.3.3 Dynamic Software Updates of Non-Java Applications 39

3 Technical Research Approach **42**

3.1 Updating VM-based Applications Dynamically 43

3.2 Virtual Superclass Binary Refactoring 45

3.3 Translating Updates for HotSwap Conformance 51

3.3.1 Structural Changes to Class 52

4 Design and Implementation **60**

4.1 Representing the Dynamic Updating System Using UML Diagrams 60

4.2 Transformation Rules 64

4.3 The ClassDifferencing Algorithm 65

4.4 Synchronizing Concurrent Updates of Cluster Applications 67

4.4.1 Design Considerations 68

4.4.2 Synchronization Algorithm 70

4.4.3 Example Implementation 73

4.5 Limitations and Issues 77

5 Experiments and Results Analysis **80**

5.1 Performance Evaluation 80

5.2 Updating an RMI-based Application Dynamically 82

5.3 Updating Smith-Waterman Parallelization Dynamically 85

5.3.1 Smith-Waterman Algorithm 85

5.3.2 Parallelizing a Program of the Smith-Waterman Algorithm 87

5.3.3 Experimental Results 92

5.4 Updating a Scientific Application Dynamically 96

5.4.1 Enhancing a Successive Over-Relaxation Program 96

5.4.2 Updating a Molecular Dynamics Simulation System Dynamically . . 98

5.5 Updating High Performance Cluster Applications Dynamically 100

6 Future Work and Conclusions **104**

6.1 Future Work 104

6.2 Conclusions 106

6.2.1 Guidelines for Applying DSU to CIAs 106

6.2.2 Summary 108

Bibliography **110**

A Sample Code **124**

A.1 Generation of a proxy and virtual superclass 124

A.2 Bytecode rewriting using Javassist 127

A.3 HotSwap Screenshot 130

A.4 MPJ Express 131

List of Figures

1.1 Software development of CIAs using DSU. 8

2.1 Proxy design pattern. 21

2.2 Java dynamic proxy. 22

2.3 Invoking the m1() method via Java dynamic proxy. 23

2.4 Eugster's uniform proxy. 23

2.5 Indirecting using an interface. 24

2.6 Isomorphic proxies for JDK system classes. 25

2.7 Offline software updates VS. dynamic software updates. 25

2.8 JVM HotSwap facility. 33

3.1 Overall flow of the approach to dynamic software updates. 43

3.2 Creating updateable applications. 44

3.3 Bytecode transformation. 45

3.4 Virtual interface binary refactoring. 47

3.5 Virtual Superclass binary refactoring. 48

3.6 `invokeinterface` and `invokespecial` bytecode instructions. 50

3.7 Limitations of HotSwap Technique 53

3.8 HotSwap Models. 54

3.9 Adding a new method. 56

4.1 Use case diagram of the DSU system presented in this research. 61

4.2 Class diagram of the DSU system presented in this research. 63

4.3 Sequence diagram of the *makingProgramUpdateable* scenario. 64

4.4 Sequence diagram of the *generateNewerVersion* scenario. 65

4.5 Indirection using superclasses. 66

4.6 Indirecting constructors and methods. 66

4.7 Indirecting the superclass's non-private fields. 67

4.8 Generating a special *invoke* method. 67

4.9 Generating special methods to get helper classes. 68

4.10 The $ClassDifferencing$ algorithm. 69

4.11 Waiting for update information from the user. 71

4.12 Synchronizing concurrent dynamic updates on multiple nodes. 72

4.13 Making HPC applications updateable. 75

4.14 An example of the synchronization code. 75

4.15 Updating parallel HPC at runtime. 76

5.1 Overhead of binary refactoring microbenchmark. 81

5.2 Overhead of binary refactorings on SpecJVM98. 83

5.3 The RMI-based simple banking program. 84

5.4 Changes to the RMI program using extended HotSwap. 84

5.5 Screenshot of a serial JAligner program 88

5.6 Growth of GenBank (1982-2008). 89

5.7 High-level view of parallelizing JAligner. 93

5.8 Changes to Smith-Waterman alignment program 93

5.9 Refactoring overhead of Smith-Waterman alignment program. 95

5.10 Refactoring overhead on Successive Over-Relaxation. 97

5.11 Updating the rescaling module of a molecular dynamics simulation. 99

5.12 Performance overhead on the update management code. `Bcast` and `Barrier` have been executed 200 and 20 times, respectively. 100

5.13 Performance overhead on the update management code. `Bcast` and `Barrier` have been executed 500 and 20 times, respectively. 102

A.1 Screenshot of Transformer class. 130

A.2 Executing a target program. 131

A.3 Executuing a HotSwap program. 132

A.4 Starting MPJ Express daemons. 133

A.5 Executing an MPJ Express program. 134

A.6 Stopping MPJ Express daemons. 134

List of Tables

2.1 Comparison to related work on dynamic software updating for Java software 38

5.1 SpecJVM98 test programs. 82

5.2 Dynamic updates to the Banking program. 85

5.3 Changes to the Molecular Dynamics Simulation. 99

Chapter 1

Introduction

Among the most challenging computing application domains is parallel programming for distributed memory multiprocessors. Such systems range from compute clusters to ad-hoc grids, but fundamentally they coordinate a collection of distributed computing resources to solve a computationally-intensive problem in parallel. Distributed memory multiprocessors help solve important computational problems in science and engineering, in domains including scientific simulation, image processing, and bioinformatics.

Writing software for distributed memory multiprocessors has been notoriously difficult due to a variety of factors. Parallel programming models over distributed memory abstractions are difficult to utilize effectively to achieve good performance. Distributed coordination is challenging and error-prone. The runtime behavior of a parallel program is difficult to predict from looking at its source code. Finally, applications in this domain are often written by domain experts—scientists and engineers—who are extremely knowl-

edgeable in their respective domains but may lack a deep understanding of computing or experience with modern developments in software engineering.

The raison d'être of parallel computationally-intensive applications (CIAs) is to reduce *time-to-discovery*, the total time it takes from posing a problem to arriving at a solution. This metric is the time it takes to develop and fine-tune an application and the sum of the time it takes to run it. While computationally-intensive computing researchers have traditionally focused on reducing the time to run applications, another avenue for reducing time-to-discovery is to apply solid software engineering principles, novel techniques, and advanced tools [21, 56, 98]. That is, the software engineering approaches traditionally used to improve the construction and maintenance of traditional software can also benefit computationally-intensive software.

In particular, this dissertation explores how *dynamic software updates (DSU)*—an advanced software engineering approach for updating software while it runs—can be applied to computationally-intensive applications, thus reducing their time-to-discovery. The idea behind dynamic software updates is simple. While a program is running, the programmer changes the program's source code, compiles the program into a binary representation, and then uses a dynamic update system to replace the running binary representation with the updated one. Implementing a safe and efficient dynamic update system is strewn with challenges, including the need to replace a binary representation of a program while preserving its runtime state as well as dealing with various features of the underlying programming language and the runtime environment.

Dynamically updating parallel computationally-intensive applications presents its own set of challenges. A typical parallel application includes multiple concurrent tasks

executed on different processors, each running at its own pace and only periodically communicating with other tasks. These concurrent tasks must be updated consistently, which is non trivial. In particular, maintaining consistency during a dynamic update of multiple concurrent processes requires a distributed coordination protocol. This protocol must not only ensure some runtime invariants (e.g., no two divergent versions of a task are running simultaneously) but also do so without imposing an undue performance overhead on the parallel program.

Despite these challenges of applying dynamic software updates to a parallel computationally-intensive application, this research foresees that this advanced software engineering approach is capable of drastically improving how we build and fine-tune applications in this important domain. Computationally-intensive applications running on large numbers of processors are difficult to develop incrementally, as they are often time-consuming to deploy; the typical *try-change-try again* development cycle does not fit well for applications in this domain. In particular, any code change involves stopping the execution, changing the program, re-deploying the changed program, and re-starting the computation anew. In a computationally-intensive computing environment, all these actions can be quite time-consuming and disruptive. Besides, this development model may not utilize expensive computing resources most effectively, wasting valuable processing time.

Furthermore, two important trends in computationally-intensive applications exacerbate the cost of starting and restarting a computation: *grid computing* and *real-time simulation*. In the case of grid computing, restarting a computation from scratch requires repeating several steps, all of which are expensive and some of which may be impossible (i.e., resource discovery and reservation, resource allocation, data staging, data streaming, and job

launching). In the case of real-time simulation, "system-level" simulations, which use high performance computing (HPC) resources, often have hard-to-anticipate requirements and real-time constraints, thus necessitating dynamic code updates to avoid stop/restart cycles. Examples of such computations include hurricane modeling, infrastructure and environment monitoring, epidemic modeling, and personalized medicine.

Dynamic software updating offers a pragmatic approach that can save human time and computing resources. With dynamic updates, one could change parameters, specify a different precision, or even switch to a different model or algorithm on the fly, observing the results of the updates in near real time. All of these can significantly reduce the time-to-discovery for computationally-intensive applications.

Having studied a large body of research on various aspects of DSU, this research finds that this advanced software engineering technique can provide the following benefits for computationally-intensive applications:

- **Satisfying new or changed user requirements on the fly**

  If the initial requirements change while a computationally-intensive application is executing, the computation must be interrupted, so that the code can be changed to reflect the new requirements. By contrast, dynamic software updates do not require interrupting the computation by enabling code changes on the fly.

- **Reducing software downtime**

  Off-line software updates result in software downtime—an application provides no services during updates. In computationally-intensive computing, such downtime is undesirable, as computing resources are quite expensive to operate, with large electricity

and cooling costs. Dynamic software updates can reduce software downtime, thereby avoiding the loss of intermediate data and using computing resources more productively.

- **Avoiding the re-deployment hassle**

  Deploying computationally-intensive applications often requires interacting with a scheduler, a software module that allocates the required computational resources and schedules the execution. In heterogeneous grid environments, effective scheduling can become particularly onerous. In any case, re-deploying a computationally-intensive application imposes an additional burden on the user. Dynamic software updates can eliminate the need to re-deploy the application by applying the required changes on the fly.

This research is concerned with distributed computationally-intensive applications that use the Java™technology to operate seamlessly in a heterogeneous environment. The Java technology has been successfully applied to the domain of distributed parallel computation: heterogeneous computational grids are commonly Java-based [4]. The Java Virtual Machine (JVM) is one of the most advanced virtual execution environments ported to a multitude of different platforms.

Due to the significant efforts that have gone into optimizing the performance of the JVM, it now offers a viable platform for executing computationally-intensive applications. In fact, a new high-productivity language X10 [31] already compiles to the standard JVM classes. Other high-productivity languages under development such as Fortress [17] are likely to target the standard JVM as well. Charles *et al.* [31] claim that the performance gap between high performance Java applications and C/C++ applications will eventually

be eliminated. At any rate, we expect JVM-based programming languages to be used more widely for computationally-intensive applications in the near future.

This research explores how dynamic software updates can be implemented efficiently and flexibly for virtual machine-based CIAs. Even though the experimental platform is the JVM, the presented models and techniques are applicable for other VM-based platforms and object-oriented programming languages.

1.1 Problem Formulation

Dynamic software updates have traditionally been applied to long-running applications. Some of these applications are computationally-intensive, used to solve scientific and engineering problems. Using dynamic software updates to fix problems and add new information can shorten the time-to-discovery metrics of a CIA application, the total time it takes from posing a problem to arriving at a solution.

The computationally-intensive software development cycle often consists of three steps—development or modification of computationally-intensive software, computationally-intensive software launching process, and inspection of results [108]. This computationally-intensive software launching process involves discovering resources, allocating resources, and launching the application. The launching process often is cumbersome, time-consuming, and error-prone, thereby affecting the productivity in computationally-intensive application development and utilization. Computationally-intensive applications are typically long-running, and programmers often engage in rapid application development to shorten the time-to-solution. Hence, the need to perform perfective and adaptive updates often emerges

after a computationally-intensive application has started executing. However, making even the most trivial update to a running computationally-intensive program requires stopping the computation, changing the program, and relaunching it. Relaunching a computationally-intensive application from the beginning after performing a perfective or adaptive action wastes valuable computing resources. Hence, extensible and efficient mechanisms are required to make updates without interrupting operation. That is, while the system remains in operation, maintenance is conducted, so that the system is not functionally perturbed. Unlike the typical development cycle of computationally-intensive applications, Figure 1.1 includes the activity of dynamic software updates, which can help meet new or changed user requirements after the initial delivery of computationally-intensive applications. The programmer can perform dynamic software updates, after observing intermediate results of the system execution. Since dynamic software updates can avoid the time-consuming and complex launching phase, the maintenance of computationally-intensive applications could benefit from using dynamic software updates.

Existing approaches to dynamic software updates are limited as these approaches require the user to modify the standard virtual machine for dynamic updates [45], to observe sophisticated programming rules or component models [69], or to install additional runtime modules for dynamic software updating [33]. These approaches have attempted to address the research challenges for dynamic updating by modifying the standard Java language or virtual machine or by adding subsystems or frameworks supporting dynamic software updates. While these approaches need to provide customized JVM or runtime frameworks for every supported platform, the approach presented in this dissertation enables dynamic software updates in a portable way, without imposing any modification to the standard JVM.

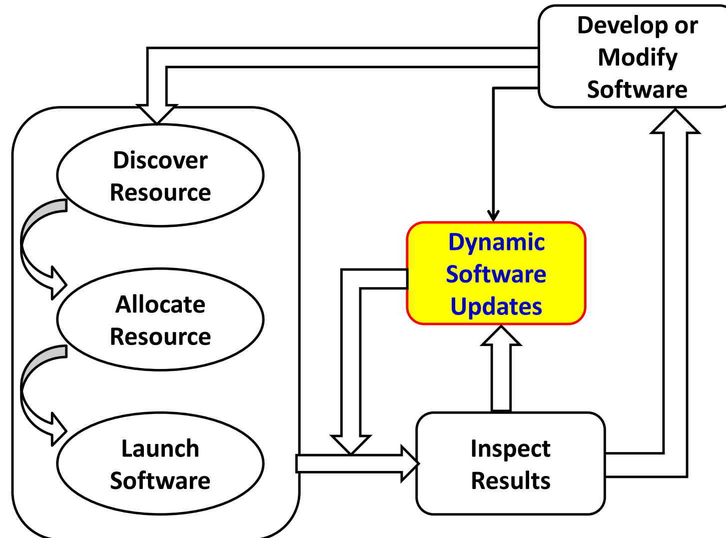


Figure 1.1: Software development of CIAs using DSU.

Among the advanced facilities provided by the JVM is the HotSwap API [7, 46]. This facility makes it possible to replace loaded classes in a running application. However, HotSwap has several limitations that constrain the kinds of changes the can be made to the updated classes. Specifically, HotSwap disallows adding new methods, fields, and constructors to the replaced classes. In addition, it also disallows changing signatures of existing methods and fields. In essence, the signature of a replaced class must remain the same, and only the method bodies can change. One consequence of this limitation is that HotSwap does not provide any support for updating the state of objects created using an older version of a class. These constraints of HotSwap substantially limit its applicability for updating classes in high-productivity languages that compile to the standard JVM. To compile a program written in such a language to regular Java classes often requires the compiler to add new methods and fields behind the scenes. Thus, even if the programmer were willing to comply with the limitations of HotSwap, the compilation model of high-productivity languages is

likely to render such compliance impossible.

To overcome the limitations of HotSwap without changing its implementation, binary refactoring techniques are used. `Binary Refactoring` [103] captures various techniques for changing the binary representation of a program without affecting its functionality. The goal of binary refactoring is to improve program maintenance, as the technique captures and provides automated tools for applying low-level transformations as a post-compilation step, without cluttering the maintained version of source code. Since most VM-based applications are developed without updating functionality, program transformation using bytecode rewriting is often used to make the VM-based applications updateable. Given a target application, a dynamic software updating system will transform the structure of the application so that the application can be dynamically updated without affecting its functionality.

A common transformation captured by binary refactoring is changing direct object references to proxy references. Such a technique has been used in prior state of the art to make it possible to profile a subset of a program's functionality [87] and to replay a program's execution [85]. A common approach for changing a direct object reference to a proxy reference is to use a binary refactoring called *Virtual Interface*. The adjective *virtual* emphasizes the fact that the introduced interface is not seen by the client program and is only used as an implementation artifact. Such Virtual Interface (VI)-based proxy patterns consist of interfaces, proxy classes, and their implementation classes. These implementation classes can be replaced at runtime while these interfaces are not changeable. Client classes communicate through the interfaces and do not need to know about their implementation classes.

To be applicable for computationally-intensive applications, however, a binary refac-

toring for introducing indirection must not impose significant performance overhead on a target program. According to our measurements on the latest JDK release (1.6), the overall overhead of the Virtual Interface binary refactoring can go as high as 44% for some benchmarks. Such an overhead can render the technique inapplicable for the domain of CIAs.

The problem statement to be addressed by this research is as follows:

a. Although dynamic software updates could significantly reduce time-to-discovery, this powerful technology has not yet been exploited in the domain of computationally-intensive applications (CIAs). Compared to applications run on a single machine, a typical CIA coordinates the execution of multiple concurrent processes running on multiple compute nodes, requiring that during updates no two divergent versions of a program be run simultaneously. Maintaining this invariant is challenging, as concurrent processors are synchronized sparingly to ensure maximum parallelism. Thus, a distributed synchronization algorithm for synchronizing dynamic software updates of parallel computationally-intensive applications must be explored.

b. The dynamic software update facilities of the JVM have limitations that significantly constrain the programmer. Furthermore, the existing binary rewriting techniques for enabling dynamic updates incur a significant performance overhead, making them unsuitable for CIAs. Thus, to realize the true potential of dynamic software updates for JVM-based applications, the dynamic update facilities' limitations must be overcome. Further, novel binary rewriting techniques must be created possessing the efficiency required to enable DSU in JVM-based CIAs.

The following specific research questions are answered in addressing the problem

statements above mentioned in this research.

a. Can a parallel computationally-intensive application be updated dynamically, while preserving correctness and without jeopardizing performance?

b. Can the HotSwap mechanism, assisted by binary refactoring techniques, improve a dynamic software updating system for virtual machine-based applications by addressing efficiency and extensibility problems without sacrificing other factors?

In these research questions, efficiency refers to the performance overhead of transformed applications, and extensibility refers to the kinds of changes supported (e.g., the change of method bodies and the addition of new fields or methods). The research focuses on models and tools for a dynamic software updating system of virtual machine-based applications, specially computationally-intensive software, by overcoming the deficiencies of HotSwap with a new binary refactoring. For this research, Java was selected as the experimental platform to establish and describe models and tools for dynamic software updating for JVM languages. However, the presented techniques should be applicable or adaptable to any VM-based platform that supports HotSwap-like dynamic code loading and bytecode engineering.

Three hypotheses are posed from these research questions, and research methods and experiments are designed and undertaken in order to test these hypotheses empirically.

Hypothesis I:

Multiple concurrent copies of a computationally-intensive application can be dynam-

ically updated while maintaining consistency of the parallel computation.

Although dynamic software updates have been applied in the domain of long-running servers, they could be applied to perfective changes of computationally-intensive applications in order to satisfy users' new or changed requirements. To carry out an efficient parallel computation, multiple concurrent tasks of a CIA should be coordinated without the tasks communicating with each other excessively. Thus, coordinating a dynamic software update of a CIA should ensure consistency with minimal use of global synchronization, which can impose an unreasonable performance overhead.

To address this challenge, this dissertation presents a distributed synchronization algorithm that coordinates concurrent dynamic updates of running processes. In addition, to apply this algorithm to an existing parallel application, special code is injected to each piece of software that executes concurrently and needs to be updated dynamically.

Hypothesis II:

A new binary refactoring using superclasses should significantly outperform prior Virtual Interface binary refactoring, alleviating the indirection overhead on each method call.

A novel binary refactoring for introducing indirect referencing that we call *Virtual Superclass* is created. Similar to Virtual Interface, the Virtual Superclass binary refactoring introduces an indirection to a target class, but it does so without incurring the often prohibitive overhead of Virtual Interface. This improved efficiency makes Virtual Superclass applicable for applications as a means of enabling efficient dynamic updates.

Virtual Superclasses are superclasses that are used internally for dynamic updating.

That is, the introduced superclass is not seen by the client program and is only used as an implementation artifact. Like a Virtual Interface, the virtual superclass does not affect the functionality of a target application. Virtual Superclass-based proxy patterns consist of proxy classes and superclasses, while Virtual Interface-based proxy patterns have proxy classes, interfaces, and implementation classes. In the Virtual Superclass-based proxy pattern, client code calls the methods of proxy classes, these calls are passed to the superclass of the proxy class, and the superclass finally handles the method call.

Hypothesis III:

The limitations of the HotSwap technique can be overcome by using Virtual Superclass binary refactoring without changing the HotSwap API or runtime libraries.

The HotSwap facility is a standard approach to dynamic software updating provided by the Java Virtual Machine in JDK 1.4. This HotSwap technique substitutes modified class code in a running application and conveniently supports class replacement without additional effort. Even if HotSwap can be used to construct dynamic software updating systems, the current implementation of the HotSwap mechanism forbids the change of the class schema (i.e., the addition of new methods or fields or the change of method signatures).

Our approach to dynamic software updates addresses the limitations of the HotSwap facilities to enable a wider range of dynamic updates (i.e., adding methods, constructors, and fields to classes as well as changing method signatures) using Virtual Superclass binary refactoring without changing the HotSwap API or adding runtime libraries. For example, HotSwap does not allow adding new methods to the swapped classes. However, the presented approach makes it possible for the programmer to add new methods to the updated classes, while using HotSwap as the swapping mechanism. The added indirection enables additional

degrees of bytecode rewriting, but it does so while fully complying with the constraints of HotSwap.

1.2 Contributions

Applying dynamic software updates to a computationally-intensive application can reduce time-to-discovery. Dynamic software updates can help adapt a running computationally-intensive application to new requirements, thereby facilitating its perfective maintenance.

The main contributions of this work are:

- **A distributed concurrency algorithm for synchronizing dynamic software updates of parallel computationally-intensive applications**

  Updating multiple concurrent tasks consistently entails receiving the update information from the user and applying the update consistently to all the running nodes. Each of these activities requires a distributed consistency algorithm. This dissertation presents a synchronization algorithm for updating parallel computationally-intensive applications. Case studies with a parallel scientific application have demonstrated the algorithm imposes a negligible performance overhead.

- **A new efficient implementation of the Proxy pattern, suitable for computationally-intensive applications**

  This contribution shows that the Proxy pattern can be implemented incurring a negligible performance overhead, and thus be effectively applied to the domain of computationally-intensive applications. The implementation employs bytecode rewriting to change di-

rect references to proxy references, which can then be used to extend the system's functionality.

- **A bytecode rewriting technique that ensures the JVM HotSwap conformance in the presence of arbitrary changes in updated applications**
  This contribution provides a bytecode rewriting approach that effectively addresses constraints of the JVM HotSwap facility using the Proxy pattern, which does not support any changes to class schemas. The approach enables the programmer to modify class schemas (i.e., adding new fields, constructors, or methods). This, in turn, enables greater flexibility in updating JVM-based systems dynamically.

This research explores opportunities for applying the state of the art in dynamic software updates to computationally-intensive applications, in areas including bioinformatics applications, scientific applications, medical image processing applications, etc. In this work, parallel programs have been considered as examples of computationally-intensive applications, and their dynamic adaptive maintenance has been investigated. This dissertation has presented flexible and efficient dynamic software updates for JVM-based, distributed, computationally-intensive applications. Case studies, including a parallelization of the Smith-Waterman algorithm and a molecular dynamics simulation, have demonstrated the presented approach is applicable for updating parallel computationally-intensive applications on the fly.

1.3 Outline of the Dissertation

In this introductory chapter, the problems that motivated this research and the contributions of this research have been discussed. Chapter 2 provides an overview of the state of the art in dynamic software updates, including comparisons of the presented approach with work previously done. Chapter 3 describes the technical research approach including models and techniques. Chapter 4 details the design and implementation of the efficient and flexible dynamic software updates as well as a synchronization algorithm and its reference implementation for updating parallel high performance applications dynamically. To demonstrate the applicability of the presented approach, Chapter 5 shows experimental results of dynamically updating a popular bioinformatics program and a scientific simulation program. Finally, Chapter 6 presents future work and draws conclusions.

Chapter 2

Background

To provide a novel approach to dynamic software updates for computationally-intensive applications, this research presents a new efficient implementation of the Proxy pattern using bytecode transformation. The Proxy pattern can be implemented incurring a negligible performance overhead, and thus be effectively applied to the domain of computationally-intensive applications. The implementation employs bytecode transformation to change direct references to proxy references, which can then be used to extend the system's functionality. Bytecode transformation is also used to overcome constraints of the standard JVM facility for dynamic software updating. The Proxy pattern, bytecode transformation, and Java bytecode transformation frameworks including ASM [13], Byte Code Engineering Library (BCEL) [3], Javassist [9, 34], and FERRARI [27] are described in Sections 2.1 and 2.2.

Prior research has described a common set of concepts or guidelines for dynamic

software updating, and its models, techniques, and frameworks could be used for other research projects. However, since dynamic software updating systems change the behavior of a running system, they are very dependent on programming languages or systems. Some dynamic updating systems require installing a specific operating system or middleware system. As the target environment of this research is the JVM, the Section 2.3 gives an overview of dynamic software updating techniques for non-JVM applications and then compares in detail the approach presented in this work with prior state of the art in dynamic software updates of Java.

2.1 Bytecode Transformation

Bytecode is an intermediate representation of a program which is designed for reducing platform dependence–the same bytecode can be executed on different hardware and operating systems. Bytecode may either run on virtual machines directly, or it may be compiled into platform-specific machine code to minimize the performance overhead incurred by bytecode interpretation. An example of bytecode is Java bytecode which is executed on Java virtual machines. Java compilers translate Java source code into Java bytecode and each bytecode instruction or opcode is one byte in length.

Bytecode rewriting techniques can be used to transform the bytecode structure of Java software. Java libraries for bytecode engineering include BCEL, ASM, FERRARI, and Javassist. BCEL provides a Java library and tools to analyze, create, and manipulate Java class files. BCEL uses a customizable class loader to instrument code at load-time. To use the bytecode manipulation of BCEL, prior knowledge of bytecode instructions is required.

ASM is a lightweight bytecode manipulation framework focusing on simplicity of use and performance. FERRARI is a recently-developed bytecode instrumentation framework which supports full coverage of bytecode manipulation including application code and standard JDK classes.

Javassist is a structural reflection framework and enables Java programs to define a new class at runtime. Specially, Javassist provides source-level APIs as well as bytecode-level APIs. Therefore, developers can edit directly class files or insert bytecode in the form of source text by using the Java language without fully understanding Java bytecode instructions. Javassist includes a simple Java compiler, and source text for class manipulation is compiled into Java bytecode, which is inlined in the appropriate place. The APIs provide the ability to insert source code at the beginning or end of a method body, to alter a method body, and to add a new method or field. Listing 2.1 shows an example of bytecode manipulation using Javassist. This example inserts new code at the beginning of the `move` method in the `Point` class. After the bytecode manipulation, the `move` method prints the values of `dx` and `dy` whenever it is called. `$1` and `$2` of the `insertBefore` method represent the first actual parameter `dx` and the second actual parameter `dy`, respectively.

2.2 Proxy Pattern

Design patterns are reusable software modules (e.g., a set of related classes) which can solve frequently recurring problems in the software engineering community. Since design patterns are well-understood and throughly tested, the programmer uses them to shorten software development time and improve the quality of a final software product. Proxies, one of the

```
1  //Before bytecode rewriting
2  class Point {
3    int x, y; void move(int dx, int dy) {
4      x += dx; y += dy;
5    }
6  }
7
8  //Bytecode rewriting using Javassist
9  ClassPool pool = ClassPool.getDefault();
10 CtClass cc = pool.get("Point");
11 CtMethod m = cc.getDeclaredMethod("move");
12 m.insertBefore("{ System.out.println($1); System.out.println($2); }");
13 cc.writeFile();
14
15 //After bytecode rewriting
16 class Point {
17     int x, y;
18     void move(int dx, int dy) {
19         { System.out.println(dx); System.out.println(dy); } //new code
20         x += dx; y += dy;
21     }
22 }
```

Listing 2.1: Insert new code at the beginning of a method body.

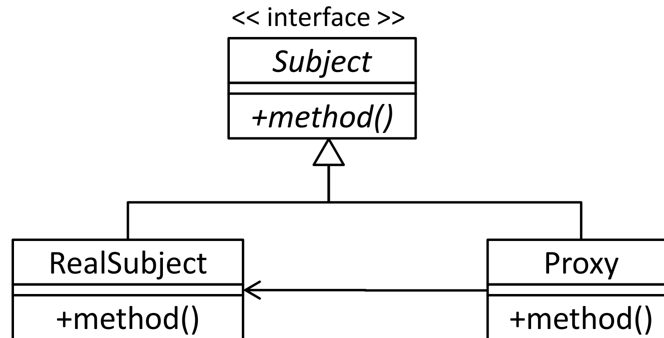


Figure 2.1: Proxy design pattern.

most common design patterns, control access to an object by introducing an indirection that interposes additional functionality, such as distribution, persistence, or profiling. Figure 2.1 illustrates the Proxy design pattern which provides a surrogate or placeholder for another object to control access to it [42]. The classes participating in this pattern are Proxy, Subject, and RealSubject. In Java, classes Proxy and RealSubject implement the methods defined in class Subject so that a proxy can be substituted for the real subject.

As a standard library of the Java language, Reflection API provides Java dynamic proxy [6] which defines a set of classes and interfaces to support a uniform interface for a list of Java interfaces. A *dynamic proxy class* implements a list of interfaces defined at runtime and method calls on an instance of the dynamic proxy class are delegated to its *invocation handler*. Figure 2.2 depicts a proxy class and invocation handler. Each proxy object has an associated InvocationHandler object. When a method is invoked on a proxy object, the invocation is dispatched to the *invoke()* method of its InvocationHandler. The *invoke* method is able to manage any method invocation by including an array of objects as its argument and returning an object as its return value. Figure 2.3 illustrates how method m1 is invoked via Java dynamic proxy. The invocations of *m1()* are reified and passed to the

InvocationHandler via its *invoke* method which invokes the actual method *m1()* of a target object.

Although Java dynamic proxy is powerful, it only provides proxies for interfaces; it does not allow creation of proxies for a class type.

To overcome this limitation of Java dynamic proxy, Eugster [37] presents uniform proxies for a Java class type and interface type. His uniform proxies generate a dynamic proxy for a class type and *access handlers* for fields of a class. A proxy class inherits a target class and the access handler has methods **get** and **set** to read and write a field, respectively. Figure 2.4 depicts a proxy class, invocation handler, and access handler.

Although Eugster's approach generates a proxy for both Java class types and interface types, his approach using a subclass cannot create proxies for **final** classes and methods because **final** constructs do not allow for overloading. Eugster describes this limitation of his approach as follows:

> *"There is no magic behind the solution to circumventing the limitations introduced by the* ***final*** *keyword. It consists in handling final classes and methods*

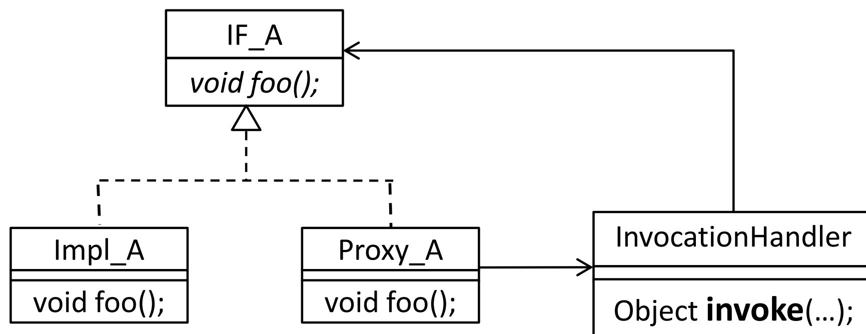


Figure 2.2: Java dynamic proxy.

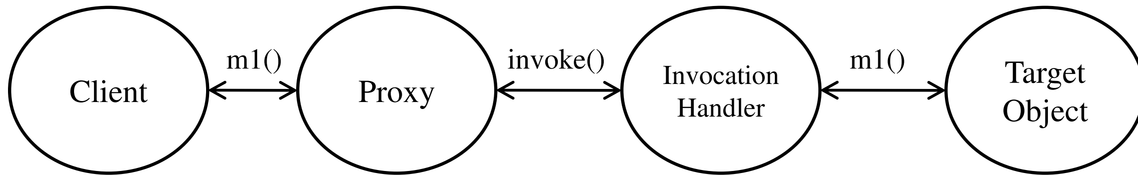


Figure 2.3: Invoking the m1() method via Java dynamic proxy.

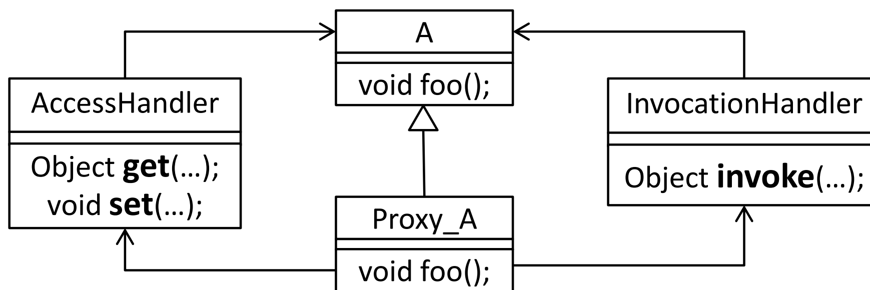


Figure 2.4: Eugster's uniform proxy.

as non-final ones when linking corresponding classes, yet keeping track of these occurrences for the verification of classes..."

A common implementation for proxies introduces an interface for a target class which is called *Virtual Interface bytecode transformation* shown in Figure 2.5. The adjective *virtual* emphasizes that the introduced interface is not seen by a client program and is only used as an implementation artifact. The Virtual Interface binary rewriting transforms a class into proxy, interface, and implementation classes and the client code never accesses the introduced "virtual" interface directly. This technique makes it possible to profile only a subset of a program's functionality [88] and to replay a program's execution [86].

However, the Virtual Interface-driven transformation implements the Proxy pattern in a way that can incur significant performance overhead on the rewritten programs. It can

incur as much as 44% performance overhead for some benchmarks.

JDK system classes are read-only in most environments for ease of deployment and portability. Further, moving a JDK class to the application space can be nontrivial due to the complication caused by the presence of native methods [104]. Thus, special care should be taken when one creates proxies for JDK system classes. Isomorphic proxies [104] depicted in Figure 2.6 can be used to create proxies for JDK system classes. In the class diagram, *pkg.A* and *upkg.A* denote a target system class and an isomorphic class, respectively. Isomorphic proxies are placed in a different package and have no relationship to their system classes. Proxy naming and package hierarchies are isomorphic to their corresponding system classes. For example, a proxy for *java.lang.String* is *upkg.java.lang.String*.

2.3 Dynamic Software Updating Systems

Dynamic software update refers to the process of changing the software code of a running program to support software evolution while the program runs. Figure 2.7 compares dynamic software updates with offline software updates. Suppose a bioinformatics program

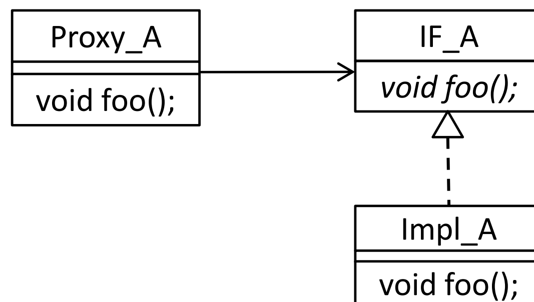


Figure 2.5: Indirecting using an interface.

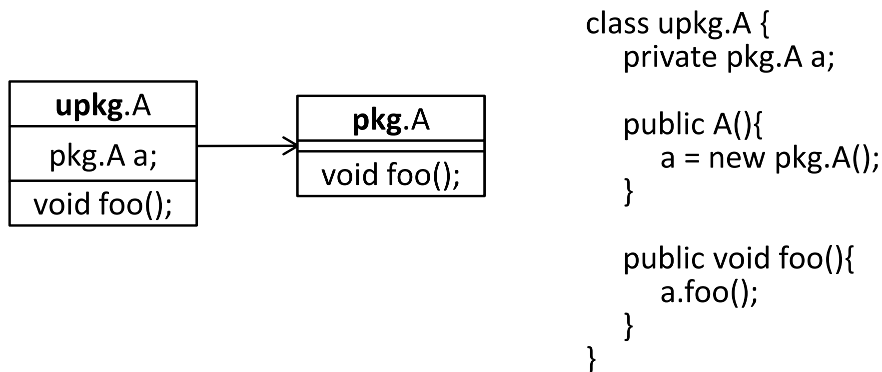


Figure 2.6: Isomorphic proxies for JDK system classes.

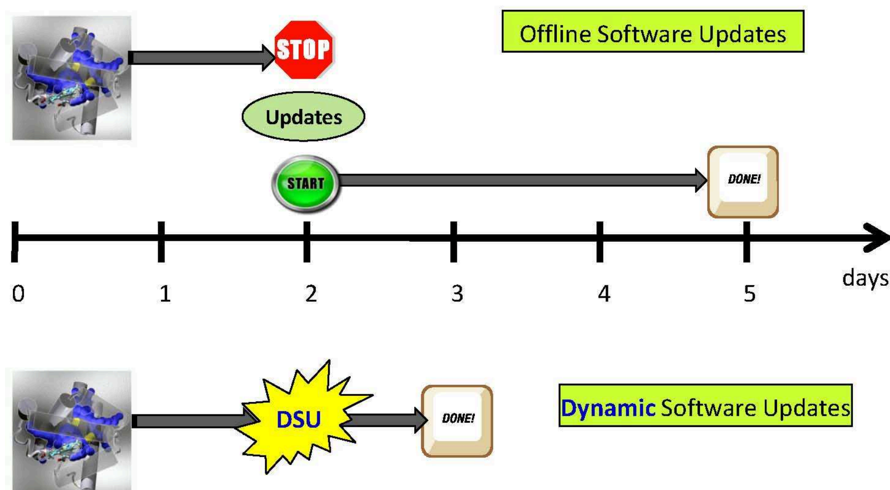


Figure 2.7: Offline software updates VS. dynamic software updates.

normally executes for `3 days` in order to arrive at a solution. What if the program has to be changed after `2 days` to meet certain requirements? If offline software updates are used, the bioinformatics application has to be stopped and then restarted after `2 days`. As a result, it takes about `5 days` to complete the computation. By contrast, since dynamic software updates does not need to stop the computation for updates, it takes about `3 days` to arrive at a solution.

2.3.1 DSU System Properties

Research on dynamic software updates hails back to the early 1980's. Many researchers have focused on the challenge of changing a running application without stopping its execution. Each shift in the design of programming languages and models has brought about a new wave of research on dynamic software updates.

The following discussion, inspired by a similar overview included in Michael Hicks' doctoral dissertation [53], presents key properties of dynamic software update systems and demonstrates how representative systems support the presented properties.

Range of updates

The *range of updates* property refers to supporting arbitrary updates without significantly constraining which programming language features can be used in the updated applications. Ideally, DSU systems should have no limitations on the update unit such as classes, procedures, and processes.

DYMOS [68] is an integrated environment which consists of a command interpreter, source code manager, editor, compiler, and run-time support system. DYMOS permits changes to module definitions, data definitions, and infinite loop bodies. Ginseng [78] supports changes to function types and the type of global variables by tracking concrete uses of functions and global variables. Since the updating facility of PROSE [79, 81] is based on HotSwap in the Sun JDK implementation, PROSE only supports updates to method bodies. Unlike PROSE, JVOLVE [101] can flexibly support schema changes such as addition and replacement of fields and methods because it uses a modified virtual machine. To support

various kinds of updates, Bialek's approach [25] automatically partitions Java applications into a set of classes or Java packages.

Robustness

The *robustness* property represents safety, well-timedness, and the ability to rollback DSU systems. Whenever a robust DSU system updates an application dynamically, the risk of the updated application crashing has to be minimized.

The DYMOS system mentioned above ensures resiliency to crash or system failure of the updated applications by enforcing type checking. A new version of a procedure is compiled within the environment used by the old version. To improve safety further, DYMOS uses update pre-conditions provided by the programmer and enforced by the system to ensure that the updated system does not become unstable as a result of an update. Argus [28] provides crash recovery facilities that interact with the Argus' process abstractions called *guardians*. A portion of a guardian's running state is stored in permanent storage. If the program crashes, a replacing guardian can be restarted using the persistent old state. Guarded Software Updating (GSU) [102] can update extremely long-running applications (e.g., satellite software running for a decade). GSU allows different versions of a program to be deployed simultaneously, so that the new version could be tested under the current runtime conditions. If the testing establishes confidence, then the system can be transitioned to the new version, using the old one as a backup accomplished through message logging, checkpointing, and rollback recovery.

JVOLVE verifies the updated bytecode for type safety and checks the running thread's activation stack to reach a safe update point, delaying the updates if necessary. POLUS [32]

can rollback committed updates to the original versions of the code and data, which are stored in memory. While using extra memory, this approach provides fast and easy rollbacks.

Performance overhead

The *performance overhead* property refers to the total amount of extra time it takes to enable dynamic updates. To be applicable for high performance applications, DSU systems should impose a negligible performance overhead.

POLUS, a DSU system for C applications, uses binary rewriting to redirect function calls between versions. The introduced function redirection may impose some performance overhead. However, Chen *et al.* claim that such overhead is minimal (less than 1% for most applications) [32]. PROSE updates Java applications dynamically with a negligible run-time performance overhead, thanks to the aggressive inlining of the existing method calls in the replaced method bodies. JVOLVE uses a custom virtual machine, providing a highly efficient DSU service with virtually no performance penalty.

Ease of use

The *ease of use* property refers to the simplicity with which a DSU system can be used to apply dynamic updates. Ideally, the update processes should be transparent to the programmer, with most of the functionality completely automated. That is, programmers should not have to write any dynamic update code, and the development process should be separated from the dynamic update process. Segal and Frieder have observed that the degree to which a particular DSU system is used is directly proportional to its transparency [95].

To minimize user involvement, Bialek's system automatically adds dynamic update functionality to the applications. POLUS also needs only minimal update information from programmers and automatically generates update patches. Furthermore, to provide good usability, POLUS enables patch processes to be visible to programmers. Ginseng features a compiler and tool suite for constructing updateable C applications from programs written without dynamic updates in mind—it produces updateable programs dynamically and generates dynamic patches automatically.

Portability

The *portability* property refers to the ability to perform dynamic updates on multiple platforms. Some dynamic updating systems only work on specific operating and/or middleware systems.

Dynamic C++'s approach [55] requires no special preprocessor or compiler support. Its lightweight proxy classes can be compiled by any standard C++ compiler.

On the other hand, some DSU systems need comprehensive infrastructure support such as a custom operating or middleware system. Hauptmann *et al.*'s approach [52] leverages certain properties of the Chorus operating system, including dynamic process loading, port migration, and thread scheduling. The Eternal system [75] dynamically updates CORBA-based systems, thus being portable across CORBA-enabled platforms.

Kang *et al.* [59, 60] adapt HPC programs, in Fortran, C, and C++, for new requirements. Their approach leverages the function interception capabilities of a framework that operates at the assembly language level, thus ensuring programming language independence.

Some Java DSU systems, including JDrums [11], Jvolve, and Dynamic Virtual Machine (DVM) [73], use custom JVMs. The portability of these approaches is thus constrained by the necessity to provide a customized JVM for every supported platform.

Multi-threading support

The *multi-threading support* property refers to the ability to safely update a program that has multiple concurrent execution threads.

DYMOS can update multi-threaded applications due to its design, which uses the multi-thread variant of the Modula language called StarMod; this particular dialect makes it easier to synchronize those DYMOS function calls that access global structures. The design of Argus naturally supports the replacement of *guardians*, which are groups of distributed, multi-threaded processes.

To update multi-threaded C programs efficiently, Ginseng improves on barrier-style synchronization. Its extension, Stump [76], further facilitates safe and timely updates of multi-threaded programs by enabling the programming to easily reason about the safety of updates. To that end, Stump takes programmer's input in the form of safe program update locations and calculates an extended set of such locations. It then relaxes the built-in properties of synchronization constructs to ensure that threads not be blocked at these update locations.

Jvolve can safely update multi-threaded programs by ensuring that all the threads have reached a safe state, in which updated methods are no longer allocated on the runtime stack. If a safe state cannot be reached, the updates are postponed until a later time.

Distribution support

The *distribution support* property refers to the ability to update distributed applications, whose execution spans multiple address spaces, possibly separated by a network.

Some systems dynamically update distributed applications by using custom middleware. PolyLith [57] runs C programs on top of a special reconfiguration-enabled runtime system. The Conic [63] distributed programming system coordinates multiple distributed processes. The provided entry points, called channels, are used for inter-process communication, and a configuration manager can redirect the channels on the fly. JDRUMS provides a special communication layer that enables accessing a remote JVM through JINI using RPC. The JD reconfiguration tool coordinates distributed components using a JINI communication protocol. To adapt HPC applications without degrading their performance, Kang *et al.*'s approach injects adaptive code at the existing global synchronization points (e.g., where MPI_Bcast and MPI_Barrier are invoked).

Language constructs support

The *language constructs support* property refers to accommodating various programming language constructs during dynamic updates. Of interest to this discussions are various language constructs found in C, C++, and Java.

PODUS, PolyLith, On-line Software Version Change (OSVC) [49], and Ginseng are specifically designed for dynamically updating programs written in C. Dynamic C++, Eternal, and Chorus aim at dynamic updates of C++ applications. Argus, Conic, Erlang [105], Dynamic ML [47], and DYMOS work with custom versions of various languages that facili-

tate dynamic updates.

The rapid, widespread adoption of Java technology has served as an impetus for creating many DSU systems for updating Java applications on the fly. Orso *et al.*'s technique [88] refactors bytecode to enable dynamic updates. Bialek *et al.*'s system also rewrites the updated software at the source or bytecode levels to enable dynamic updates. Several approaches [45, 73, 93] have introduced custom virtual machines to support dynamic updates of Java applications. Some approaches [26, 33, 69, 106] introduce new languages features, middleware systems, or require that software developers abide by specific component models or programming rules. Bierman *et al.*'s UpgradeJ [26] is a Java-like language for type-safe upgrading classes dynamically.

2.3.2 Dynamic Software Updates of Java Applications

Previous work on dynamic updating of Java software can be classified into five groups based on their implementation techniques: Java HotSwap, program transformation, custom virtual machines or runtime libraries, specific programming models, and extension of the Java language like Aspect-Oriented Programming (AOP). Some approaches using program transformation have attempted to use the standard virtual machine and Java language, while others use custom virtual machines and AOP, additional runtime libraries or extended language constructs.

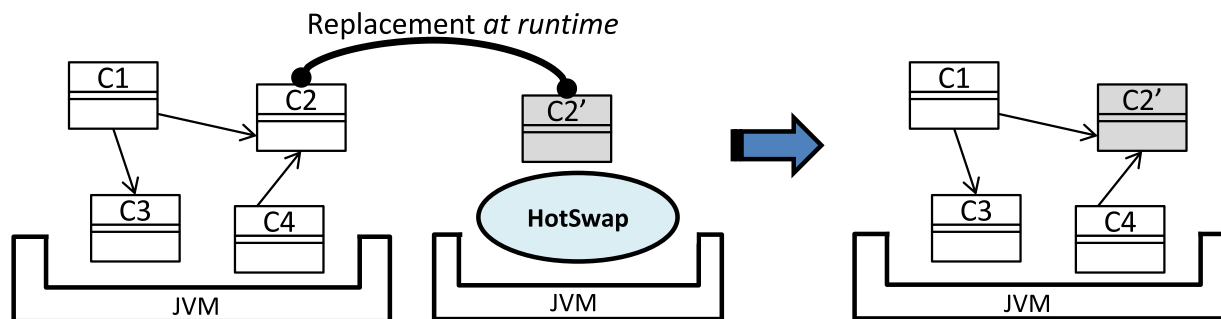


Figure 2.8: JVM HotSwap facility.

HotSwap

The HotSwap mechanism (added in Java 1.4) is a feature that encapsulates the ability to substitute modified code in a running application. One can replace the classes in a running JVM on a physically remote machine. An already loaded class can be replaced with another binary compatible version of that class. The restriction of the current HotSwap feature is that it does not support schema changes to a target class, such as addition of new methods, signature change of the existing methods, or addition of new fields.

Figure 2.8 shows how the JVM HotSwap reloads class *C2'* on the fly. The replaced application with four classes and the HotSwap program with a newer version *C2'* execute on two different JVMs; one executing the target application needs to start with the appropriate debugging options, and the other where HotSwap runs is connected to the JVM with the target application. The rightmost part depicted in Figure 2.8 shows the updated application includes the new version *C2'*.

Program Transformation

Program transformation refers to a technique to change the software structure without affecting its functionality. A typical approach to dynamic software updating using program transformation is based on Java interfaces, which have public method signatures without their implementation bodies. As an initial work, Liang and Bracha [70] have introduced dynamic class loading that allows Java classes to be reloaded without restarting Java applications. This approach suggested Java reflection [8] or interface facilities for dynamic class loading. The idea of this approach is to use separate namespaces for different versions of classes in order to allow those classes to be reloaded.

However, program transformation using reflection or interfaces may not be applicable for computationally-intensive applications since the frequent usage of reflection or interfaces leads to performance degradation. Orso *et al.* [87] have proposed a software-based technique for dynamic updating of Java software, permitting replacement, addition, and deletion of Java classes at runtime. This approach involves two steps-transformation and dynamic update. The transformation step allows Java applications to be swapping-enabled by splitting them into wrapper, implementation, interface, and state classes.

The replacement of a target class is performed in the step of the dynamic update. When a newer version of a class is available, a corresponding implementation class is generated and is replaced during dynamic updates. A special swap-request method in a wrapper class is called, and the new version of the implementation class is passed as an argument. The wrapper performs the dynamic update by iterating through the instances vector that stores references to all instances of the target class.

However, this structural transformation imposes indirection overhead on each method call, and can incur significant performance degradation. According to our measurements on the latest JDK release (1.6), the overall overhead of the Virtual Interface binary refactoring can go as high as 44% for some benchmarks. Furthermore, this approach is not flexible since it does not allow modifying public methods of interfaces.

Bialek [25] suggested an approach to dynamic software updates of existing Java applications by partitioning applications and using the partitions as update units. Java software can be updated dynamically by inserting a software module, called a `dynamic update functionality`, at source code or bytecode level. He claimed the dynamic update functionality allows replacing update units of an application while running and attempted to provide an approach to dynamic software updates which focuses on flexibility, simplicity, and transparency design criteria.

Custom Virtual Machines or Runtime Libraries Required

Several approaches [45, 73, 93, 94] have introduced custom virtual machines to support dynamic software updating. They provide their own virtual machines to overcome limitations of current standard JVMs by adding new classes or modifying standard JVM classes. The major weaknesses of these approaches are 1) users must install the custom virtual machines, 2) the custom virtual machines could limit other functionality of the standard JVM, and 3) the custom virtual machines might not interoperate with the standard JVM. Chen *et al.* [33] have proposed an extended Java RMI to manipulate dynamic dependencies between objects in a distributed computing environment. Java Distributed Run-time Updating Management System (JDrums) [93] uses a modified virtual machine to allow class updates during exe-

cution and version management of all installed classes, and it provides tool support for the auto generation of update code.

Specific Programming models or Frameworks

Some approaches [33, 69] have used specific programming languages or middleware systems for dynamic software updating. These approaches force software developers to abide by specific component models or programming rules for dynamic software updating. These approaches, therefore, cannot work for general Java applications. A component framework presented in [69] contains a reconfiguration management subsystem for dynamic reconfiguration. Normal Java classes must be transformed into components with an interface so that they can be reconfigured at runtime. Warth *et al.* have presented Expanders [106], a new programming language construct that specifies new methods, fields, and interfaces. Expanders enable the programmer to express new methods and fields to be added to an existing program statically. Bierman *et al.* [26] have introduced UpgradeJ, which is a Java-like language with support for type-safe dynamic class upgrading. UpgradeJ allows classes to be updated dynamically with newer versions, permitting co-existing versions of classes and the declaration of a class with explicit version numbers. The class upgrades of UpgradeJ include the extension of classes, the revision of existing subclasses, and the evolution of classes. However, the implementation of UpgradeJ is underway, and the authors plan to implement a prototype based on Java using JVM and HotSwap.

Aspect-Oriented Programming (AOP) for Dynamic Software Updating

Aspect-Oriented Programming (AOP) [62] is a programming paradigm that promotes the principle of separation of concerns. AspectJ [1, 61] is an aspect-oriented extension to the Java programming language that provides convenient constructs for supporting separation of concern concepts.

While AspectJ provides compile-time and load-time weaving, dynamic AOP systems including JBoss AOP [10], PROSE [80], and AspectWerkz [2] support runtime weaving. Dynamic AOP systems support a process for unanticipated changes to a running system. However, their techniques are less mature than AspectJ and have some limitations of interoperability, flexibility, and performance.

AOP-based approaches to dynamic software updates consider dynamic software updating as a crosscutting concern. These approaches often support two steps of dynamic software updates: the first step is to make a program adapt-ready, and the second step is to enact dynamic software updating. Like the program transformation, AOP-based approaches need to insert dynamic update modules, usually aspects, into a target application before the application is executed [35, 50, 90, 107]. Previtali *et al.* [90] have presented an approach to support dynamic software evolution based on aspect-oriented programming. They describe how to use the pointcut and the weaving technique of AOP for the support of runtime evolution. Their approach to runtime evolution consists of two steps. At first, to determine version differences, the original and new versions of a software system are compared. As the second step, a list of update actions and pointcuts are constructed based on the first step. Aspects for dynamic software updating are generated at compile time and are woven with relevant application classes. Yang *et al.* [107] have demonstrated how AOP can be used to support

Criteria	Orso[87]	Bialek [25]	Malabarba[73]	Lee[69]	Previtali[90]	This work
Use of Standard						
-Standard virtual machine	+	+	-	+	-	+
-HotSwap	-	-	-	-	-	+
-No coding constraints	+	+	+	-	+	+
-No runtime library required	+	+	-	-	-	+
Extensibility						
-Addition of new fields/methods	-	+	+	+	+	+
-Update of fields/methods	-	+	+	+	+	+
-No source modification required	+	+	+	+	+	+
Efficient code	-	-	+	+/-	-	+

Table 2.1: Comparison to related work on dynamic software updating for Java software(supported:+, unsupported:-, and partially supported:+/-).

dynamic adaptations. Their dynamic updating system makes a program adapt-ready, and then achieves dynamic adaptation. At software development time, a core program and aspects are developed separately and the aspects are woven with the core program at compile time. At runtime, the adaptation kernel performs a dynamic adaptive process according to the pre-defined rules. Gustavsson *et al.* [50] have viewed the possibility for runtime evolution as an aspect and have implemented such an aspect in AspectJ. This approach introduces the empty interface *Versionable*, and classes need to implement this interface to be prepared for updates. Aspects weave the code for runtime evolution with those classes.

Table 2.1 compares the approach presented in this research with closely related work. While these approaches to dynamic software updates are powerful and effective, none of

them have been applied to computationally intensive applications such as scientific and bioinformatics programs.

2.3.3 Dynamic Software Updates of Non-Java Applications

Platform-independent and generalized approaches to dynamic software updating systems adopting software architectures or adaptation models have been provided [43, 44, 64, 65, 83].

Garlan and Schmerl [44] have introduced model-based adaptation. In this approach, externalized adaptation supports a kind of closed-loop control system paradigm. The outside modules are responsible for monitoring the target system. When they notice the system parameters are above the predefined thresholds, they attempt to adapt the system. The system model is an abstract representation of a system e.g. performance models and architecture models. Garlan and Schmerl also have described four major steps; Monitoring, Interpretation, Resolution, and Adaptation. Monitoring collects runtime data from the running system and Interpretation analyzes low-level monitored information to find a problem. Resolution repairs the system and Adaptation causes the adaptation to occur in the running system without stopping or restarting it.

Oreizy *et al.* [83] have described how to apply software architectures for runtime change of software systems. They claimed an architecture-based approach to runtime software evolution provides unique benefits, such as a common representation of runtime changes, separation of computation from communication, and encapsulation of change application policies. A tool suite for supporting the runtime modification of software architectures, called *ArchStudio*, was presented.

Performance management of a running software system is a key area using dynamic software updating techniques [30, 67, 74]. Generally, a reconfiguration manager dynamically updates the configuration of a system according to the performance of the system. To measure the performance of the system, response time, throughput, and utilization are monitored. Caporuscio *et al.* [30] have presented a model-based system for reconfiguration for dynamic performance management. They proposed a framework to manage performance of software systems at runtime based on monitoring and model-based performance evaluation. According to the predefined performance threshold, the framework reconfigures the topology of components to maintain performance.

In addition, autonomic systems can benefit from using dynamic software updates. Autonomic computing [15, 44] is a type of computing model in which the system is self-healing, self-configured, self-protected, and self-optimized. The essence of autonomic computing systems is self-management, the intent of which is to free system administrators from the details of system operation and maintenance and to provide users with a machine that runs at peak performance 24/7. Dynamic software updates are applicable to implement self-healing or self-configured systems by dynamically adding, removing, and replacing parts of systems.

Neamtiu *et al.* [54, 77] have presented a dynamic software updating implementation called *Ginseng* for C applications. Ginseng consists of a compiler, a patch generator, and a runtime system; the Ginseng compiler generates an updateable executable and checks if changes to a running program violate type-safety, the Ginseng patch generator automatically creates patch code representing differences of two versions of C software, and the Ginseng runtime system applies the changes to the program when a safe update point has been

reached.

Hjálmtýsson and Gray [55] have introduced an upgrade technique to dynamically add new code to a running C++ application using existing language features without language extensions or runtime libraries. They have proposed dynamic C++ classes with an abstract interface and one or more implementation classes. This approach is a Virtual Interface-based Proxy pattern, which is also used for the Java language [87]. New code is inserted into a running system by replacing the implementation class, and this approach allows the coexistence of older and newer objects until the older objects are destroyed.

Rasche *et al.* [92] have presented an approach to reconfigure multithreaded Microsoft .NET applications using a dynamic aspect weaving tool, called LOOM.NET. This approach creates proxies dynamically for all public interfaces of a component at load-time. Those proxies intercept the method calls of a client and control their execution. A reconfiguration algorithm is implemented within aspect code using a readers-writer lock and the state of all updated components is transferred from the old to the new version.

Ajmani *et al.* [16] have proposed an upgrade system for large-scale, long-lived distributed systems. They introduced gradual, controlled update techniques for distributed systems with multiple nodes by defining scheduling functions to control upgrade deployment and transform functions to transform persistent state.

K42 [100] is an operating system that supports online reconfiguration of active components using interposition and hot-swapping; interposition wraps the interfaces of an active component and adds new functionality, and hot-swapping replaces an active component with a new component instance.

Chapter 3

Technical Research Approach

Prior research has addressed dynamic software updating of JVM-based applications by using program transformation, custom JVMs, proprietary programming models, or aspect-oriented programming [25, 69, 73, 87, 90]. Unlike those approaches, this research is intended to focus on the efficiency and extensibility of dynamic software updating of VM-based computationally-intensive applications using binary refactoring and HotSwap without additional runtime libraries.

More detailed technical approaches and issues of this presented dynamic software updating are described in this chapter:

- The overall approach to dynamic software updating
- An efficient binary refactoring
- An approach for overcoming the constraints of HotSwap
- Software tools for experiments and development of a software prototype

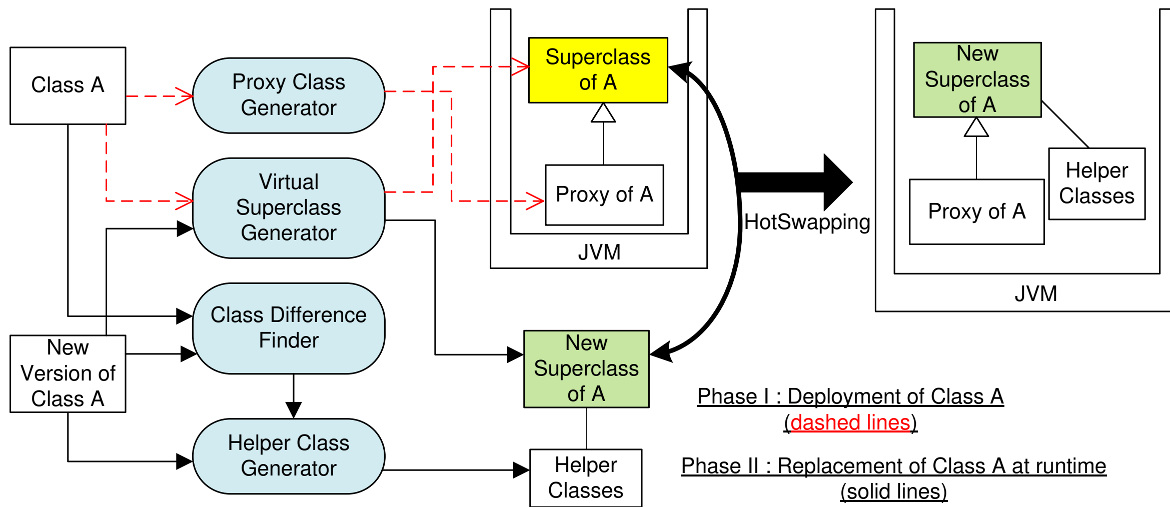


Figure 3.1: Overall flow of the approach to dynamic software updates.

3.1 Updating VM-based Applications Dynamically

The modules for binary refactoring and HotSwap are key components of the presented dynamic software updating system. At the first phase illustrated in Figure 3.1, the updating system refactors all the loaded classes at the bytecode level and generates their corresponding virtual superclasses. The virtual superclasses have actual methods implementing application-specific logic and are swapped by the updating system. Therefore, the output of the bytecode rewriting is updateable software, which is structurally different from the original version and can be deployed on a virtual machine. When the initial program is changed, the programmer gives the changed classes as input to the updating system, which refactors them into virtual superclasses and special helper classes. HotSwap can then replace older class versions of virtual superclasses with newer versions, as they have the same schema. Helper classes make the updates conform to the HotSwap API when new methods or fields are added. The new members are added to helper classes, so that the signatures of virtual superclasses remain

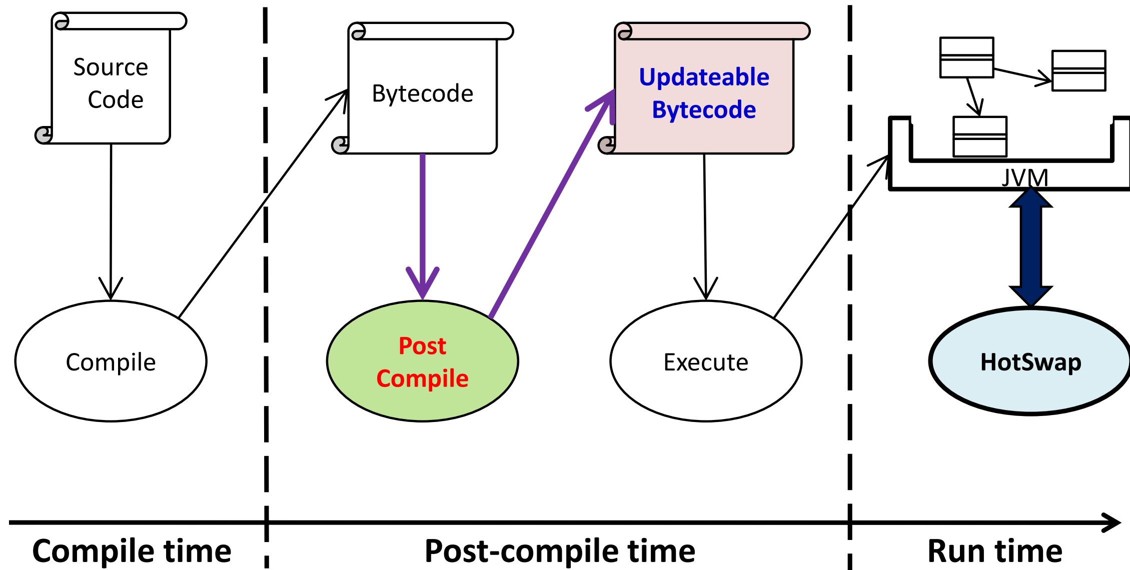


Figure 3.2: Creating updateable applications.

the same.

Figure 3.2 illustrates the phases for dynamic software updates including compile, post-compile, and run time. At compile time, the programmer compiles source code without assumption of dynamic software updates. At post-compile time, the presented dynamic update system enables the programmer to inject the update management code into bytecode. The bytecode is transformed into an updateable bytecode. Finally, at run time, updates are applied to the updateable application using HotSwap.

To manage class versions, there are three approaches to object update models. The first approach does not allow the coexistence of older and newer versions at any point, and the creation of new instances from a newer version is blocked until all old objects have expired. After the updating, only new objects from the newer version are alive. The second approach updates all old instances defined by an old version at the given updating point. Each object's state needs to be transferred to an instance of a new version to keep the integrity of

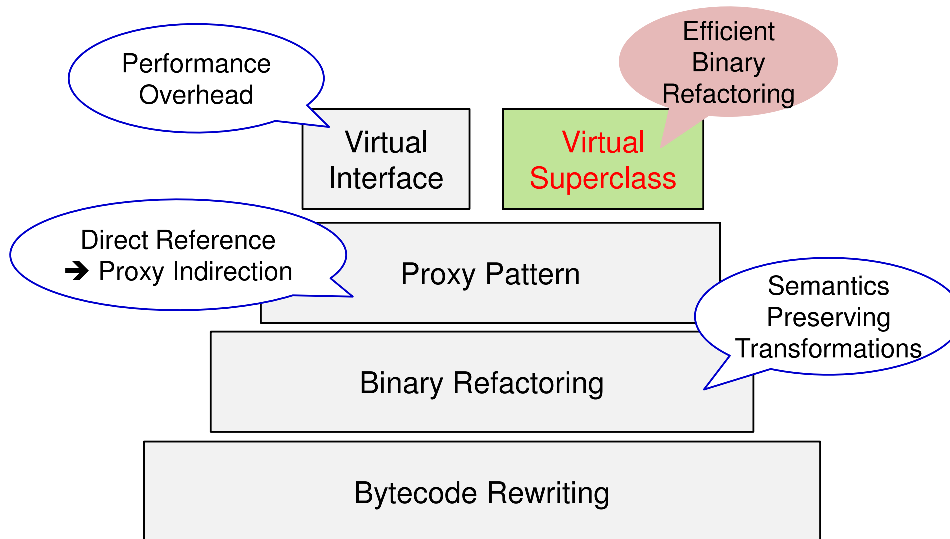


Figure 3.3: Bytecode transformation.

an application. In this approach, to find the proper updating point is non-trivial, and if old and new instances have a different data structure, the state transfer should be performed carefully. The last approach allows for the coexistence of old and new instances. All new instances of a new version are created regardless of existing old objects of an old version. These old objects are alive with the new objects. When a new version has a new field, a helper class manages the new field for old objects because the old objects do not have that field. This approach can be implemented efficiently [55] using a standard JVM and Java features, therefore we have chosen to use this approach.

3.2 Virtual Superclass Binary Refactoring

Figure 3.3 shows a hierarchy of bytecode transformation used to provide efficient and flexible dynamic software updates in this work. Bytecode rewriting refers to a technique to

manipulate bytecode that allows the programmer to alter the semantics of program. The process of changing a software system that improves its internal structure without altering its external behavior at the source level is called `refactoring` [103]. `Binary refactoring` alters the binary representation of a program without affecting its functionality (i.e., semantics preserving transformation). Application domains of binary refactoring include Proxy pattern, Split class and removal of indirection. To avoid unnecessary network traffic in a distributed environment, the Split Class binary refactoring sends only the field required by a server by splitting a class. The Proxy pattern can be used for profiling and testing by changing a direct object reference to a proxy indirection. The state-of-the art binary refactoring for Proxy indirection is `Virtual Interface binary refactoring`. However, since this approach shows significant performance overhead, it is not applicable for dynamic software updates of computationally-intensive applications. To address the performance overhead of Virtual Interface binary refactoring, `Virtual Superclass binary refactoring` is presented in this research. Next, Virtual Interface binary refactoring and Virtual Superclass binary refactoring will be explored in detail.

With the increasing use of computationally-intensive applications in scientific and engineering disciplines, there is a need for more efficient binary refactoring techniques for the maintenance of computationally-intensive applications. One of major challenges of this research is to improve the performance overhead of existing Virtual Interface binary refactoring [55, 84, 85, 87], which has been used for dynamic software updating systems of object-oriented applications.

Figure 3.4 shows an original and transformed method call. Virtual Interface binary refactoring consists of a proxy class, an interface, and an implementation class. The im-

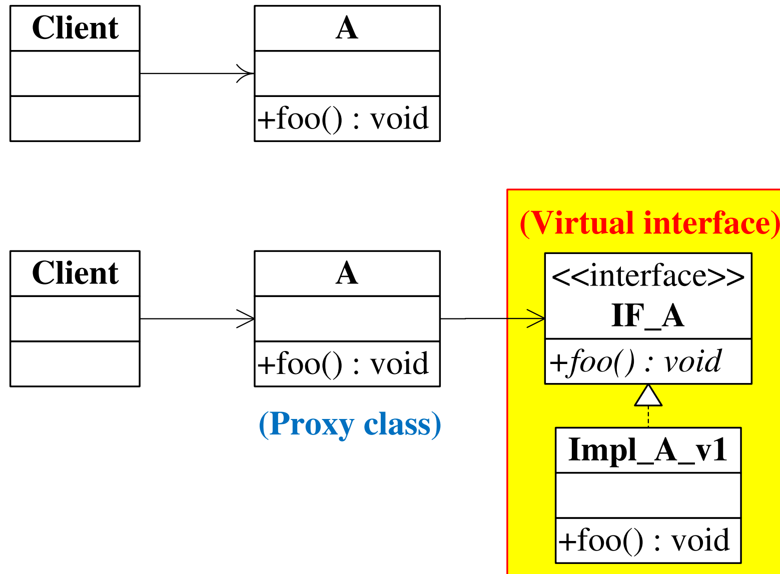


Figure 3.4: Virtual interface binary refactoring.

plementation class has actual methods representing application logic. The proxy class has a set of proxy methods, which have exactly the same signature as these actual methods, but different implementations while the interface has the same signatures as those actual methods without implementations. All method calls to proxy methods from client classes are passed to corresponding actual methods of implementation classes through interfaces. Thus, significant performance overhead is imposed on each method call. This Proxy pattern has been used for dynamic updates for object-oriented programming languages, such as C++ and Java, permitting the replacement of implementation classes at runtime. One of the limitations of this pattern is that it does not allow changes to the public methods of an interface.

In Listing 3.1, three classes *VI_A*, *Impl_A*, and *Proxy_A* have a public method *foo*. When a client class calls the *foo* method of *Proxy_A*, this call is passed to *VI_A*. Finally, the *foo* method of *Impl_A* is executed.

```
1  interface IF_A{
2    public void foo();
3  }
4
5  class Impl_A implements IF_A{
6    public void foo(){
7      System.out.println("I'm foo.");
8    }
9  }
10
11 class Proxy_A{
12   IF_A iFace = ..;
13   public void foo(){
14     iFace.foo();
15   }
16 }
```

Listing 3.1: Example code of Virtual interface binary refactoring.

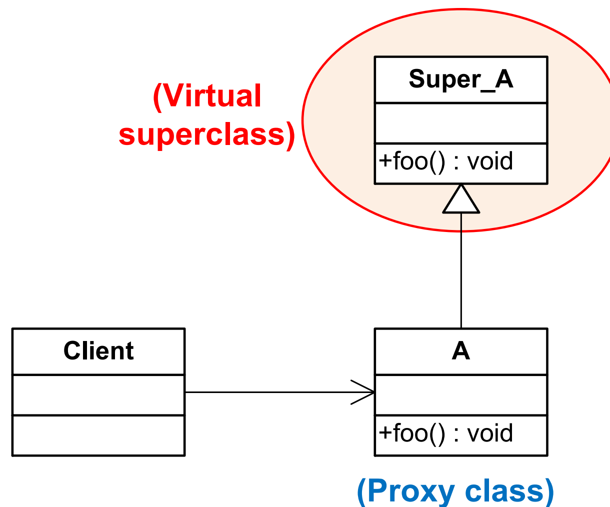


Figure 3.5: Virtual Superclass binary refactoring.

Figure 3.5 illustrates the Virtual Superclass binary refactoring, which overcomes the performance overhead of the Virtual Interface binary refactoring. The proxy class depicted in Figure 3.5 plays the same role as that of the Virtual Interface binary refactoring. The superclass has actual methods like the implementation class of the Virtual Interface binary refactoring and can be replaced with a newer version by HotSwap.

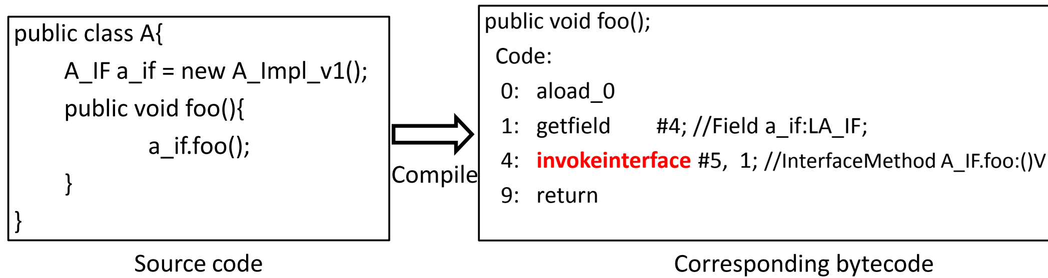
The implementation of the Virtual Superclass binary refactoring relies on the sophisticated optimization capabilities of modern JVMs. Among the advanced optimization facilities that they provide is the ability to inline delegating method calls, if the delegation does not involve dynamic dispatch [1].

Method calls are translated into one of the four bytecode instructions `invokespecial`, `invokestatic`, `invokevirtual`, and `invokeinterface` [72]. `invokespecial` is used to call private methods, superclass methods, and object constructors and `invokestatic` is used to invoke static methods. Since both bytecode instructions are statically linked, they can be inlined. By contrast, `invokevirtual` and `invokeinterface` cannot be inlined because they can be involved in dynamic method dispatch. `invokevirtual` is used for virtual calls and `invokeinterface` is only used to call methods defined in interfaces.

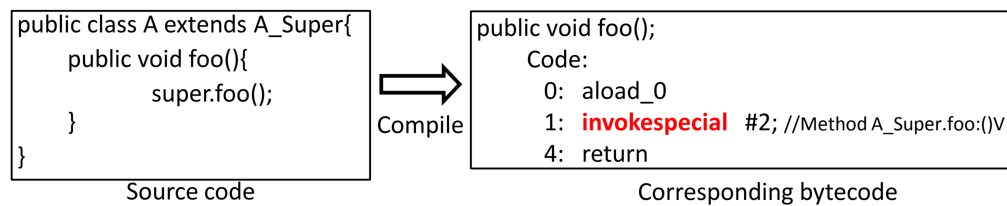
In Listing 3.2, the call to `super.foo` is translated into the `invokespecial` bytecode instruction, reserved for invoking constructors and methods in superclasses. Modern JVMs can effectively inline this call, completely eliminating any indirection overhead in most cases. By contrast, Virtual Interface uses the `invokeinterface` instruction in its im-

---

[1] When *obj.m*() is called, the actual implementation of *m*() depends on the runtime type of the receiver object *obj*.



(a) *invokeinterface* bytecode instruction



(b) *invokespecial* bytecode instruction

Figure 3.6: `invokeinterface` and `invokespecial` bytecode instructions.

plementation. While the performance of `invokeinterface` has been improved significantly in modern JVMs [18], this instruction implements a form of dynamic method dispatch, and as such cannot be safely inlined. Thus, it is the underlying invocation instructions used by Virtual Interface and Virtual Superclass that explain the inherent differences in their performance characteristics. Figure 3.6 shows source code and its corresponding bytecode code –`a_if.foo()` and `super.foo()` are translated to `invokeinterface` and `invokespecial` bytecode instructions, respectively.

Listing 3.2 shows a superclass *Super_A* and its subclass *Proxy_A*. Both classes have a public method *foo*, and method calls on the *foo* method of *Proxy_A* are delegated to its superclass *Super_A*.

```
1   class Super_A{
2     public void foo(){
3       System.out. println ("I'm foo.");
4     }
5   }
6
7   class Proxy_A extends Super_A{
8     public void foo(){
9       super.foo();
10    }
11  }
```

Listing 3.2: Example code of Virtual Superclass binary refactoring.

3.3 Translating Updates for HotSwap Conformance

The HotSwap limitations are particularly constraining due to the nature of compiling languages other than Java to run on a standard JVM, the approach used by high-productivity languages such as X10. When such compilation takes place, one cannot assume a one-to-one correspondence between a source file and a corresponding class file.

To further clarify this point, consider the implementation of inner classes in the Java language. Rather than changing the JVM, the Java language team implemented this new language construct by compiling inner classes to standard JVM classes, an approach used for the new language constructs in VM-based high-productivity languages. Listing 3.3 is an example of adding an inner class `I` to class A.

From the programmer's perspective, no new methods were added to class `A`. Nevertheless, the Java compiler adds a `synthetic` method `access` to class `A` to allow the inner class access to the non-public fields of its enclosing class. Thus, using HotSwap to update

```
1  class A
2  {
3    private int i;
4    class I
5    {
6      void bar() { int j = i; }
7    }
8  }
```

Listing 3.3: Adding an inner class “I” to class “A”.

class `A` will fail due to the limitation of not allowing the addition of new methods. It is highly likely that the translation of new language constructs in high-productivity languages such as X10 will require the compiler to add such `synthetic` methods and fields.

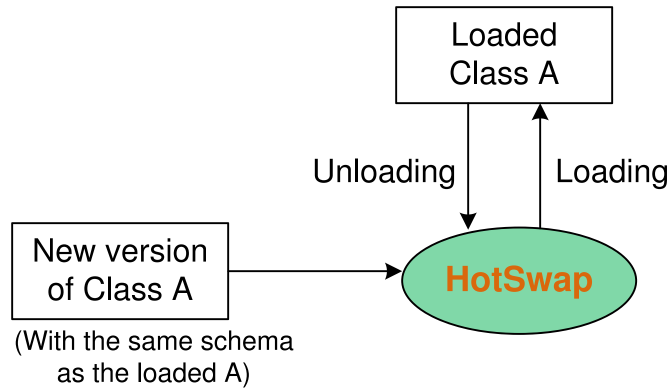
Figure 3.7 summarizes the constraints of the JVM HotSwap. HotSwap disallows any changes to the signature of a class. The swapped class has to contain the same set of methods and fields as the currently deployed version, and the only kind of change allowed is changing method bodies.

3.3.1 Structural Changes to Class

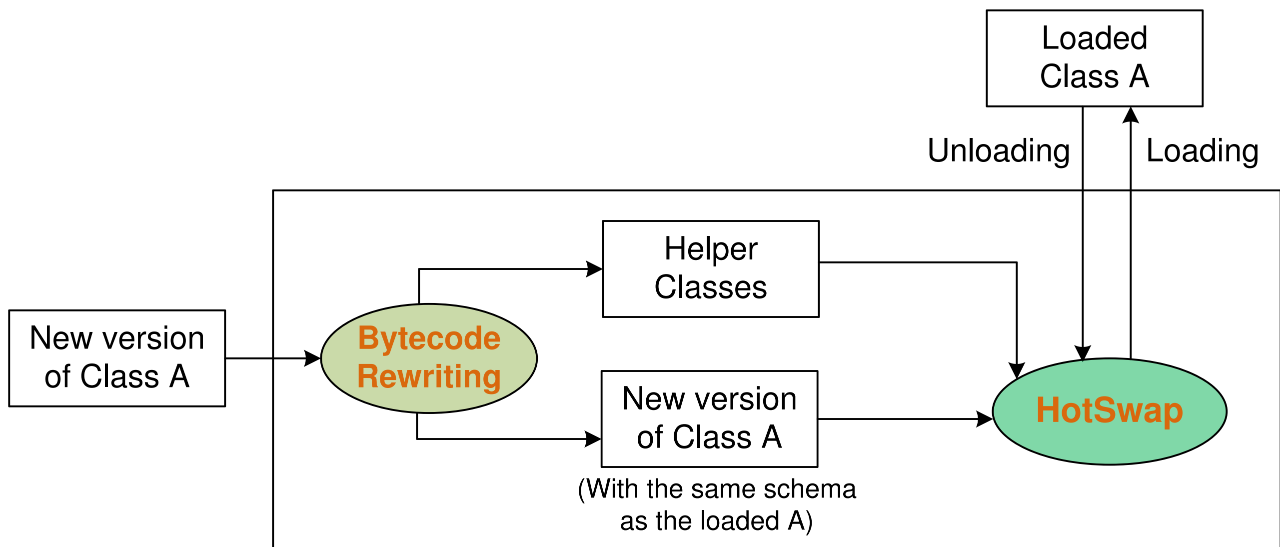
In Figure 3.8, (a) is the typical HotSwap, which can successfully swap a class with the same schema as a loaded class, and (b) is the HotSwap assisted by binary refactoring, which is intended to overcome the limitations of (a). The bytecode rewriter generates Virtual Superclasses, proxy classes, and helper classes to overcome the constraints of HotSwap. As seen in Figure 3.8 (b), the newer version of class `A` as an input could have a different schema from the loaded class `A`. The bytecode rewriter translates this class having a different schema

Targets of Change	Limitations of Java HotSwap	Exceptions
Method	M1: Addition of a new method	UOE: add method not implemented
	M2: Removal of an existing method	UOE: schema change not implemented
	M3: Addition of formal arguments of a method	UOE: delete method not implemented
	M4: Removal of formal arguments of a method	UOE: delete method not implemented
	M5: Change to the return type of a method	UOE: delete method not implemented
	M6: Change to method modifiers	UOE: changes to method modifiers not implemented
Field	F1: Addition of a new field	UOE: schema change not implemented
	F2: Removal of an existing field	UOE: schema change not implemented
	F3: Change to the type of a field	UOE: schema change not implemented
	F4: Change to field modifier	UOE: schema change not implemented
Class	C1: Addition of a new superclass	UOE: hierarchy change not implemented
	C2: Removal of an existing superclass	UOE: hierarchy change not implemented
	C3: Addition of a new interface	UOE: hierarchy change not implemented
	C4: Removal of an existing interface	UOE: hierarchy change not implemented
	C5: Change to class modifiers	UOE: changes to class modifiers not implemented

Figure 3.7: Limitations of HotSwap Technique (UOE is the abbreviation for `java.lang.UnsupportedOperationException`).



(a) HotSwap disallowing schema change of classes



(b) HotSwap supporting a full range of dynamic updates

Figure 3.8: HotSwap Models.

for HotSwap conformance and generates helper classes to enable programmers to replace this class flexibly using the following features:

Adding new methods/constructors The addition of a new method is a typical example of the schema change of a class—it is necessary for changing the behaviors of a class or

```
1   class Client {
2     A myA = new A();
3     public void callNewMethod_Bar() {
4       //Invoke bar(42); through invoke() method
5       myA.invoke(''bar'', new Class[]{int.class}, new Object[]{new Integer(42)});
6     }
7   }
```

Listing 3.4: Calling a new method “bar”.

inserting new functionalities to a class. To support adding a new method, the presented approach uses a helper class, which contains newly added methods because the schema change is not allowed. A replacing class can call these new methods on the helper class. All the call sites of newly added methods are translated into the calls to `invoke` like Listing 3.4. Each new constructor is translated into an invocation of a “do-nothing” constructor and a special initialization method that contains the added constructor’s logic.

Figure 3.9 shows an example of adding a new method; the newer version of `A` has a new method `bar`. The first and second columns in Figure 3.9 illustrate class diagrams representing classes and their relationships at the source code and the corresponding bytecode, respectively. The special helper class `HelperClass` contains the new method `bar` and each proxy class contains the `invoke` method. Each invocation of `bar` is translated to invoke `invoke` instead.

We may have to consider a special case in which newly added methods need access to existing `private` fields. Since `private` fields can be only accessed by the class declaring them, helper classes containing added methods cannot access these `private` fields. To allow helper classes to access `private` fields, we may need to change the fields’ access to `public` or

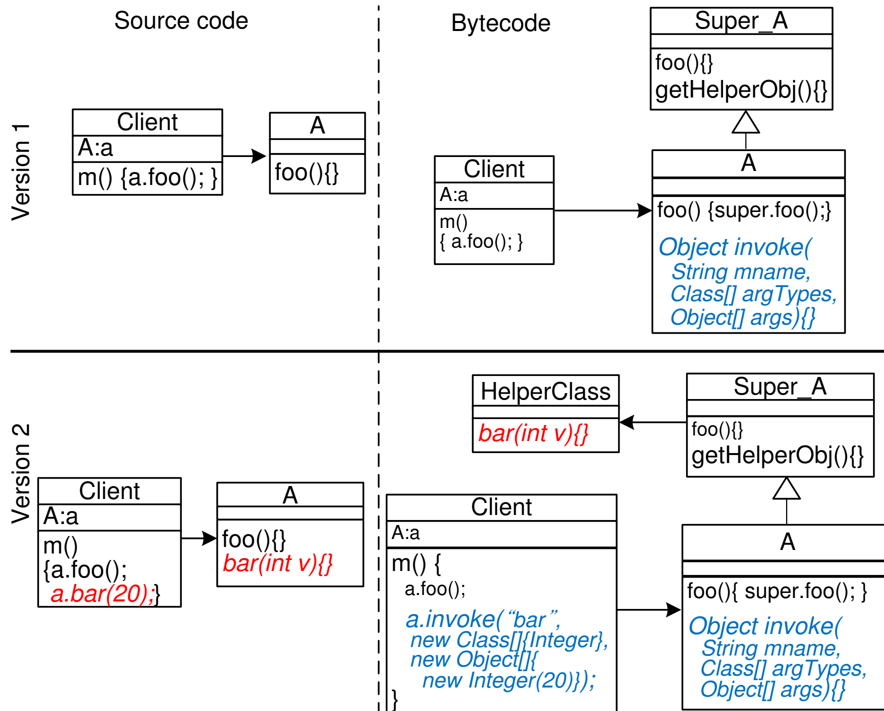


Figure 3.9: Adding a new method.

`protected`. However, this approach violates the encapsulation principle and raises security issues. Bhowmik and Pugh [24] proposed an effective way to solve such a problem for the Java inner classes rewrite. By using their approach, a virtual superclass enables intended helper classes to access its `private` fields. At load time, a virtual superclass obtains a secret key and passes it to its method helper class. Therefore, the virtual superclass and the method helper class share the same secret key. When objects of the method helper class need to access `private` fields of the virtual superclass, they call a public method that receives and checks the secret key. If the secret key is the same, the virtual superclass gives access to its `private` field. Otherwise, a security exception is thrown.

Adding new fields This approach to dynamic software updates presents two optional implementations for the addition of new fields to a class; one implementation uses a helper class having a variable declared as data type `java.util.HashMap` to store new fields, and the other implements an additional class for each new field. The former can be implemented easily, and the latter can show better performance when the same field is accessed many times.

Version management of classes This approach to dynamic software updates allows for the coexistence of old and new instances, which are created from different versions of a class. All new instances of a new version are instantiated regardless of existing old objects of an old version. Older objects are alive with the new objects. To prevent application crashes, the class version of each instance must be managed so that the changes introduced by newer versions are coordinated with older objects. To keep track of the version information, the presented rewrite introduces a static version field and method `checkVersion`. When `checkVersion` is invoked on a proxy, it checks if the version of a class is older than the latest version. If so, a special initialization method is invoked for the newly added state (i.e., extra fields). The class version then is updated to the latest version.

Listing 3.5 shows a proxy class `A` and its virtual superclass `A_virtSuper` with HotSwap-supporting and application-specific methods.

This approach for dynamic software updating supports the update of a set of related classes at one time. Those classes can be within one Java package or different packages. Such updates of closely related classes often involve considering their relationships for successful updating. For example, changes to one class can affect other classes, which invoke methods of

```
1  class A extends A_virtSuper {
2    private int myVersion = −1;
3
4    //HowSwap−supporting methods
5    public A(Bogus b) { //add new constructors
6      super(b);   //call super do−nothing constructor
7    }
8    final Object invoke(String methodName, Class[] argTypes, Object[] args) {
9      //invoke any method
10   }
11   //check if the version of this class is older than the latest version
12   private final void checkVersion() {
13     if (myVersion < getLatestVersion()) {
14       ...
15       //invoke a method for initializing new state
16       myVersion = getLatestVersion();
17     }
18   }
19   //Application−specific method
20   int foo(int i) {
21     checkVersion();   //check the version of this class
22     return super.foo(i);
23   }
24 }
25 class A_virtSuper {
26   private int j = 0;
27   public static final int VERSION = 1;
28
29   //HowSwap−supporting methods
30   public SuperA(Bogus b) {...}  //for adding constructors
31   public int getLatestVersion() {
32     return VERSION;
33   }
34   //Application−specific method
35   int foo(int i) {
36     return (i + j);
37   }
38 }
```

Listing 3.5: A proxy and virtual superclass.

the changed class. Therefore, invoke and access relationships between updated classes should be considered for successful dynamic software updating. Should dynamic software updating not be applied correctly, the system should be designed to automatically restore its original state right before the dynamic software updating is performed. As a result, the system is not affected by the failure of the dynamic software updating. If dynamic updating has been finished partially, the integrity of a system could be damaged. A model and implementation for an update transaction mechanism for multiple class updates will be provided.

Chapter 4

Design and Implementation

To demonstrate the approach to dynamic software updates discussed in Chapter 3, a prototype of a dynamic updating system using HotSwap and bytecode engineering has been developed. This chapter describes the design and the implementation of the prototype.

4.1 Representing the Dynamic Updating System Using UML Diagrams

Use case diagrams [82] are used to depict a high-level view of a system by identifying and partitioning system functionality during the analysis phase. Figure 4.1 depicts a UML use case diagram which represents the presented approach to dynamic software updates in this research using JVM HotSwap and binary rewriting. Software developers and the JVM HotSwap facility interact with the updating system. A software developer can find differ-

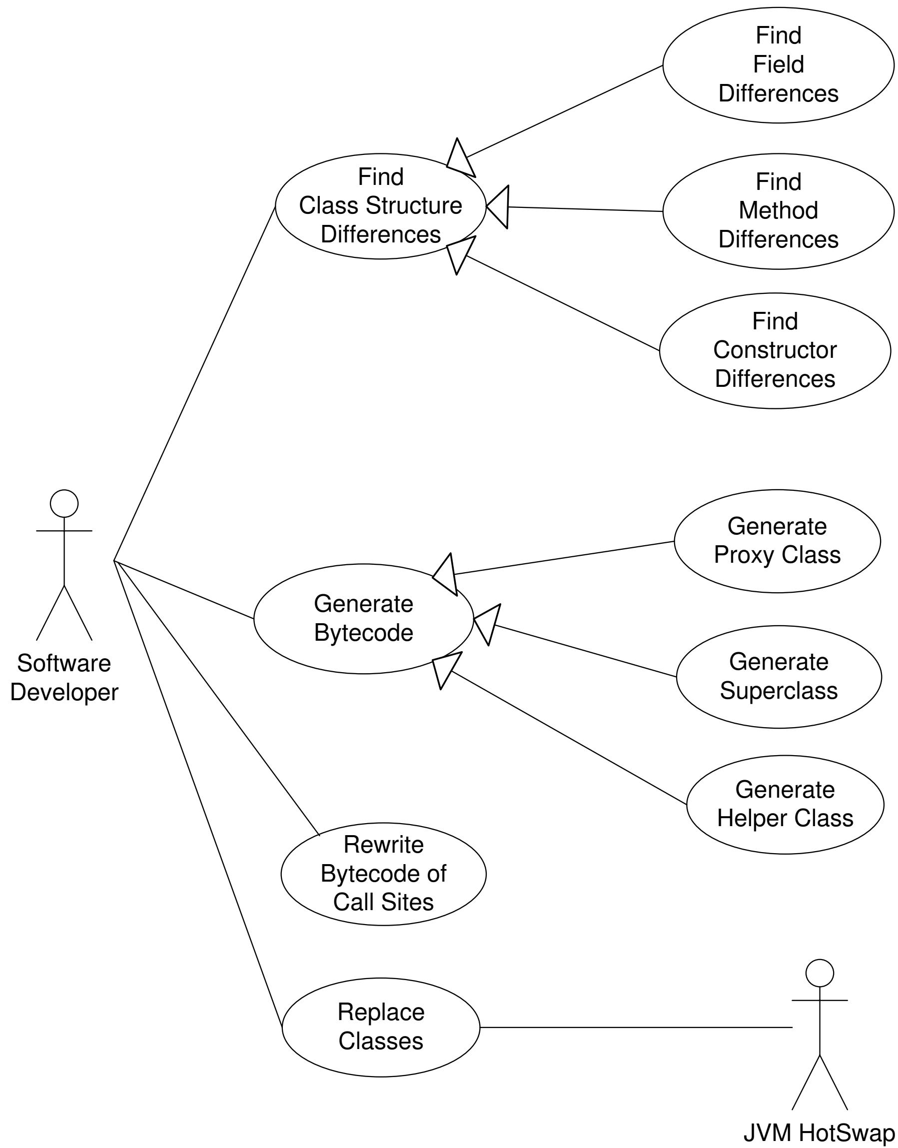


Figure 4.1: Use case diagram of the DSU system presented in this research.

ences between classes, generate bytecode representing the found differences, and rewrite the bytecode of call sites which will invoke the generated bytecode. The software developer can replace an older version with a newer one using the JVM HotSwap.

UML class diagrams represent the structural information of a system by identifying classes of a system and their relationships including inheritances and associations. Figure 4.2 represents a class diagram of the presented dynamic software updating system in this work. This diagram is drawn from the extracted use cases above. Class *BytecodeGenerator* includes common methods for generating bytecode and the three classes *SuperclassGenerator*, *ProxyClassGenerator*, and *HelperClassGenerator* extend these methods. *SuperclassGenerator* and *ProxyClassGenerator* generate virtual superclasses and proxies, respectively. *ClassStruDifferencer* compares the structure of an older version with one of a newer version. The identified differences will be used to generate bytecode. *UpdateableVerGenerator* rewrites the bytecode of a target application to make it an updateable application, which is structurally different from the original version.

While class diagrams illustrate the structure of an application, sequence diagrams are used to represent the behavioral information of the application. Figure 4.3 models the logic of making an updateable application using the presented updating system in this work. The classes across the top of the diagram are identified in Figure 4.2. When a software developer wants to make a target application updateable, he/she has to contact *UpdateableVerGenerator*. The software developer sends a set of classes which might be updated dynamically during the execution of the target application. He/she can send all application classes or some of them. Within the updating system, such a method call triggers a filtering method of *ClassNameFilter*. A set of classes to be updated is sent to two code generators–*ProxyClassGenerator* and *SuperclassGenerator*. *UpdateableVerGenerator* receives the generated proxy classes and superclasses and returns an updateable application to the software developer.

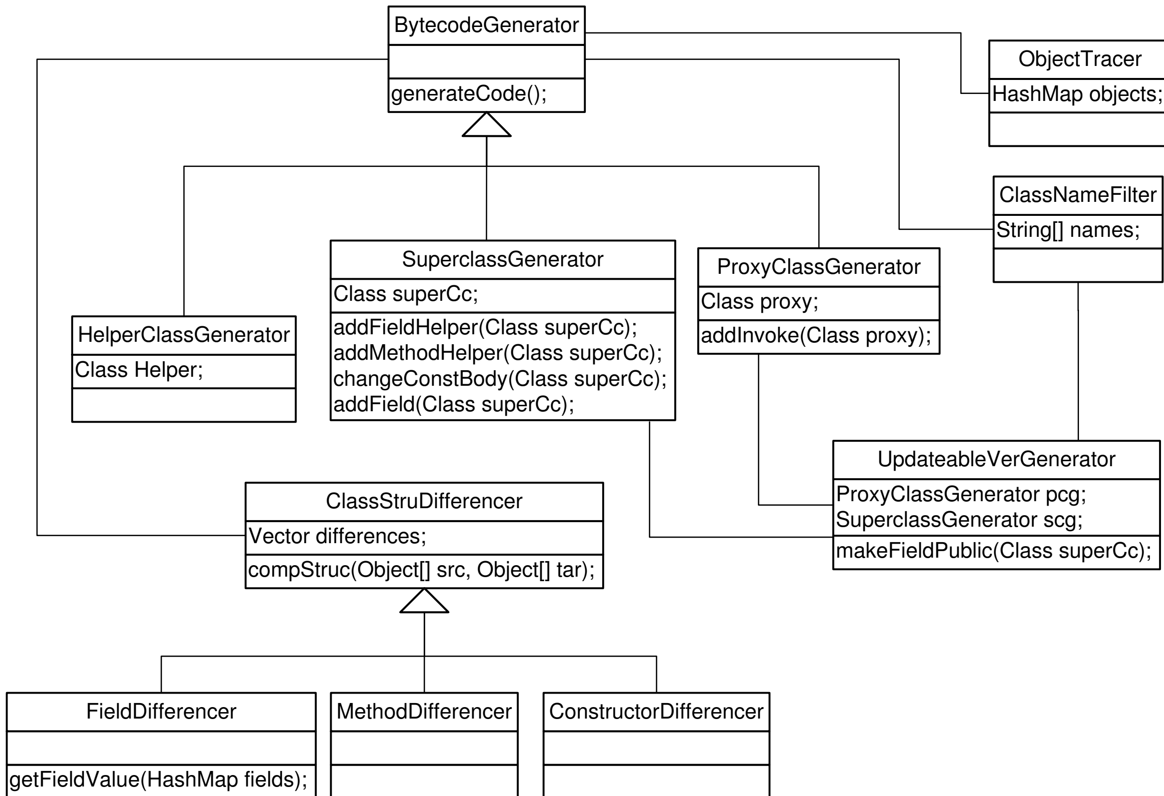


Figure 4.2: Class diagram of the DSU system presented in this research.

Figure 4.4 depicts a sequence diagram for generating newer versions of virtual superclasses and helper classes. Given a newer version of a class, the updating system compares the newer version with the older one and then generates bytecode of a virtual superclass and helper class by following the flow of logic illustrated in Figure 4.4. To get new versions of classes, a software developer requests comparison of the structure of old and new classes. *ClassStruDifferencer* invokes a filtering method of *ClsssNameFilter* and determines structural differences between old and new classes. *HelperClassGenerator* and *Superclass-Generator* automatically generate helper classes and superclasses based on the identified structural differences. Finally, the software developer can replace the old versions of the

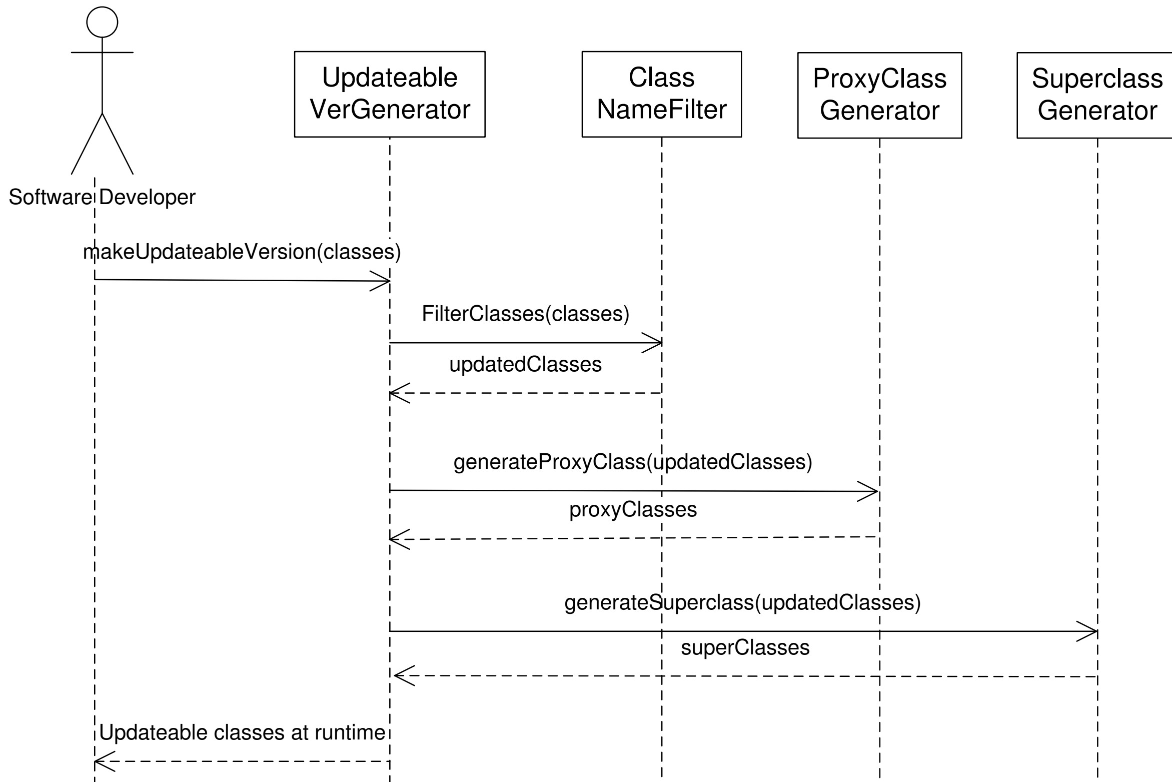


Figure 4.3: Sequence diagram of the *makingProgramUpdateable* scenario.

classes with the newly generated classes.

4.2 Transformation Rules

A bytecode rewriter for Virtual Superclass binary refactoring observes the following transformation rules to manipulate bytecode of an application. In Figure 4.5, c is an original class, transformed into a proxy and the added superclass. The new superclass c_{vs} is inserted between the proxy c_{proxy} and the initial superclass s of the original class c. Figure 4.6 depicts how the indirection works for methods and constructors. Figure 4.7 details how the presented

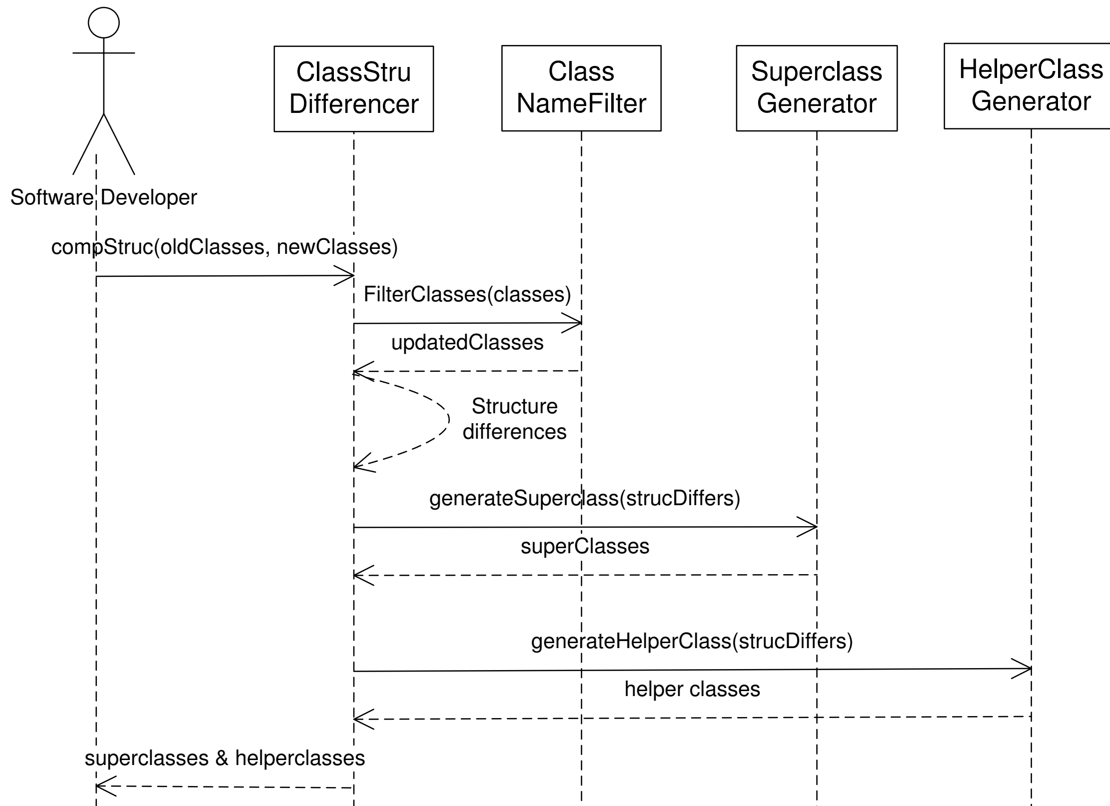


Figure 4.4: Sequence diagram of the *generateNewerVersion* scenario.

approach introduces an indirection when accessing non-private fields. The double vertical bar ($\|$) specifies pre- and post-conditions of the structural enhancement operations. In $\frac{X}{Y}$, X denotes an enhancement, while Y denotes its corresponding symbolic undo operation. $private <_v V$ denotes the visibility V which is stronger than *private* visibility.

4.3 The ClassDifferencing Algorithm

To generate the helper classes, the presented approach identifies the structural changes between two versions of a class. The class differencing algorithm shown in Figure 4.10 takes

two versions of the same program as input and returns a collection of differences in fields, constructors, and methods. This algorithm simply compares the fields, methods, and constructors of the classes by using the Java Reflection API. To find out field differences, the differencing algorithm compares the modifier, type, and name of a field. The method differences are identified by examining the modifier, return type, name, and parameter types of a method and constructors are distinguished by their modifier and parameter types.

A set of interfaces, $I = \{i_1, i_2, \ldots, i_i\}$
$VSuper(c \| c_{proxy}, c_{vs})$: Class c is transformed into c_{proxy} and c_{vs}.
c: refactored class, c_{proxy}: proxy class of c, c_{vs}: new superclass of c

$$VSuper(c \| c_{proxy}, c_{vs}) = \frac{c\ extends\ s\ implements\ I}{c_{proxy}\ extends\ c_{vs},\ c_{vs}\ extends\ s\ implements\ I}$$

Figure 4.5: Indirection using superclasses.

P_{vs} ‖...‖ denotes the rewriting by the presented approach.
//The transformation of constructors
P_{vs} ‖ *public* $k(T_1, \ldots, T_n)$ *throws* $C_1, \ldots, C_i$‖ =
  public $k(T_1, \ldots, T_n)$ throws $C_1, \ldots, C_i$ { super($T_1, \ldots, T_n$); }
//The transformation of methods
P_{vs} ‖ *public T* $m(T_1, \ldots, T_n)$ *throws* $C_1, \ldots, C_i$‖ =
  public T m($T_1, \ldots, T_n$) throws $C_1, \ldots, C_i${
    if (the return type of m is *void*) super.$m(T_1, \ldots, T_n)$;
    else return super.$m(T_1, \ldots, T_n)$;
  }

Figure 4.6: Indirecting constructors and methods.

$$
\begin{array}{l}
//Access\ the\ superclass's\ non-private\ fields \\
G_{vs}\ \|...\|\ \text{represents the generation of getters and setters for fields.} \\
G_{vs}\ \|\ private <_v V\ T\ x\| = \\
\qquad private <_v \text{V}\ \ \text{T}\ \ \text{get}X()\ \{\ \ \text{return}\ \ x;\ \} \\
\qquad private <_v \text{V}\ \ \text{void}\ \ \text{set}X(\text{T}\ x)\ \{\ \ \text{this}.x = x;\ \} \\
//Access\ non-private\ fields\ via\ a\ proxy \\
P_{vs}\ \|\ private <_v V\ T\ x\| = \\
\qquad private <_v \text{V}\ \ \text{T}\ \ \text{get}X()\ \{\ \ \text{return}\ \ \text{super.get}X();\ \} \\
\qquad private <_v \text{V}\ \ \text{void}\ \ \text{set}X(\text{T}\ x)\ \{\ \ \text{super.set}X(x);\ \}
\end{array}
$$

Figure 4.7: Indirecting the superclass's non-private fields.

Proxy class – invoke method

```
1 public final Object invoke
2     (String methodName, Class[] argTypes, Object[] args){
3   ...
4   Method m = SomeClass.getDeclaredMethod(methodName,
        argTypes);
5   Object returnValue = m.invoke(SomeObject, args);
6   ...
7 }
```

Figure 4.8: Generating a special *invoke* method.

4.4 Synchronizing Concurrent Updates of Cluster Applications

Due to their cost efficiency, compute clusters are among the most widely-used high performance computing environments. A typical compute cluster features a large number of homogeneous processors connected to each other with a high performance interconnect. To submit a computational task to a cluster, users interface with a scheduler that queues up the submitted tasks for their turn to be run. To coordinate the execution between differ-

$Virtual\ superclass -\ getMethHClass()\ and\ getFieldHClass()$

```
1  public final static Object getMethHClass(){
2    if (methodHelperObj is null)
3      methodHelperObj = new MethodHelperClass();
4    return methodHelperObj;
5  }
6  public final static Object getFieldHClass(){
7    if ( fieldHelperObj is null)
8      fieldHelperObj = new FieldHelperClass();
9    return fieldHelperObj ;
10 }
```

Figure 4.9: Generating special methods to get helper classes.

ent processors, compute cluster applications use the Message Passing Interface (MPI) [12] middleware library, which has a standardized interface and is almost universally available. Next we describe our dynamic software update system that targets compute clusters and uses MPI for coordination. We start by outlining the main design goals of our system, then provide details of our implementation, and finally present an evaluation of the performance of our system.

4.4.1 Design Considerations

Compared to applications run on a single machine, cluster applications coordinate the execution of multiple concurrent processes running on multiple compute nodes. Therefore, a dynamic update system targeting cluster applications must possess several properties.

One issue is delivering the updated binary representation of a program to each compute node. Since the nodes on a cluster typically run on a shared file system, copying a new

```
    INPUT: A set C = {(c^1_v1, c^1_v2), (c^2_v1, c^2_v2), ..., (c^p_v1, c^p_v2)} of class pairs to be compared.
    OUTPUT: A collection of differences of fields, constructors, and methods
 1  while (a set C is not empty) do
 2    //Compare fields of classes
 3    fieldsOfOldClass ⟵ c^i_v1.allFields(), fieldsOfNewClass ⟵ c^i_v2.allFields()
 4    for (fieldsOfNewClass is not empty) do
 5      eachFieldOfNewClass ⟵ fieldsOfNewClass.nextItem()
 6      for (fieldsOfOldClass is not empty) do
 7        eachFieldOfOldClass ⟵ fieldsOfOldClass.nextItem()
 8        if (sig. of eachFieldOfOldClass == sig. of eachFieldOfNewClass) then
 9          isSameField = true; break;
10        end if
11      end for
12      if (NOT isSameField) then
13          differentMembers.addElement(eachFieldOfNewClass)
14      end if
15    end for
16    //Compare constructors and methods of classes
17    methodsOfOldClass ⟵ c^i_v1.allMethods(), methodsOfNewClass ⟵ c^i_v2.allMethods()
18    for (methodsOfNewClass is not empty) do
19      eachMethodOfNewClass ⟵ methodsOfNewClass.nextItem()
20      for (methodsOfOldClass is not empty) do
21        eachMethodOfOldClass ⟵ methodsOfOldClass.nextItem()
22        if (sig. of eachMethodOfOldClass == sig. of eachMethodOfNewClass) then
23          isSameMethod = true; break;
24        end if
25      end for
26      if (NOT isSameMethod) then
27          differentMembers.addElement(eachMethodOfNewClass)
28      end if
29    end for
30  end while
```

Figure 4.10: The *ClassDifferencing* algorithm.

version of the binary will immediately make it available to all the nodes.

Another issue is updating multiple concurrent copies of a program consistently. In other words, some distributed runtime invariant has to be maintained. The nature of this invariant depends on the kind of updated application. For example, for embarrassingly parallel applications, in which the Master communicates exclusively with Workers and Workers do not communicate among each other, the required consistency guarantees are quite relaxed; each Worker code could be updated independently and the correctness of the computation will still be preserved.

Nevertheless, most high performance parallel applications have much more stringent update consistency requirements than a typical Master Worker application. One invariant that may have to be maintained is that *no two divergent versions of a program are run simultaneously.* Since to ensure maximum parallelism, concurrent processors are synchronized sparingly, maintaining this invariant can be quite challenging.

Our distributed synchronization implementation has two parts. First, we have created a distributed synchronization algorithm that implements a consistency scheme using standard MPI calls. In addition, to apply this algorithm to an existing parallel application, we have to inject special code to each piece of software that will be run concurrently and will need to be updated dynamically. We describe each of these parts in turn next.

4.4.2 Synchronization Algorithm

Updating multiple concurrent tasks consistently entails receiving the update information from the user and applying the update consistently to all the running nodes. Each of these

OUTPUT: Update information which will be used for updating on all nodes.

```
1   updateInfoQueue ⇐ ϕ
2   create a communication channel ch_p with a port number p
3
4   REPEAT
5     await the update data ud from the user
6     IF ∃ ud THEN
7       ch_p receives ud through the port number p
8       add ud to the update queue updateInfoQueue
9     ENDIF
10  UNTIL the application is running
```

Figure 4.11: Waiting for update information from the user.

activities requires a distributed consistency algorithm that we now describe.

Dynamic updates are initiated by the user who interacts with an update front end. The purpose of the front end is to accept from the user a new version of the running program. Then the front end may choose to calculate the delta between the running and new versions of the program to reduce the amount of code that would have to be replaced. Finally, the front end instructs the running nodes to update themselves to the new version.

The step of this procedure that requires distributed coordination is for the nodes to receive the update request from the front end. Figure 4.11 shows the algorithm, according to which the root node of the distributed computation receives update requests. To that end, the root node spawns a new *update thread*, that waits for an incoming socket connection request. An asynchronous IO facility can be used to avoid busy waiting. When a connection from the front end is received, the update thread creates a user communication channel (line 2), receives update data from the user (line 5), and enqueues it (line 8). The thread can repeat the above procedure continuously while the application is running, thus possibly

```
1  updateInfoQueue ⇐ φ
2  N ⇐ a set of the nodes involved in the computation
3
4  /* The root node (rank0) creates a thread which will wait for update information. */
5  IF nodeRank ≡ 0 THEN
6    create an update thread udThread
7  ENDIF
8  ...
9
10 /* Broadcast the update information from the root node to all nodes. */
11 IF nodeRank ≡ 0 THEN
12   get updateInfoQueue from the update thread udThread
13   broadcast updateInfoQueue to all nodes N
14 ENDIF
15
16 IF updateInfoQueue ≠ φ THEN
17   /* Each worker node updates its application on the fly. */
18   update the application app_j on n_j based on updateInfoQueue
19   /* A worker node waits until the others finish their updates. */
20   WAIT UNTIL the updates on the remaining N − n_j have been completed
21 ENDIF
```

Figure 4.12: Synchronizing concurrent dynamic updates on multiple nodes.

enqueuing multiple update requests.

The code on each running node is patched with update management code, which constitutes a short sequence of method calls and conditional statements. The exact code location at which dynamic updates should take place depends on the semantics of the updated application. For some applications, update management can be seamlessly added to the beginning of the main compute loop; however, this particular location may be unsuitable for other applications. Figure 4.12 displays the algorithm implemented by the update management code. To synchronize updates, the root node (i.e., the process with rank zero)

broadcasts update information to all nodes. Upon receiving the update information, all the nodes, including the root node, update themselves dynamically using whatever local DSU infrastructure is in place, and synchronize on a barrier.

4.4.3 Example Implementation

To enable efficient and safe dynamic software updates for parallel high performance applications, our approach taps into their development cycle and also adds some runtime functionality. Figure 4.13 describes the process by which an unaware parallel high performance application is enhanced to be dynamically updateable. The enhancement process starts after the application has been implemented, thus following the principle of separating concerns. The programmer focuses on implementing the application logic, while a special post-compilation step adds the required functionality to enable dynamic updates. Furthermore, our approach does not assume that the application is built with dynamic updates in mind.

As our example implementation, we have targeted parallel applications that use the Java™technology to operate seamlessly in a heterogeneous environment. The Java technology has been successfully applied to the domain of distributed parallel computation: heterogeneous computational grids are commonly Java-based [4]. The JVM also is one of the most advanced virtual execution environments ported to a multitude of different platforms. However, our primary motivation for choosing JVM as our experimentation platform is convenience. The relatively high level of Java bytecode makes it easier to add functionality to classes at the binary level, thus simplifying our proof-of-concept implementation. We believe, however, that our example implementation can be ported to other HPC platforms

with only minor adjustments in the programming tools.

One of the key issues of dynamic software updates is safety. That is, updating a running application must not put it in an unsafe state that may affect the application's correctness and stability. Thus, it becomes important to identify a point in the execution of a parallel task at which its software can be safely updated dynamically. Upon reaching this point, the task should check whether the update is available, perform the requisite synchronization with the other tasks, and then update itself if so instructed.

Identifying a specific program execution point at which it is safe to perform a dynamic update requires a thorough understanding of the semantics of the parallel application. While identifying such safe update points automatically can be an immensely complex task, the programmer can easily specify them by hand. To that end, our approach allows the programmer to specify update information through a simple configuration file that contains the Java class, method, and line number, before which the synchronization code should be added. The code is added directly at the bytecode level, a common software practice for Java applications with multiple bytecode engineering toolkits readily available [3, 34].

Figure 4.14 shows that the synchronization code is quite straightforward. The process with rank zero creates a thread to receive update requests from the user. To ensure that the thread does not waste computing resources, all the communication takes place through Java asynchronous IO, which wakes up the thread only when a new update request is ready to be processed. The rank zero process then broadcasts the update information to the remaining processes. When the user initiates a dynamic update, each process updates itself and synchronizes on a barrier. If no update is needed, then the only additional cost is the broadcast (line 5).

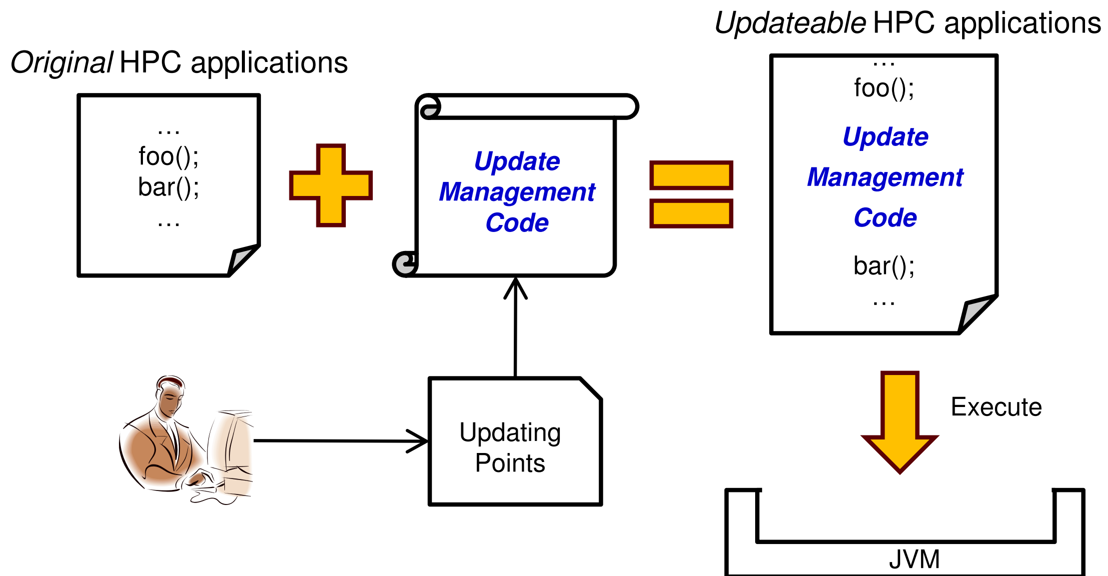


Figure 4.13: Making HPC applications updateable.

```
1   if (MPI.COMM_WORLD.Rank() == 0) {
2     classNames = SelectSockets.INSTANCE.getClassNames();
3   }
4
5   MPI.COMM_WORLD.Bcast(classNames, 0, classNames.length, MPI.OBJECT, 0);
6
7   if (!classNames.isEmpty()){
8     ExecutionManager.update(classNames);
9
10    MPI.COMM_WORLD.Barrier();
11
12  }
```

Figure 4.14: An example of the synchronization code.

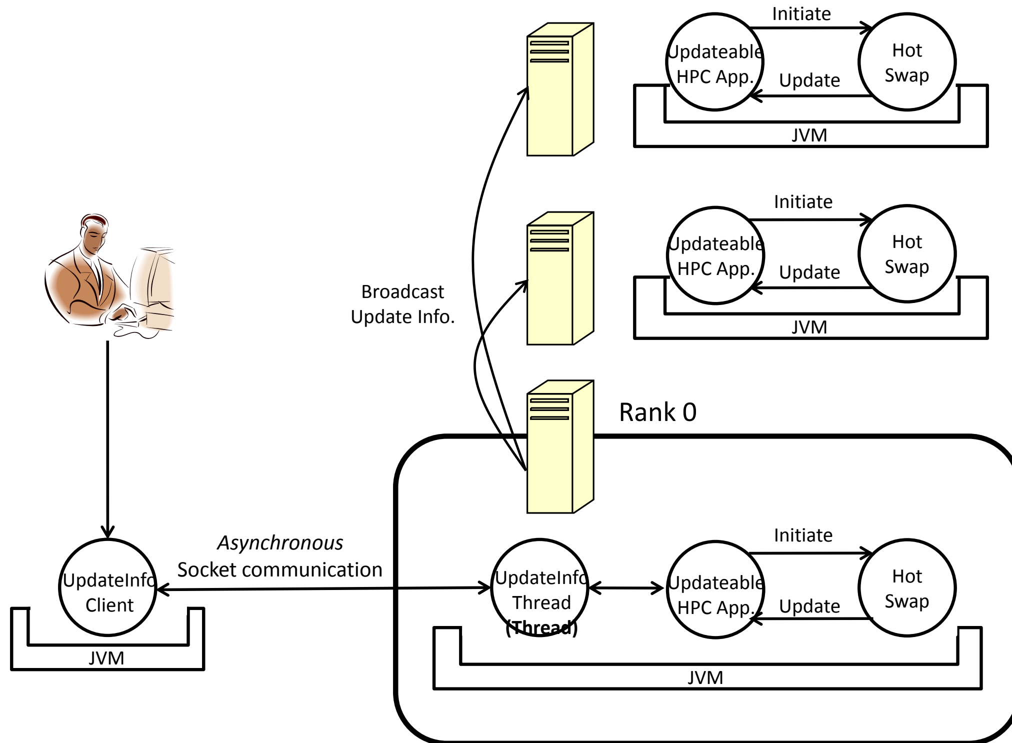


Figure 4.15: Updating parallel HPC at runtime.

Figure 4.15 illustrates our runtime support for dynamic updates. Our approach leverages the capabilities of the standard JVM HotSwap facility. Upon receiving an update request, the running JVM sends itself a HotSwap request, relying on the HotSwap machinery to safely interrupt the execution, reload the updated version of the specified classes, and restart the computation. Upon restarting the computation, the first executed method is a barrier synchronization that ensures that all the concurrently-running copies of the computation have been updated consistently.

4.5 Limitations and Issues

JVM-based applications are composed of objects and their interactions, and these objects are instantiated from classes. Since the presented dynamic software updating system uses HotSwap, it will support class updates, not object updates. These class updates cover the modification of fields or methods inside a class and the schema change of a class (e.g., addition or removal of fields or methods). However, when a class is replaced with a newer version, all object instances from the older version of that class must be managed so that they do not raise exceptions or errors during system operation. Special helper classes will transfer state between old and new objects. In addition, the approach to dynamic software updates will support various update units that range from a single class to multiple classes within a Java package.

This approach does not address the deletion of existing methods or fields, or changing class hierarchies. However, one notable exception is emulating the change in polymorphic dispatch due to the deletion of a method. If a method in a subclass is deleted, the superclass version should be invoked instead. If the presented rewriter detects such a case through a simple static analysis, the methods affecting polymorphic dispatch are replicated as methods with different names, and all their call sites are updated accordingly. Allowing these changes to the updated class would require an additional engineering effort, which is beyond the scope of this research. Unused methods and fields do not affect the program's execution, and changing class hierarchies is too destructive a change to warrant support as part of a dynamic update.

Since this research supports bytecode rewriting of class files, the approach to dy-

namic software updating can be used for VM-based computationally-intensive applications like X10 applications. However, it does not support current Fortress [17] and Erlang [19] computationally-intensive applications which are not based on a virtual machine. Security and real-time issues for dynamic software updating are not considered in this work. Thus, this work is not appropriate for real-time applications because the time delay due to dynamic software updating could lead to the failure or malfunction of these real-time applications.

Here are other issues and limitations considering the features of object-oriented programming languages.

Monitor Concurrency Control For `synchronized` original methods, this attribute is left only for methods in subclasses, but it is removed from their proxies. Thus, if indirected methods call each other directly, they will lock on the same object, avoiding a potential deadlock.

Reflection The use of reflection to locate and invoke newly-added methods will render the presented approach invalid, as the new methods are not members of an updated class but of a helper class instead. Fortunately, the use of reflection is uncommon in high performance applications written in Java, and JVM-based languages such as X10 disallow reflection completely to enable a wider range of optimizations.

Native Code Some of the functionality in the JDK is provided as native code, executing as part of the JVM libraries. These native libraries are used in Java to achieve better performance and to obtain access to system resources. Since the presented approach changes bytecode, it would not be applicable for dynamic updates of native code. This research does

not foresee the need to update such native code dynamically in this application domain.

Removing methods and fields The presented approach does not support the deletion of methods or fields, as the presence of unused methods and fields in a program does not affect its execution with one notable exception. If polymorphic dispatch is present and an overriding method in a subclass is deleted, the overridden method must be invoked in the new version. This research ensures this behavior by changing the body of the deleted method in a subclass to delegate to the overridden method in a superclass.

Class hierarchy changes In addition to ignoring the deletion of methods and fields, which does not affect the program execution, this research also does not aim at supporting changes in class hierarchies. While proxies presented in this work maintain the original inheritance relationship, which allows subclass proxies to be used in place of a superclass, changing a class inheritance hierarchy is too destructive to the overall structure of a program to be of value for a dynamic update.

Chapter 5

Experiments and Results Analysis

The performance and effectiveness of the presented approach to dynamic software updates have been evaluated. First, the respective performance of Virtual Superclass and Virtual Interface have been compared, and then a parallelization of Smith-Waterman alignment algorithm has been updated dynamically.

5.1 Performance Evaluation

The following micro and macro benchmarks demonstrate the performance advantages of Virtual Superclass. The experimental environment consisted of a workstation with an Intel Pentium 4 (3.6GHz) processor, 1GB RAM, running Ubuntu Linux 7.10 (Gutsy Gibbon) and JDK version 1.5.0_14.

The first micro benchmark assessed the overhead of indirecting a single method invo-

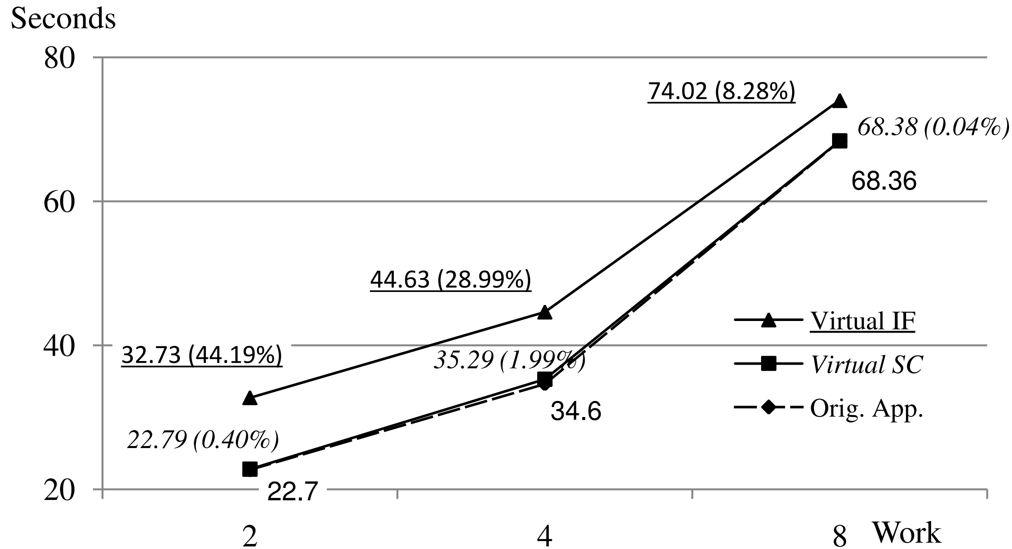


Figure 5.1: Overhead of binary refactoring microbenchmark.

cation. Figure 5.1 shows that the cost of indirection depends on the amount of computation performed by the indirected method. In this benchmark, the indirected method performed two, four, and eight multiplications, increments, and test operations. Each invocation is repeated $1 * 10^9$ times. The maximum overhead of less than 2% makes this refactoring suitable for introducing indirection to the majority of computationally-intensive applications.

To assess the performance of the Virtual Superclass indirection in more realistic programs, five different full program benchmarks from SpecJVM98 [14] were used. SpecJVM98 is a benchmark suite for measurement of Java virtual machine performance and requires a Java virtual machine compatible with JDK 1.1. API, or later. It contains eight test programs shown in Table 5.1.

Figure 5.2 shows that similar to the micro benchmark numbers above, the total overhead of Virtual Superclass never exceeds 2%, whereas for Virtual Interface it can go as

Program	Description	Real Application
_200_check	Check JVM and Java features	X
_201_compress	A popular utility used to compress and uncompress files	O
_202_jess	A Java expert system shell	O
_209_db	A small data management program	X
_213_javac	The Java compiler	O
_222_mpegaudio	An MPEG-3 audio stream decoder	O
_227_mtrt	A dual-threaded program that ray traces an image file	X
_228_jack	A parser generator with lexical analysis	O

Table 5.1: SpecJVM98 test programs.

high as 16%.

5.2 Updating an RMI-based Application Dynamically

This section demonstrates that presented approach to dynamic software updates in this dissertation can be applied to not only computationally-intensive applications but also normal Java applications.

A Remote Method Invocation (RMI)-based program was used from a Java book [38], as an example of a long-running server application that can benefit from being updated at runtime. The program consists of a remote interface `Account`, its implementation `AccountImpl`, and `AccountClient`. Figure 5.3 depicts those three classes and the relationships among them. To satisfy users' new or changed requirements, `Account` and `AccountImpl` will be dynamically updated using the presented approach to dynamic software updates in this research. Figure 5.4 shows the initial version of `AccountImpl` has changed to a newer

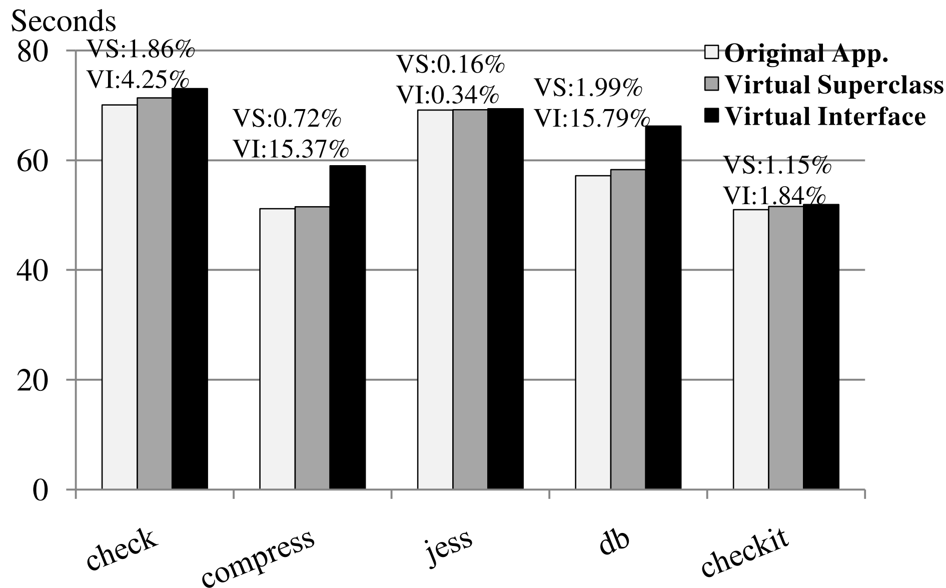


Figure 5.2: Overhead of binary refactorings on SpecJVM98.

version by adding new fields and methods.

The updated cases are illustrated in Table 5.2, which add a method, change the signature of a method, and add a new field. To add a new method, `transfer`, the presented updating system generates a proxy class with the `invoke` method and a helper class with the `transfer` method. All invocations to `transfer` are replaced with invocations to `invoke` method of the proxy class.

To modify the signatures of remote methods, a new argument `access` that checks the user's permission was added.

To add a new field, `coowner`, a special helper class containing the new field is generated. The virtual superclass will return the helper class when any object tries to access the field.

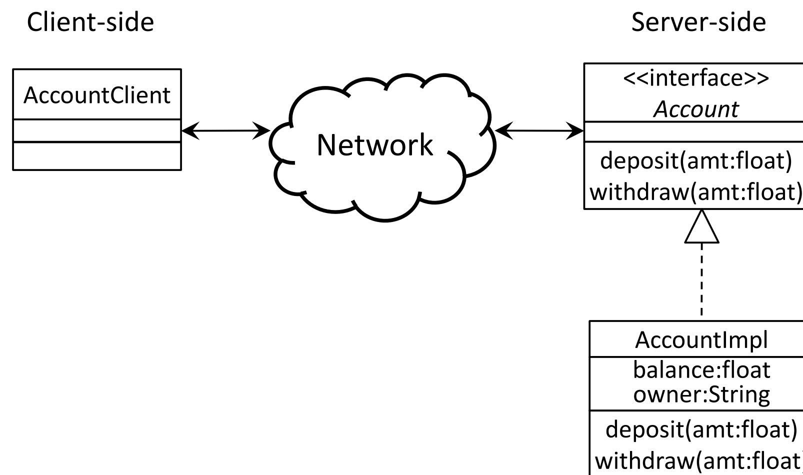


Figure 5.3: The RMI-based simple banking program.

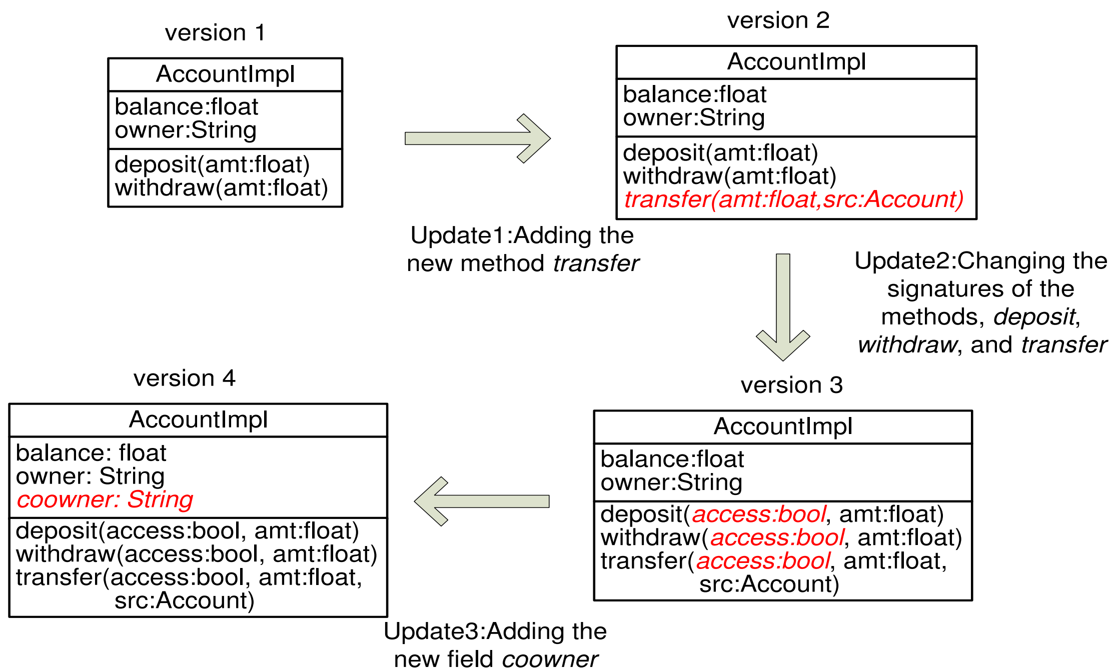


Figure 5.4: Changes to the RMI program using extended HotSwap.

Changes	Implementations
Add methods	Add `transfer(float, Account)`
Change method signatures	deposit(float) → deposit(*boolean*, float) withdraw(float) → withdraw(*boolean*, float) transfer(float, Account) → transfer(*boolean*, float, Account)
Add fields	Add a new field `coowner` and getter/setter
Call sites	*.transfer(..) → *.invoke(”transfer”, ...)

Table 5.2: Dynamic updates to the Banking program.

5.3 Updating Smith-Waterman Parallelization Dynamically

The motivating example is concerned with pairwise sequence alignment, a well-known problem in bioinformatics. The use of computers has made it possible to answer a large spectrum of questions in biology. Most of these problems are solved by representing a biological entity such as a gene computationally and manipulating the resulting representation using a variety of algorithms.

5.3.1 Smith-Waterman Algorithm

The Smith-Waterman algorithm [71] is a dynamic programming method [22] for computing the optimal local alignment of two sequences. The algorithm was first proposed in 1981 by Smith and Waterman and a scoring matrix including a set of specified gap penalties is used [99]. The Smith-Waterman algorithm 1) initializes a similarity scoring matrix, 2) finds the biggest value among $(i-1, j)$, $(i, j-1)$, and $(i-1, j-1)$ in the matrix to decide the value

of the position (i, j), and 3) traces back the path that leads to the maximal score in order to find the optimal local alignment.

The Smith-Waterman algorithm can compare two sequences: $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$. Starting from row 0 and column 0, a similarity score matrix M is constructed with the following recurrence and its size is $(n+1)$ x $(m+1)$:

For $0 \leq i \leq n$, and $0 \leq j \leq m$,

M[0,0] = M[i,0] = M[0,j] = 0.

For $1 \leq i \leq n$, and $1 \leq j \leq m$,

$$M[i,j] = max \begin{cases} 0, & \\ M[i,j-1] + scoreInsertion(B[j]) & \text{Insertion,} \\ M[i-1,j-1] + scoreSubstitution(A[i],B[j]) & \text{Match/Mismatch,} \\ M[i-1,j] + scoreDeletion(A[i]) & \text{Deletion.} \end{cases}$$

The value $scoreSubstitution(A[i], B[j])$ is the score for representing a match or a mismatch. The $scoreInsertion(B[j])$ and $scoreDeletion(A[i])$ are the penalty for inserting a gap character '–'. Either a_i or b_j could be aligned to a gap '–'.

The Smith-Waterman algorithm can be exemplified by aligning two sequences:

Sequence1 = ACACA and Sequence2 = AGCAC,

scoreSubstitution(match) = 2,

scoreInsertion(B[j]) = scoreDeletion(A[i]) = scoreSubstitution(mismatch) = -1,

$$
M = \begin{pmatrix}
 & - & A & C & A & C & A \\
- & 0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & 2 & 1 & 2 & 1 & 2 \\
G & 0 & 1 & 1 & 1 & 1 & 1 \\
C & 0 & 0 & 3 & 2 & 3 & 2 \\
A & 0 & 2 & 2 & 5 & 4 & 5 \\
C & 0 & 1 & 4 & 4 & 7 & 6
\end{pmatrix}
$$

The alignment represented in the similarity matrix above is:

Sequence1: A – C A C A

Sequence2: A G C A C –

This research used JAligner which is a Java implementation of the Smith-Waterman algorithm providing biological local pairwise sequence alignment [5]. JAligner takes two sequence files and prints out alignment results on the console. Figure 5.5 shows a screenshot of JAligner which is executed from the command line. JAligner aligns p53_human against p53_mouse using a BLOSUM62 scoring matrix and the similarity score is 1554.00. The output shows which sequences are matched or mismatched and how many sequences are matched.

5.3.2 Parallelizing a Program of the Smith-Waterman Algorithm

Parallel processing has started to become an indispensable component of bioinformatics, as the used datasets keep growing larger in response to discovering new biological knowledge. For example, genome databases are estimated to double every 12 months [23]. Figure

```
Sequence #1: p53_human
Sequence #2: p53_mouse
Length #1: 393
Length #2: 393
Matrix: BLOSUM62
Gap open: 10.0
Gap extend: 0.5
Length: 393
Identity: 304/393 (77.35%)
Similarity: 326/393 (82.95%)
Gaps: 6/393 (1.53%)
Score: 1554.00

p53_human          1 MEEPQSDPSVEPPLSQETFSDLWKLLPENNVLSPLPSQAMDDLMLSPDDI      50
                     |||.|||.|:|.||||||||.||||||..::| |.| ..||||:| |.|:
p53_mouse          4 MEESQSDISLELPLSQETFSGLWKLLPPEDIL-PSP-HCMDDLLL-PQDV      50

p53_human         51 EQWFTEDPGPDEAPRMPEAAPRVAPAPAAPTPAAPAPAPSWPLSSSVPSQ     100
                     |::|   .||.||.|:..|.....|....|.|||||||..|||||.||||
p53_mouse         51 EEFF---EGPSEALRVSGAPAAQDPVTETPGPAAPAPATPWPLSSFVPSQ      97

p53_human        101 KTYQGSYGFRLGFLHSGTAKSVTCTYSPALNKMFCQLAKTCPVQLWVDST     150
                     |||||:|||.||||.|||||||.|||||.|||:||||||||||||||.:|
p53_mouse         98 KTYQGNYGFHLGFLQSGTAKSVMCTYSPPLNKLFCQLAKTCPVQLWVSAT     147

p53_human        151 PPPGTRVRAMAIYKQSQHMTEVVRRCPHHERCSDSDGLAPPQHLIRVEGN     200
                     ||.|:||||||||:|||||||||||||||||||.||||||||||||||||
p53_mouse        148 PPAGSRVRAMAIYKKSQHMTEVVRRCPHHERCSDGDGLAPPQHLIRVEGN     197

p53_human        201 LRVEYLDDRNTFRHSVVVPYEPPEVGSDCTTIHYNYMCNSSCMGGMNRRP     250
                     |..|||:||.||||||||||||||.||:.|||||.|||||||||||||||
p53_mouse        198 LYPEYLEDRQTFRHSVVVPYEPPEAGSEYTTIHYKYMCNSSCMGGMNRRP     247

p53_human        251 ILTIITLEDSSGNLLGRNSFEVRVCACAGRDRRTEEENLRKKGEPHHELP     300
                     |||||||||||||||||:|||||||||.||||||||||.|||.....|||
p53_mouse        248 ILTIITLEDSSGNLLGRDSFEVRVCACPGRDRRTEEENFRKKEVLCPELP     297

p53_human        301 PGSTKRALPNNTSSSPQPKKKPLDGEYFTLQIRGRERFEMFRELNEALEL     350
                     |||.|||||..||:||..||||||||||||:||||:||||||||||||||
p53_mouse        298 PGSAKRALPTCTSASPPQKKKPLDGEYFTLKIRGRKRFEMFRELNEALEL     347

p53_human        351 KDAQAGKEPGGSRAHSSHLKSKKGQSTSRHKKLMFKTEGPDSD    393
                     |||.|.:|.|.||||||:||:|||||||||||.|.|..|||||
p53_mouse        348 KDAHATEESGDSRAHSSYLKTKKGQSTSRHKKTMVKKVGPDSD    390
```

Figure 5.5: Screenshot of a serial JAligner program: JAligner aligns p53_human against p53_mouse using a BLOSUM62 scoring matrix. The similarity score is 1554.00.

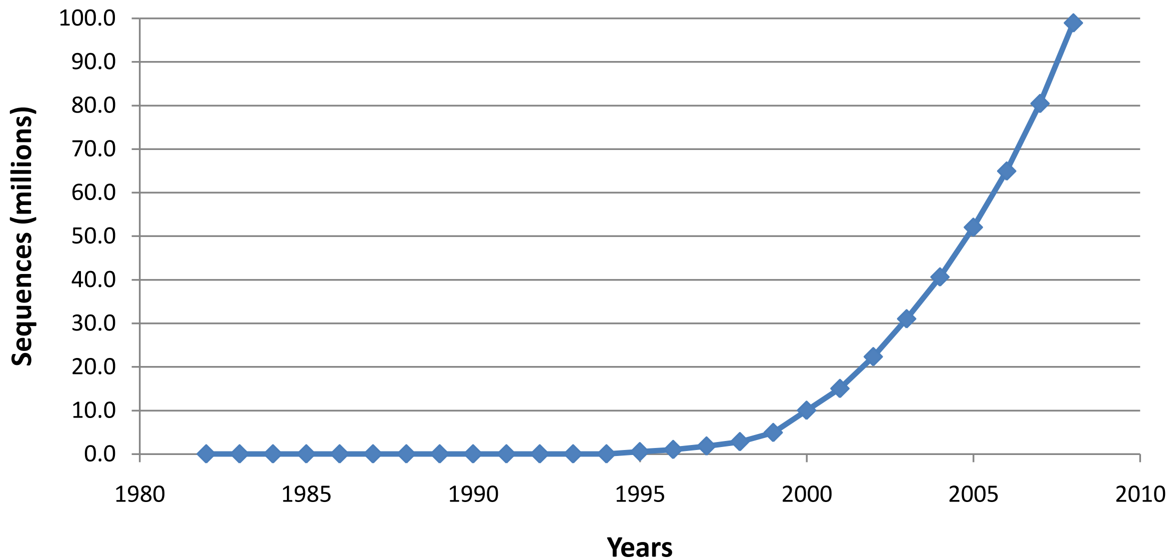


Figure 5.6: Growth of GenBank (1982-2008).

5.6 shows the growth of NCBI's GenBank database– the size of sequence databases have increased dramatically since the 1990s. To deal with the tremendous amount of data, bioinformatics applications are often computationally-intensive. As a result, the computational resources of a single computer are insufficient to meet the computation demands of aligning sequences against the latest databases. Consequently, parallel computation has entered the mainstream of bioinformatics research, and distributed multiprocessing systems are being used to solve bioinformatics problems.

To motivate the need for flexible dynamic updates in in-vivo enhancement of distributed applications, how the well-known Smith-Waterman alignment algorithm could be parallelized and developed incrementally to run in an ad-hoc grid environment is described. The sequential version of this algorithm [5] calculates a similarity score between two sequences. A parallelization of this algorithm will align an unknown sequence against an

entire database of known sequences, with the database partitioned among different computational nodes. The resulting computation will follow a simple Master Worker model, with the Master node assigning tasks to the Worker nodes as well as collecting and filtering the results. Specifically, the Master accepts an unknown sequence as input and sends it to individual Worker nodes. Each worker node aligns the unknown sequence against its portion of the partitioned database. The sequences having the highest similarity scores (e.g., above a given threshold) are then sent back to the Master. The Master collects the results, sorts them, and reports the top-ranked results to the user.

Parallelizing sequential algorithms is often fraught with unexpected complications. In fact, many sequential algorithms do not even have an efficient parallel version. Thus, it is common first to create a correct parallel version of a sequential algorithm and then to fine-tune it for efficient parallel execution in a given deployment environment. Incidentally, this approach aligns well with what is known as *separation of concerns* [36], a guiding principle for reducing the complexity of software development. The key insight behind separating concerns is that a computing problem is divided into parts, so that its different facets are isolated, enabling the programmer to reason about them independently. In this context, the concerns of correctness and performance could be dealt with independently. In addition, sometimes the programmer cannot even fully understand all the functional requirements of a parallel program, until it has been deployed and run under realistic conditions.

Sophisticated simulators make it possible for the programmer to test a grid application on a single machine and get a realistic picture how the application would work when deployed on the grid. Nevertheless, such simulators only come as a part of a mature grid infrastructure, and are not available for light-weight environments such as ad-hoc grids using

the JVM. Even if such a simulator were available, parameterizing it with the exact information about an ad-hoc grid would be a prohibitively difficult undertaking, on par with deploying and running the application on the grid.

Thus, after creating an initial parallelization of the Smith-Waterman alignment algorithm described above, the programmer could deploy and test it in its intended deployment environment. One common difference between sequential applications and their parallelizations is that the parallel version produces much more output data. It is quite likely, for example, that while in the sequential version of Smith-Waterman alignment algorithm, all the results could comfortably fit on the same output window, in the parallel version, the results would be more numerous. As a result, it is possible that the output data in the parallel version could only be properly examined, if they were saved to a disk file. Thus, the programmer may wish to change the piece of functionality that simply dumps the results to standard output to write them to a disk file instead.

It also may turn out that certain assumptions made during the design phase would no longer hold true. For example, the programmer may have assumed that the `float` precision would be sufficient for representing similarity scores, while after seeing the initial results realize that the `double` precision is needed.

Finally, it may turn out that the implementation of the alignment algorithm does not satisfy the expected performance or accuracy requirements. A slight variation of the algorithm could satisfy these requirements to a greater extent.

In-vivo enhancement makes it possible to fine-tune and troubleshoot a distributed application in its real deployment environment, while sidestepping the inefficiencies of a typical upgrade cycle through flexible dynamic updates.

Specifically, this implementation targets JVM-based distributed environments, which already come equipped with a facility that allows swapping classes in a running virtual machine, called HotSwap. However, the functionality provided by HotSwap is grossly insufficient to support the dynamic updates required for distributed computationally-intensive applications. As a concrete example, the dynamic updates required to address the deficiencies outlined above would be impossible with HotSwap. To address the insufficiencies of HotSwap to support the utility and efficiency of in-vivo enhancement, a novel binary rewriting approach that transforms the bytecode of a distributed computationally-intensive application is presented. These transformations enhance the bytecode with the capabilities required to enable flexible dynamic updates with the standard HotSwap, but they do so while incurring only minuscule performance overhead on the rewritten programs, as detailed in Section 5.1. To validate the expressive power of this approach, the Smith-Waterman alignment algorithm is dynamically updated to address the deficiencies described above and thoroughly document experiences.

5.3.3 Experimental Results

The novel binary rewriting techniques presented above enable effective dynamic updates of the Smith-Waterman parallelization used to motivate this work, thereby making it possible to perfect this application on the fly. As the experimental environment, a small grid of five nodes connected by a LAN has been assembled. The Ibis [4] grid infrastructure is used, even though the grid nodes communicated with each other through Java sockets rather than through the MPJ middleware [29] provided by Ibis. Figure 5.7 shows an example deployment of the parallelization.

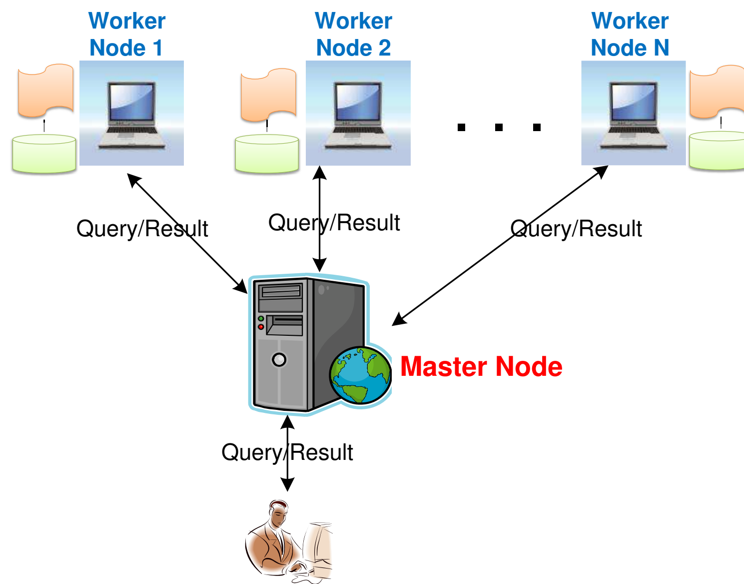


Figure 5.7: High-level view of parallelizing JAligner.

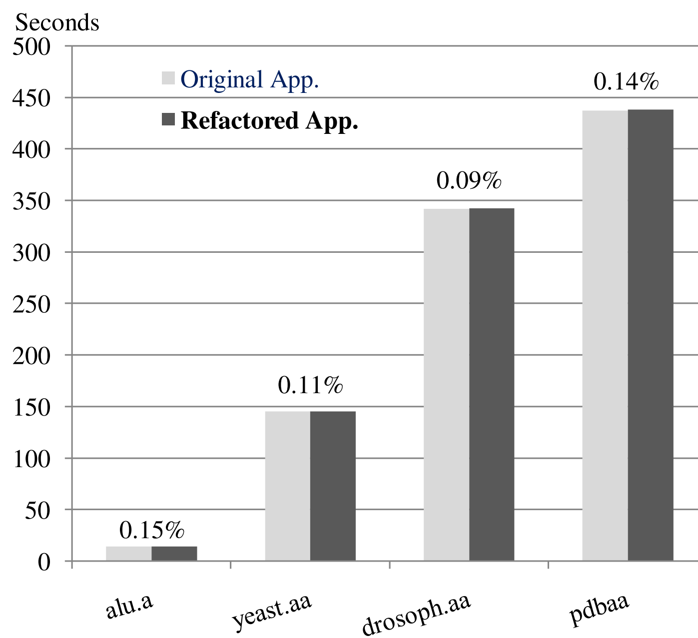
Cases	Requirements	# of updates				
		Field	Method	Class		
				Method body	Sig. change	Replaced classes
Case1: Console ⇨ File	Saving alignment results as a file	1	1	1	1	2
Case2: float ⇨ double	Displaying alignment results in a double precision	5	11	6	4	9
Case3: SW ⇨ SWG	A need of more practical alignment algorithm	0	4	1	1	2

Figure 5.8: Changes to Smith-Waterman alignment program using extended HotSwap. SW:Smith-Waterman algorithm [99], SWG:Smith-Waterman-Gotoh algorithm [48].

Recall that the required dynamic updates included changing the display method, the precision of the results, and the alignment algorithm used. As it turns out, all of these three updates involve structural changes to the bytecode, rendering the standard HotSwap facilities unsuitable for the task. Specifically, changing the display from the console to a disk file required replacing classes `AlignCommentLine` and `FileOutput`, as well as adding a new method `writeToFile`, thereby changing the signature of class `FileOutput`. Such a seemingly trivial change as using the `double` rather that the `float` precision for the similarity scores required modifications of 5 fields, 11 methods, and 9 classes! Because the similarly score is computed through the interaction of multiple methods in different classes, changing its type (i.e., from `float` to `double`) requires changing the signatures of all of the involved methods. Finally, modifying the alignment algorithm required modifying the signatures of 4 methods in 2 different classes. Because the base algorithms use different parameter sets, the methods' signatures, invoked when the algorithm is executed, had to be changed accordingly. Figure 5.8 presents the exact statistics of the changes involved.

For this case study, the presented binary rewriting infrastructure has been included into the standard class loading process. All the dynamic updates are initiated from the Master node, which has remote debugging connections to each Worker node.[1] The programmer interacts with an upgrade script that takes the classes of a new program version, compares this version with the current version, computes the necessary updates, and applies them dynamically through the remote debugging connection to the remote nodes. Figure 5.9 shows the indirection overhead on the rewritten Worker code. Because the cost of indi-

[1]Starting from JDK 1.4, remote debugging connections do not impose performance overhead, allowing programs to run at full speed.



DB	Size (KB)	# of seq.	DB	Size (KB)	# of seq.
alu.a	221	1,962	drosoph.aa	8,300	14,331
yeast.aa	3,321	6,312	pdbaa	22,169	37,882

Figure 5.9: Refactoring overhead on the worker portion of Smith-Waterman parallelization. x-axis: the databases names in FASTA format, y-axis: the total execution time.

rection is incurred *only* when invoking methods, and the Worker process does most of the computation within a single method, the overall overhead is negligible. Thus, the presented novel binary rewriting approach made it possible to use the standard HotSwap to update a running distributed application, without either having to modify the JVM or having to degrade the performance. Furthermore, the updates were applied without having to stop the parallel execution and wasting valuable HPC resources. These results indicate that in-vivo enhancement can become a valuable tool for delivering parallel solutions under tight deadlines.

5.4 Updating a Scientific Application Dynamically

In this section two scientific applications have been used to demonstrate the efficiency and flexibility of the presented approach to dynamic software updates.

5.4.1 Enhancing a Successive Over-Relaxation Program

To demonstrate the efficiency of the presented approach to dynamic software updates, the total execution time of a Successive Over-Relaxation (SOR) [51, 89] program is compared with that of its rewritten version.

In SOR, an iterative method is used to solve a large system of equations by finding successive approximations to the solution starting from an initial guess. The iterative method changes the approximation in several steps to bring it closer to the solution. Successive Over-Relaxation is an iterative method for solving a linear system of equations and speeds

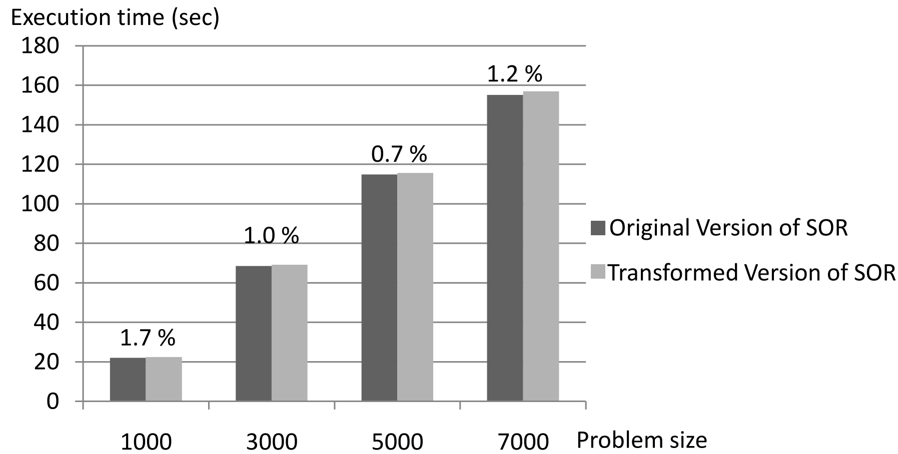


Figure 5.10: Refactoring overhead on Successive Over-Relaxation. x-axis: the problem size; y-axis: the total execution time of both the original and the enhanced versions.

up convergence of a slow-converging process. The parallel implementation of SOR is based on the Red/Black SOR algorithm which divides the grid points into two classes, red and black and orders left to right, bottom to top within each class. A program divides the red and the black array into roughly equal size bands of rows, assigning each band to a different processor. Each processor has (number of rows/number of processors) rows and needs to communicate with its neighbors to exchange boundary rows. All red points are calculated in parallel using the black points and then all the black points are calculated in parallel using the red points.

The original SOR program written in MPJ is rewritten by the bytecode rewriter which changes direct references to proxy references to make the program updateable. Each CPU exchanges one row of the matrix with its neighbors at the beginning of each iteration. The measurements were conducted on a compute cluster, with each node running a dual processor AMD Opteron 240 (1.4Ghz), 1GB RAM, CentOS version 4.2, JDK version 1.5.0,

connected by Myrinet (4Gbit). Figure 5.10 shows the total overhead of the rewritten version never exceeds 2%.

5.4.2 Updating a Molecular Dynamics Simulation System Dynamically

A parallel Molecular Dynamics Simulation (MDS) program [66, 91] which was deployed on Ibis [4], a Java-based grid programming environment was used. Among other services, Ibis provides a Java API for MPI-like message passing among cluster nodes.

Molecular Dynamics Simulations [97] is a form of computer simulation to model the structure, motions, and interactions of molecular systems, including proteins, cell membranes, and DNA, at an atomic level of detail. The MDS program was updated dynamically twice, updating the thermostat algorithm and the number of molecules. The thermostat algorithm maintains or rescales the temperature constant of a molecular system by increasing or decreasing the velocity of the molecules. Therefore, the selection of an appropriate thermostat method depends on the molecular system in use. Also the initial number of the molecules may need to change during the simulation. Figure 5.11 depicts the main modules of an MDS program and the thermostat module that are updated dynamically. Table 5.3 summarizes the aforementioned scenarios, which motivate dynamic changes.

While the changes above may seem simple, without dynamic software updating facilities, they would require stopping the parallel execution and losing valuable computing resources. Furthermore, these updates could not be accomplished by using HotSwap alone. In fact, trying to use HotSwap for these updates would throw an exception terminating the

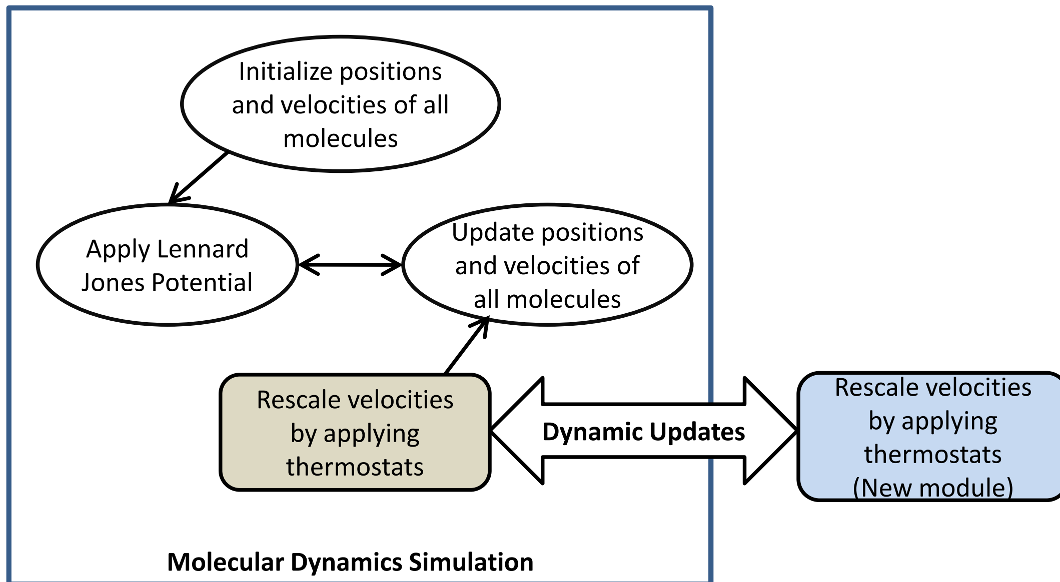


Figure 5.11: Updating the rescaling module of a molecular dynamics simulation.

Updates	Requirements	Implementations
Thermostat algorithm	Rescale velocities of molecules by replacing the thermostat algorithm	Adding a new method `rescale(mol[] m, int size)` and fields
The number of molecules	Increase/decrease the number of molecules to be simulated	Adding a new method `updateNumOfMols(int size)`

Table 5.3: Changes to the Molecular Dynamics Simulation.

program's execution. Finally, these changes are a natural consequence of delivering solutions under tight deadlines. It is not always possible to put enough care into designing a distributed parallel application, so that it always satisfies the requirements of different users.

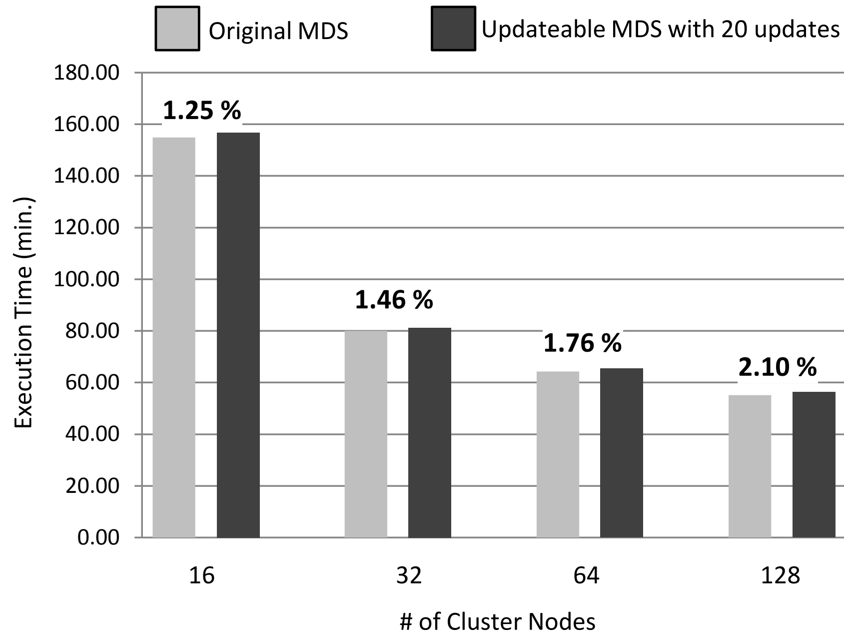


Figure 5.12: Performance overhead on the update management code. `Bcast` and `Barrier` have been executed 200 and 20 times, respectively.

5.5 Updating High Performance Cluster Applications Dynamically

As our evaluation environment, we have used SystemGreen (SystemG)—a state-of-the-art research supercomputer recently constructed at Virginia Teach. SystemG features top-of-the-line components, typically found in a modern compute cluster: each compute node runs two Intel Xenon processors with 4 cores each (a total of 8 cores), 8GB RAM, and Fedora Core 10. The nodes are connected by InfiniBand (10Gbit+). For all the experiments, we used JDK 1.6.0_0.

The goal of our evaluation was to ensure that the injected update management

code, discussed above, imposes a negligible performance overhead. If the goal of dynamic software updates is to reduce time-to-discovery, then high costs of executing the update management code would offset—if not eliminate altogether—the desired overall performance improvements. Since our update management code implements a distributed synchronization algorithm, its performance is thus dependent on the number of nodes. The higher the number of nodes that have to be updated consistently, the higher the cost of synchronizing them.

As our benchmark application, we have used a parallel Molecular Dynamics Simulation (MDS) application, a benchmark that is distributed with MPJ Express [20], a middleware library that provides Java bindings to the majority of MPI calls. Thus, although each individual node runs Java, the nodes communicate with each other through MPI. Therefore, our findings should be representative of a typical compute cluster application, and our setup does not unfairly benefit our approach.

MDS applications [66, 97] model the structure, motions, and interactions of molecular systems, including proteins, cell membranes, and DNA, at an atomic level of detail through a parallel computer simulation. The primary method of an MDS application includes the main execution loop, which simulates the movement of particles by running code on multiple nodes in parallel and synchronizing their execution through MPJ calls. Specifically, the main loop includes the routines that move particles, calculate the force on particles, and average the velocity of all particles. The size of the simulation is controlled by adjusting the number of times the particles are moved `nstep`. We ran the simulation for `nstep` = 200 and 500. The update management code was inserted in the beginning of the main execution loop. Thus, the update management code was executed once per iteration step. We also randomly performed 20 dynamic updates. To capture the overhead of managing the updates, we have

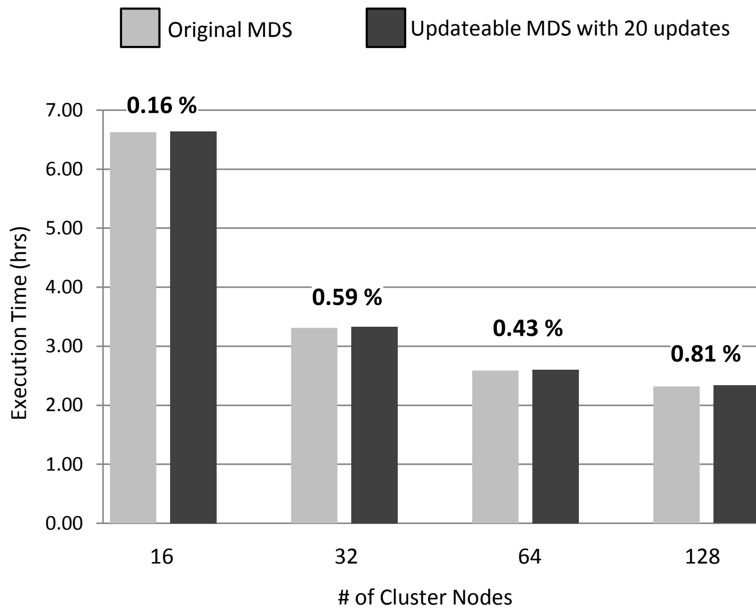


Figure 5.13: Performance overhead on the update management code. `Bcast` and `Barrier` have been executed 500 and 20 times, respectively.

swapped the running class versions with the same classes. Thus, even though classes were swapped, the changes did not have any effect on the execution of the application.

Figures 5.12 and 5.13 show the performance overhead imposed by our update management code for 200 and 500 simulation steps. We compared the total execution time of the original and the enhanced for dynamic updates versions of the MDS application on 16, 32, 64, and 128 nodes. The overhead is incurred mainly by the MPI `Bcast` and `Barrier` calls: while `Bcast` is executed on each iteration of the main execution loop, `Barrier` is called only if a dynamic update has taken place (i.e., `Barrier` is called 20 times). As expected, the performance overhead tends to increase with the number of nodes. Nevertheless, the total overhead never exceeds 2%. Furthermore, as the number of iterations increases, the total overhead of executing our update management code decreases.

Note that the total execution time of each simulation can be as high as 8 hours depending on the configuration. If the running code needs to be changed during these 8 hours, the ability to update the code dynamically, in our view, will be well worth the overhead of the update management code, which adds up to less than 10 minutes [2].

[2] 2% of 8 hours is $480\,mins * .02 \approx 10\,mins$.

Chapter 6

Future Work and Conclusions

6.1 Future Work

There are a wide variety of possible extensions to the work described in this dissertation. This section considers some of the research directions that can be followed to further develop the insights brought about by this dissertation.

- **Extending the HotSwap implementation**

  This work has used bytecode rewriting to overcome limitations of HotSwap which can constrain dynamic software updates. However, we can extend the current implementation of HotSwap so that it can support schema changes of classes. If virtual machines provide such an extended HotSwap facility, the user can easily update applications on the fly.

- **Supporting VM-based high-productivity languages**

  The presented approach in this work can benefit new VM-based high-productivity languages including X10 and Fortress. Since their new language features are likely to pose new challenges for dynamic software updates, flexible approaches to dynamic software updates may be needed.

- **Dynamically updating .NET applications**

  The standard JVM was selected as the experimental platform in this dissertation. However, the presented approach to dynamic software updates in this work might be applicable not only for Java but also for other object-oriented programming languages. Specifically, as Java programs execute on the JVM, C# programs execute on the Common Language Runtime in .NET platform which supports dynamic code loading. Virtual Superclass binary refactoring presented in this research could be used to provide proxies for C# classes by modifying the C# bytecode representation.

- **Formalizing the Virtual Superclass binary refactoring**

  Virtual Superclass binary refactoring makes it possible to update computationally-intensive applications on the fly without imposing significant performance degradation. Even though thorough experiments have demonstrated the applicability of this work, formal approaches including FJ [58] and MJ [40] should be used in order to represent the formal semantics of Virtual Superclass binary refactoring.

- **Applying the developed techniques to conscientious software**

  Conscientious software is a recently proposed paradigm for developing reliable, self-sustaining software systems [39]. To keep conscientious software systems alive, they gather information on the running application, reason about problems, and apply rec-

ommended remedies [41]. Since conscientious software systems are continuously updated over time, dynamic software updates is an essential part of them. Therefore, the efficient and flexible dynamic software updates presented in this dissertation can be used to realize JVM-based conscientious software systems.

6.2 Conclusions

This dissertation has shown that updating computationally-intensive applications dynamically can reduce their time-to-discovery metrics—the total time it takes from posing a problem to arriving at a solution—and, as such, should become an intrinsic part of their software lifecycle. In light of this observation, we believe that there is value in other DSU researchers applying their expertise to the domain of computationally-intensive applications. To help their efforts, this section shares several guidelines for applying DSU to CIA applications. It then summarizes the contributions of this dissertation.

6.2.1 Guidelines for Applying DSU to CIAs

- **Create intuitive user interfaces to dynamic updates for non-experts**

  CIAs are often written by domain experts—scientists and engineers—who possess detailed knowledge in their respective domains but may lack a deep understanding of computing or experience with the state of the art in software engineering. Therefore, to ensure the success of dynamic update systems for this type of users, the barrier to adoption must be lowered by providing intuitive user interfaces such as the graphical interfaces typically implemented for modern desktop applications.

Providing an intuitive graphical user interface, in and of itself, is not sufficient to accommodate non-expert users, but it enables an important property—it minimizes the need for direct human intervention in the tedious and error-prone aspects of interacting with a typical dynamic update system.

- **Making a CIA dynamic updates capable must not hurt its performance**

  Since the raison d'être of parallel CIAs is to reduce *time-to-discovery*—the total time it takes from posing a problem to arriving at a solution—dynamic updates must be considered in that broader context as well. The only reason to even consider enhancing a CIA with dynamic update capabilities is because DSUs can help avoid having to restart the computation from scratch, thereby saving valuable computation cycles. However, making a CIA dynamic updates capable must not impose a significant performance overhead, lest it will negate all the benefits of DSUs. A good general rule of thumb is for the performance overhead imposed by introducing dynamic update capabilities to a CIA not to exceed a couple of percentage points. Furthermore, as based on our experiences, the overhead of a DSU system must be assessed not only using micro-benchmarks but also using realistic CIAs.

- **Use portable dynamic update systems**

  Since most CIAs are executed in environments featuring commodity hardware, introducing DSU must take this fact into consideration. Specifically, those dynamic update systems that were designed to work with special-purpose hardware or software configurations (requiring specialized OS/middleware) are likely to be ill-suited for a typical CIA. To alleviate the need for conformity when operating within the constraints im-

posed by commodity hardware and standard system software, one can introduce DSUs to a CIA by means of virtualization. A typical virtual machine is an abstraction layer that can provide a common execution platform in the presence of heterogeneous OS and hardware environments on a CIA, thereby increasing the portability of a DSU solution at hand. For example, this research has leveraged the portability benefits provided by the Java Virtual Machine.

6.2.2 Summary

This dissertation has established the value of and recommended practices for applying dynamic software updates to computationally-intensive applications, providing models and tools for practical dynamic software updates. This research proposes using dynamic software updates to reduce the time-to-discovery metrics of high performance parallel applications.

This dissertation has provided an extensive overview of the state of the art in dynamic software updates and presented our own approach for dynamically updating high performance parallel applications. Our approach demonstrates the benefits of dynamic software updates for the domain of high performance parallel applications and contributes several innovative techniques. One technique employs binary rewriting to overcome constraints of the HotSwap API to support a wider range of dynamic changes. Another technique introduces an algorithm for ensuring consistency when dynamically updating multiple concurrent parallel tasks.

The effectiveness of the presented approach was demonstrated through case studies that involved HPC applications from the bioinformatics and molecular dynamic simulation

domains. We have benchmarked a parallel molecular dynamics simulation, comparing the performance of the original version and the one enhanced with the implementation of our distributed synchronization algorithm. The results show that the overhead of our approach is negligible and can be justified by the added ability to update parallel execution on the fly. These case studies as well as additional performance benchmarks have shown that dynamic software updates can be a powerful tool for shortening the time-to-discovery in distributed computationally-intensive applications.

High performance computing researchers and practitioners alike are starting to realize the potential of reducing the development and fine-tuning component of the time-to-discovery in parallel high performance applications. This dissertation has demonstrated the benefits of applying dynamic software updates to parallel high performance applications. Furthermore, the efficiency of our dynamic update system provides useful insights that can benefit the design of similar systems in the future. These and other potential benefits, in our view, warrant making dynamic software updates an essential part in the software life cycle of engineering computationally-intensive applications.

Bibliography

[1] AspectJ, http://www.eclipse.org/aspectj/.

[2] AspectWerkz, http://aspectwerkz.codehaus.org/.

[3] Byte Code Engineering Lab (BCEL), http://jakarta.apache.org/bcel/.

[4] Ibis: Grids as Promised, http://www.cs.vu.nl/ibis/.

[5] JAligner, http://jaligner.sourceforge.net/.

[6] Java dynamic proxy, http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[7] Java HotSwap, http://java.sun.com/j2se/1.4.2/docs/guide/jpda/enhancements.html.

[8] Java Reflection, http://java.sun.com/docs/books/tutorial/reflect/index.html.

[9] Javassit, http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[10] JBoss AOP, http://labs.jboss.com/jbossaop/.

[11] JDrums, http://www.ida.liu.se/~jengu/jdrums/.

[12] Message Passing Interface, http://www.mcs.anl.gov/mpi.

[13] ObjectWeb ASM, http://asm.objectweb.org/.

[14] SpecJVM98 Benchmarks, http://www.spec.org/jvm98/.

[15] Practical autonomic computing: Roadmap to self managing technology. *White Paper prepared for IBM Corporation*, January 2006.

[16] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[17] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The fortress language specification. *Sun Microsystems, Inc.*, March 2007.

[18] B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 108–124, New York, NY, USA, 2001. ACM.

[19] J. Armstrong, R. Virding, C. Wikstrm, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996,.

[20] M. Baker, B. Carpenter, and A. Shaft. MPJ Express: Towards thread safe Java HPC. *Cluster Computing, IEEE International Conference on*, 0:1–10, 2006.

[21] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Software*, 25(4):29–36, 2008.

[22] R. Bellman. *Dynamic Programming*. Dover Publications, 2003.

[23] D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, B. Rapp, and D. Wheeler. GenBank. *Nucleic Acids Res.*, 30:17–20, 2002.

[24] A. Bhowmik and W. Pugh. A secure implementation of Java inner classes. *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, poster session*, May 1999.

[25] R. P. Bialek. Dynamic updates of existing Java applications. *Ph.D. Thesis, the University of Copenhagen*, pages 1–216, June 2006.

[26] G. Bierman, M. Parkinson, and J. Nob. UpgradeJ: Incremental typechecking for class upgrades. 2007.

[27] W. Binder, J. Hulaas, and P. Moret. Advanced Java bytecode instrumentation. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144, New York, NY, USA, 2007. ACM.

[28] T. Bloom and M. Day. Reconfiguration and module replacement in argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.

[29] M. Bornemann, R. van Nieuwpoort, and T. Kielmann. Mpj/ibis: A flexible and efficient message passing platform for Java. In *PVM/MPI*, pages 217–224, 2005.

[30] M. Caporuscio, A. D. Marco, and P. Inverardi. Model-based system reconfiguration for dynamic performance management. *J. Syst. Softw.*, 80(4):455–473, 2007.

[31] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[32] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.

[33] X. Chen. Extending RMI to support dynamic reconfiguration of distributed systems. *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 401–408, 2002.

[34] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE'03)*, pages 364–376, 2003.

[35] P. David, T. Ledoux, and N. Bouraqadi. Two-step weaving with reflection using aspectj. OOPSLA'01 Workshop on "Advanced Separation of Concerns in Object-Oriented Systems", October 2001.

[36] E. Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[37] P. Eugster. Uniform proxies for Java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 139–152, 2006.

[38] D. Flanagan, J. Farley, W. Crawford, and K. Magnusson. *Java Enterprise in a nutshell.* O'Reilly & Associates, 1999.

[39] S. Fleissner and E. L. A. Baniassad. Epi-aspects: aspect-oriented conscientious software. *SIGPLAN Not.*, 42(10):659–674, 2007.

[40] M. P. G. Bierman and A. Pitts. Mj: An imperative core calculus for java and java with effects. In *Technical Report 563, University of Cambridge Computer Laboratory,*, 2004.

[41] R. P. Gabriel and R. Goldman. Conscientious software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 433–450, New York, NY, USA, 2006. ACM.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[43] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[44] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM.

[45] B. Gharaibeh, D. Dig, T. N. Nguyen, and J. M. Chang. dReAM: Dynamic refactoring-aware automated migration of Java online applications. *Technical Report, Iowa State University*, August 2007.

[46] C. Gillecorresponding and P. N. Robinson. Hotswap for bioinformatics: A strap tutorial. *BMC Bioinformatics*, 7(64), February 2006.

[47] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. *Technical Report, The University of Edinburgh*, December 1997.

[48] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[49] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.

[50] J. Gustavsson, T. Staijen, and U. Assmann. Runtime evolution as an aspect. *First International Workshop on Foundations of Unanticipated Software Evolution*, 2004.

[51] B. Harbulot and J. R. Gurd. Using aspectj to separate concerns in parallel scientific Java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 122–131, New York, NY, USA, 2004. ACM.

[52] S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 70, Washington, DC, USA, 1996. IEEE Computer Society.

[53] M. Hicks. *Dynamic Software Updating.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[54] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.

[55] G. Hjálmtýsson and R. Gray. Dynamic c++ classes: a lightweight mechanism to update code in a running program. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 1998. USENIX Association.

[56] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2005. IEEE Computer Society.

[57] C. Hofmeister. Dynamic reconfiguration. *Ph.D. Thesis, University of Maryland*, 1993.

[58] A. Igarashi, B. Pierce, and P. Wadler. Featherwieght java: a minimal core calculus for java and gj. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 132–146, New York, NY, USA, 1999. ACM.

[59] P. Kang, Y. Cao, N. Ramakrishnan, C. J. Ribbens, and S. Varadarajan. Modular implementation of adaptive decisions in stochastic simulations. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 995–1001, New York, NY, USA, 2009. ACM.

[60] P. Kang, N. K. C. Selvarasu, N. Ramakrishnan, C. J. Ribbens, D. K. Tafti, and S. Varadarajan. Modular, fine-grained adaptation of parallel programs. In *ICCS*, pages 269–279, 2009.

[61] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[62] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[63] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11(4):424–436, 1985.

[64] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11(4):424–436, 1985.

[65] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[66] A. Kumar. Molecular Dynamics Simulations. `http://www.personal.psu.edu/auk183/MolDynamics/Molecular%20Dynamics%20Simulations.html`.

[67] O. Layaida, S. B. Atallah, and D. Hagimont. Reconfiguration-based qos management in multimedia streaming applications. In *EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference*, pages 248–255, Washington, DC, USA, 2004. IEEE Computer Society.

[68] I. Lee. DYMOS: A Dynamic Modification System. *Ph.D. Thesis, University of Wisconsin, Madison*, April 1983.

[69] Y.-F. Lee and R.-C. Chang. Java-based component framework for dynamic reconfiguration. *IEE Proceedings - Software*, 152(3):110–118, June 2005.

[70] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 36–44, 1998.

[71] H.-Y. Liao, M.-L. Yin, and Y. Cheng. A parallel implementation of the smith-waterman algorithm for massive sequences searching. volume 2, pages 2817–2820 Vol.4, 2004.

[72] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[73] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. *Lecture Notes In Computer Science, Proceedings of the 14th European Conference on Object-Oriented Programming*, 1850:337–361, June 2000.

[74] D. A. Menascé, H. Ruan, and H. Gomaa. A framework for qos-aware software components. In *WOSP '04: Proceedings of the 4th International Workshop on Software and Performance*, pages 186–196, New York, NY, USA, 2004. ACM.

[75] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the internet inter-orb protocol interface to provide corba with fault tolerance. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 6–6, Berkeley, CA, USA, 1997. USENIX Association.

[76] I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2009.

[77] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 72–83, New York, NY, USA, 2006. ACM.

[78] I. G. Neamtiu. Practical dynamic software updating. *Ph.D. Thesis, University of Maryland*, pages 1–212, August 2008.

[79] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. In *Conference on Advanced Information Systems Engineering (CAiSE)*, pages 125–138, 2005.

[80] A. Nicoara and G. Alonso. Dynamic aop with PROSE. In *Conference on Advanced Information Systems Engineering (CAiSE)*, pages 125–138, 2005.

[81] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, systematic, and efficient code replacement for running Java programs. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246, New York, NY, USA, 2008. ACM.

[82] OMG. Unified Modeling Languages, `http://www.uml.org/`.

[83] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[84] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant Component Interactions with JINSI. In *Proceedings of the Fourth International ICSE Workshop on Dynamic Analysis (WODA 2006)*, pages 3–9, Shanghai, China, May 2006.

[85] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, may 2005.

[86] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, May 2005.

[87] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, October 2002.

[88] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, October 2002.

[89] J. M. Ortega. *Introduction to Parallel Vector Solution of Linear Systems*. Plenum Press, New York, NY, USA, 1988.

[90] S. C. Previtali and T. R. Gross. Dynamic updating of software systems based on aspects. *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 83 – 92, September 2006.

[91] D. C. Rapaport. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, New York, NY, USA, 1996.

[92] A. Rasche, W. Schult, and A. Polze. Self-adaptive multithreaded applications: a case for dynamic aspect weaving. *Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, 2005.

[93] T. Ritzau and J. Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, May 2000.

[94] T. Ritzau and J. Andersson. Dynamic deployment of Java applications. *In Java for Embedded Systems Workshop*, May 2000.

[95] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

[96] A. Shafi, A. Hussain, and J. Raza. A parallel implementation of the finite-domain time-difference algorithm using MPJ Express. pages 1–6, April 2008.

[97] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. *Commun. ACM*, 51(7):91–97, 2008.

[98] F. Shull, J. Carver, L. Hochstein, and V. Basili. Empirical study design in the area of high-performance computing (hpc). *Empirical Software Engineering, International Symposium on*, 0:10 pp., 2005.

[99] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[100] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*, 2003.

[101] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates for Java: A VM-centric approach. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2009.

[102] A. T. Tai, K. S. Tso, L. Alkalai, S. N. Chau, and W. H. Sanders. On low-cost error containment and recovery methods for guarded software upgrading. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, page 548, Washington, DC, USA, 2000. IEEE Computer Society.

[103] E. Tilevich and Y. Smaragdakis. Binary refactoring: Improving code behind the scenes. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 264–273, May 2005.

[104] E. Tilevich and Y. Smaragdakis. Transparent program transformations in the presence of opaque code. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 89–94, New York, NY, USA, 2006. ACM.

[105] R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[106] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with Expanders. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN con-*

ference on Object-oriented programming systems, languages, and applications, pages 37–56, 2006.

[107] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 85–92, New York, NY, USA, 2002. ACM.

[108] I.-C. Yoon, A. Sussman, and A. Porter. And away we go: understanding the complexity of launching complex HPC applications. In *SE-HPCS '05: Proceedings of the second International Workshop on Software Engineering for High Performance Computing System Applications*, pages 45–49, New York, NY, USA, 2005. ACM.

Appendix A

Sample Code

A.1 Generation of a proxy and virtual superclass

Virtual Superclass binary refactoring changes class `A` to extend a *virtual* superclass `SuperA`. In other words, the virtual superclass is inserted into the class's inheritance hierarchy. The original class `A` becomes a proxy, and the virtual superclass contains all the original functionality, including both method bodies and fields. The proxy simply delegates to the virtual superclass. Code listing A.1 shows a target class, which is transformed into a proxy class in Listing A.2. A new superclass listed in A.3 has actual methods instead of the proxy class. The target class listed in A.1 is transformed into the proxy class shown in A.2–some fields and methods, such as `myVersion`, `invoke`, and `checkVersion` are added. Those new constructs are necessary to support the extended hotswap. A special `invoke` method provides a facility for expressing the invocation of newly added methods without changing the updated class's

signature. To keep track of the version information, a static version field `myVersion` and method `checkVersion` are introducted. `checkVersion` on a proxy checks if the version of a class is older than the latest version. If so, a special initialization method is invoked for the newly added state and the class version is updated to the latest version. Listing A.3 shows a virtual superclass of the target class, which has some Hotswap-supporting methods and an application method, `foo`. This `foo` method has an actual method body and is executed when the proxy class delegates a method call.

```
1   //An original class
2   public class A {
3     int myI=0;
4     int myJ=0;
5
6     void foo(int i, int j) {
7       myI=i;
8       myJ=j;
9     }
10  }
```

Listing A.1: Source code of a target class.

```
1   //A proxy class
2   public class A extends SuperA{
3     //HotSwap-supporting fields
4     private int myVersion = -1;
5
6     //HotSwap-supporting methods
7     public A(Bogus b) {
8       super(b);
9       ObjectTracer.currentObject.put("A", this);
10    }
11    final Object invoke(String methodName, Class[] argTypes, Object[] args) {
12      Object obj=null;
13      try{
14        Class[] argtypes = {this.getClass()};
15        Object[] as = {this};
```

```
16            Method setSuperMethod = getMEObject().getClass().getDeclaredMethod("setSuper",
                  argtypes);
17            setSuperMethod.invoke(null, as);
18
19            Method m = getMEObject().getClass().getDeclaredMethod(methodName, argTypes);
20            obj = m.invoke(null, args);
21         }catch(Exception e){e.printStackTrace();}
22         return obj;
23      }
24      private final void checkVersion(){
25         if (myVersion < getCurrentVersion()) {
26            try{
27               Class[] argtypes = {this.getClass()};
28               Object[] as = {this};
29               Method newStateInitMethod = getMEObject().getClass().getDeclaredMethod("
                     newStateInit", argtypes);
30               newStateInitMethod.invoke(null, as);
31               myVersion = getCurrentVersion();
32            }catch(Exception e){e.printStackTrace();}
33         }
34      }
35
36      //Application-specific methods
37      void foo(int i, int j) {
38         checkVersion();
39         super.foo(i, j);
40      }
41   }
```

Listing A.2: Source code of a proxy class.

```
1    //Virtual Superclass
2    class SuperA {
3       int myI=0;
4       int myJ=0;
5
6       //HotSwap-supporting fields
7       public static final int VERSION = 1;
8       public Object newFieldObj = null;    //for a new class for new fields
9
10      //HotSwap-supporting methods
```

```
11        public SuperA(Bogus b) { //For adding new constructors
12          super(b);
13        }
14        private void initializeNewFields (){
15        }
16        public int getCurrentVersion(){
17          return VERSION;
18        }
19        public static Object getMEObject(){
20          return null;
21        }
22
23        //Application-specific methods
24        void foo(int i, int j) {
25          myI=i;
26          myJ=j;
27        }
28      }
```

Listing A.3: Source code of a virtual superclass.

A.2 Bytecode rewriting using Javassist

Javassist enables programmers to edit class files by inserting Java-like source code without full understanding Java bytecode instructions. The inserted source code is compiled into bytecode by a simple Java compiler provided with Javassist. Those source-level APIs provide the ability to insert source code at the beginning or end of a method body, alter a method body, and add a new method or field. Listing A.4 shows an example of adding an `invoke` method using Javassist. This example code takes a class `A` and creates a proxy class `A` with an `invoke` method and a superclass `SuperA`. `addNewMethods` shown in Listing A.4 is used to add the `invoke` method, which expresses the invocation of newly added methods. Some code in this `addNewMethods` indicates the method body of the `invoke` method. This source

code is compiled into bytecode, which is inserted into the proxy class.

```
1   import javassist.*;
2   import javassist.bytecode.*;
3   import java.io.*;
4
5   public class Transformer {
6     static ClassPool pool = null;
7     static CtClass superCc = null;
8     static CtClass modifiedA = null;
9     static CtClass originalA = null;
10
11      public static void main(String[] args) throws Exception{
12      pool = ClassPool.getDefault();
13      pool.importPackage("java.lang.reflect");
14      originalA = pool.get("A");  //a target class
15
16      InputStream classstream = new FileInputStream(new File("A.class"));
17      superCc = pool.makeClass(classstream);
18      superCc.setName("SuperA"); //a superclass
19      superCc.writeFile();
20
21      modifiedA = pool.makeClass(originalA.getName());  //a proxy class
22      modifiedA.setSuperclass(superCc);
23
24      CtConstructor[] ccs=originalA.getDeclaredConstructors();
25      for(int i=0;i<ccs.length;i++){
26        CtConstructor newc = CtNewConstructor.copy(ccs[i], modifiedA, null);
27        newc.setBody("super($$);");
28        modifiedA.addConstructor(newc);
29      }
30
31      CtMethod[] cms=originalA.getDeclaredMethods();
32      for(int i=0;i<cms.length;i++){
33        CtMethod newm = CtNewMethod.copy(cms[i], modifiedA, null);
34        newm.setBody("return super."+cms[i].getName()+"($$);");
35        modifiedA.addMethod(newm);
36      }
37
38        try{
39          addNewMethods();
40        }catch(Exception e){e.printStackTrace();}
```

```
41          }
42          public static void addNewMethods() throws Exception{
43          try {
44            modifiedA.getDeclaredMethod("invoke");
45              System.out. println ("invoke() is already defined in A.");
46              modifiedA. writeFile () ;
47          }catch (NotFoundException e) {
48            CtMethod invoke = CtNewMethod.make(
49              "Object invoke(String methodName, Class[] argTypes, Object[] args) {
                  return null;}", modifiedA);
50
51            String helperClass = "AddedMethods_" + modifiedA.getSimpleName();
52            StringBuffer code = new StringBuffer();
53            code.append("{ \n");
54            code.append("Object obj=null; \n");
55            code.append("try{ \n");
56            code.append("Class c = Class.forName(\"" + helperClass + "\"); \n");
57            code.append("Method m = c.getDeclaredMethod($1, $2); \n");
58            code.append("obj = m.invoke(null, $3); \n");
59            code.append("}catch(Exception e){e.printStackTrace();} \n");
60            code.append("return ($r)obj; \n");
61            code.append("} \n");
62
63            modifiedA.addMethod(invoke); //Add an invoke method to A
64            invoke.setBody(code.toString());
65            modifiedA. writeFile () ;
66
67              System.out. println ("invoke() was added.");
68          }
69        }
70      }
```

Listing A.4: Adding “invoke” method.

Figure A.1 shows the results of `Transformer` class listed in Listing A.4. The original class `A` has no `invoke` method–the upper part of Figure A.1 shows that `A` has one constructor and one public method. A bytecode instrumenter `Transformer` was run, and an output message says the `invoke` method was added successfully. The `Transformer` class changed

the bytecode of `A` by adding a new `invoke` method. The result of `javap` execution as illustrated in the lower part of Figure A.1 shows an `invoke` method has been added.

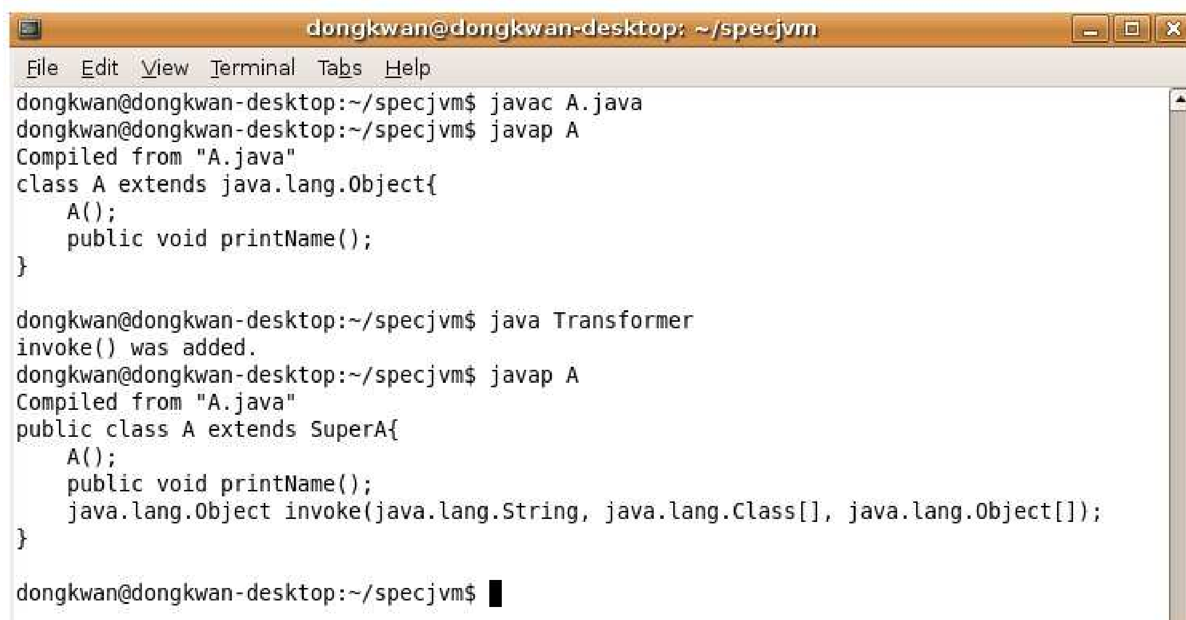


Figure A.1: Screenshot of Transformer class.

A.3 HotSwap Screenshot

This section describes how to use the HotSwap program with standard Java APIs. The HotSwap program uses `tools.jar` file, thereby `%JDK_HOME%\lib\tools.jar` must be added to the CLASSPATH. Then a swapped program can be executed illustrated in Figure A.2. The target program needs to start with the appropriate socket transport debugging options like

java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=8000 <classnames>

The target program prints out its version number shown in Figure A.2. The version number

of the program has been changed from 1 to 2. That indicates the target class has been replaced with a new one during the hotswap activity.

```
Command Prompt
C:\Original>java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=8000 Main
Listening for transport dt_socket at address: 8000
Version 1.
Version 1.
Version 1.
Version 1.
Version 1.
Version 1.
Version 1.
Version 1.
Listening for transport dt_socket at address: 8000
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
====> Version 2.<===
```

Before HotSwap

HotSwap is happening

After HotSwap

Figure A.2: Executing a target program.

The HotSwap program runs like

java HotSwap <host> <port> <a list of classnames>

Figure A.3 shows the successful execution of the HotSwap program. If the swapping process is failed, the program prints out an exception message.

A.4 MPJ Express

MPJ Express [20, 96] is an implementation of MPI-like bindings for Java and a Java messaging system that enables the programmer to write parallel programs with Java under a

Figure A.3: Executuing a HotSwap program.

```
1  import mpi.*;
2
3  public class HelloWorld {
4    public static void main(String args[]) throws Exception {
5      MPI.Init(args);
6      int me = MPI.COMM_WORLD.Rank();
7      int size = MPI.COMM_WORLD.Size();
8      System.out.println("Hello World from <"+me+">");
9      MPI.Finalize();
10   }
11 }
```

Listing A.5: "Hello World" program written in MPJ Express.

compute cluster environment. MPJ Express provides a set of methods for message passing among MPJ Express processes including blocking (e.g., `Send()` and `Recv()`) and non-blocking methods (e.g., `Isend()` and `Irecv()`). It also supports collective communications including `Broadcast`, `Barrier`, `Reduce`, etc.

This section shows a simple MPJ Express program and how to run the program with the MPJ Express runtime. MPJ Express includes a runtime module (i.e., daemon) that should be executed on cluster nodes involved. Programmers can run MPJ Express applications using the `mpjrun` module on the head node which communicates with daemons and returns standard output to the console.

```
ikek70@systemg:~/mpjexpress/mpj/test
[ikek70@systemg test]$ mpjboot machines4
Starting mpjd...
wrapper  | Spawning intermediate process...
Starting mpjd...
wrapper  | Spawning intermediate process...
Starting mpjd...
wrapper  | Spawning intermediate process...
Starting mpjd...
wrapper  | Spawning intermediate process...
[ikek70@systemg test]$
```

Figure A.4: Starting MPJ Express daemons.

Listing A.5 is an example of a Hello World program written in MPJ Express. The program can be compiled with the standard `javac`. To execute the program, we have to start the MPJ Express daemons on all nodes which will participate in computation. Figure A.4 shows the daemons of MPJ Express have been started on four nodes. MPJ Express provides a configuration file which specifies machine names or IP addresses of the nodes where the programmer wants to execute MPJ Express processes.

Figure A.5 shows the execution of a parallel program written in MPJ Express. The programmer can execute the mpjrun module using mpjrun.sh or mpjrun.bat scripts. In this example, four MPJ Express processes are created on nodes `n1` through `n4`.

MPJ Express allows the programmer to shut down MPJ Express daemons on nodes using `mpjhalt`. Figure A.6 shows four daemons of MPJ Express have been stopped. `machines4` includes the machine names to be halted.

```
ikek70@systemg:~/mpjexpress/mpj/test
[ikek70@n1 test]$ mpjrun.sh -np 4 -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=8000
20:49:10.914 EVENT  Starting Jetty/4.2.23
20:49:10.921 EVENT  Started HttpContext[/]
20:49:10.923 EVENT  Started SocketListener on 0.0.0.0:15000
20:49:10.923 EVENT  Started org.mortbay.http.HttpServer@77c99159
20:49:10.923 EVENT  Starting Jetty/4.2.23
20:49:10.923 EVENT  Started HttpContext[/]
20:49:10.924 EVENT  Started SocketListener on 0.0.0.0:15001
20:49:10.924 EVENT  Started org.mortbay.http.HttpServer@6d4c1103
Listening for transport dt_socket at address: 8000
Listening for transport dt_socket at address: 8000
Listening for transport dt_socket at address: 8000
Listening for transport dt_socket at address: 8000
Starting process <0> on <n1>
Starting process <2> on <n3>
Starting process <1> on <n2>
Starting process <3> on <n4>
Java Grande Forum MPJ Benchmark Suite - Version 1.0 - Section 3 - Size B
Executing on 4 processes
iter: 1000, cnt: 1000, rank: 0
Section3:MolDyn:Run:SizeA       11.743 (s)       8.6966776E7     (Interactions/s)
Section3:MolDyn:Total:SizeA     11.768 (s)       0.084976204     (Solutions/s)
Section3:MolDynB:ElapsedTime   11.774 (s)
Stopping process <0> on <n1>
Stopping process <2> on <n3>
Stopping process <1> on <n2>
Stopping process <3> on <n4>
EXIT20:49:23.360 EVENT  Stopping Acceptor ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=15000]
20:49:23.361 EVENT  Stopped SocketListener on 0.0.0.0:15000
20:49:23.361 EVENT  Stopped HttpContext[/]
20:49:23.361 EVENT  Stopped org.mortbay.http.HttpServer@77c99159
20:49:23.361 EVENT  Stopping Acceptor ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=15001]
20:49:23.361 EVENT  Stopped SocketListener on 0.0.0.0:15001
20:49:23.361 EVENT  Stopped HttpContext[/]
20:49:23.362 EVENT  Stopped org.mortbay.http.HttpServer@6d4c1103
[ikek70@n1 test]$ exit
logout
Connection to n1 closed.
[ikek70@systemg test]$
```

Figure A.5: Executing an MPJ Express program.

```
ikek70@systemg:~/mpjexpress/mpj/test
[ikek70@systemg test]$ mpjhalt machines4
Stopping mpjd...
Stopped mpjd.
Stopping mpjd...
Stopped mpjd.
Stopping mpjd...
Stopped mpjd.
Stopping mpjd...
Stopped mpjd.
[ikek70@systemg test]$
```

Figure A.6: Stopping MPJ Express daemons.