ADDING SECURITY CONTROLS TO DYNAMICALLY OPTIMIZED MOBILE PROGRAMS

by

Anteneh Addis Anteneh

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer and Information Sciences with Distinction

Spring 2005

ADDING SECURITY CONTROLS TO DYNAMICALLY OPTIMIZED MOBILE PROGRAMS

by

Anteneh Addis Anteneh

Approved: _____

Lori Pollock, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Lisa Marvel, Ph.D.
Committee member from the Department of Computer and Information Sciences

Approved: _____

James Glancey, Ph.D.
Committee member from the Board of Senior Thesis Readers

Approved: _____

Mohsen Badiey, Ph.D.
Chair of the University Committee on Student and Faculty Honors

ACKNOWLEDGMENTS

First, I wish to thank my family and friends for all their support through the years. Thank you to everyone at HiperSpace lab for the help over the years, including Ben, Sreedevi, Emily, Dave, Anthony, Sara, and Lewis. Prof. Glancey, thank you for taking the time to read this thesis. A big thank you to Dr. Marvel for all the great help and guidance. A special thanks to Mike Jochen for being a great mentor. Also, my special thanks to Prof. Pollock for all the help, advice, and guidance she provided, and the hard work she has done for me to realize this.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Mobile programs can provide great functionality to modern computing systems. Allowing mobile programs to evolve during execution as they move through a network of hosts can improve program performance. Dynamic optimization systems can achieve this performance gain by taking into consideration the current runtime characteristics of the program to make better-informed optimization decisions. Performing this kind of program transformation on multiple computing hosts throughout a network will provide the same performance gains while reducing the overhead on the local machine. The decrease in overhead is gained by distributing different responsibilities of the dynamic optimization process to multiple hosts. The use of efficient mobile programs can greatly benefit computing systems; however, a host must be able to confirm the authenticity of a program before it can proceed to run the software.

A serious issue concerning mobile programs is that the code may be forged or altered enroute from a server to the local machine. Existing program validation and authentication methods such as digital signatures and checksums are not adequate when programs are allowed to evolve or change as they move through a network. Other validation methods involve the execution of a program, which is not desirable if the authenticity of the program is not yet confirmed. Therefore, there exists a need for new security measures that enable users to reap the benefits of dynamically evolving mobile code while mitigating the risks of the use of these mobile programs. We propose a first step in developing a security framework that will restrict how a program can change as it passes through a network of hosts. The system will

allow transformations to occur based on a defined program transformation policy. Restricting what parts of a program can change as it is being transformed will make mobile programs a safe and efficient technology.

Chapter 1

INTRODUCTION

It has been shown that applying dynamic optimization to programs can provide substantial performance gains [1, 2]. A dynamic optimization system applies transformations to programs during execution. Dynamic optimization takes into account the current runtime input, state, and environment of a program when applying optimizations. This additional information gives the dynamic optimizer an advantage over traditional static compilation/optimization methods by performing more specialized optimizations for the current run of a program. Additional performance improvements can be gained by performing dynamic optimization in a networked environment [24]. These additional performance gains are achieved by allowing computation/execution to continue on one host while parts of the transformation process occur in parallel on different hosts.

Allowing programs to transform over a network of computing hosts introduces the same security problems that are associated with executing mobile programs on local hosts. Study in the area of mobile code security continues to be an active area of research [4, 10, 7, 20]. Dynamic optimization creates a situation where the client host is faced with a decision on whether to execute a program that has possibly been transformed by another host. In some networked environments, such as some sensor networks, it may not be possible to establish complete trust in all hosts. In such network environments failure to validate the integrity of mobile programs may result in catastrophic loss or damage to system resources. For this reason, there is a

need for a method in which a host can accept or reject transformations performed on a program before executing the code. This validation system for mobile programs should be based on a well-defined transformation policy.

Traditional program validation and authentication methods such as checksums and digital signatures [19, 22, 15] do not adequately meet the security needs of dynamically evolving mobile programs. Any change to a program after the signature or checksum has been computed renders the validation data inadequate and thus these techniques become ineffective. Other non-static validation methods involve execution of the program that the local host has not validated yet. The ideal validation framework will allow for these mobile programs to evolve as they move around a network while making sure that any changes made to a program were not of a malicious nature. In networks with limited resources, this must all be done in the most efficient manner possible. Therefore, there exists a need for new security measures that enable users to reap the benefits of dynamically evolving mobile code while mitigating the risks of the use of these mobile programs.

Our research group has been exploring and classifying the kinds of dynamic transformations that can occur in a program. These classifications have been used to develop a framework to validate dynamically evolving mobile code. The validation framework will enable the system to restrict how the code can change as it goes through a network of hosts based on a well-defined transformation policy. One example of an application environment for such a system would be a network of low power remote sensor devices connected via a wireless network. Such networks are being deployed for interests as varied as military, commercial and research applications. Another example environment would contain intelligent devices connected via a Bluetooth network [5].

Our framework will utilize specification languages to communicate controls and requests for program transformations to nodes in the network. Transformation

controls will specify how a program may change in the network environment, while client nodes will use transformation requests to specify parts of the program they want to change. The framework will allow the client nodes in a network to refer to a transformation control policy before accepting any proposed changes to a program. The security of dynamically evolving mobile programs can be ensured through a system that controls dynamic program transformations.

The particular contributions of this thesis are:

- Identification and analysis of a base system to serve as an infrastructure for implementing a prototype control framework
- Design and implementation of a transformation control specification language
- Experimental study of the use of the transformation control specification language in controlling dynamic transformation of mobile programs

The proceeding chapters are organized as follows. In Chapter 2, I will describe the relevant background material for this thesis. Chapter 3 provides an overview of the transformation control framework. Chapter 4 presents an overview of the analyzed dynamic, adaptive optimization systems, and details the framework base used in the implementation. Chapter 5 discusses the transformation specification language in detail. Test scenarios and experimental results will be discussed in Chapter 6. Chapter 7 discusses related work in the area of mobile code security. Chapter 8 presents the conclusions and future work.

Chapter 2

BACKGROUND

2.1 Mobile Code

For the purposes of this thesis, the terms *(evolving) mobile code* and *(evolving) mobile program* both refer to any software that may be modified (by itself or by some other entity) as it travels through a network of computation nodes and is not compiled on the local host. These nodes may be interconnected via a wireless or wired network in an environment where complete trust is not established. The concepts of mobile programs and self-evolving code are not new [18]. Mobile programs can provide great functionality to modern computing systems. The added functionality of mobile programs include providing dynamic updates to local software, reducing bandwidth requirements by performing distributed computation remotely on large amounts of data, and enabling a software-on-demand delivery paradigm [17].

The use of mobile code will increase with the increased use of networked wireless devices. However, the use of evolving mobile programs introduces many security risks such as virus attacks, denial of service attacks, and other code tampering attacks. A serious issue concerning mobile code is that the code may be forged or altered en route from a server to the local machine. These forgeries or alterations may introduce malicious programs. Malicious programs (i.e., worms or viruses) can steal or modify data, and temporarily or permanently damage the resources of the host computer. In the next section, I will briefly describe existing validation methods and also show why they are inadequate for evolving mobile programs.

2.2 Existing Program Validation Methods

Program validation techniques such as digital signatures and checksums are used to detect changes and modifications in a program to prevent malicious attacks [19, 22, 15]. Checksums and digital signatures are extra data appended to a program. Checksums allow the receiver to validate the original state of the program by re-computing the checksum of the received message and comparing it with the data computed by the sender. If the received program was unchanged from the original version of the program, then the two checksums will be equal. Checksums are usually computed by summing the different components of a program.

Digital signatures involve the use of public and private keys. A private key is used to encrypt a unique identifier of the program that is computed by a hash algorithm. This identifier becomes the signature of the program. Public keys, which are available to most users in the concerned networked environment, are used to decrypt the signature of a program. The receiver can then re-compute the unique hash of the program using the same hash algorithm. In the same way as checksums, the received signature will be compared with the newly computed identifier to verify that no change occurred to the program.

Both of the above mentioned methods only check if there was a change in the program. It may be possible that a new checksum may be computed whenever any change occurs to a program. Checksums are not encrypted; therefore these unique identifiers may themselves be changed. Although digital signatures are encrypted, re-signing programs at every occurrence of a transformation may be computationally intensive and inefficient. In other instances where every node in a network may not have a signing/private key, generating new signatures will not be possible. A step further from digital signatures are digital certificates. Certification involves a third party entity that will certify the signatures. In addition to the problems of digital signatures mentioned above, digital certificates may also be communicationally

intensive due to the added step of confirming with the trusted third party.

Another existing program validation method is program encryption. This method is even more computationally intensive than methods mentioned above because it involves the encryption of the entire program rather than the unique program identifier. Program encryption also limits the usage/access of a program. Other existing program validation methods involve dynamic techniques, which involve the actual execution of the program. These types of methods defeat the purpose of our proposed framework as they execute the program before it has been validated that any change that might have occurred en route to the local host was not done with malicious intent.

There exist new techniques to steganographically embed authentication data within a program [12, 16, 15]. This approach takes advantage of certain properties within a program to encode a Tamper Detection Mark (TDM) without increasing program size, or altering program structure, semantics, or performance. While this novel approach is appealing as it simplifies management and distribution of authentication data and reduces the probability of success for certain kinds of attacks, this technique also fails in instances where the program evolves as it moves through a network and the TDM is not updated. This presents the same vulnerability as previously mentioned for traditional checksum/digital signature techniques.

In summary, existing validation methods either do not take into account the effect of transformations on mobile programs, or are too inefficient for environments with limited resources. Therefore, there is a need for new security measures that allow performance-improving optimizations to be made on mobile programs while protecting the program from malicious changes. This research will focus on dynamic optimization of mobile programs because of the advantage this method of transformation provides over static optimization methods as discussed in Chapter 1. Dynamic optimization is discussed in more detail in the following section.

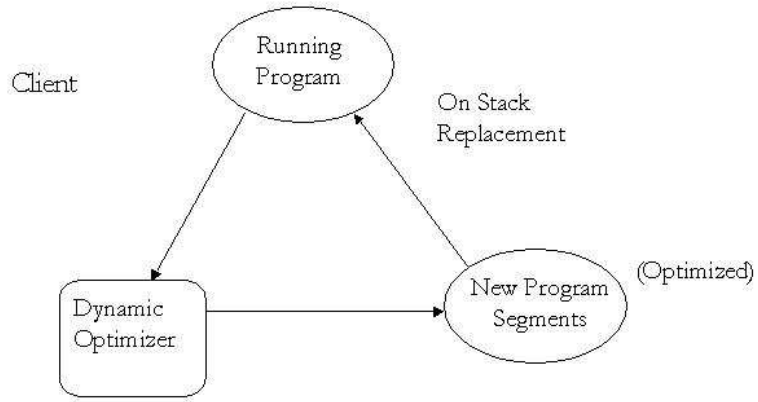

Figure 2.1: High-level view of dynamic optimization on a local machine.

2.3 Dynamic, Adaptive Optimization

While many present day examples of self-modifying software are of a malicious nature (e.g., worms and viruses), research is beginning to explore the positive benefits of this software paradigm [12, 16, 15]. The terms *dynamic* and *adaptive optimization/transformation* refer to the process of applying transformations to programs during runtime/execution. Dynamic optimization takes into account the current runtime input, state, and environment of a program when applying optimizations. Figure 2.1 presents a very high-level view of a dynamic optimization system. The dynamic optimizer makes optimization decisions based on profiling data it receives about the running program. The newly optimized segments will replace the corresponding segments of the running program through some on-stack replacement method.

An example of dynamically evolving mobile code can be seen in systems that utilize just-in-time compilers (JITs) [9, 3], dynamic translators [11], and dynamic code instrumentation [23]. A JIT, typically used for interpreted languages (e.g., Java or LISP), dynamically compiles portions of a program down to native machine instructions for faster execution (because instruction interpretation is normally slower than native instruction execution). JITs are normally designed not to

modify the semantic behavior of a program, but to improve performance on a given host during a particular instance of a program run. Dynamic translation attempts to increase software re-use by translating program instructions originally written for one architecture to a different target architecture at run time. Dynamic program instrumentation adds code to a program (thereby instrumenting the program) at execution time for the purpose of profiling program behavior. This enables targeted optimization and program testing based on program execution with real user input. The various benefits of any kind of dynamic modification of a program (e.g., self-modifying mobile code, JIT, and instrumented code) are increased flexibility and adaptability within the system (e.g., code optimized for current input sequences or code that learns from its environment and modifies its behavior).

Dynamic, adaptive optimization systems like Jikes RVM [2] and ADAPT [24] recompile portions of a program during execution while applying targeted performance improving transformations to the program. The newly optimized sections of the program are swapped with the older code once they are made available by the dynamic optimizer. Research demonstrates that the performance gains of this approach make up for the overhead of performing the required analysis and the time to complete the program transformations [2]. These gains can be achieved over static compile time optimization because the dynamic optimizer has more information about the data and state of the program than does the static compile time optimizer. Voss and Eigemann showed that additional performance improvements can be gained by performing the transformations in parallel on a separate host while execution proceeds on the Client Node [24]. My research builds on this initial idea and generalizes to a distributed dynamic optimization system running in a networked environment.

Chapter 3

A FRAMEWORK FOR ADDING SECURITY CONTROLS

3.1 Operating Network Environment

Figure 3.1 presents a high-level view of the operating network environment in which our framework is designed to operate. As mentioned above our operating networked environment may contain wired and wireless computation nodes. The network will contain:

- A *Trusted Server* that distributes the original version of the program and defines the transformation control policy
- At least one *Client Node* that is considering execution of the software and possibly requesting transformations to the program
- One or more *Benign Intermediate Nodes* (i.e., other Client Nodes) that may have hosted the software in the past or requested program transformations
- Three or more *Server Pool Nodes*, which are essentially a pool of nodes designed to assist the program transformation process – the Server Pool is treated as a single entity, not a collection of nodes
- Perhaps one or more *Malicious Nodes* that may attempt to transform the program in a nefarious manner

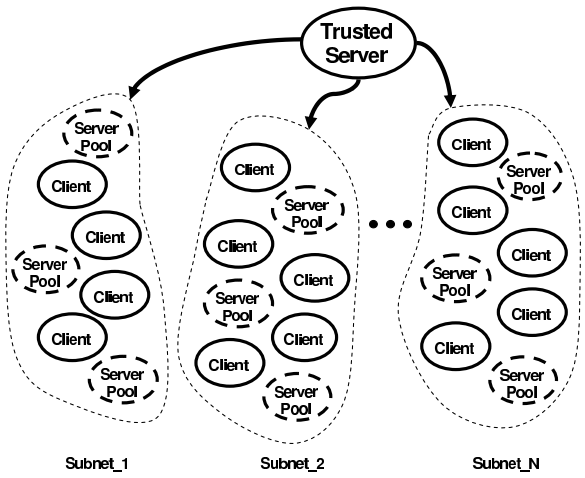

Figure 3.1: Generalization of network operating environment.

The network can be divided into two or more sub-networks. This network can utilize either a wired or wireless transmission medium. The configuration of the network can be either static (i.e., set once and the network does not change) or dynamic (nodes can enter/leave the network as time progresses). Client Nodes in the network trust all content from the Trusted Server. Client Nodes need only marginally trust Server Pool Nodes. Content from any other source in this network (i.e., other Client Nodes) is not trusted.

The network in Figure 3.1 is partitioned into N distinct subgroups. The classification for this grouping can be by geographic location, function of the client node, environmental conditions, or other criteria. Based upon the grouping classification, any Client Node from a given group represents all Client Nodes from the group. Within each subgroup of the network, a collection of Client Nodes functions as the Server Pool.

Server Pool technology has been devised to provide reliable services to networks of hosts [13]. The basic concept behind a Server Pool is to create a pool of several servers which provide the same service for a network. Through this pooled redundancy, service interruptions can be reduced.

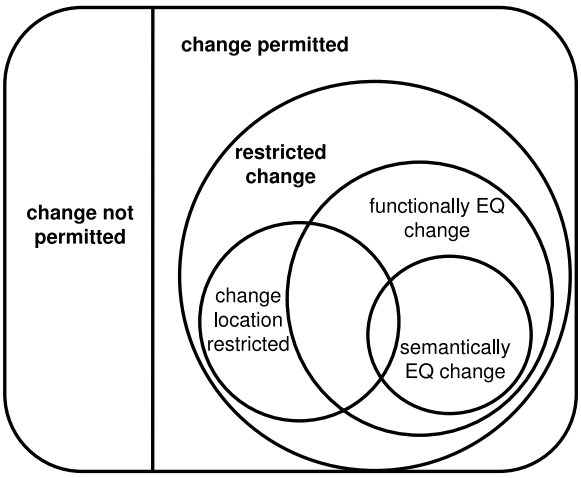

Figure 3.2: Classification of program transformations.

3.2 Program Transformation Policy

The three steps towards designing a system that validates the integrity of dynamically evolving mobile code are to (1) define the program transformation policy, (2) securely deliver this policy to all client nodes in the operating network environment, (3) enforce the transformation policy during program transformation. Before defining the program transformation policy, it is first necessary to classify the kinds of changes that can occur during program transformation.

3.2.1 Types of Transformations

Figure 3.2 presents a Venn diagram showing a high level classification of the kind of changes that can occur during program transformation, and how these classifications relate to each other. During a given program's lifetime, either no changes are allowed at all or some degree of change is performed. If transformations are allowed, they may take on different forms. Transformations may be applied to only some segments of the program (*restricted change*), or the whole program may be replaced by an entirely different program (*unrestricted change*). Restricted changes are further classified into changes limited to a certain location or segment of the

program, or changes limited by the semantics or function of the program. Two programs are functionally equivalent if they perform the same functions. Functionally equivalent programs may be coded in very different ways semantically.

The concepts of semantic and functional equivalence of programs are illustrated by the following example. Consider that a programmer solves a problem with two different programs to compare which is the best solution to the problem. Program A uses algorithm a, and Program B uses algorithm b. Each program is compiled with and without compiler optimizations to compare how different optimizations affect runtime performance. Thus, we have two final versions of each program: Program A_{opt} and Program B_{opt} are programs compiled with optimization, and Program A_{no_opt} and Program B_{no_opt} are programs compiled without optimization. All four programs are *functionally equivalent*; they all solve the same problem. Both versions of Program A are *semantically inequivalent* to both versions of Program B; however, Program A_{opt} is *semantically equivalent* to Program A_{no_opt}, and Program B_{opt} is *semantically equivalent* to Program B_{no_opt}.

3.2.2 Control Framework

The objective of this framework is to provide for a system that can control or restrict how a program can change once it is in the operating network environment. One difficult problem that arises when developing such a system is automating the process of the control specification for unspecified programs [21, 8]. Many present-day methods for detecting malicious code use pattern matching techniques to identify previously identified code patterns. These techniques are reactive in nature; they can only identify previously known and categorized malicious patterns. Good examples of software that use a reactive detection method are anti-virus software, which usually get updates for newly discovered virus patterns. The ideal validation scheme would employ a proactive technique which can recognize and restrict certain patterns based on a set of generalized and/or user-based specifications.

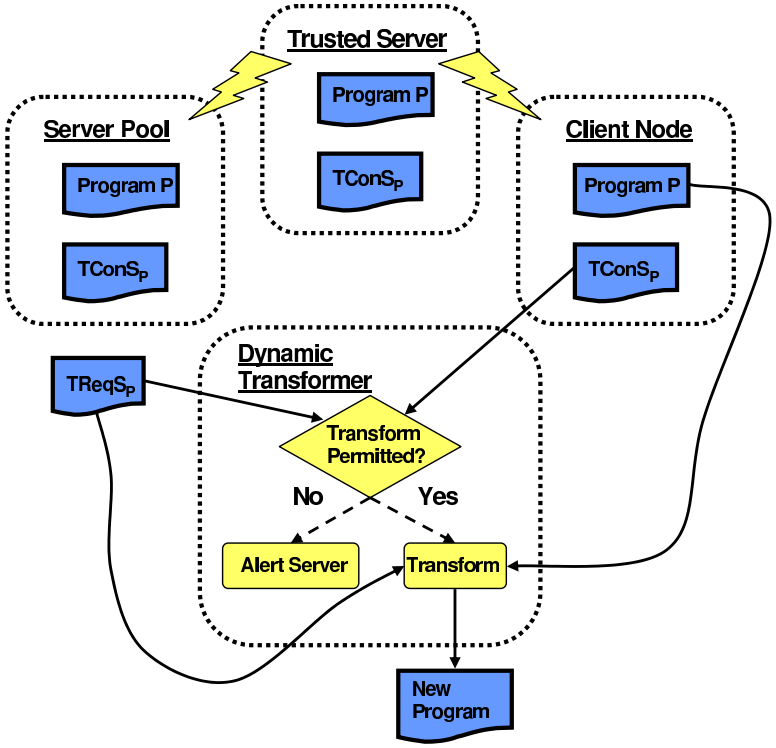

Figure 3.3: A program transformation control network.

The approach to this framework focuses on the point before a change is to be applied to a program. A general overview of the transformation control framework is presented in Figure 3.3. The system contains of a Trusted Server, the Server Pool, and Client Nodes (as described in Section 3.1). The dynamic transformer may reside on the Server Pool and the Client nodes.

An example scenario giving the details of our system follows. The Trusted Server publishes a program, P, to the network. P performs some critical function or computation that the Trusted Server seeks to protect from alteration. To protect this functionality or computation, the Trusted Server also publishes a Transformation Control Specification (TConS_P) which is associated with P. This content (P and TConS_P) is signed with the appropriate keys for future validation. Each host

in the network receives P and TConS_P, validates both objects with the appropriate keys, and proceeds with computation pending successful validation. During the execution lifetime of P, opportunities for transformation may arise. These opportunities may exist based on the nature or frequency of program input, the nature of the computation, or context changes. When these opportunities occur, the request to perform the change is encoded as a Transformation Request Specification for P, (TReqS_P). The TReqS_P is generated on the node running P and is a specification that requests specific transformations to be applied to a precise point in P. Before the transformation can be applied to P, TReqS_P must be validated at the very least by the Server Pool with TConS_P. If the validation succeeds, the transformation is applied. If the validation fails, the Trusted Server is notified and the transformation is not applied.

Depending on the security requirements of the operating environment, the above framework will follow one of two approaches in controlling program transformations. A *permissive* approach would allow all program transformations, unless they are explicitly disallowed. This approach creates a flexible approach for program transformations by only targeting specific changes. But a permissive approach also poses the most security risk as unspecified and potentially malicious optimizations are allowed to be applied. On the other hand, a *restrictive* approach would disallow all program transformations, unless they are explicitly allowed. A restrictive approach would be easier to manage and verify by specifying the list of all possible transformations that might occur. This approach ensures that any unwanted transformations will not occur, while a permissive approach will allow malicious transformations that are not explicitly restricted. Our framework is designed to handle both approaches, the default being a restrictive approach. In Chapter 5, I will go into detail about the control specification language. I will first discuss the framework infrastructure, in which the specification language is implemented, in

Chapter 4.

Chapter 4

BUILDING THE CONTROL FRAMEWORK

4.1 Identifying Potential Dynamic Transformation Systems

The first step towards designing a control framework system was to find a suitable research infrastructure that can perform most of the dynamic program analysis and actual optimization. A preferred dynamic modification system allows the programmer to take control of some aspects of optimization, profiling, and analysis methods. Two well-built dynamic code modification systems that we explored were DynamoRIO [6] and Jikes [1]. DynamoRIO, developed by Hewlett-Packard and MIT, is a system for modifying a binary program during execution. With the objective of minimizing the overhead of compilation and analysis at runtime, DynamoRIO stores optimized segments of code in a code cache. Later executions of corresponding segments use the cached code. DynamoRIO provides an API to perform program optimization and instrumentation on binary code as it runs. The DynamoRIO source code is not available for direct manipulation.

Jikes RVM, a research virtual machine developed by IBM, performs dynamic optimizations on Java bytecode programs. Jikes also provides ease in gaining control of optimizations since the source code is available for modification. We chose to use the Jikes RVM as our framework base because of the access it provides to its source code and other advantages it provides in terms of access to program analysis data. Jikes is described in further detail in the next section.

4.2 Details of the Selected Base System

4.2.1 Brief Overview

Jikes RVM is entirely implemented in Java. Before execution, all methods are compiled to native code. Jikes has two fully operational compilers: the *baseline* and *optimizing* compilers. The baseline compiler translates bytecode into native code[1]. It also performs some basic optimizations that are typically performed by static optimizing compilers. The optimizing compiler first translates bytecode into an intermediate representation (IR), to which it performs various optimizations at several levels.

One of the ways that the compilers are invoked is by the Jikes adaptive (dynamic) optimization system (AOS). The AOS invokes a compiler when it deems recompilation/optimization beneficial to the program performance based on runtime profiling data analysis. The AOS itself has three components: the *runtime measurements subsystem* which analyzes the profiling data, the *controller* which makes compilation decisions, and the *recompilation subsystem* which invokes the different compilers based on compilation decisions. The AOS also has a database that serves as a repository for all optimization decisions that it makes.

Figure 4.1 presents a simplified view of the AOS focusing on the controller. The controller, with information from the AOS database and the profiling data from the executing program, will invoke one of the compilers with an optimization plan. The optimized code segments will be returned to the AOS and then replace corresponding segments in the current program.

4.2.2 Optimizations Performed in Jikes

Jikes performs optimizations one method at a time. The optimizing compiler has three levels of optimization (0 – 2). In level 0 optimizations are performed

[1] Native code is programming code that is written to run on a specific processor.

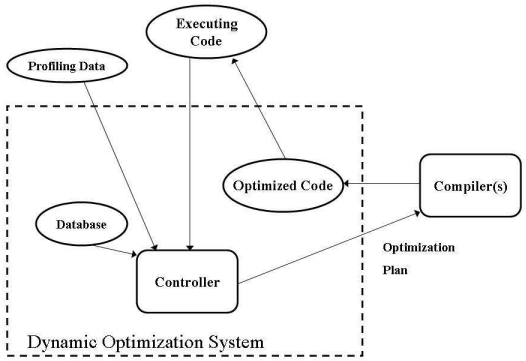

Figure 4.1: Simplified view of the Jikes Adaptive (Dynamic) Optimization System.

during IR generation. Level 1 applies additional local optimizations to level 0. Level 2 applies flow sensitive optimizations. The list of all optimizations performed in Jikes are as follows (levels are indicated in parentheses) :

- Constant propagation (0, 1)
- Copy propagation (0, 1)
- Type propagation (0)
- Non-null propagation (0)
- Constant Folding (0)
- Arithmetic simplification (0)
- Dead code elimination (0, 1)
- Elimination of redundant checks (0)
- Common subexpression elimination (1)
- Redundant load elimination (1, 2)

- Scalar replacement of aggregates (1)
- Array bounds check elimination (1, 2)

When a code segment is planned for re-optimization, it is optimized at the next level of optimization above its current level, as it has been identified as a frequently executed segment warranting more optimization.

Chapter 5

TRANSFORMATION CONTROL SPECIFICATION LANGUAGE

A vital part of the transformation control framework is the definition of a formal specification language that will be used to write specifications to communicate to different nodes how a program may change once it is deployed in the network environment. As mentioned in Chapter 3 the transformation specification language has two components: TConS (Transformation Control Specification) and TReqS (Transformation Request Specification). The grammar for the TConS language and some examples of its use are presented in this chapter. As of this writing, TReqS has not yet been designed, but we anticipate that it will have a form similar to TConS.

Figure 5.1 shows a simplified grammar for the TConS language. In this figure, productions for the language take the form of `<non_terminal>` := *production*, where *production* can be zero or more terminals or non-terminals representing other productions. The symbol "|" is used to indicate the disjunction "or" for a compound production. From Figure 5.1, we see that a TConS specification takes the general form:

```
TCONS
"class_name" {
  "method_name" {
    "transformation" { "specific_rules" }
  }
}
```

Since Jikes only applies transformations to Java bytecode programs, all our specifications will be aimed at Java bytecode files. Transformations can be specified on a class, method, or address level. The `TCONS` keyword tells the framework parser that this is a specification for a transformation control. `class_name` and `method_name` are the names of the Java class and method within that class for which we want to specify an optimization, respectively. `transformation` would be any of the transformations for which we want to write `specific_rules` to control their application on different variables or within different regions of the code, for example. The initial transformations that this system is designed to handle are:

- Constant Propagation: In this optimization, constants assigned to a variable will replace the variable at later uses of the variable, provided that the variable has not changed between the assignment and the use of the variable, and no other definitions of the used variable reach that use. The following example illustrates constant propagation.

  ```
  x := 7

  y := x + 4
  ```

  after optimization:

  ```
  x := 7

  y := 11
  ```

- Copy Propagation: In this optimization, when copy instructions of the form `x := y` occur, later occurrences of x will be substituted with y, provided that neither x or y have changed between the copy instruction and the use of x, and no other definitions of x reach the use of x.

- Method Inlining: In this optimization, a method call is replaced with the actual code of the method being called.

Figure 5.2 shows a simplified grammar for controlling the application of constant propagation. Figures 5.3 and 5.4 give an example of a simple Java class `foo` and its corresponding TConS specifications for constant propagation (`CONST_PROP`) and copy propagation (`COPY_PROP`). The statements of each method from the example program in Figure 5.3 have been labeled with contrived addresses, shown along the left side. The addresses given in this example represent the addresses of the corresponding sequence of bytecode instructions.

The TConS specification in Figure 5.4 contains transformation policies for methods `main` and `subtract` of class `foo`. The copy propagation policy for method `main` restricts the propagation of the value of `w` to the variable `y` for the address range 0 – 36, while it allows the propagation of the value of `x` to the variable `z` for the same address range. Also, for method `main`, constant propagation of variable `y` is restricted for all values in the address range 20 – 36, while variable propagation of values 1 – 199 is allowed to variable `x` in the same address range. Within method `subtract`, copy propagation is permitted for variable `a` and variable `b` (only if assigning `b` to `c` or `d`) between addresses 0 – 20.

Currently, the TConS specification is generated manually making use of knowledge of specific optimizations performed on test files by Jikes. We use JavaCC (Java Compiler Compiler) as our parser generator. JavaCC is a tool that reads a grammar specification and creates a Java program that can recognize that specific grammar. We use JavaCC in conjunction with our TConS system. The TConS system is entirely implemented in Java. TConS stores the different specified transformations into different structures and also acts as the interface for transformation checks in Jikes. TConS is implemented in such a way that there is an `isAllowed()` method for every transformation that is defined. `isAllowed()` returns `true` if the queried transformation is allowed, or returns `false` otherwise. When the permissive

transformation approach is being used, in which all unrestricted/unspecified transformations are allowed, `isAllowed()` will return `true` for all transformations that are not explicitly disallowed. An example call to `isAllowed()` for method inlining is as follows:

```
isAllowed(opt_class, opt_calling_method, TCONS_CONSTANTS.INLINE,
call_site_address, opt_method_to_inline);
```

An example generation and application of the TConS specification is as follows: A program P is deployed to the network which contains an instruction, assignment, computation, value, call, or operation that the Trusted Server wants to protect. The Trusted Server also develops the control specification TConS_P to specify restrictions on optimizations that may be performed on P. TConS_P is deployed from the Trusted Server along with P to client nodes. When Jikes is evoked on a client node to run P, it will first evoke the TConS system to parse and store the specification file. In the optimization classes of Jikes, right before an optimization is to be performed, the system will call the corresponding `isAllowed()` method with all the necessary information to determine whether that specific transformation is allowed. If the transformation is disallowed, then that transformation is not performed.

At this point, we manually write the specification(s) that restrict changes to the above types of values under one or more transformation types. One future goal for this work is to automate this process through interaction within an integrated programming environment where a programmer can select regions of program source code to apply high-level abstractions of transformation restrictions. Detailed restrictions for individual transformations will always be permitted in this system; however, we do not want to require this kind of low-level involvement from the programmer. To generate a policy that avoids restricting more optimizations than

intended, a backward slice[1] can be performed and a policy consistent with that analysis can be generated.

The grammar has been designed in a generic manner to be able to control many kinds of program transformations and to request individual or specific program transformations. For now, the system is being designed to handle only those program transformations as implemented by the Jikes RVM. As such, the transformation request at this point is the optimization level passed to the optimizer (i.e., level 0, 1, or 2 of the Jikes optimizing compiler).

[1] Program slicing is a program analysis technique that, for a given point and value in a program, examines the program and produces a listing of all the statements that either affect the computation of the value at that point (i.e., a backward slice) or that could be affected by the value from that point on (i.e., a forward slice) [14].

```
<specification> :=
  TCONS <class_name> {
    <method_rule>
  }

<method_rule> :=
  /* epsilon (empty production) */
| METHOD <method_name> {
    <transformation>
  }

<transformation> :=
  /* epsilon (empty production) */
| <transformation_name> {
    address(<addrange>,
    <transformation_rules>);
  }

<transformation_name> :=
  /* identifier token */

<transformation_rules> :=
  /* epsilon (empty production) */
| /* <transformation
     specific rule
     productions> */

<class_name> :=
  /* identifier token */

<method_name> :=
  /* identifier token */

<addrange> :=
  <address> : <address>

<address> :=
  /* address token */
```

Figure 5.1: Grammar for TConS language.

```
<const_prop> :=
   CONST_PROP { <addrange> (, <cp_constraints>)* };

<cp_constraints> :=
   <legal_vars>
|  <illegal_vars>

<legal_vars> :=
   <varlist>

<illegal_vars> :=
   ! <varlist>

<varlist> := <var_identifier> ( <var_range_list> )

<var_identifier> :=
   /* identifier token */

<var_range_list> :=
   /* epsilon (empty production) */
|  var_range (, var_range)*

<var_range> :=
   <integer> : <integer>
```

Figure 5.2: Sample grammar for controlling constant propagation.

```
    public class foo {

      public static void
 0    main(String args[]) {
 4      int w, x, y, z;
 8      w = 5;
12      x = 10;
16      y = w;
20      z = x;
24      System.out.println(
           "The answer is: " +
           subtract(w, z) );
      }

      public static int
 0    subtract(int a, int b) {
 4      int c, d;
 8      c = a;
12      d = b;
16      return c - d;
      }
    }
```

Figure 5.3: Example Java program.

```
TCONS Foo {
  METHOD main {
    COPY_PROP {
      /* At address 0 - 24, may propagate w
         unless to y, may propagate x to z */
      address(0:24, w(), !w(y), x(z) );
    }
    CONSTANT_PROP {
      /* At address 8 - 24, no propagation
         to y, may propagate to x if
         constant is between 1 - 199 */
      address(8,24, !y(), x(1:999) );
    }
  }
  METHOD subtract {
    COPY_PROP {
      /* At address 0 - 16, may propagate
         a anywhere and b to c and d */
      address(0:16, a(), b(c, d) );
    }
  }
}
```

Figure 5.4: Example TConS specification.

Chapter 6

EMPIRICAL EVALUATION STUDY

At the time of this writing, three optimizations have been defined and implemented in the TConS language for specification. These optimizations are:

- Constant Propagation
- Copy Propagation
- Method Inlining

Table 6.1: SPEC JVM 98 benchmark file details.

Benchmark	Number of Lines of Code	Action
check	1818	NA
compress	927	Compression Program
jess	6,123	Expert System
db	644	Database
mpegaudio	NA[a]	MP3 File Decoder
mtrt	NA	Ray Tracer
jack	NA[a]	Parser Generator

[a]Derived from commercial applications; source code not available

This chapter presents initial experimental results that were collected with the objective of measuring the overhead incurred by introducing the TConS system. All experiments were conducted on a 2GHz Linux machine running Intel Pentium II with a 512MB RAM and Jikes RVM v 2.3.3 installed. All presented results are the median of five execution runs. Two sets of timing data are presented, collected

under the following conditions: (1) Normal optimization performed in Jikes, no TConS specifications, (2) Normal optimization performed in Jikes, with TConS restrictions. These sets of experiments were run with seven SPEC benchmark files described in Table 6.1. For all runs, TConS is set to the permissive approach, which allows all optimizations unless explicitly disallowed.

Table 6.2: Compilation Times for runs with Uncontrolled (no TConS) vs Controlled (with TConS) Optimization.

Benchmark	Uncontrolled Run (ms)	Controlled Run (ms)	% Increase
check	435	435	0
compress	3026	3906	29.1
jess	4780	4849	2.3
db	4653	4750	3.2
mpegaudio	7342	8085	24.2
mtrt	5688	5748	2.0
jack	6144	6420	9.1

Table 6.3: Execution Times for runs with Uncontrolled and Controlled Optimization.

Benchmark	Uncontrolled (ms)	Controlled (ms)	Cntrl - Uncntrl
check	2288	2748	460
compress	21877	22842	965
jess	27195	27744	549
db	43644	44971	1327
mpegaudio	27434	28656	1222
mtrt	32170	33738	1568
jack	34259	36027	1768

Table 6.2 presents a comparison of the compilation times between uncontrolled (no TConS specifications) and controlled (with TConS specifications) runs for each of the benchmark files. Table 6.3 presents a comparison of execution times between uncontrolled and controlled Jikes runs. There was generally an increase in the compilation and execution time of the benchmarks when TConS specifications

Table 6.4: Number of Optimized Methods (Uncontrolled vs Controlled Runs).

Benchmark	Uncontrolled Run	Controlled Run	Cntrl - Uncntrl
check	2	2	0
compress	76	78	2
jess	163	172	9
db	145	143	-2
mpegaudio	207	217	10
mtrt	164	161	-3
jack	242	225	-17

Table 6.5: Controlled Run Statistics.

Benchmark	TConS Load	TConS Query	Total # of Queries	Restricted Opts
check	412.6	0	0	0
compress	406	67	159	12
jess	410	287	612	5
db	403	68	164	16
mpegaudio	435	1335	2227	58
mtrt	408	309	473	14
jack	435	560	464	8

were introduced to control optimizations. The differences in execution time are a result of the added overhead of reading in and initializing the TConS specification, the time spent querying the TConS system from Jikes, and the effect (if any) of restricted optimizations on the overall optimization decisions by the optimizing compiler. Table 6.4 presents a comparison of the number of methods that were optimized (re-optimized) by the Jikes optimizing compiler. In some cases there was an increase in the number of methods optimized/re-optimized, while there was a decrease in others. This difference may be the result of the restricted optimizations and their effect on future optimization decisions by the optimizing compiler. At this point, the TConS test files that are being used during controlled runs do not take into consideration the effect of specified restrictions on future optimization decisions. A more conclusive result can be obtained with experiments that take into

account the effect restricted optimizations have on future optimization decisions.

Table 6.5 presents detailed controlled run statistics. For each benchmark, the following information is shown in order: the time to load the TConS specification file (ms), the total time spent querying the TConS system from Jikes, the total number of queries, and the total number of restricted optimizations. Most of the TConS files for these benchmarks have an average of 10 individual restriction specifications for different variables (copy and constant propagation) and method names (method inlining). The TConS load overhead ranges from 400-500 ms, with the average TConS specification having about 20 lines and 10 restrictions.

Chapter 7

RELATED WORK

Mobile code security continues to be an active area of research. In this chapter, I discuss three most closely related works. The ADAPT system is a tool that performs de-coupled adaptive program transformations. Two tools developed by Mike Jochen, a PhD student at the Computer and Information Sciences Department at the University of Delaware, are also discussed.

7.1 ADAPT

The Automated de-coupled adaptive program transformation tool, ADAPT [24], was developed with the goal of improving the performance of dynamic optimization systems. By overlapping optimized code generation with program execution, through the use of parallel tasks, the compiler is able to perform the same tasks as highly specialized static transformers with little extra performance overhead.

7.2 Static Fingerprinting

The work described in this thesis is a derivative of the Static Fingerprinting tool. The goal of this research was to develop a unique program fingerprint to identify a mobile program. The overall purpose for this research was to be able to compare two different programs in binary form (a trusted version of the program from a trusted source, and a new program version) and determine the probability that the two programs were derived from the same source within some amount of transformation. The fingerprint tool could serve a number of purposes. A fingerprint

can be used to protect the interests of a program producer where the fingerprint can be seen as a program signature. It could also be used in programming courses to determine if any plagiarism occurred. A fingerprint could also be used to determine whether the transformations performed on the original program enroute to the local host were malicious. Therefore, it is important to achieve a constant, stable, unique fingerprint for a program regardless of the compilation process and number/type/order of optimizations.

The fingerprint is generated as follows. The binary of the original code is first transformed into an equivalent Intermediate Representation (IR). In the prototype, this IR is generated by the binary translator UQBT tool which produces an IR during its static analysis. Transformations are applied to this IR to achieve a canonical form. I was involved in this phase of the project to produce a canonical form of the IR. This canonical form is important in reversing any type of optimizations made to the program in an attempt to make it look different. The canonical form of the IR can then be analyzed to generate a program fingerprint. The original fingerprint may be stored with a trusted third party for safe keeping in cases where the desired use is copyright protection. Comparison of two fingerprints would give the degree of similarity between two programs. The result of the comparison, depending on the use of the fingerprint, can be used to decide whether to run the mobile program, determine if code theft occurred, or determine if two students have programs that are derived from the same source code.

7.3 Tamper Detection Marking

A Tamper Detection Marking [15] system was developed with the objective of enabling mobile code validation with authentication data that is derived from the code itself. The authentication data will not be separated from the code itself and no additional bandwidth will be required for data transmission. The authentication data is the Tamper Detection Mark (TDM), a cryptographic checksum. A code with

TDM is semantically equivalent to the original code. This allows users at local hosts to execute TDM-carrying code without any pre-processing when no authentication is desired. This system may be utilized to detect virtually any degree of tampering or change to an object file.

The TDM system works in two phases: an *Embed* and a *Validate Phase*. In the *Embed Phase*, a TDM is created by computing a hash value of the object file. The hash value is then encrypted with a secret key. In the *Validation Phase*, a new TDM is computed locally. The TDM in the file is extracted and decrypted before it is compared with the newly computed local TDM. If there was no alteration to the mobile program, the two TDMs will be the same. This kind of system cannot be used on evolving mobile code, as it is fragile and cannot distinguish safe changes from malicious changes.

Chapter 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

This thesis describes a framework for validating dynamically transforming software. The framework leverages the benefits of dynamic optimization with the ability to control how a program is transformed. This system can be of great benefit to dynamic transformers by providing a proactive approach to protect certain parts of the program from unwanted change. The specification language is robust in that any type of change (not just compiler optimizations) can be controlled by the system. As of this writing, the Transformation Control Specification (TConS) language has been implemented for three optimizations as well as other more general transformations. The three implemented optimizations are constant propagation, copy propagation, and method inlining. The Transformation Request Specification (TReqS) has not yet been implemented. The TReqS, when implemented, will allow nodes to request specific transformations. Restricting the parts of a program that can change as it is being transformed will make mobile programs a safe and efficient technology.

The specific contributions of this thesis are: (1) identification and analysis of a base system to serve as an infrastructure for implementing a prototype control framework, (2) design and implementation of a transformation control specification language, and (3) experimental study of the use of the transformation control specification language in controlling dynamic transformation of mobile programs.

8.2 Future Work

Future tasks to be completed in this project include the following:

- Extend TConS to handle a larger set of optimizations
- Perform a more extensive experimental study to determine the effect of transformation restrictions on future optimizations in the same run of the program
- Implement the TReqS language to communicate transformation requests to optimizing hosts in the network (At this point, Jikes optimization decisions are used as transformation requests.)
- Perform an experimental study to explore the efficiency and expressiveness of the specification language
  - Effectiveness
    - Does the control framework work properly?
    - Does it scale well?
  - Cost: overhead
    - time
    - space
    - communication
  - Flexibility/Adaptability
    - Study different network configurations
    - Which configurations provide the best performance for a given network environment?

Another future goal for this project is to explore ways to ease the generation of TConS and TReqS specifications. At this point, specification files are manually

defined. This task may involve defining a more general transformation policy for the whole or parts of the operating network from which specific TConS specifications may be developed for individual mobile programs. One way to achieve this goal might be through interaction within an integrated programming environment where a programmer can select regions of program source code to apply high-level abstractions of transformation restrictions.

BIBLIOGRAPHY

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 47–65. ACM Press, 2000.

[2] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2002.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2000.

[4] E. Bierman and E. Cloete. Classification of malicious host threats in mobile agent computing. In *Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology*, 2002.

[5] Bluetooth SIG. Specification of the Bluetooth System. http://www.bluetooth.org, 2003.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *"International Symposium on Code Generation and Optimization"*, 2003.

[7] D. M. Chess. Security issues in mobile code systems. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[8] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 11th USENIX Security Symposium*, pages 169–186, Aug. 2003.

[9] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, volume 35.5 of *ACM Sigplan Notices*, pages 13–26, June 2000.

[10] P. T. Devanbu and S. Stubblebine. Software engineering for security: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*. ACM Press, 2000.

[11] K. Ebcioğlu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37. ACM SIGARCH and IEEE Computer Society TCCA, June 2–4, 1997.

[12] R. El-Khalil and A. D. Keromytis. Hydan: Hiding information in program binaries. In *Proceedings of the 6th International Conference on Information and Communications Security*. ICISA, Springer-Verlag, Oct. 2004.

[13] M. A. Fecko, U. C. Kozat, S. Samtani, M. Ümit Uyar, and I. Höklek. Reliable and dynamic access to service in battlefield ad hoc networks. In *Proceedings – IEEE Military Communications Conference MILCOM*. IEEE, Nov. 2004.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[15] M. Jochen, L. Marvel, and L. L. Pollock. A framework for tamper detection marking of mobile applications. In *Proceedings of the Fourteenth International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Nov. 2003.

[16] M. Jochen, L. Marvel, and L. L. Pollock. Tamper detection marking for object files. In *Proceedings – IEEE Military Communications Conference MILCOM*. IEEE, Oct. 2003.

[17] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting self-modification mechanism for program protection. In *Proceedings of the 27th Annual International Computer Software and Applications Conference*. IEEE, 2003.

[18] N. M. Karnik and A. R. Tripathi. Security in the Ajanta mobile agent system. *Software–Practice and Experience*, 31(4):301–329, Apr. 2001.

[19] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, 1997.

[20] G. McGraw and G. Morrisett. Attacking malicious code: A report to the Infosec Research Council. *IEEE Software*, 17(5):33–41, Sept./Oct. 2000.

[21] M. G. Pleszkoch and R. C. Linger. Improving network system security with function extraction technology for automated calculation of program behavior. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences*. IEEE, 2004.

[22] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, Feb. 1978.

[23] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 86–96. ACM Press, 2002.

[24] M. J. Voss and R. Eigemann. High-level adaptive program optimization with ADAPT. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 93–102. ACM Press, 2001.