Fall 2013

Ant: A Framework for Increasing the Efficiency of Sequential Debugging Techniques with Parallel Programs

Jae-Woo Lee
Purdue University

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Jae-Woo Lee

Entitled
Ant: A Framework for Increasing the Efficiency of Sequential Debugging Techniques with Parallel Programs

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

SAMUEL P. MIDKIFF
Chair

MITHUNA S. THOTTETHODI

RUDOLF EIGENMANN

VIJAY S. PAI

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): SAMUEL P. MIDKIFF

Approved by: M. R. Melloch 12-02-2013
Head of the Graduate Program Date

ANT: A FRAMEWORK FOR INCREASING THE EFFICIENCY OF SEQUENTIAL DEBUGGING TECHNIQUES WITH PARALLEL PROGRAMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Jae-Woo Lee

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2013

Purdue University

West Lafayette, Indiana

For my wife, Sookyung and my children, Seunghyun and Yejee

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Lee, Jae-Woo Ph.D., Purdue University, December 2013. Ant: A Framework for Increasing the Efficiency of Sequential Debugging Techniques with Parallel Programs. Major Professor: Samuel P. Midkiff.

Bugs in sequential programs cost the software industry billions of dollars in lost productivity each year. Even if simple parallel programming models are created, they will not reduce the level of sequential bugs in programs below that of sequential programs. It can be argued that the complexity of current parallel programming models may increase the number of sequential bugs in parallel programs because they distract the programmer from the core logic of the program.

Tools exist that identify statements related to sequential bugs and allow those bugs to be more quickly located and fixed. Their use in parallel programs will continue to be useful. Many of these debugging tools require runtime monitoring of program points of interest in a program and the overhead of this monitoring is usually very high.

We propose Ant, a framework that increases the efficiency of sequential debugging techniques when used with parallel programs. The Ant framework takes two different strategies depending on whether the program to be debugged is a distributed memory program or shared memory program. For MPI programs, the Ant compiler analyzes the program and identifies two different types of code regions: those that all processes execute and regions that only part of the processes execute. For shared memory Pthreads programs, Ant uses a combination of static and dynamic analyses to determine similar parts of the program executing in parallel and the number of threads executing those parts of the program. The programs are instrumented with calls to Ant runtime libraries and debugging libraries based on the Ant compiler's static analysis results. Relative to a naive port of a debugging tool (C-DIDUCE, in our cases), Ant's technique, by exploiting the application's parallelism, reduces the monitoring overhead by up to 15.85 times (and on average 9.23 times) for MPI programs executing with 32 processes and up to 18.14 times (and on average 8.73 times) for Pthreads programs executing with 8 threads, while maintaining high accuracy.

1. INTRODUCTION

1.1 Motivation

Writing correct sequential programs is a difficult task – bugs in these programs cost the software industry billions of dollars in lost productivity each year [1]. Using more complicated parallel programming models will not reduce the number of sequential bugs, and may increase their number by adding to the overall complexity of programming. Tools exist that identify statements that may be related to sequential bugs and that allow the bugs to be identified quickly and fixed. Because sequential bugs will continue to exist in parallel programs, these tools will continue to be useful in parallel programming environments.

Many of these debugging tools require runtime monitoring of program points of interest in a program. An important class of these tools detects *invariant violations*, and includes tools such as DIDUCE [2], C-DIDUCE [3] and AccMon [4][1] that, in sequential programs, have runtime overheads of up to 20X, 1.21X and 3X (with specialized hardware support), respectively, even when a whole program is not monitored. A second class of debugging tools (e.g., [5–7]) looks for statistical variations in program behaviors between correct and incorrect runs, and can also have high runtime overheads. These overheads result from needing to monitor fine grained program actions at runtime. A naive port of these tools to parallel programs will have high overheads while executing on expensive parallel hardware.

[1]AccMon uses special hardware

1.2 Ant Framework

We propose *Ant*, a framework that increases the efficiency of sequential debugging techniques when used with parallel programs. The Ant framework allows sequential debugging tools that do not require all events of interest to be monitored to efficiently and accurately target parallel programs. We show the effectiveness of the Ant framework using a case study involving C-DIDUCE [3], an implementation of DIDUCE [2] that targets C instead of Java programs.

The Ant framework picks between two different strategies depending on whether the program to be debugged is a distributed memory program or a shared memory program. We call the technique for distributed memory parallel programs *AntDM*, and the technique for shared memory parallel programs *AntSM*.

1.2.1 AntDM: Exploiting Parallelism of Distributed Memory Programs

AntDM allows the sequential tools mentioned above to efficiently and accurately target distributed memory parallel programs (MPI programs in our study). AntDM does this by solving two important problems. First, AntDM allows the results gathered on many processes to be merged in a theoretically sound way that gives useful results. Second, AntDM uses the inherent parallelism of the program being monitored to reduce the overhead of the debugging tool, while maintaining a high level of accuracy.

Statistical and invariance based debugging tools such as DIDUCE and C-DIDUCE assert a hypothesis that serves as the foundation of the tool. DIDUCE and C-DIDUCE assert the *value invariant hypothesis*, which states that a given variable takes on a small set of values during its lifetime, even with different input data, and rarely occurring deviations from this set of values indicate buggy or anomalous behavior. Detecting where these deviations occur aids in debugging. The literature on these and similar techniques (e.g., [2–4, 7, 8]) empirically validate the utility of the asserted hypotheses in sequential programs.

In our case study, AntDM asserts a parallel version of the value invariant hypothesis. AntDM asserts that a value invariant holds across different input datasets, across similar processes executing the program, and across executions involving different numbers of processes. In Section 3.3.1, we present, from [9], the result that the merging of the monitoring data gathered over many processes will yield the same result as if the data was gathered in a single sequential execution.

Empirical evidence gathered from our case study targeting C with MPI programs and C-DIDUCE shows validity of the value invariant detection and the practicality of exploiting the parallel value invariant hypothesis with AntDM. We use four MPI parallel benchmarks that have had bugs injected into them. Each process performs *replicated* monitoring, that is, each process performs the monitoring required by C-DIDUCE as if it were an independent program, with the results of the individual processes' monitoring collected and merged. This monitoring provides effective detection of the injected bugs, as explained in Section 4.4.

Using the replicated monitoring described above, C-DIDUCE and DIDUCE suffer high overheads in parallel programs just as they do in sequential programs. One way to reduce these overheads is to have each of the P processes executing the program monitor $\frac{1}{P}$ of the events. This performs a sampled monitoring by distributing the monitoring evenly across the P processes. We call this type of monitoring *distributed* monitoring. As we show in Section 4.4, distributed monitoring significantly reduces the monitoring overhead, but suffers from reduced accuracy in detecting anomalous events of interest. The inaccuracy results from each process only sampling $\frac{1}{P}$ events, even in program regions that are not executed by all P processes.

AntDM takes a more intelligent approach that achieves low overhead similar to that of distributed monitoring, and accuracy similar to that of replicated monitoring. It does this by using a static, compile time analysis to divide the program into regions that are executed by all processes (*All-process Region*s or *AR*s) and regions that are not executed by all processes (*Not-All-process Region*s or *NAR*s). In ARs, AntDM acts like distributed monitoring and each process monitors $\frac{1}{P}$ of the accesses. In

NARs, all processes monitor all accesses, as with replicated monitoring. We present experimental results showing that AntDM's strategy achieves the best of both replicated and distributed monitoring: it has nearly the overhead reduction of distributed monitoring with accuracy that is close to replicated monitoring.

1.2.2 AntSM: Exploiting Parallelism of Shared Memory Programs

While a naive port of sequential debugging tools to a parallel, shared memory platform is possible, doing so is inefficient. The tools often rely on having a single data item monitored for each program point of interest (e.g., every reference of a non-floating point variable). The key insight of this thesis is that different instances of the same code executing in parallel in different threads are likely to behave similarly, and that sampled monitoring over that code can reduce overheads with only a small impact on accuracy.

The Ant Shared Memory (or AntSM) system exploits this key observation to reduce the overhead of debugging tools when used with shared memory parallel programs. AntSM uses the parallelism of the multi-threaded shared memory program being monitored to reduce the overhead of the debugging tool, while maintaining a high level of accuracy. It does this by first instrumenting the program with calls to the AntSM runtime library to collect and maintain information about parallelism in the program. The program is then instrumented with monitoring and other calls for the bug detection technique being used. At runtime, the parallel structure of the program and the number of threads executing some region of the program are used to perform an intelligently sampled monitoring.

We measure the effectiveness of AntSM with a case study using multi-threaded, parallel Pthreads programs from the PARSEC benchmark suite [10] with injected bugs like those in the Siemens Benchmark Suite [11]. Our debugging tool is C-DIDUCE [3], an implementation of DIDUCE [2] targeting C instead of Java. AntSM reduces the running time of the monitored program by up to 18.14 times (and on average 8.73

times) on an eight-core machine relative to a naive port that performs no sampling, with an accuracy that is close to monitoring all accesses.

1.3 Contribution

Ant framework with AntDM and AntSM technique presents the following technical contributions:

- A debugging framework that allows sequential debugging tools to be used with parallel programs;
- A monitoring technique that uses an intelligent sampling strategy to exploit the parallelism within an application;
- The uses of ARs and NARs to guide MPI program instrumentation for debugging tools, and data showing that this leads to accurate monitoring with a low overhead;
- Experimental results showing the validity of the parallel value invariant hypothesis with AntDM and the effectiveness of C-DIDUCE on distributed memory parallel programs;
- The uses of the fork site analysis with Pthreads programs to enable sampled monitoring to reduce program debugging overheads;
- A case study showing the effectiveness of Ant framework with the C-DIDUCE [3] value invariant tool with shared memory parallel programs;
- Experimental results showing the usefulness of our sampling results and overhead reduction strategies used by AntSM when monitoring shared memory parallel programs.

1.4 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 presents the background and related work. Chapter 3 discusses AntDM, the technique used by the Ant framework targeting distributed memory parallel programs. Chapter 4 describes AntSM, the technique used by the Ant framework targeting shared memory parallel programs. Chapter 5 gives our conclusions.

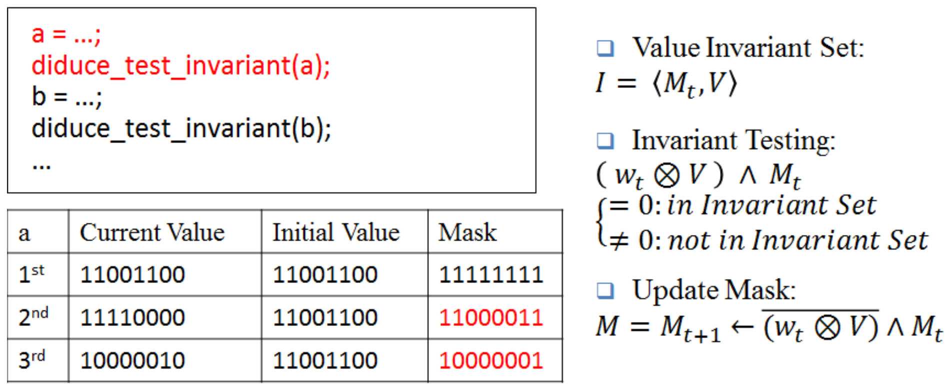
2. BACKGROUND AND RELATED WORK

2.1 Overview of Value Invariant Detection

This thesis uses C-DIDUCE [3], a C implementation of DIDUCE. The differences between DIDUCE and C-DIDUCE result from DIDUCE targeting Java and C-DIDUCE targeting C. Details of these differences can be found in [3]. Both DIDUCE and C-DIDUCE first perform a training run to determine an approximation to the set of all values seen by each reference in the program. DIDUCE associates each reference of a variable with an invariant $I = \langle M_t, V \rangle$, where V is the variables' initial value, and M_t is the value of an *invariant mask* after the t-th access. V is initialized to the variable value that is seen when the reference is first executed, and M is initialized to be all 1's. Let w_t be the t-th value of V observed at the program point.

As each value w_t is observed, the test $(w_t \otimes V) \wedge M_t \neq 0$ is performed, where $\otimes$ is the bitwise XOR operation. If the test is true, the invariant is relaxed by updating the mask so that $M = M_{t+1} \leftarrow M_t \wedge \overline{(w_t \otimes V)}$. Intuitively, each update of the mask results in the mask having a value of '0' in bit positions where both a '0' and a '1' have been previously seen. A mask position containing a '1' indicates that all previous values only had a '1' in that position, or that all previous values only had a '0' in that position. Whether only a '0' or '1' value was seen is determined by inspecting the corresponding bit of V. Thus the test determines if the value w_t differs in one or more bits from all previously seen values, and if it does, the mask is relaxed to indicate this.

For example, as shown in Figure 2.1, if the variable, "`a`" in the program snippet, has the values in the "Current Value" column, the invariant test is performed each time and the mask information is updated accordingly. This example shows only 8 bits but the actual implementation keeps 32 bits for each value and mask. After



```
a = …;
diduce_test_invariant(a);
b = …;
diduce_test_invariant(b);
…
```

a	Current Value	Initial Value	Mask
1st	11001100	11001100	11111111
2nd	11110000	11001100	11000011
3rd	10000010	11001100	10000001

❑ Value Invariant Set:
$I = \langle M_t, V \rangle$

❑ Invariant Testing:
$(w_t \otimes V) \wedge M_t \begin{cases} = 0: \mathit{in\ Invariant\ Set} \\ \neq 0: \mathit{not\ in\ Invariant\ Set} \end{cases}$

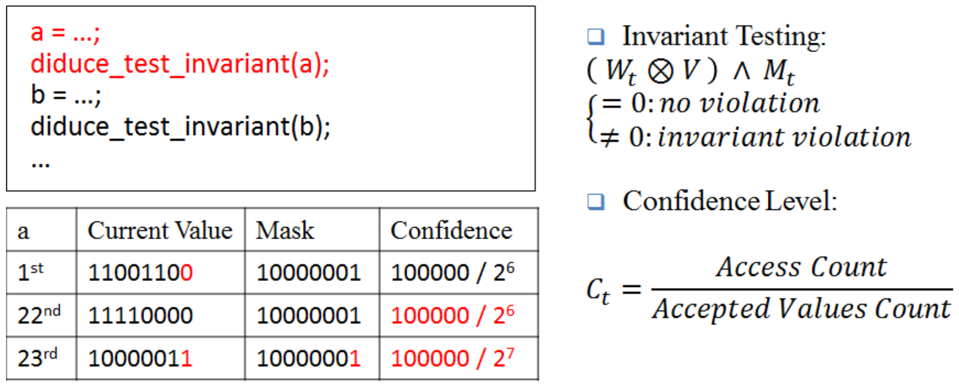
❑ Update Mask:
$M = M_{t+1} \leftarrow \overline{(w_t \otimes V)} \wedge M_t$

Fig. 2.1. Example of C-DIDUCE in training mode.

three accesses of the variable, `a`, the resulting mask shows the first and the last bit is invariant and this mask is used for detecting the invariant violation in checking mode.

In a production run with a different input, values that are not in the (approximate) set of seen values are detected by applying the test above. However, not all invariant violations are treated equally. In particular, violations with values that are seen many times are treated as being less important than violations with values that occur only a few times. The intuition behind this is that values that are seen many times are more likely to be values that should have been in the invariant set. At the end of the run, the different violations are ranked, and a listing of violations, in rank order, is produced. As with other debugging and anomaly detection tools, the assumption is that lower ranked violations are less likely to correlate to a bug, and that a programmer debugging a program will examine the highly ranked violations, fix any indicated errors, and then either re-execute the program, or re-train and re-execute the program.

Figure 2.2 shows an example of information used in checking mode. When the value of variable `a` is “10000011”, it violates the invariant test. This is because that the initial value is “11001100” and the mask is “10000001” so the result of invariant



```
a = ...;
diduce_test_invariant(a);
b = ...;
diduce_test_invariant(b);
...
```

a	Current Value	Mask	Confidence
1st	11001100	10000001	100000 / 2^6
22nd	11110000	10000001	100000 / 2^6
23rd	10000011	10000001	100000 / 2^7

❑ Invariant Testing:

$$(W_t \otimes V) \wedge M_t \begin{cases} = 0: \textit{no violation} \\ \neq 0: \textit{invariant violation} \end{cases}$$

❑ Confidence Level:

$$C_t = \frac{\textit{Access Count}}{\textit{Accepted Values Count}}$$

Fig. 2.2. Example of C-DIDUCE in checking mode. In the confidence level computation, `Access Count` contains the number of accesses to the reference of a variable and `Accepted Values Count` contains the number of accepted values to the reference, i.e., 2 to the power of the number of bits which are marked as "not invariant" in the mask of an invariant set.

testing is "00000001", i.e., not zero. The newly computed mask will be "1000000", which has 7 bits marked as "not invariant" so the confidence level is computed based on this. The new and old confidence levels are compared and if the confidence level drop (the difference between the confidence level for the invariant set in the table and for the current value) is higher than the previous drop, the new value is recorded in the invariant violation table.

2.2 Related Work

2.2.1 Debugging Sequential Programs

There has been previous work focusing on the development of tools to aid the debugging of sequential programs, and we have mentioned some of them in Chapter 1. Ernst, et al. [8, 12] introduce DAIKON, a system that detects program invariants at runtime. The DAIKON infers invariants, such as the set of constant values in a variable and range limits, at specific program points such as procedure entries, exits, and loop heads. At runtime, the instrumented program provides DAIKON with the values of variables in scope and DAIKON detects the violation of the invariants. Hangal, et al. [2] propose DIDUCE, a debugging technique of value invariant violation detection and this was discussed in the previous section. Zhou, et al. [4] discuss a program counter(PC) based invariant detection tool called AccMon. Their work asserts that in most programs, a given memory location is typically accessed by only a small set of instructions and by extracting the invariant of the set of PCs accessing a given variable, it can detect accesses by outlier instructions, which may be related to a memory related bugs such as memory corruption, buffer overflow, stack smashing, etc. Other tools [5–7] describe debugging techniques using statistical variations in program behaviors between correct and incorrect runs.

Fei, et al. [3] provide a debugging framework, called Artemis, to reduce the overhead of debugging tools. Their work defines the dynamic context of a program region to be the program state accessed in that region, and approximates a context at the entrance of a procedure by approximating a variable's value by an integer value and a memory object being pointed to by the pointer's type. Artemis collects the context invariant information during the correct runs of a target program and reduces the overhead of debugging tools in production run by avoiding remonitoring the same code region under the same context.

These tools are complementary to our work in that Ant framework is applicable to these tools. Unlike our work, these tools target sequential programs.

2.2.2 Debugging Parallel Programs

There are several previous works on debugging parallel programs. TotalView [13], Mantis [14], and Prism [15] support typical debugging methods such as adding breakpoints at the program points and specifying the processes or threads of interest. These tools support a GUI to make it possible to debug the target programs interactively by browsing the source code at runtime.

Stringhini et al. [16] introduce PADI, a debugging tool that offers a mechanism to select the processes to be debugged. PADI's group selection mechanism allows users to select pre-defined groups of processes as well as to define their own group and this mechanism helps reducing the amount of processes to be visualized and controlled. Cheng et al. [17] discuss a parallel and distributed program debugger that focuses on portability by incorporating a client-server model. To support portability, their work provides a protocol that specifies the interaction between a message-passing library and the debugger. Wismuller et al. [18] introduce a parallel debugger called DETOP that applies the event-action paradigm to avoid unnecessary user interaction. Their work also describes a performance analyzer called PATOP that measures the system utilization to detect the performance loss caused by idle processor states.

Ant differs from these tools in that Ant exploits the parallelism of the application to reduce the overhead of sequential debugging tools when used with parallel programs.

2.2.3 Process Clustering

Another research area looks for outliers in the behavior of processes in a cluster. These often use statistical techniques to find clusters of similarly behaving processes based on metrics such as communication patterns, volumes, stack traces, and so forth, and then look for outliers in terms of control flow behavior, or the previously mentioned metrics among the processes in a cluster. Mirgorodskiy et al. [19] describe an approach for locating the causes of anomalies in distributed systems by collecting function-level traces from each process, comparing them to each other if the application fails, and identifying a function that is likely to explain the anomalous behavior. Gao et al. [20] introduce a tool called DMTracker that extracts data movement (DM)-based invariants at program runtime and checks the violations of these invariants. Their work asserts that these violations of DM-based invariants indicate potential bugs such as data races and memory corruption bugs. Arnold et al. [21,22] discuss a tool called STAT to aid in debugging large-scale applications. The STAT collects stack traces over a sampling period to form process equivalence classes exhibiting similar behavior and the collected information is used for the root cause analysis of problems such as deadlocks and performance bottlenecks. One problem of the techniques described in these works is that they are slow, likely too slow to use at runtime [22].

Our Ant framework is orthogonal to these approaches. It does *not* use statistical information to form clusters or find outliers within a cluster. The AntDM uses statically determined partitioning of regions to drive instrumentation for sequential bug-finding tools to improve the performance of the tools. The AntSM does a simple function of clustering by checking the entering/exiting of the root functions and maintaining the number of threads that execute the root reachable code in the same thread group at runtime.

3. ANTDM: ANT FRAMEWORK TARGETING DISTRIBUTED MEMORY PARALLEL PROGRAMS

3.1 The AntDM Framework

We call our technique that targets MPI programs, *AntDM* [9]. AntDM allows sequential tools to efficiently and accurately target distributed memory parallel programs. AntDM does this by solving two important problems. First, AntDM allows the results gathered on many processes to be merged in a theoretically sound way that gives useful results. Second, AntDM uses the inherent parallelism of the program being monitored to reduce the overhead of the debugging tool, while maintaining a high level of accuracy. The target MPI program is statically analyzed and marked as two different regions, one where all the processes executes the source code (*All-process Regions* or ARs) and the other where only subset of processes execute the source code (*Not-All-process Regions* or NARs). AntDM applies distributed monitoring over ARs of the target program among all the processes and replicated monitoring over NARs of the target program.

The AntDM framework, shown in Figure 3.1, has two main components: (1) a static analysis component, whose input is a C/MPI program that identifies and instruments All-process Regions(ARs) and Not-All-process Regions(NARs), and (2) a debugging runtime. The compiler analysis and instrumentation is discussed in Section 3.2, and the use of an invariant violation detection and monitoring (runtime) technique is discussed in Section 3.3, when we discuss a case study using the C-DIDUCE value invariance debugging tool.

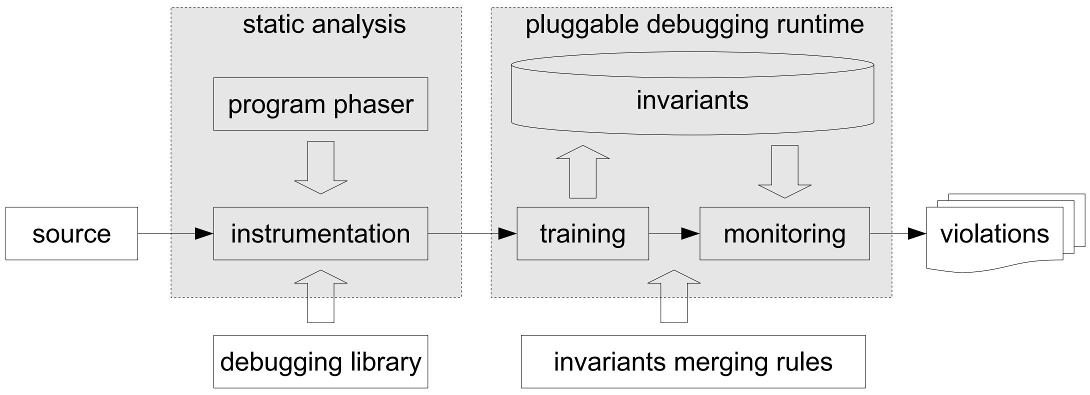


Fig. 3.1. Overview of the AntDM Framework.

3.2 Compile Time Analysis and Instrumentation

In this section, we describe AntDM's compile time analysis and instrumentation strategies.

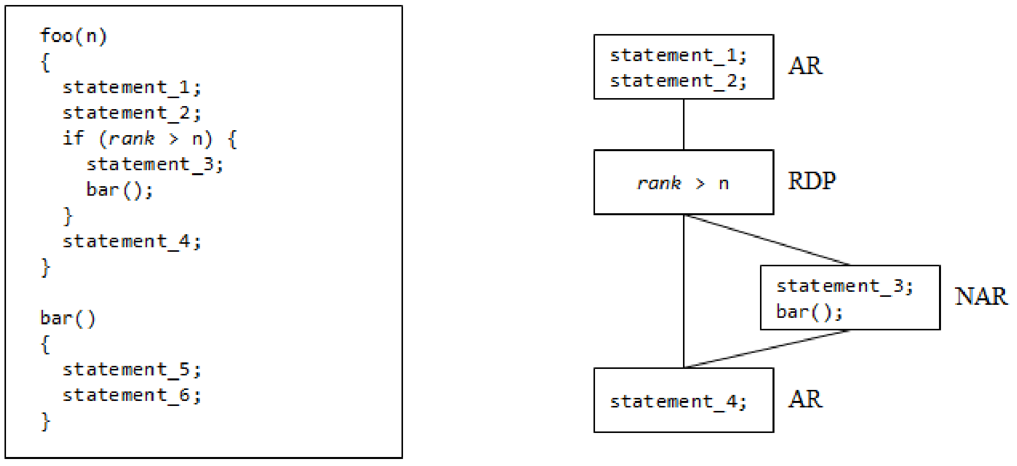
3.2.1 Region Demarcation Points (RDPs), ARs and NARs

We consider code to be in a NAR when it is control dependent on a branch whose conditional is a function of the process *rank* (i.e., process id). We call these conditional branches *region demarcation points* (RDPs). AntDM's static analysis detects ARs and NARs by identifying RDPs, and this is done by following DEF-USE chains from the `MPI_Comm_rank` function calls. We are not interested in the value of the control expression or its variables, only that it is dependent on an MPI rank, and therefore that some processes may follow the true path from the conditional branch and others may not. All statements that are control dependent on the RDP are members of a NAR. We note that our analysis may be conservative – i.e., we may identify regions as NARs that are actually ARs, but the effect of this is to increase the monitoring overhead and (possibly) the accuracy.

Algorithm 1 AntDM Static Analysis for marking NARs and ARs

```
1: $pid\_handles$ ← gather_process_id_info($control\_flow\_graphs$, $def\_use\_chains$)
2: $RDP$s ← gather_RDP_info($pid\_handles$, $control\_flow\_graphs$)
3: **for all** $statement$ in $program$ **do**
4:     $pp$ ← get_control_dependent_point($statement$)
5:     **if** $pp$ in $RDP$s **then**
6:         mark $statement$ as $NAR$
7:     **else**
8:         mark $statement$ as $AR$
9:     **end if**
10: **end for**
11: **while** $change$ in $NAR$s **do**
12:     **for all** $callsite$ in $NAR$s **do**
13:         $procedure$ ← get_procedure($callsite$)
14:         mark all $statements$ in $procedure$ as $NAR$
15:     **end for**
16: **end while**
```

Algorithm 1 describes the AntDM static analysis for marking NARs and ARs of the input parallel program. First, RDPs are determined using the process id handles (i.e., the variables that contain the process id itself or the resulting value from the function of the process id) by traversing the control flow graph and following def-use chains (lines 1 and 2). The `gather_RDP_info` function finds all the expressions containing any in *pid_handles* set. Next, all statements in the program are checked to see if they are control dependent on an *RDP* and marked as either a *NAR* or an *AR* (lines 3 to 10). Finally, the callee procedures from NARs are iteratively marked as NARs (lines 11 to 16). The resulting program is ready for instrumentation as described in the following section.



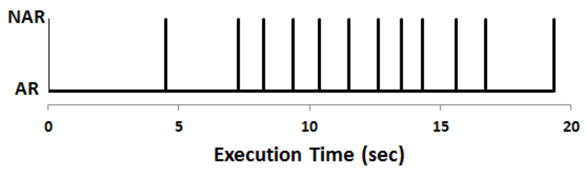
```
foo(n)
{
  statement_1;
  statement_2;
  if (rank > n) {
    statement_3;
    bar();
  }
  statement_4;
}

bar()
{
  statement_5;
  statement_6;
}
```

(a) Target Program

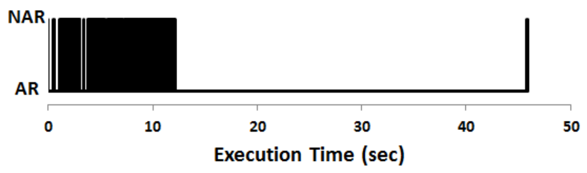
(b) Control Flow Graph

Fig. 3.2. Example of AntDM static analysis for marking ARs and NARs based on RDPs.

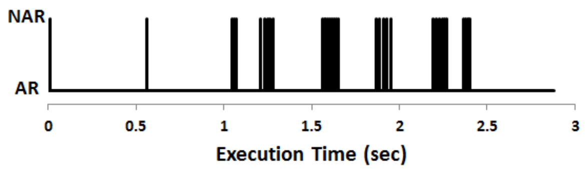
Figure 3.2 shows an example of the AntDM static analysis and its resulting codes being marked as ARs or NARs. From the `gather_process_id_info` function, the variable *rank* is added to *pid_handles* set. As shown Figure 3.2(b), the conditional expression, "`rank > n`", is marked as RDP since it contains the process id variable, *rank*, in the expression and this changes the execution path of the process depending on the value of the process id variable. Next, the program is traversed over the control flow graph and marked as either AR or NAR depending on the control dependence on the RDPs. In the `foo` function of Figure 3.2(a), `statement_1`, `statement_2`, and `statement_4` are marked as ARs and `statement_3` and the function call, `bar()`, are marked as NARs. Finally, all the statements (`statement_5` and `statement_6`) in the callee function, `bar`, in Figure 3.2(a), are marked as NARs.



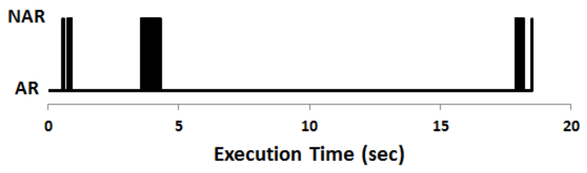
(a) NPB IS



(b) ASCI SMG2000



(c) SPEC MPI2007 TACHYON



(d) SPEC MPI2007 MILC

Fig. 3.3. Time spent in ARs and NARs during program execution.

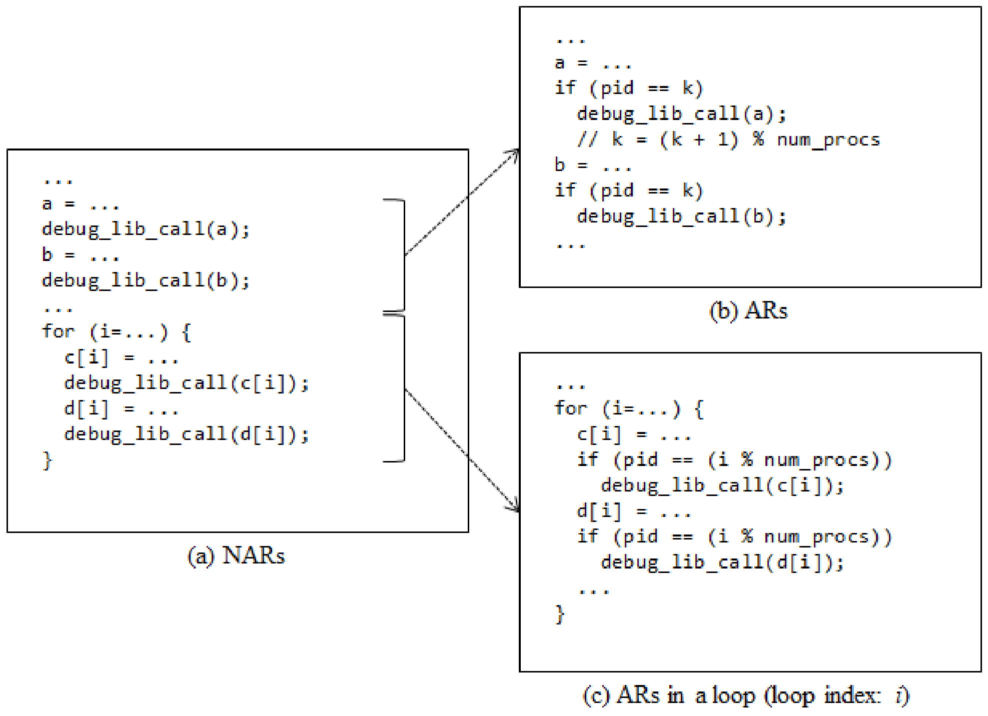
Figure 3.3 shows the time each benchmark spends in ARs and NARs. The graph shows that programs spend the overwhelming part of their execution in ARs where we can distribute monitoring. This observation motivates our instrumentation strategy that provides much lower runtime monitoring cost and good accuracy in locating bugs.

3.2.2 RDP guided instrumentation

Our goal is to spread the monitoring across all processes when all processes are executing a region (i.e., are in an AR) and to ensure that all code is monitored when in a NAR. Thus, within a NAR, the instrumentation at a program point is *replicated*, i.e., performed by all processes, as shown in Figure 3.4(a), with the instrumentation (i.e., the `debug_lib_call`) being executed by all processes. If the spreading happens in a NAR, it is possible that the monitoring task is assigned to a process that is not actually executing the path containing the monitoring point. This case may result in a reduced accuracy as shown in Section 3.4.3. Therefore, we follows a conservative approach when monitoring in a NAR. Within an AR, however, the instrumentation is *distributed* over the processes as shown in Figure 3.4(b) and (c). In this case, the instrumented code is invoked every $\frac{1}{P}$ executions within a process, where P is the number of processes.

If the program point is in an AR, the guard expression controlling the execution of the instrumentation is different in straight line code and in a loop. In straight line code as shown in Figure 3.4(b), the guard expression is *pid* `== k`, where `k` $\in \{0, \ldots, P-1\}$. After being used to guard an instrumentation call, `k` is set to `(k + 1) % P`. In a loop as shown in Figure 3.4(c), the loop index is used to assign the monitoring task of each loop iteration to the different process.

Although we provide a case study and implementation using C-DIDUCE, we believe that the framework can be used with at least two major classes of tools. The first



```
...
a = ...
debug_lib_call(a);
b = ...
debug_lib_call(b);
...
for (i=...) {
  c[i] = ...
  debug_lib_call(c[i]);
  d[i] = ...
  debug_lib_call(d[i]);
}
```

```
...
a = ...
if (pid == k)
  debug_lib_call(a);
  // k = (k + 1) % num_procs
b = ...
if (pid == k)
  debug_lib_call(b);
...
```

```
...
for (i=...) {
  c[i] = ...
  if (pid == (i % num_procs))
    debug_lib_call(c[i]);
  d[i] = ...
  if (pid == (i % num_procs))
    debug_lib_call(d[i]);
  ...
}
```

Fig. 3.4. Instrumentation example by different regions.

class is *invariant violation detection* tools (e.g., [2–4, 8]), which look for violations of program invariants, such as what program location(s) normally access a memory location [4] and what values a variable normally has [2,3]. The second class is tools that find statistical variations in program behaviors that are correlated to bugs (e.g., [5–7]). In both classes, AntDM can reduce overheads compared to monitoring all accesses in all processes, with a minimal impact on precision, and offer similar overheads and improved precision relative to simply distributing the monitoring across processes.

3.3 Parallel Value Invariant Detection – A Case Study

We now present a case study of the AntDM framework and its instrumentation technique using the C-DIDUCE [3] value invariant detection (VID) technique [2] adapted to parallel programs. The details of the C-DIDUCE and VID are described in Section 2.1. In this section, we focus on how the C-DIDUCE and VID is extended to parallel programs with our AntDM framework.

3.3.1 Extending VID to Parallel Programs

We extend the value invariants hypothesis to adapt VID to parallel programs. The following [23][1] shows how to compute the merged invariant set $I' = \langle M'_t, V_i \rangle$, used to extend VID and the detail explanation on how the equation computing the mask is derived is provided in Appendix A:

$$M'_t = \left[\overline{V_k} \vee (\bigwedge_{i=1}^{t} w_{k,i} \wedge V_j \wedge \bigwedge_{i=1}^{t} w_{j,i}) \right] \wedge \left[V_k \vee (\bigwedge_{i=1}^{t} \overline{w_{k,i}} \wedge \overline{V_j} \wedge \bigwedge_{i=1}^{t} \overline{w_{j,i}}) \right],$$

here, $I_k = \langle M_{k,t}, V_k \rangle$ and $I_j = \langle M_{j,t}, V_j \rangle$ are the invariant sets built in two different processes (p_k and p_j) and V_i equal to either V_k or V_j.

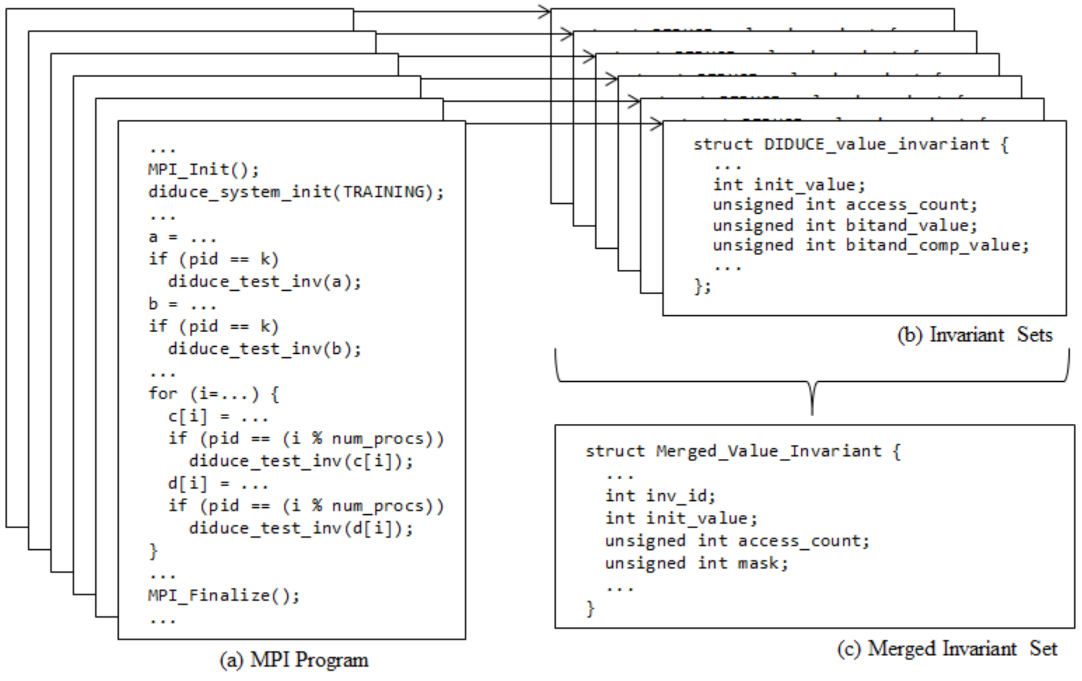
We note that our I' is exactly the I' that would be formed if all dynamic references to the monitored variable at this program point, in all processes of the parallel program, had been used to form a single I, and the variable's value is not a function of the number of processes. Our formulation allows the approximate invariant set for each variable reference to independently collected during the parallel run, and then merged in time proportional to the static number of monitoring points in the program, as required by our parallel value invariance hypothesis.

3.3.2 Using C-DIDUCE with the AntDM framework

As described in Figure 3.1, C-DIDUCE can be easily used with the AntDM framework. For the static analysis, the debugging library information, such as the function names (and relevant parameters) for the invariant training/checking, needs to be pro-

[1]This equation and its derivation was done by Leonardo R. Bachega in an earlier version of this project that used clustering but never published.

vided. This information is used by the AntDM framework when instrumenting the function calls. The initialization function information for C-DIDUCE is also required, therefore a runtime initialization call is inserted right after the MPI runtime initialization. This initialization sets the training/checking mode and allocates memory for the invariant data structures. Upon exiting the program, the invariant information is written to output files and the post-run tools merge the output files. In training mode, the output files contain the value invariant training data and are merged into one training file as described in the previous section. In checking mode, the output files contain the invariant violation information and this information is also merged into one violation list. The different debugging tools may require different rules for merging the output so tools implementing the merging rules are also required.



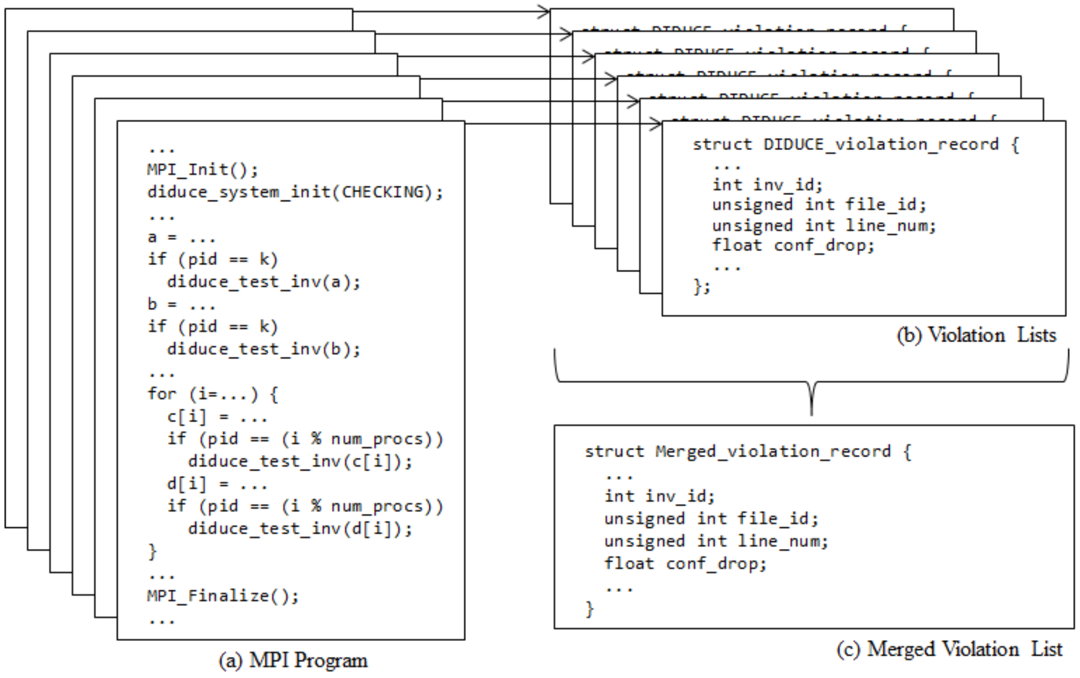
```
...
MPI_Init();
diduce_system_init(TRAINING);
...
a = ...
if (pid == k)
  diduce_test_inv(a);
b = ...
if (pid == k)
  diduce_test_inv(b);
...
for (i=...) {
  c[i] = ...
  if (pid == (i % num_procs))
    diduce_test_inv(c[i]);
  d[i] = ...
  if (pid == (i % num_procs))
    diduce_test_inv(d[i]);
}
...
MPI_Finalize();
...
```

```
struct DIDUCE_value_invariant {
  ...
  int init_value;
  unsigned int access_count;
  unsigned int bitand_value;
  unsigned int bitand_comp_value;
  ...
};
```

```
struct Merged_Value_Invariant {
  ...
  int inv_id;
  int init_value;
  unsigned int access_count;
  unsigned int mask;
  ...
}
```

Fig. 3.5. Example of AntDM's merging Invariant Sets in training mode.

For example, Figure 3.5 shows an example of merging invariant sets gathered by multiple processes in training mode. Each process in multiple machines executes the same code and performs the invariant set testing on variables of interest as shown in Figure 3.5(a). The resulting invariant set is stored in the invariant set files in the form of Figure 3.5(b). As shown in Section 3.3.1, C-DIDUCE's value invariant detection is extended to the parallel program so the bitwise-AND of all the values shown in the variable and the bitwise-AND of all the values' complements are stored in the each process' invariant set file. This information is required to create a file with a single invariant set by applying the merging equation in Section 3.3.1. The merged invariant set file is in the same form as the invariant set file for sequential programs. This single invariant set file is used to detect the violation of the invariant set in checking mode.


```
...
MPI_Init();
diduce_system_init(CHECKING);
...
a = ...
if (pid == k)
  diduce_test_inv(a);
b = ...
if (pid == k)
  diduce_test_inv(b);
...
for (i=...) {
  c[i] = ...
  if (pid == (i % num_procs))
    diduce_test_inv(c[i]);
  d[i] = ...
  if (pid == (i % num_procs))
    diduce_test_inv(d[i]);
}
...
MPI_Finalize();
...
```

```
struct DIDUCE_violation_record {
  ...
  int inv_id;
  unsigned int file_id;
  unsigned int line_num;
  float conf_drop;
  ...
};
```

```
struct Merged_violation_record {
  ...
  int inv_id;
  unsigned int file_id;
  unsigned int line_num;
  float conf_drop;
  ...
}
```

Fig. 3.6. Example of AntDM's merging Violation Lists in checking mode.

The merging of violation lists in checking mode is performed as shown in Figure 3.6. Each process records its detected violations of the merged invariant set into the violation list file. The processes within the same machine write records into the same violation list file. Multiple violation list files from different machines are collected after the execution of the program and merged into a single violation list file. Each file is sorted by the confidence level drop and when the files are merged, the biggest confidence drop in each invariant set is recorded and sorted for the final violation list.

3.3.3 Scalability

Although C-DIDUCE with the AntDM framework uses post-run analysis, it is scalable to a large number of processes and large data sets. In training mode, the number of records in each output file is at most the static number of invariant monitoring points, i.e., it is proportional to the program size and not the program execution time. Merging these files requires a fixed number of set operations on each file as described in Section 3.3.1. Therefore, the execution time for merging training data is linear in the number of processes. In checking mode, the number of records in each output file is also at most the number of invariant monitoring points. Since C-DIDUCE only writes to output files when there are invariant violations in checking mode, the number of records in each file is typically less than the number of monitoring points. Merging these files requires a fixed number of comparisons based on the confidence drop, as described in the DIDUCE paper [2]. Therefore, the execution time for merging the invariant violation data is also linear in the number of processes. The larger data set does not affect the scalability of our post-run analysis within the AntDM framework because the analysis depends on the number of invariant monitoring points, not the size of data set. Here, the larger data set size causes more updating or checking of the invariant at each program point at runtime but does not increase the amount of data being merged from the output files at post-runtime, nor, in the worst case, is the monitoring overhead higher than it would have been without our technique. As shown in our experimental results, the overhead is, in practice, much less than when the technique is applied to sequential programs.

3.4 Experimental Results

In this section, we provide quantitative evidence of the effectiveness of AntDM framework in reducing overheads and detecting buggy behavior with C-DIDUCE.

3.4.1 Implementation and Experimental Setup

Static analysis and instrumentation, described in Section 3.1, are implemented in the Cetus compiler [24–26]. All variable writes in the program and all variable reads of control expressions in the program are monitored. The benchmarks used in the DIDUCE and C-DIDUCE studies [2, 3] are sequential, and so we use the four benchmarks described in Table 3.1: NPB-IS [27], ASCI-SMG2000 [28], SPEC MPI2007-TACHYON and SPEC MPI2007-MILC [29].

Table 3.1
Benchmark characteristics: "NARs count" is the number of NAR code regions; "NARs ratio" is the number of procedures with NARs/total procedures. For both NARs count and ratio, a smaller value is better for Ant performance.

	Description	Lines of code	Executable size	Instrument count	NARs count	NARs ratio
IS	Bucket Sorting	1.2K	680 KB	348	12	1/11
SMG2000	Semi. Multigrid Solver	22.7K	1.1 MB	7278	10	10/349
TACHYON	Parallel Ray Tracing	12.9K	890 KB	1732	22	17/413
MILC	Quantum Chromodynamics	15.8K	871 KB	3560	115	48/310

The same kinds of bugs were injected as with the original DIDUCE and C-DIDUCE studies, and the bug types are the same as those found in the Siemens bug benchmarks [11]. Eight to eleven bugs were injected into each benchmark, with each bug injected into a different copy of the benchmark. The bugs are triggered by all processes that execute a path containing the bug.

Table 3.2 shows the types and the number of injected bugs in each benchmark. Bugs types are: *Value Mutation* which changes an assignment like `a = x` to `a = x + c`; *Loop Mutation* which changes loop bounds from `i < mp` to `i < mp+1`; *Control Mutation* mutates the operator of conditional expression in which changes an `if` statement condition from `(a > b)` to `(a <= b)`. Bugs were injected into both NARs and ARs, and most bugs were placed into ARs.

We used machines with two quad-core Intel Xeon 2.33GHz processors, 16 GB of memory, Linux 2.6.18 and the mpich2-1.0.8. The training run for all benchmarks was done with 2 processes, and the detection run was done with 16 processes for MILC and 32 processes for the other three benchmarks.

Table 3.2
The types and the number of injected bugs. “NAR percentage” is the percentage of code in NARs; “Bug count” is the total bug count and the number by each type (V: value mutation/ L: loop mutation/ C: control mutation); and “NAR Bug percentage” is the percent of bugs in NARs.

	NAR Line of Code	NAR percentage	Bug count (V/L/C)	Bug count in NAR	NAR Bug percentage
IS	126	10.5 %	8 (7 / 1 / 0)	1	12.5 %
SMG2000	373	1.6 %	10 (8 / 0 / 2)	1	10.0 %
TACHYON	576	4.0 %	11 (11 / 0 / 0)	1	9.1 %
MILC	2887	18.0 %	9 (4 / 0 / 5)	1	11.1 %

3.4.2 Performance of Optimized Parallel Value Invariant Detection

Figure 3.7 compares C-DIDUCE monitoring overhead among replicated ("Replicated"), AntDM's AR/NAR based monitoring ("AntDM") and naive distributed ("Distributed") schemes for our benchmarks. The figure shows that there is significant overhead reduction going from Replicated to AntDM and Distributed, with a reduction of 15.85X for NPB IS, 4.28X for ASCI SMG2000, 11.14X for SPECMPI TACHYON and 5.63X for SPECMPI MILC. The reason why the maximum overhead reduction for Distributed is less than the number of the processes is that Distributed monitoring itself incurs the overhead of checking the process rank at each monitoring point, as described in Section 3.2.2. The AntDM and Distributed overheads are very similar (differing by 1.4% to 13%) and low, because the programs are usually executing ARs, as seen in Figure 3.3. As discussed in the next section, accuracy is better with AntDM than Distributed. Since AntDM's monitoring is distributed in ARs, and analysis and instrumentation occur offline, our technique is inherently scalable with increasing process counts.

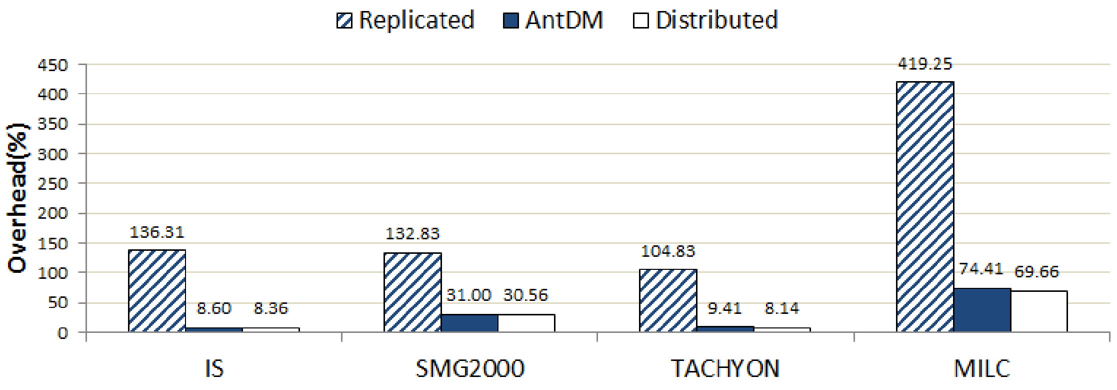


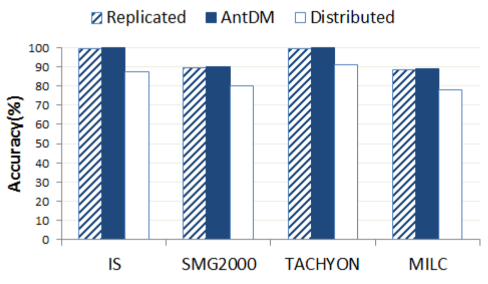
Fig. 3.7. The comparison of C-DIDUCE overhead against the execution time with no instrumentation.

3.4.3 Accuracy of Optimized Parallel Value Invariant Detection

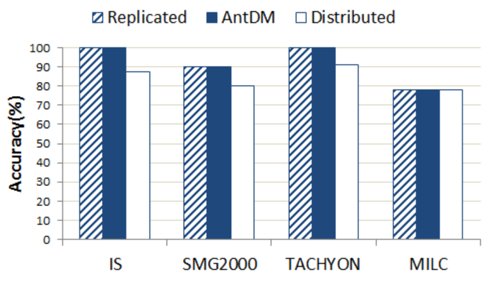
We now present experimental data showing the effectiveness of the three different monitoring schemes in detecting the injected bugs.

Note that even with fully replicated monitoring, some bugs go undetected because (1) they may not be executed by C-DIDUCE, or (2) they may not appear as bugs because the statement is only executed a small number of times and all values appear equally valid, or the approximation (V and M_t) used by DIDUCE misses outlier values. This happens with DIDUCE and C-DIDUCE in sequential programs.

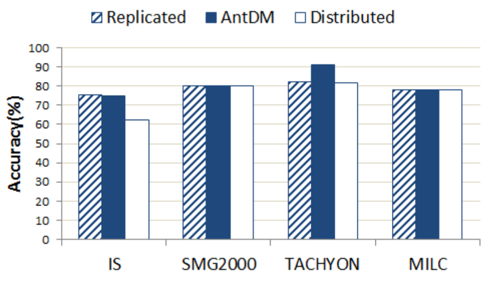
Training runs were done with the original, correct benchmarks using small data sets. After training, each copy of a benchmark containing an injected bug was run with the large data set under all three monitoring versions.



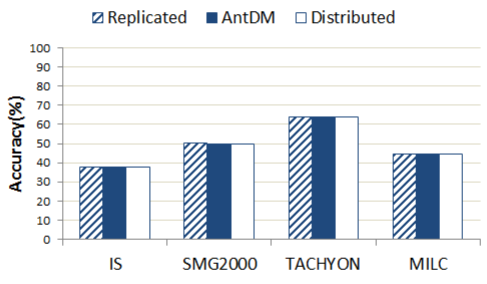
(a) Any place



(b) Top 40



(c) Top 20



(d) Top 10

Fig. 3.8. Accuracy of bug detection by the ranking in the violation list. Any place means any rank in the violation list was considered as successful detection. Top 40 means the ranking is within top 40 of violation list. Top 20 is within first 20 of violation list. Top 10 is within first 10.

Figure 3.8 presents bug detection rates for each version of C-DIDUCE. A detection rate of 100% means that all injected bugs are detected by C-DIDUCE. Because DIDUCE and C-DIDUCE rank anomalies as to the likelihood of them being a bug, we report the rates for bugs that occur in the top 10, 20, or 40 anomalies, or that are detected anywhere. Note that AntDM is nearly as accurate as Replicated, despite having a much lower overhead, showing the effectiveness of the AntDM monitoring technique. AntDM is also more accurate than Distributed because of AntDM monitoring all accesses in all processes within NARs, with the bugs found by AntDM and not Distributed all being in NARs. Thus AntDM uses a distributed scheme when it is safe to and otherwise uses a replicated scheme. With TACHYON in Figure 3.8(c), AntDM does better than Replicated because a program crash causes AntDM to lose violation data which coincidentally causes an injected bug to be ranked higher.

3.4.4 Discussion

Our experimental results show that the distributed versions of C-DIDUCE increased the performance of the invariant detection by up to 15X while (unlike Distributed) maintaining almost the full accuracy of the expensive Replicated monitoring. We could further reduce the overhead of AntDM's monitoring by using *clustering* of similarly behaving processes [19, 20] to determine the clusters within NARs, and distributing the monitoring across the processes within each cluster, in the same way AntDM does with ARs. This will be particularly important for programs that spend more time in NARs.

4. ANTSM: ANT FRAMEWORK TARGETING SHARED MEMORY PARALLEL PROGRAMS

4.1 The AntSM framework

We call our technique that targets Pthreads programs, *AntSM* [30]. The key insight of the AntSM framework is that different instances of the same code executing in parallel in different threads are likely to behave similarly, and that sampled monitoring over that code can reduce overheads with only a small impact on accuracy. The Ant Shared Memory (or AntSM) system exploits this key observation to reduce the overhead of debugging tools when used with shared memory parallel programs. AntSM uses the parallelism of the multi-threaded shared memory programs being monitored to reduce the overhead of the debugging tool, while maintaining a high level of accuracy. It does this by first instrumenting the program with calls to the AntSM runtime library to collect and maintain information about parallelism in the program. The program is then instrumented with monitoring and other calls for the bug detection technique being used. The data used for the bug detection can be collected in two different ways. One is using a centralized table protected by synchronization and the other is using a per-thread table which will be merged later. At runtime, the parallel structure of the program and the number of threads executing some regions of the program are used to perform an intelligently sampled monitoring.

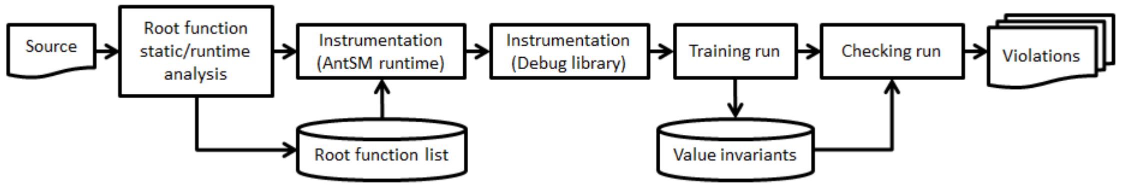
To provide insights into AntSM's strategy, we now contrast how it, and a straightforward port of a monitoring-based debugging tool, function. In this thesis, we use the statistical and invariance-based debugging tool called C-DIDUCE, used in Chapter 4's case study, that asserts the value invariant hypothesis. The value invariant hypothesis states that a given variable takes on a small set of values during its lifetime, even with different input data, and rarely occurring deviations from this set of values indicate buggy or anomalous behavior.

DIDUCE and other invariance based tools typically have a training phase and a checking phase. During the training phase, the program being debugged is run with data that gives a correct answer. Each action of interest is monitored and the outcome of that action is recorded. For DIDUCE, each variable reference is monitored and the value seen is recorded in a compressed form. These outcomes form an invariant set of outcomes that are true for correct executions. During checking runs, each action of interest is monitored and the outcomes that are deviations from the invariant set are monitored and recorded. This monitoring and recording often incurs a high overhead, and it is this overhead that we seek to reduce with our techniques. After the program executes, the deviations from the invariant set are ranked. Frequently occurring deviations are considered more likely to be invariants that simply were not seen during the training runs, and are ranked lower. Rarely occurring deviations are considered more likely to be signs of a bug, and are ranked higher.

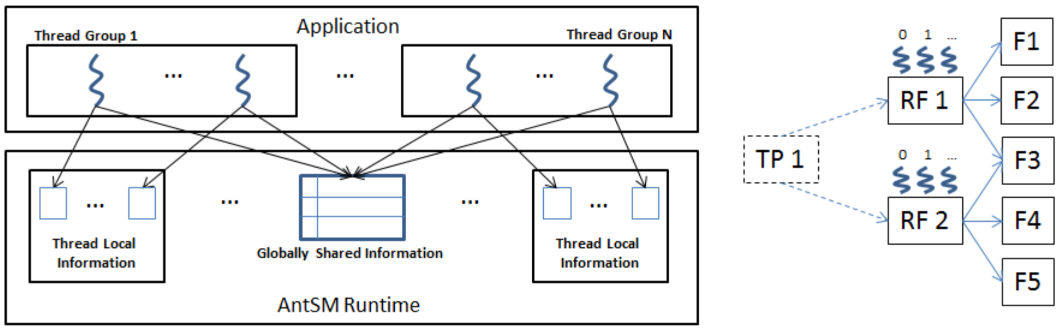
A straightforward port of a tool would simply instrument a parallel program as if it were a sequential program, and monitor all actions of interest in the program. Ignoring overheads induced by the tool running in a parallel environment and needing to be thread-safe, this would produce the same overhead as a sequential execution of the program that executed the same number of monitored actions. Thus each thread executes all of the monitoring, a mode that we call *replicated* monitoring.

One way to reduce the overhead of replicated monitoring is to have each of the T threads executing the program monitor $\frac{1}{T}$ of the events. This performs a *distributed* sampled monitoring across all T threads. This significantly reduces the monitoring overhead but can lead to less accuracy in detecting anomalous events that indicate a bug, as shown in Section 4.4.4. The loss of accuracy results from each thread only sampling $\frac{1}{T}$ events, even in program regions that are not executed by all T threads. This leads to some actions being severely under-monitored or completely missed by monitoring.

AntSM takes a more intelligent approach, and by doing so achieves nearly the low overhead of distributed monitoring and accuracy close to that of replicated monitoring. A typical Pthreads program either spawns threads that directly call a function that performs the thread's share of the computation, or spawns threads that are in a thread pool, check a work queue, and perform the computation defined by the function that is implied by the queue entry. We call all these functions that perform the computation *root* functions. By instrumenting and analyzing the thread spawning points and the root functions, and tracking when threads enter and exit root functions, the exact number of threads performing the computation associated with the root function can be determined. This count, T_c, can be used to perform sampling of $\frac{1}{T_c}$ actions rather than $\frac{1}{T}$ actions, and avoid severely undersampling program actions of interest. Moreover, because the functions associated with a given root function are engaged in the same operation on different data, sampling within these functions is more likely to be sampling from a set of similar actions than simply randomly sampling across the entire program, which should lead to higher accuracy. Within loops, this sampling is implemented by each thread executing $\frac{1}{T_c}$ instances of statements, and within straight-line code, by each thread executing each $\frac{1}{T_c}$ statements in the textual representation of the program. For example, if we consider the situation in which there are 10 threads spread equally among 2 root functions and assume that 1000 invariant checks are done by the code executed from each root function, the *replicated* monitoring will perform 2000 samplings. The *AntSM* monitoring will do 400 samplings ($\frac{1}{5} \times 1000 + \frac{1}{5} \times 1000$) and the *distributed* monitoring will do 200 samplings ($\frac{1}{10} \times 2000$). This shows that *distributed* monitoring scales as $\frac{1}{T}$ whereas AntSM scales within each parallel region executing the same code. Thus AntSM samples at a higher rate than *distributed* but less than *replicated*, giving better performance than *replicated*. AntSM also samples based on the parallelism in regions executing common code and thus gives better precision than *distributed*.


Source
Root function static/runtime analysis
Instrumentation (AntSM runtime)
Instrumentation (Debug library)
Training run
Checking run
Violations
Root function list
Value invariants

(a) Overview of the AntSM debugging system.



(b) AntSM Runtime and a runtime call graph. `RF` is a root function and the `Fs` represent other functions. `TP1` shows the case with a thread pool. Squiggly lines represent threads, numbered by thread group ID.

Fig. 4.1. Steps performed by AntSM and the AntSM runtime system.

4.2 AntSM Runtime and Instrumentation

We now describe how code is instrumented and the information gathered by that instrumentation is used to enable AntSM's intelligent sampling strategy (see Figure 4.1.)

4.2.1 Finding root functions

First, root functions must be identified directly from the *start_routine* argument to the `pthread_create` function. Programs using a thread pool require instrumenting all functions to log when the function is entered and exited, and printing the function name and system thread ID. From this, a simple script can extract root function names. The logging of function names with the thread id incurs about a 20X runtime overhead, but this task is required only once at the time the root functions are identified. This is unnecessary if the programmer already knows what functions are used for the root functions by having the programmer provide the function name list to the AntSM. Even when this is not the case, we could identify the root functions within a few minutes to an hour at most.

4.2.2 Instrumentation for tracking code executed by root functions

After root functions are identified, they are instrumented with calls to the AntSM runtime library to monitor when a thread starts and finishes executing the root function. This information is made available to any code that is executed within the root function or any function called (directly or indirectly) from the root function. We refer to this code as a *root reachable code*, and the threads executing it as a *thread group*. Each thread in the program is given a "*thread_id*" by the system. AntSM also maintains for each thread a local ID, called the "*group_id*", where $0 \leq group\_id < T_c$ and T_c is the number of threads executing a particular root function. AntSM also maintains a mapping between *thread_id*s and *group_id*s. This allows the thread to test if it should perform a particular monitoring operation.

4.2.3 Instrumentation for collecting monitoring data

Next, the program is instrumented with calls to the debugging tool's library to perform sampling. A training run is then performed to build the initial invariant sets, and then one or more checking runs are performed to identify potentially buggy program points. As mentioned in Section 4.1, the data used for the bug detection can be collected in two different ways.

The first way uses a typical shared memory programming style, i.e., the shared information updated by multiple threads is protected by synchronization. In this way, a single shared invariant table and a single violation list are used by multiple threads. All the accesses of the table by multiple threads are synchronized using Pthreads mutexes and conditional variables. This is a space-efficient way for the scalable systems. As the number of threads increases, the memory usage of invariant information does not change.

However, if the memory efficiency is not an issue, an alternative way is possible when implementing the multi-threaded data collection that gives better performance. Instead of using a single shared invariant table and violation list, a separate invariant table and violation list for each thread can be created and updated during execution of multiple threads and these tables are then merged at the end of the program execution. This is how AntDM extends C-DIDUCE for distributed memory parallel programs. This separation causes some space overhead. For example, assuming that each value invariant requires 32 bytes of memory and there are 10,000 monitoring points in a target program, 320 Kbytes of memory is required for a value invariant table. Using the separate tables by 8 threads causes the space overheads of 2.2 Mbytes (7×320 KB) and using the tables by 32 threads causes the space overheads of 9.9 Mbytes (31×320 KB). Although this separation causes some space overhead, we can prevent the interaction between the multiple threads and thus remove the high overhead of synchronization.

4.2.4 AntSM runtime algorithm

We now describe the algorithm of AntSM runtime in more detail. To maintain the information about parallelism, i.e., how many threads are executing some root reachable code, the AntSM runtime provides two library functions - *antsm_enter_root* and *antsm_exit_root*. One purpose of this instrumentation is to track when a thread begins executing a particular root function, and when a thread stops executing a root

Algorithm 2 AntSM runtime library, antsm_enter_root

Input: *root_addr* - an address of a root function

Output: Set of thread-local variables and thread-global variables

```
 1: // thread-local: each thread keeps own copy of these variables
 2:     thread_id ← syscall(SYS_gettid) // system thread ID
 3:     group_id // unique thread ID in its thread group assigned by AntSM
 4:     my_root_addr ← root_addr // root function address
 5: // thread-global: all threads share these variables
 6:     root_map // thread_id → root function address used by antsm_exit_root
 7:     group_id_map // thread_id → thread ID in its thread group
 8:     thread_cnt_map // my_root_addr → runtime thread count
 9: root_map[thread_id] ← my_root_addr
10: if thread_cnt_map[my_root_addr] is not set then
11:     // this is the first thread that enters the function
12:     group_id_map[thread_id] ← group_id ← 0
13:     thread_cnt_map[my_root_addr] ← 1
14: else
15:     group_id_map[thread_id] ← group_id ← thread_cnt_map[my_root_addr]
16:     thread_cnt_map[my_root_addr] ← thread_cnt_map[my_root_addr] +1
17: end if
```

function. This allows the AntSM runtime to know how many threads are executing each root function, i.e., the value T_c for each root function. The second purpose of this instrumentation is to ensure that all *group_id*s lie between 0 and $T_c - 1$. When the thread that finishes executing a root function is not the thread with the highest valued *group_id*, it is necessary to adjust the *group_id*s of the remaining threads to maintain the constraint that all *group_id*s lie between 0 and $T_c - 1$.

A call to *antsm_enter_root*, described in Algorithm 2, is inserted at the beginning of each root function. The *thread_id* and root function address are captured in thread-local variables (lines 2 and 4). Line 9 associates the root function's address with the current thread. Because parallelism information is kept for each root function, line 10 checks if another thread is already executing the root reachable code. If not, in line 12 the current thread is given the *group_id* of 0 in the current thread group (the set of threads executing this root reachable code) and the thread count is set to 1 (line 13). If other threads are executing code from this root, the *group_id* for this group is set to the number of threads that were already executing code from the root (line 15) and the thread count for this root is incremented (line 16). At this point, each thread executing root reachable code has access to its position within its thread group, and the total number of threads in the thread group. Note that thread-local variables are used to avoid unnecessary synchronization for better performance in the AntSM runtime. In Algorithm 2, all the accesses to the thread-global data structures must be guarded by the proper synchronization techniques. The hashmap variables (*root_map*, *group_id_map* and *thread_cnt_map*) may be accessed by multiple entering/exiting threads at the same time. Pthreads mutexes and condition variables with the read/write counters are used to synchronize the accesses.

A call to *antsm_exit_root*, described in Algorithm 3, is inserted at the exit points of each root function. This function updates the count of threads executing a root reachable code, and ensures each *group_id* has values between 0 and $T_c - 1$. The function first decrements the thread count for the current thread group (lines 4 and 5) and nulls out its entry in the *group_id* map and the root function map (lines

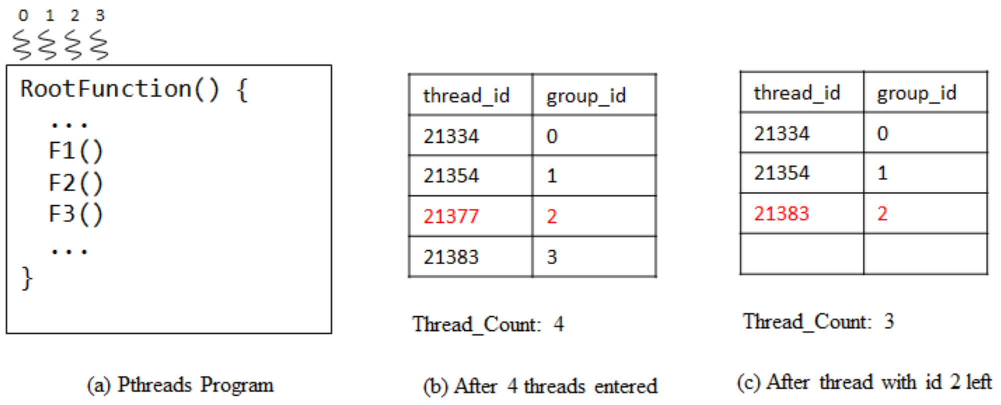
Algorithm 3 AntSM runtime library, antsm_exit_root

Input: *root_addr* - an address of a root function

Output: Set of thread-local variables and thread-global variables

```
1: // thread local and global variables are as in antsm_enter_root in Algorithm 2
2: // local (automatic) variable:
3:     thread_cnt
4: thread_cnt ← thread_cnt_map[my_root_addr] −1
5: thread_cnt_map[my_root_addr] ← thread_cnt
6: if group_id_map[thread_id] = thread_cnt then
7:     group_id_map[thread_id] ← NULL
8:     root_map[thread_id] ← NULL
9: else if group_id_map[thread_id] < thread_cnt then
10:    find group_id_map[$thread\_id_i$] where root_map[$thread\_id_i$] = root_addr
       and group_id_map[$thread\_id_i$] = thread_cnt
11:    // i between 0 and size[group_id_map] −1
12:    group_id_map[$thread\_id_i$] ← group_id_map[thread_id]
13:    group_id_map[thread_id] ← NULL
14:    root_map[thread_id] ← NULL
15: end if
```

7, 8, 13 and 14) since the thread is no longer active in executing a root reachable code. Because of the way the monitoring code is generated (as described below), the *group_id* must always be in the range, $0 \leq group\_id < thread\_cnt$. Thus, if the current thread's *group_id* is less than the decremented thread count (line 9), then the thread with the highest *group_id* in its group will have a *group_id* equal to the thread count in its group. In this case, the thread with its *group_id* equal to thread count is found (line 10), and assigned the current thread's *group_id* in its group (line 12). As in Algorithm 2, all accesses to the thread-global data structures (*root_map*, *group_id_map* and *thread_cnt_map*) must be also protected by the synchronization techniques. The same Pthreads mutexes and condition variables with the read/write counters that are used in the *antsm_enter_root* function are also used to synchronize these accesses.



```
RootFunction() {
  ...
  F1()
  F2()
  F3()
  ...
}
```

(a) Pthreads Program

thread_id	group_id
21334	0
21354	1
21377	2
21383	3

Thread_Count: 4

(b) After 4 threads entered

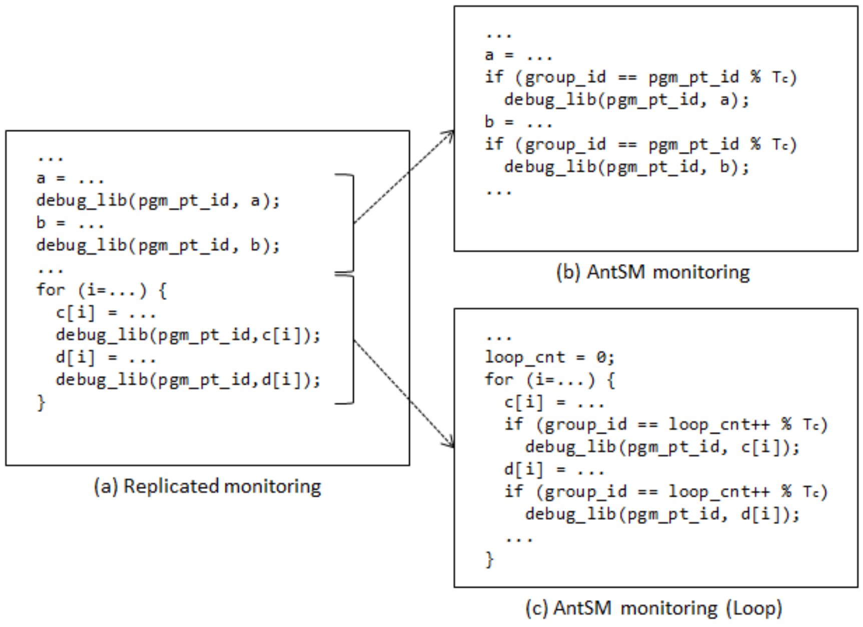
thread_id	group_id
21334	0
21354	1
21383	2

Thread_Count: 3

(c) After thread with id 2 left

Fig. 4.2. Example of updating *group_id* when entering/exiting root function.

For example, in the Pthreads program snippet of Figure 4.2(a), when an initial thread enters a root function, `RootFunction`, the *group_id* of `0` in its thread group, is assigned and the associated thread's count is increased by one (line 12 and 13 of Algorithm 2). If three more threads execute the `RootFunction` function, each will execute lines 15 and 16 of Algorithm 2 (i.e., thread_cnt_map[`RootFunction`] is 4) as shown in Figure 4.2(b). If a thread with *group_id*, 2, exits `RootFunction` (giving a "true" condition in line 9 of Algorithm 3), the thread with *group_id*, `3`, is found and its *group_id* is replaced with the leaving thread's *group_id*, `2` (lines 10 and 12 of Algorithm 3). Therefore, the range of thread IDs in this group is maintained within the updated thread count of 3. The resulting thread id mapping is shown in Figure 4.2(c). The *group_id* for each thread group, set in lines 12 and 15 of Algorithm 2, is also used to distribute monitoring as described in Section 4.2.5.



```
...
a = ...
debug_lib(pgm_pt_id, a);
b = ...
debug_lib(pgm_pt_id, b);
...
for (i=...) {
  c[i] = ...
  debug_lib(pgm_pt_id,c[i]);
  d[i] = ...
  debug_lib(pgm_pt_id,d[i]);
}
```

```
...
a = ...
if (group_id == pgm_pt_id % Tc)
  debug_lib(pgm_pt_id, a);
b = ...
if (group_id == pgm_pt_id % Tc)
  debug_lib(pgm_pt_id, b);
...
```

```
...
loop_cnt = 0;
for (i=...) {
  c[i] = ...
  if (group_id == loop_cnt++ % Tc)
    debug_lib(pgm_pt_id, c[i]);
  d[i] = ...
  if (group_id == loop_cnt++ % Tc)
    debug_lib(pgm_pt_id, d[i]);
  ...
}
```

Fig. 4.3. Debugging library instrumentation example.

4.2.5 Sampled monitoring of AntSM

With AntSM, the sampled monitoring of program points is done within a thread group. If the program point is in straight-line code (Figure 4.3(b)), AntSM generates the conditional statement:

$$\texttt{if (}\ group\_id \texttt{ == } pgm\_pt\_id \texttt{ \% } T_c \texttt{)}$$

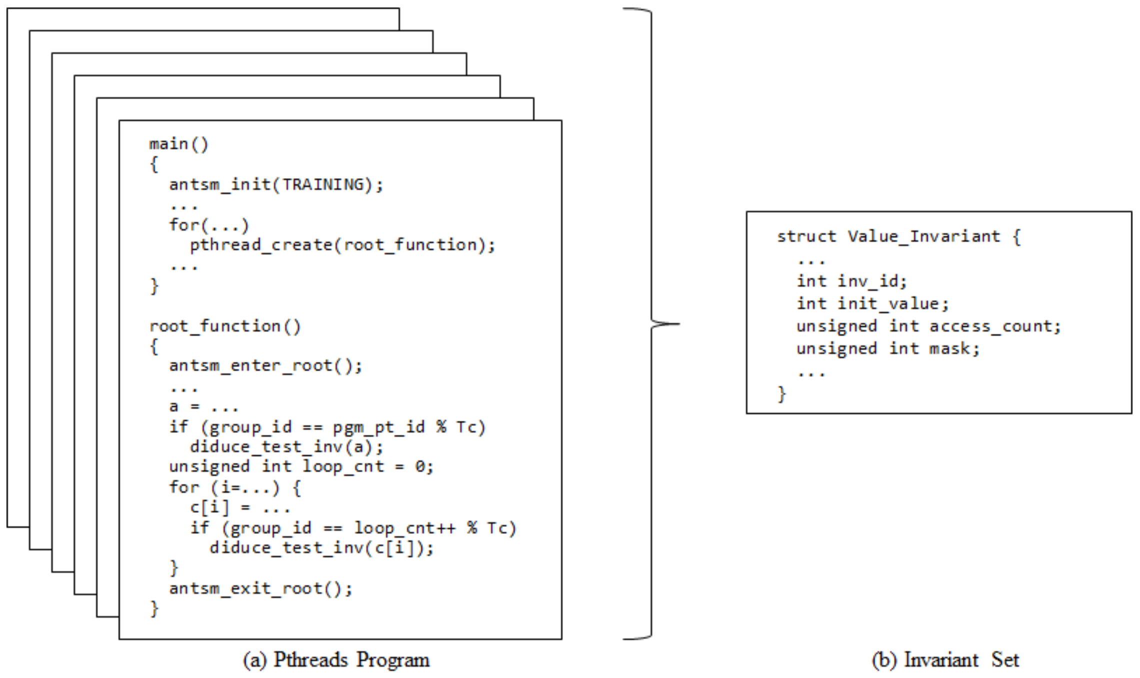
where *group_id* $\in \{0, \ldots, T_c - 1\}$ and *pgm_pt_id* is the numerical ID given for the current program point. Sampling in loops (Figure 4.3(c)) is done using an inserted count variable, *loop_cnt*, which helps the monitoring of each statement's instances within the loop to be distributed evenly among the threads in their group. *antsm_enter_root* and *antsm_exit_root* cause the number of threads, T_c, to change dynamically as threads enter and exit the root function, as was described in Section 4.2.4. AntSM performs replicated monitoring (Figure 4.3(a)) on code that is never executed in parallel, i.e., not reachable from a root function.

4.3 A Case Study with C-DIDUCE and Value Invariant Detection

We now present a case study of the AntSM framework and its instrumentation technique using the C-DIDUCE [3] value invariant detection (VID) technique [2] adapted to shared-memory parallel programs. The details of C-DIDUCE and VID are described in Section 2.1.

When using AntSM with C-DIDUCE and Pthreads programs, a runtime initialization call is inserted at the beginning of the program to initialize the C-DIDUCE runtime. This initialization records whether the run is a training or checking run, allocates memory and initializes the invariant data structures. Upon exiting the program, the invariant information is written to an output file. In training mode, the output files contain the value invariant set for all monitored points. In checking mode, the output files contain invariant violation information.

For example, Figure 4.4 shows an example of the instrumented target program with C-DIDUCE debugging and AntSM runtime libraries. In the `main` function in Figure 4.4(a), the initialization function of AntSM, `antsm_init`, is inserted with the mode option, `TRAINING`, and in the root function, `root_function`, the AntSM runtime functions, `antsm_enter_root` and `antsm_exit_root` are inserted at the beginning and end of the root function respectively. The C-DIDUCE libraries are inserted at the monitoring point of interest as described in Section 4.2.5. The resulting invariant



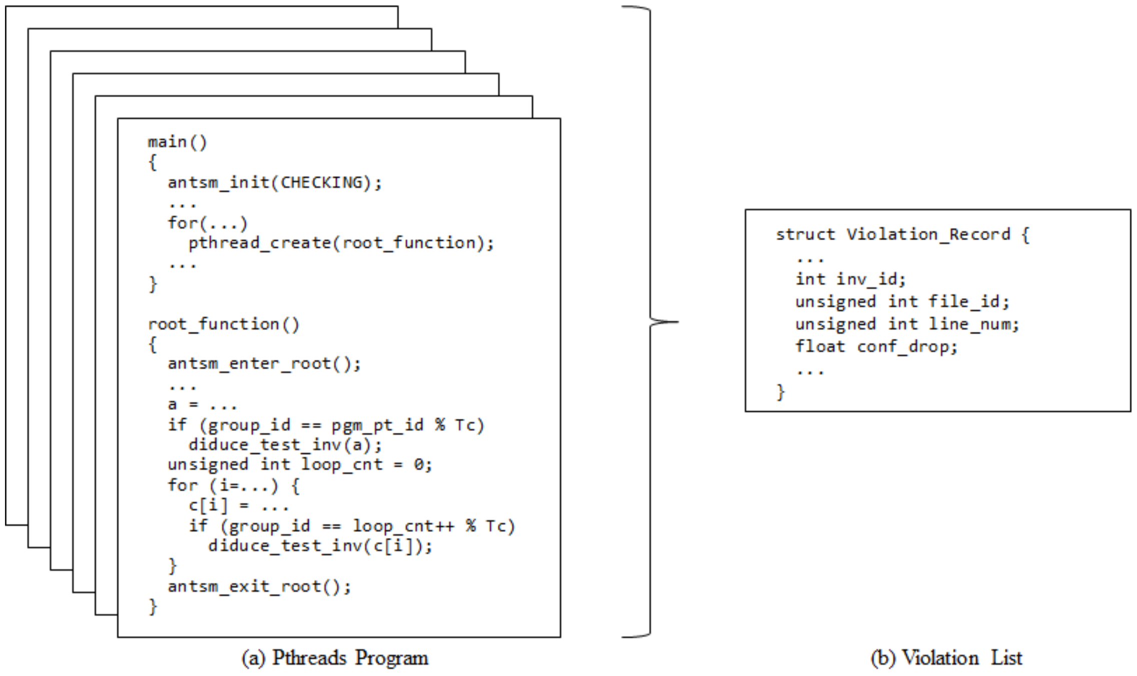
```
main()
{
  antsm_init(TRAINING);
  ...
  for(...)
    pthread_create(root_function);
  ...
}

root_function()
{
  antsm_enter_root();
  ...
  a = ...
  if (group_id == pgm_pt_id % Tc)
    diduce_test_inv(a);
  unsigned int loop_cnt = 0;
  for (i=...) {
    c[i] = ...
    if (group_id == loop_cnt++ % Tc)
      diduce_test_inv(c[i]);
  }
  antsm_exit_root();
}
```

```
struct Value_Invariant {
  ...
  int inv_id;
  int init_value;
  unsigned int access_count;
  unsigned int mask;
  ...
}
```

Fig. 4.4. Example of C-DIDUCE with AntSM in training mode.

information is recorded in the invariant set in the form of Figure 4.4(b). When the value invariant testing fails, the current invariant mask of the variable is updated by applying the newly added invariant information.

As shown in Figure 4.5, the initialization function of AntSM, `antsm_init`, is also inserted with the mode option, `CHECKING`, and the root function is instrumented with the entering/exiting AntSM runtime functions in the same way as training mode. When the invariant testing fails, the confidence level drop is computed and if the drop is greater than the previous confidence drop, this violation is recorded and sorted in the violation list.


(a) Pthreads Program

```
main()
{
  antsm_init(CHECKING);
  ...
  for(...)
    pthread_create(root_function);
  ...
}

root_function()
{
  antsm_enter_root();
  ...
  a = ...
  if (group_id == pgm_pt_id % Tc)
    diduce_test_inv(a);
  unsigned int loop_cnt = 0;
  for (i=...) {
    c[i] = ...
    if (group_id == loop_cnt++ % Tc)
      diduce_test_inv(c[i]);
  }
  antsm_exit_root();
}
```

(b) Violation List

```
struct Violation_Record {
  ...
  int inv_id;
  unsigned int file_id;
  unsigned int line_num;
  float conf_drop;
  ...
}
```

Fig. 4.5. Example of C-DIDUCE with AntSM in checking mode.

4.4 Experimental Results

4.4.1 Implementation and Experimental Setup

Static root function analysis and instrumentation, described in Section 4.2, are implemented in the LLVM compiler, v 3.1 [31–33]. When a thread pool is used, we find root functions as described in Section 4.2. All memory loads, stores, and return values from function calls are monitored. We use eleven programs from the PARSEC Pthreads benchmark suite [10] described in Table 4.1. Two programs from this suite are not used: *freqmine*, which uses OpenMP, not Pthreads[1], and *facesim*, which LLVM cannot compile.

The bugs which are injected into our benchmark programs are the same kind as those used in the original DIDUCE and C-DIDUCE studies and in the Siemens bug

[1]No significant technical challenge prevents us from using OpenMP.

benchmarks [11]. Five to fifteen bugs were injected into each benchmark, with each bug injected into a different copy of the benchmark. To allow an accurate comparison of our technique with C-DIDUCE, bugs are injected at frequently executed program points. If a program point is not frequently executed, it is possible that our sampling will miss “noise” and capture relatively more buggy actions. This in turn makes our sampled executions appear better than full monitoring. Because of this, the number of injected bugs is not proportional to the lines of code in Table 4.1.

We used machines with two quad core Intel Xeon 2.33GHz processors, 16 GB of memory, and Linux 2.6.32 for the performance and accuracy experiments; machines with 48 AMD Opteron 6176, 2.3 GHz processors, 256 GB of memory, and Linux 2.6.32 were used for the scalability test. For the performance and accuracy experiments, training runs were done using 2 threads and the small dataset. Checking runs were done using 8 threads and the large dataset.

Table 4.1
Summary of the PARSEC benchmark characters: "Monitored Points" is the number of static program points monitored; "Thread Pool" says if the benchmark uses a thread pool; "Injected Bugs" is the number of bugs injected; and "Original Speedup is the speedup of the un-instrumented benchmark going from 1 to 8 threads.

Name	Application Domain	Lines of Code	Monitored Points	Thread Pool	Injected Bugs	Original Speedup
blackscholes	Financial Analysis	408	180	No	8	4.63
bodytrack	Computer Vision	3066	6544	Yes	15	5.62
canneal	Engineering	371	207	No	7	1.37
dedup	Enterprise Storage	398	553	Yes	8	2.03
ferret	Similarity Search	8940	9141	Yes	5	2.86
fluidanimate	Animation	2733	1329	No	6	4.02
raytrace	Visualization	3553	2757	Yes	7	1.28
streamcluster	Data Mining	1720	978	No	6	3.46
swaptions	Financial Analysis	994	898	No	14	7.98
vips	Media Processing	98940	21168	No	10	7.63
x264	Financial Analysis	26437	14705	No	15	5.71

4.4.2 Performance of C-DIDUCE with AntSM

Figure 4.6 compares the overhead of different monitoring schemes. The baseline is the original benchmark execution time (without any monitoring). The bars labeled "Replicated", "AntSM" and "Distributed" are for the naive replicated monitoring scheme, AntSM's sampled monitoring, and the distributed monitoring scheme, respectively. Note that the vertical axis is on a log scale. The benchmark names are labeled with the reduction in overhead going from the replicated scheme to the AntSM scheme ("Replicated" to "AntSM"). AntSM shows up to 18.14 times overhead reduction (*dedup*) and an average reduction of 8.73 times.

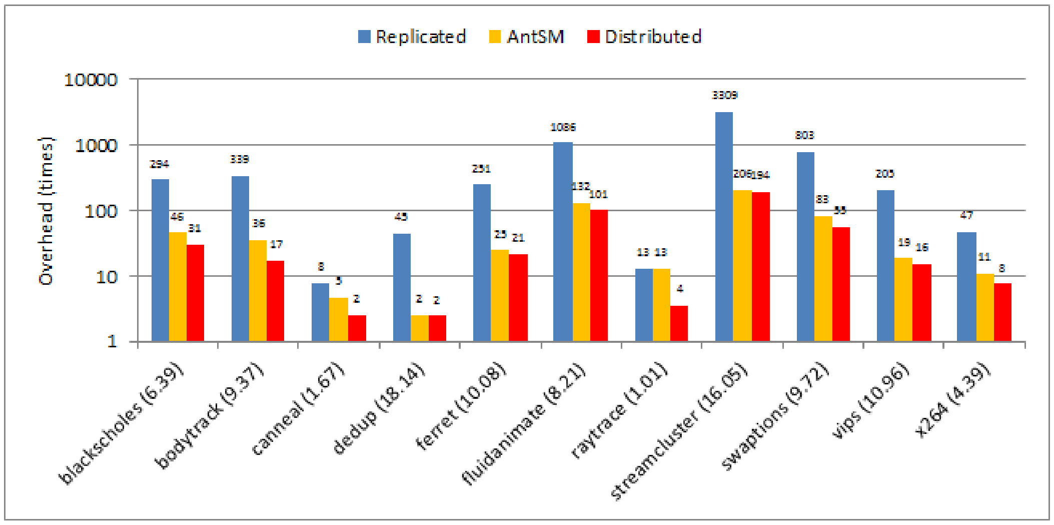


Fig. 4.6. Comparison of C-DIDUCE execution time overhead in checking mode. The baseline is the execution time of the original benchmark with large dataset and no instrumentation. Note that the vertical axis is on a log scale. The data label on each bar shows the overhead (times) rounded to the nearest one. The number next to each benchmark's name represents the overhead reduction from *Replicated* to *AntSM*.

Two benchmarks with low overhead reduction are *canneal* (1.67X) and *raytrace* (1.01X). As shown in Table 4.1, these benchmarks have a low original speedup, indicating little parallelism and few opportunities for AntSM to perform sampled monitoring. In particular, the *raytrace* benchmark executes almost entirely sequentially. Measuring overhead in only the parallel section of *raytrace* gives an overhead reduction of 2.17 for AntSM. The naive "Distributed" scheme gives the best performance because this scheme performs a $\frac{1}{T}$, where T is the number of threads, sampling even in sequential areas of the program. As shown in Section 4.4.4, "Distributed" has a lower accuracy than the other two schemes.

Training runs were done with 2 threads and the overhead reductions from "Replicated" to "AntSM" (measured as with the checking runs) are 4.40X for *blackscholes*, 3.55X for *bodytrack*, 1.39X for *canneal*, 0.97X for *dedup*, 2.17X for *ferret*, 4.23X for *fluidanimate*, 1.08X for *raytrace*, 2.46X for *streamcluster*, 3.86X for *swaptions*, 1.78X for *vips*, and 1.80X for *x264*. Low overhead reductions occur because the initial AntSM startup overhead is not amortized on a small number of threads and smaller data set used for some benchmarks.

4.4.3 Scalability Results

We now present experimental data showing the scalability of AntSM when monitoring value invariants. Figure 4.7 and Table 4.2 present the speedup of AntSM with an increasing number of threads. The baseline for the speedup is the execution of C-DIDUCE with AntSM in checking mode, executing with a single thread. As the table shows, AntSM scales in most benchmarks as the number of threads increases. There are three benchmarks showing low scalability (*raytrace*, *canneal*, and *dedup*) but as shown in the last column of Table 4.2, the original speedup of those benchmarks are low, resulting in the low scalability in AntSM as well.

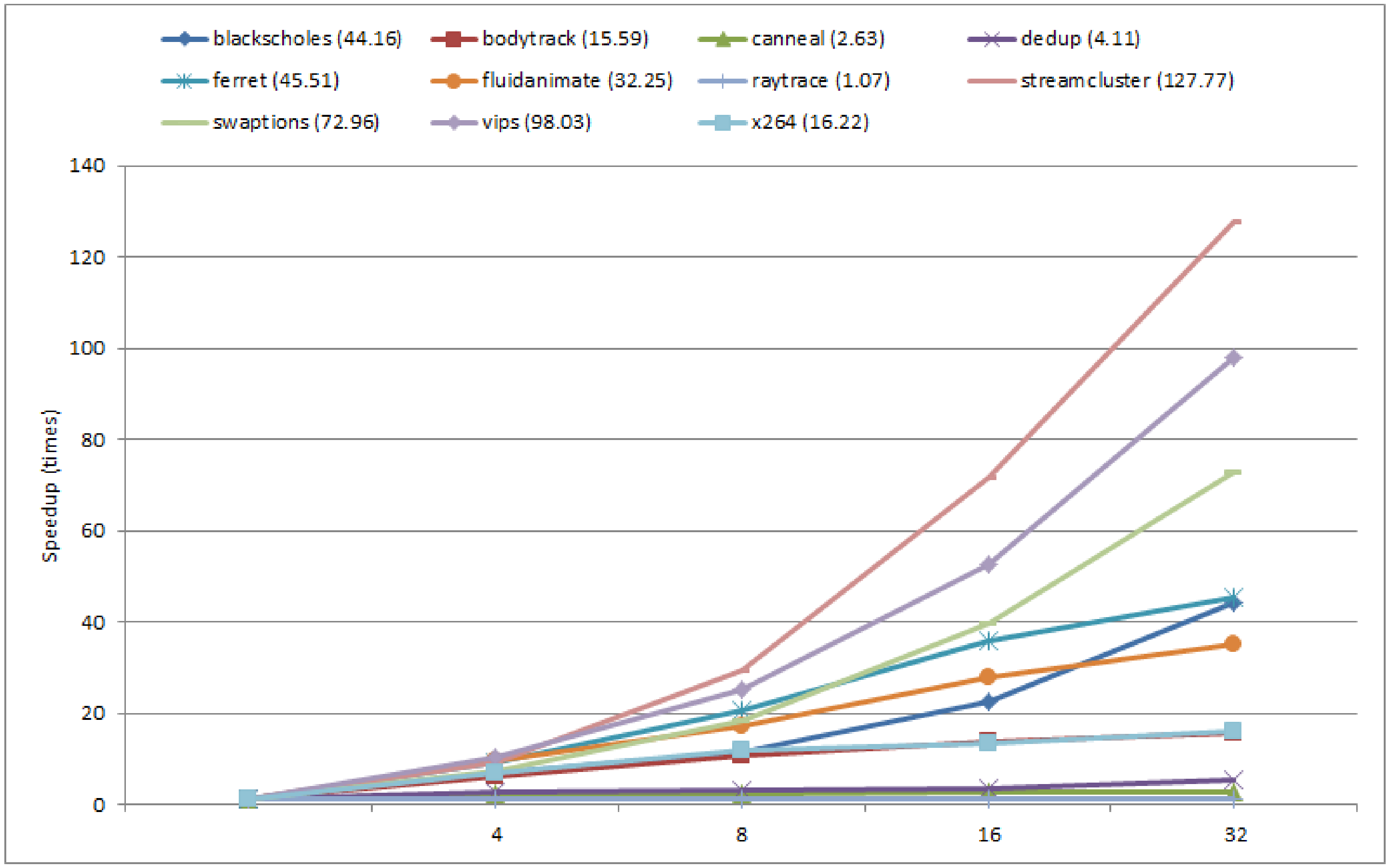


Fig. 4.7. The scalability of AntSM. The baseline is the execution time of benchmark with instrumentation in checking mode, executing with a single thread.

Table 4.2
The speedup of AntSM by different thread counts. The baseline is the execution time of benchmark with instrumentation in checking mode, executing with a single thread.

	4 Threads	8 Threads	16 Threads	32 Threads	Original Speedup
blackscholes	6.94	11.64	22.49	44.16	4.63
bodytrack	5.96	10.54	13.81	15.59	5.62
canneal	2.25	2.48	2.59	2.64	1.37
dedup	2.81	3.13	3.58	5.48	2.03
ferret	9.03	20.69	35.67	45.51	2.86
fluidanimate	9.44	17.35	28.03	35.25	4.02
raytrace	1.03	1.05	1.06	1.07	1.28
streamcluster	9.17	29.48	71.52	127.77	3.46
swaptions	7.15	18.39	39.80	72.96	7.98
vips	10.25	25.37	52.64	98.03	7.63
x264	6.96	11.92	13.27	16.22	5.71

Figure 4.8 shows the ratio of AntSM's speedup to the original speedup by the number of threads (i.e., the ratio, $y = \frac{SP_A}{SP_O}$, here, SP_A is the speedup of AntSM and SP_O is the speedup of the original benchmark without instrumentation). As shown in the figure, AntSM scales better than the original benchmark except for *raytrace* which shows a very low original speedup(1.28) in Table 4.2. This is because as the program runs with more threads, the task of monitoring is more distributed among the threads, therefore, the effect of distribution becomes bigger as the number of threads increases.

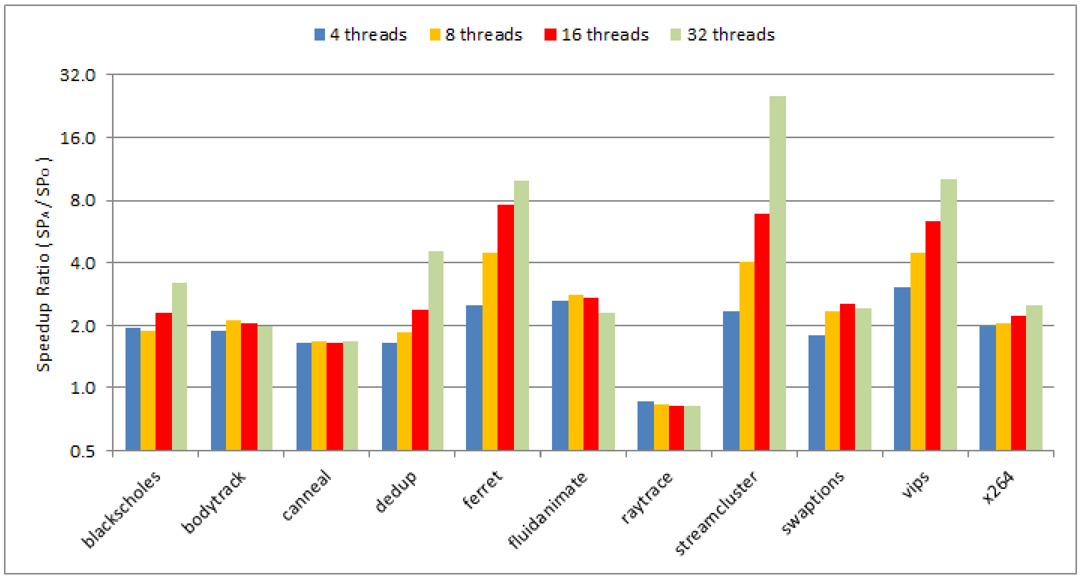


Fig. 4.8. The ratio of AntSM's speedup to the original speedup by the number of threads (4, 8, 16, and 32 threads). If the ratio, y, is greater than 1, AntSM scales better than the original PARSEC benchmark.

4.4.4 Accuracy of C-DIDUCE with AntSM

Figure 4.9 shows the accuracy measurements for C-DIDUCE in a checking mode run with "Replicated", "AntSM" and "Distributed" monitoring. We injected 5 to 15 bugs into each benchmark. Each bug is a form of *Value Mutation*, which changes an

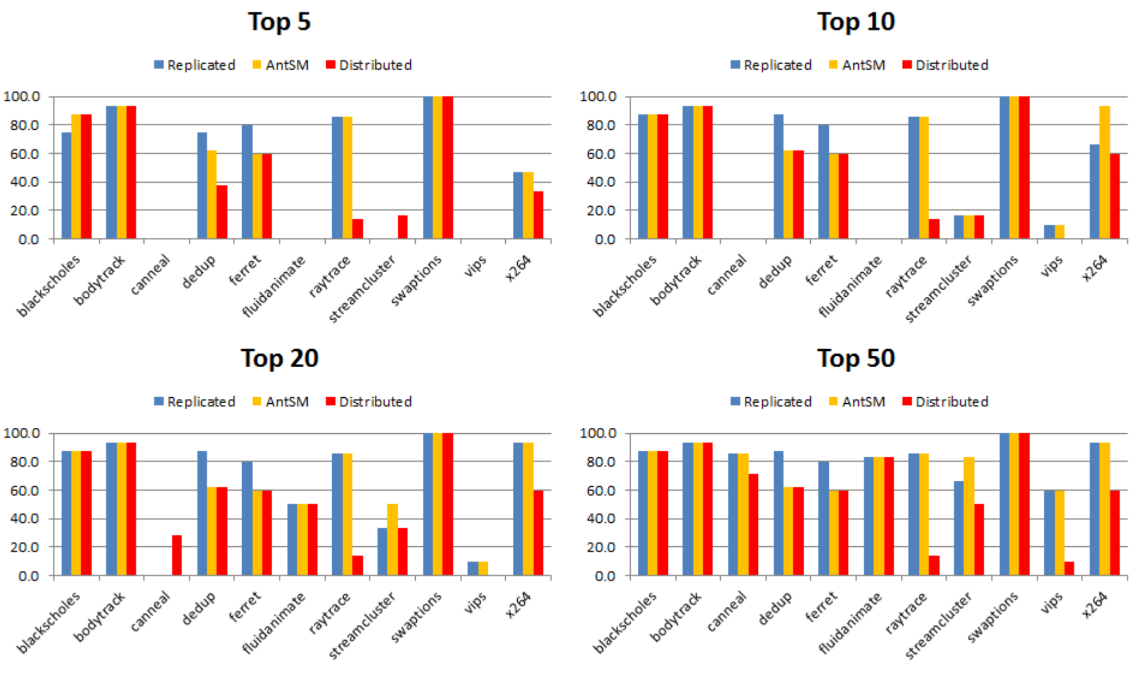


Fig. 4.9. The comparison of accuracy among *Replicated*, *AntSM*, and *Distributed*.

assignment like "`a = x`" into "`a = x + c`", where `c` is an integer constant. DIDUCE and C-DIDUCE rank anomalies as to how likely they are to be a bug, and we report the rates for bugs occurring in the top 5, 10, 20, or 50 ranked anomalies. `Top X` in Figure 4.9 means that only bugs ranked in the top X violations are considered to be successfully detected. Figure 4.9 shows that "AntSM" has accuracy similar to "Replicated" in most cases while providing much better performance. "AntSM" accuracy is equal to, or better than (5 cases) "Distributed" in all cases. In particular, "AntSM" has higher accuracy than "Distributed" for the two benchmarks with a large sequential portion (*raytrace* and *vips*) in Top 50 because the "AntSM" checking uses replicated monitoring in the sequential parts of the program while "Distributed" uses sampling.

Table 4.3
Comparison of average accuracy for Replicated, AntSM, and Distributed monitoring.

	Replicated	AntSM	Distributed
Top 5	50.5	48.7	40.2
Top 10	57.0	55.4	44.9
Top 20	65.5	62.9	53.6
Top 50	83.9	81.3	62.9

Note that sometimes the sampling schemes ("AntSM" and "Distributed") are ranked higher than "Replicated" as with *streamcluster* of Top 5, *raytrace* and *x264* of Top 10, *canneal* and *streamcluster* of Top 20, and *streamcluster* of Top 50. This is because the "Replicated" scheme performs more monitoring, and thus can see more violation data, which may lower the ranking of detected violations. The same reason holds between "AntSM" and "Distributed."

Table 4.3 shows the comparison of the average accuracy for the three different monitoring scheme by each ranking. This also shows "AntSM" monitoring has accuracy similar to "Replicated" monitoring, better than "Distributed" monitoring in average.

4.4.5 Discussion

In our C-DIDUCE case study and experiments, multi-threaded C-DIDUCE was implemented in a typical shared memory programming style, i.e., the shared information among threads was protected by synchronization. In Section 4.2.3, we discussed an alternate implementation using per-thread tables to avoid synchronization. In this section, we provide experimental results when using this alternate implementation that show its higher performance and memory use.

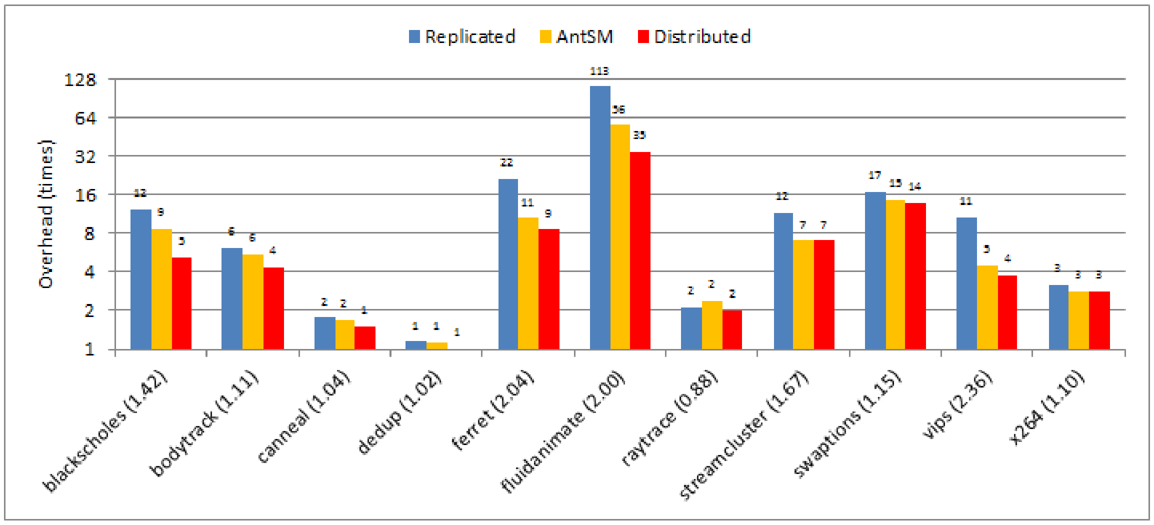


Fig. 4.10. Comparison of C-DIDUCE execution time overhead in checking mode when using separate table for each thread and merging the tables at the end of execution. The baseline is the execution time of the original benchmark with no instrumentation. Note that the vertical axis is on a log scale. The data label on each bar shows the overhead (times) rounded to the nearest one. The number next to each benchmark's name represents the overhead reduction from *Replicated* to *AntSM*.

Figure 4.10 compares the overhead when a separate table for each thread is used during the execution and merged at the end of execution with 8 threads. When compared with overheads from Figure 4.6, which uses synchronization for globally-accessed invariant data, Figure 4.10 shows significant overhead reduction by using separate tables for each thread in all three different monitoring schemes. This overhead reduction is the effect of removing synchronization at the cost of memory for each thread. As shown in Figure 4.10, AntSM's sampled monitoring scheme based on runtime thread counts further reduces the overhead. This shows the effect of sampled monitoring by exploiting parallelism of the programs even with additional overhead of checking the runtime thread counts and checking the current thread's turn of monitoring. To support this assertion, Table 4.4 shows the actual access counts of the C-DIDUCE debugging library. The ratio of "Replicated" to "AntSM" at the right-

Table 4.4
The runtime monitoring counts for different monitoring schemes. This shows how many times C-DIDUCE debugging libraries are actually invoked in checking mode with 8 threads.

	Replicated	*AntSM*	*Distributed*	$\frac{Replicated}{AntSM}$
blackscholes	2.84×10^8	4.26×10^7	3.83×10^7	6.67
bodytrack	4.96×10^8	1.74×10^8	4.24×10^7	2.85
canneal	2.48×10^8	2.02×10^8	6.54×10^7	1.23
dedup	2.48×10^8	1.41×10^8	6.76×10^7	1.76
ferret	2.05×10^9	3.66×10^8	3.27×10^8	5.59
fluidanimate	1.08×10^9	1.46×10^8	1.44×10^8	7.42
raytrace	6.69×10^8	6.63×10^8	7.54×10^7	1.01
streamcluster	4.98×10^8	1.12×10^8	1.08×10^8	4.45
swaptions	1.54×10^9	3.97×10^8	2.29×10^8	3.90
vips	7.83×10^8	1.18×10^8	9.99×10^7	6.62
x264	2.23×10^8	4.66×10^7	4.46×10^7	4.79

most column of Table 4.4 shows the ratio for the number of times that the C-DIDUCE invariant violation testing function is invoked between those two monitoring schemes. This shows how effectively the parallelism is exploited. As mentioned in Section 4.1, AntSM performs its sampling within code regions with the same root function that are executed in parallel and so "AntSM" sampling lies between "Replicated" and "Distributed." AntSM does "Distributed" monitoring only within similar regions.

In Figure 4.10, the *blackscholes*, *ferret*, *fluidanimate*, *streamcluster*, *vips* and *x264* are applications that show a larger overhead reduction than the other applications. From the access count in Table 4.4, we observed that these applications shows higher parallelism than the other applications, therefore, shows that AntSM exploits effectively the parallelism of applications on its overhead reduction. With low parallelism, it is possible for “AntSM” monitoring to show more overhead than “Replicated” as in the case of *raytrace* because of the cost of checking the runtime thread counts and introducing an additional branch to check the current thread’s turn of monitoring.

As shown from our experiments in Section 4.4 and discussion in this section, the Ant framework is effective with either using space-efficient approach or using a performance-centric approach at the cost of increased memory usage. Although the effect of removing synchronization in this section is high, separating debugging data and merging at the end of execution may not be always the best solution depending on the focus of the execution environment for targeting programs. However, AntSM shows its usefulness in either way of implementation as shown from the experimental results in Figure 4.6 and Figure 4.10.

We note that “AntSM” monitoring becomes closer to “Replicated” monitoring as the number of threads executing the program in parallel becomes small. As the number of threads increases, “AntSM” monitoring becomes close to “Distributed” monitoring because our AntSM framework adjusts the sampling rate based on the parallelism of the application by monitoring runtime thread counts. When all parallelism is from a single root function, “AntSM” and “Distributed” do identical monitoring. Our experimental results in this section and sections 4.4.2 and 4.4.4 show that this is an effective way of reducing overhead while maintaining the accuracy by exploiting the parallelism of an application.

5. CONCLUSIONS

We have presented the Ant framework for increasing the efficiency of sequential debugging techniques with parallel programs.

In AntDM, which is targeting distributed memory parallel programs such as MPI programs, our technique uses a static AR/NAR detection analysis and instrumentation strategy. We present a case study that extends to parallel programs with the C-DIDUCE debugging tool developed for sequential programs. More specifically, we have presented the design and implementation of parallel value invariant analysis and experimentally shown the validity of the parallel value invariant hypothesis and the effectiveness of C-DIDUCE on parallel programs.

AntSM, which targets shared memory parallel programs, such as Pthreads programs, the framework uses a combination of compile-time analysis and instrumentation, and runtime monitoring, to intelligently sample events of interest for these tools. We presented a case study of using AntSM with the C-DIDUCE debugging tool that was developed for sequential programs. Our techniques lead to significant performance improvements over a naive porting of these tools and much better accuracy than a less intelligently applied sampling. This work allows sequential bugging tools to be efficiently used to create more reliable and robust parallel programs.

We believe that the Ant framework will allow a broad range of debugging tools, whose monitoring can be sampled, to be efficiently used with parallel programs.

LIST OF REFERENCES

LIST OF REFERENCES

[1] "Software errors cost U.S. economy $59.5 billion annually," 2002. NIST News Release 2002-10.

[2] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering*, pp. 291–301, 2002.

[3] L. Fei and S. P. Midkiff, "Artemis: practical runtime monitoring of applications for execution anomalies," in *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, (New York, NY, USA), pp. 84–95, ACM Press, 2006.

[4] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "AccMon: Automatically detecting memory-related bugs via program counter-based invariants," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'04)*, 2004.

[5] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the ACM SIGPLAN 2005 conference on Programming Language Design and Implementation*, 2005.

[6] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pp. 141–154, 2003.

[7] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM Press, 2005.

[8] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the 22nd International Conference on Software Engineering*, pp. 449–458, 2000.

[9] J.-W. Lee, L. R. Bachega, S. P. Midkiff, and Y. Hu, "Ant: A debugging framework for MPI parallel programs," in *International Workshop on Languages and Compilers for Parallel Computing (LCPC'12)*, 2012.

[10] "The PARSEC Benchmark Suite." http://parsec.cs.princeton.edu.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th International Conference on Software engineering*, ICSE '94, (Los Alamitos, CA, USA), pp. 191–200, IEEE Computer Society Press, 1994.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” in *Proceedings of the 21st international conference on Software engineering*, ICSE ’99, (New York, NY, USA), pp. 213–224, ACM, 1999.

[13] “Totalview user guide.” last checked 9/28/2012.

[14] S. S. Lumetta and D. E. Culler, “The Mantis parallel debugger,” in *SPDT ’96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, (New York, NY, USA), pp. 118–126, ACM Press, 1996.

[15] S. Sistare, E. Dorenkamp, N. Nevin, and E. Loh, “MPI support in the Prism programming environment,” in *Supercomputing ’99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, p. 22, ACM Press, 1999.

[16] D. Stringhini, P. Navaux, and J. C. de Kergommeaux, “A selection mechanism to group processes in a parallel debugger,” in *In Proceedings 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’00)*, June 2000.

[17] D. Cheng and R. Hood, “A portable debugger for parallel and distributed programs,” in *Proceedings of Supercomputing ’94*, pp. 723–732, November 1994.

[18] R. Wismuller, M. Oberhubera, J. Krammera, and O. Hansenb, “Interactive debugging and performance analysis of massively parallel applications,” *Parallel Computing*, vol. 22, pp. 415–442, March 1996.

[19] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, “Problem diagnosis in large-scale computing environments,” in *SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 88, ACM, 2006.

[20] Q. Gao, F. Qin, and D. K. Panda, “DMTracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements,” in *SC ’07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ACM, 2007.

[21] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, “Stack trace analysis for large scale debugging,” *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 64, 2007.

[22] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit, “Lessons learned at 208k: towards debugging millions of cores,” in *SC ’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–9, IEEE Press, 2008.

[23] “Personal conversation.” Leonardo R. Bachega.

[24] S. ik Lee, T. A. Johnson, and R. Eigenmann, “Cetus - an extensible compiler infrastructure for source-to-source transformation,” in *Languages and Compilers for Parallel Computing, 16th Intl. Workshop, College Station, TX, USA, Revised Papers, volume 2958 of LNCS*, pp. 539–553, 2003.

[25] “The Cetus Project.” http://cetus.ecn.purdue.edu.

[26] H. Bae, D. Mustafa, J.-W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. P. Midkiff, “The cetus source-to-source compiler infrastructure: Overview and evaluation,” *International Journal of Parallel Programming*, vol. 41, no. 6, pp. 753–767, 2012.

[27] “NAS Parallel Benchmarks.” http://www.nas.nasa.gov/publications/npb.html.

[28] “The ASCI Purple Benchmark.” https://asc.llnl.gov.

[29] “SPEC MPI2007.” http://www.spec.org/mpi2007/.

[30] J.-W. Lee and S. P. Midkiff, “AntSM: Efficient debugging for shared memory parallel program,” in *International Workshop on Languages and Compilers for Parallel Computing (LCPC’13)*, 2013.

[31] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis and transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO’04)*, 2004.

[32] C. Lattner and V. Adve, “The LLVM compiler framework and infrastructure tutorial,” *Languages and Compilers for High Performance Computing Lecture Notes in Computer Science*, vol. 3602, pp. 15–16, 2005.

[33] “The LLVM Compiler Infrastructure.” http://llvm.org.

APPENDIX

A. EXTENDING VALUE INVARIANT DETECTION TO PARALLEL PROGRAMS

To adapt the value invariant detection (VID) to parallel programs, we extend the value invariants hypothesis to parallel programs, as follows. First, we observe that a large part of the computation performed in a parallel program across tasks is identical regardless of the number of processes used to execute the program. Intuitively, this is true because given the same input the parallel and sequential versions of the program will return the same answers, disregarding numerical stability and round-off effects. Based on this observation, we allow training runs to use a smaller input on a small number of processes, and detection runs using larger inputs on a large number of processes. While significantly lowering the cost of training runs, it creates another problem: How do we form the approximations of the sets of invariant values that will be used by each of the P' processes on the detection run from the approximations formed by the P processes on the training runs? Consider the expression for the mask in $I = \langle M_t, V\rangle$ defined above. When the expression is monitored at time t, the mask value will be

$$M_t = \bigwedge_{i=1}^{t} \overline{(w_i \otimes V)} \wedge M_0 = \bigwedge_{i=1}^{t} \overline{(w_i \otimes V)},$$

since $M_0 \equiv 1$. It follows from the DeMorgan's laws and the definition of $\otimes$:

$$\begin{aligned}
M_t &= \bigwedge_{i=1}^{t} \overline{(w_i \otimes V)} = \bigwedge_{i=1}^{t} \overline{(\overline{w_i} \wedge V) \vee (w_i \wedge \overline{V})} \\
&= \bigwedge_{i=1}^{t} ((w_i \vee \overline{V}) \wedge (\overline{w_i} \vee V)) = \bigwedge_{i=1}^{t} (w_i \vee \overline{V}) \wedge \bigwedge_{i=1}^{t} (\overline{w_i} \vee V) \\
&= (\overline{V} \vee \bigwedge_{i=1}^{t} w_i) \wedge (V \vee \bigwedge_{i=1}^{t} \overline{w_i})
\end{aligned}$$

Now, consider the invariant sets $I_k = \langle M_{k,t}, V_k \rangle$ and $I_j = \langle M_{j,t}, V_j \rangle$ of the same variable reference (i.e., the same program point) built in two different processes (p_k and p_j). We can merge both to form a single invariant set $I' = \langle M'_t, V_i \rangle$, with

$$M'_t = \left[\overline{V_k} \vee (\bigwedge_{i=1}^{t} w_{k,i} \wedge V_j \wedge \bigwedge_{i=1}^{t} w_{j,i}) \right] \wedge \left[V_k \vee (\bigwedge_{i=1}^{t} \overline{w_{k,i}} \wedge \overline{V_j} \wedge \bigwedge_{i=1}^{t} \overline{w_{j,i}}) \right],$$

and V_i equal to either V_k or V_j.

VITA

VITA

Jae-Woo Lee was born in Daegu, South Korea in June 1974. He received his B.E. in Computer Engineering from Dankook University, Seoul, South Korea, in 1997, and M.S. in Computer Science from University of Southern California, Los Angeles, CA, in 1999. From January 2000 to July 2007, he worked at Samsung SDS, Seoul, South Korea, as a software engineer and was involved in many projects within a broad area of technologies.

In August 2007, Jae-Woo joined the Ph.D. program in the School of Electrical and Computer Engineering at Purdue University, West Lafayette, IN. He began research work with Prof. Midkiff in Summer 2008. Jae-Woo interned at Nvidia Corporation, Santa Clara, CA, in Summer 2012 as a compiler software engineer for three months.

Jae-Woo's research interests lie broadly in optimizing compilers and their applications for the perfomance enhancement. The focus of Jae-Woo's Ph.D. was on the overhead reduction of sequential debugging tools with parallel programs. He was also involved in the Cetus project and developed several features in the Cetus compiler.

Jae-Woo will pursue his career as a compiler software engineer at Intel Corporation from January 2014 and wants to contribute to people in the world with his talent and best efforts.