EFFICIENT DYNAMIC PROGRAM MONITORING ON MULTI-CORE PLATFORMS

A DISSERTATION

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

GUOJIN HE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

ANTONIA ZHAI, ADVISOR

May, 2012

Acknowledgements

First and foremost, I would cordially thank my advisor, Professor Antonia Zhai, for her great help, enduring patience, and invaluable advices throughout my entire doctoral program.

I also say thank you to Professor Mats Heimdahl, Professor David J. Lilja, and Dr. Michael Whalen for their generous help, encouragement, and suggestions during my thesis research. I also thank them for serving as members of my examination committee. Moreover, I would like to thank Professor Pen-Chung Yew and Professors Wei-Chung Hsu for their encouragement and help during my Ph.D studies and studies and for providing advice about my career.

I also appreciate my fellow co-workers: Jinpyo Kim, Shengyue Wang, Jin Lin, Venkatesan Packirisamy, Yangchun Luo, Vineeth Mekkat, Ragavendra Natarajan, Jieming Yin, and Anup P Holey for their friendship and technical help.

I am also indebted to my personal friends Professor Ziguo Zhong, and Dan Wang for their enormous help and encouragement in life and in research during my Ph.D studies.

Furthermore, I want to thank the people I have worked with during my internship and current employment, along with friends in China. Their creative minds and ideas inspired me in great measure.

Lastly, I would say thanks to my parents, my sister, and her family for their unconditional love and support.

Abstract

Software security and reliability have become increasingly important in the modern world. An effective approach to enforcing software security and reliability is to monitor a program's execution at run time. However, instrumentation-based implementation of a dynamic program monitor on single-core systems suffers significant performance overhead. As multi-core architecture becomes more mainstream, implementing efficient dynamic program monitoring by assigning monitoring activities onto separate processor cores and thus reducing performance overhead becomes not only a feasible but an appealing way to enforce software security and reliability. To achieve efficient and flexible multi-core based dynamic program monitoring, however, three challenging issues must be carefully considered and adequately addressed: the hardware support, the monitoring model, and the parallelization of monitoring tasks.

This dissertation proposes novel solutions to these challenging problems. The hardware support proposed in this dissertation, which is referred to as extraction logic, selectively extracts execution information from the monitored program and forwards it to a monitor running on a separate CPU core. The extraction logic is generic and configurable by the monitor so that it can support a large spectrum of monitoring tasks. Based on this generic hardware support, this dissertation proposes a novel monitoring model, referred to as the distill-based monitor model. Monitors in this execution model is generated by special compiler supports. The distill-based

monitor model is based on the observation that a monitor needs only partial information from the monitored execution and that of this needed information, some can be easily computed by the monitor from other information that has already been communicated. We implemented a code generator and optimization techniques to decide which set of information to forward and which set to compute so as to minimize the total execution time of the monitor. This compiler support can optimize a variety of monitors with diverse monitoring requirements, taking as input the control flow graph of the monitored program and the set of monitoring requirements.

To parallelize monitoring tasks, this dissertation proposes a novel parallelization paradigm built on General-purpose Computing on Graphics Processing Unit (GPGPU) architecture. In the following chapters, we first propose a generic, purely software-based GPGPU monitor framework that is flexible enough to support parallelization of various kinds of monitoring tasks. Furthermore, we propose software-based optimization techniques built on this framework that effectively take advantage of various characteristics of monitoring tasks such as taint-propagation and memory-bug detection, and thus achieve significant performance improvement.

This dissertation reports the performance improvement achieved by the proposed monitoring model and parallelization paradigm. Relative to the performance of traditional instrumentation-based monitor for taint-propagation and memory-bug-detection, the proposed compiler support is able to bring down performance overhead by 3.7 times and 2.2 times for SPEC2006INT benchmarks. The proposed GPGPU-based monitor with optimization even achieves more for memory-bug detection, reducing performance overhead by 5.2 times.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As computer systems have become necessities in virtually all important aspects of human life today, computer software plays an increasingly important role in the safety and welfare of modern societies. Consequently, the security and reliability of software are extremely critical issues in the modern world. In modern computer systems, there are two major approaches to enforcing software security and reliability: statically examining source code [1, 2, 4, 3] and dynamically monitoring program execution [5, 11, 12, 10, 6, 7]. The latter is becoming increasingly important and common due to several trends that make static approaches more expensive and less effective: first, rapid increases in the complexity of modern software systems; second, rapid increases in data size, and last but not the least, fast-changing and unpredictable environments where software programs are deployed.

Efficiency is the key issue for successful dynamic program monitoring systems. Since dynamic program monitoring systems must run side by side with monitored execution on the same hardware platform, the performance overhead of monitoring is added to the performance of monitored program execution. In the uniprocessor era, this overhead could be prohibitively high and thus prevented dynamic program

monitors from being widely deployed. Recently, however, the emergence of multi-core architecture facilitates future dynamic program monitoring as it enables more efficient paradigms. This dissertation reports our research efforts on the design, implementation and evaluation of an efficient dynamic program monitoring system built on multi-core platforms.

1.1 Background of Dynamic Program Monitoring and Its Challenges

All dynamic program monitoring systems must implement the following key functionalities:

1. extracting information about the monitored program execution.
2. maintaining meta-data a.k.a shadow data to keep track of the status of monitored execution.

Based on the differences in implementation, dynamic monitoring systems can be classified into three categories: instrumentation-based monitoring systems [5, 11, 12, 10, 6, 7]; specialized-hardware-based monitoring systems [26, 27, 28, 29, 31, 32, 33, 34] and multi-core-based monitoring systems [53, 54].

The research reported in this dissertation falls into the multi-core-based category. Multi-core-based monitoring addresses challenges faced by the other two categories of dynamic program monitoring. This section briefly describes and compares basic ideas and key features of three different approaches. Details of multi-core-based monitoring proposed by our research is left to later chapters of this dissertation.

Instrumentation-based dynamic program monitoring tools use monitoring code to instrument the binary or the original source code of monitored program. The monitoring code tracks changes in the status of the monitored program and verifies

compliance with security or reliability rules that are defined by the monitoring requirements. Many well-known instrumentation-based program monitoring tools allow instrumentation of the user program at a fine-grained level [5, 12, 10, 6, 7]. This way, they can support popular monitoring requirements such as taint-propagation and memory-bug-detection that require detailed run-time information such as referred memory addresses, sources and destinations of data movements, as well as targets of jump instructions.

The main disadvantage of instrumentation-based dynamic program monitoring tools is that their performance overhead is significant. Instruction-grained monitoring tasks such as taint-propagation or memory-bug-detection can slow monitored programs by 10-50 times [6, 7]. This disadvantage prevents instrumentation-based monitoring tools from being employed for wider applications. The root cause of this high performance overhead is that instrumentation code competes with its monitored execution for limited hardware resources during execution/monitoring time.

As an alternative to instrumentation-based approaches, researchers have proposed using specialized hardware support to accelerate dynamic program monitoring. [26, 27, 28, 29, 31, 32, 33, 34]. Dynamic program monitoring systems with these specialized hardware supports do not instrument the monitored program. Instead, the specialized hardware supports take over the tasks of information extraction, meta-data maintaining and status checking. Depending on the monitoring requirements, these hardware supports extend the existing micro-architecture design in different ways: some add additional function units into the processor pipeline [26, 27], some augment the memory hierarchy [29, 32, 33], and others introduce new type of storage to accommodate meta-data [28, 32, 33]. Compared to instrumentation-based approaches, these specialized hardware supports significantly reduce the performance overhead of dynamic program monitoring.

The disadvantage of dynamic monitoring with specialized hardware support is that its application is limited to a specific type of monitoring activity. In other words, this approach is inflexible. Fundamentally, different monitoring tasks require different monitoring activities, and thus specialized hardware support must be customized to the characteristics of each monitoring activity to be effective.

1.2 Multi-core-based Dynamic Program Monitoring

The research reported by this dissertation is different from previous proposals in that it is built on multi-core platforms to keep the monitoring system both flexible and efficient. It is different from other multi-core-based monitoring systems [34, 39, 49], as other systems mainly rely on hardware approach to implement and accelerate monitoring. The research in this dissertation uses both hardware approach and software approach to implement efficient multi-core-based monitoring. As this will be elaborated upon later, in this section we focus on introducing the common characteristics of multi-core based dynamic program monitoring.

In typical multi-core based dynamic program monitoring systems, two software entities simultaneously execute on separate cores: the program to be monitored and the program that monitors. In this dissertation, they are referred to as the monitored program and the monitor, respectively. During a monitored execution, the execution information of the monitored program such as the addresses of accessed memory, updated values of registers, and branch direction are collected by a dedicated hardware circuit, referred to as *extraction logic* in this dissertation. The extraction logic forwards every piece of collected information to the monitor on another core in real time through a software queue implemented in the shared memory. The monitor obtains the forwarded information from the queue and uses it to track the progress of the

monitored execution and to check against security or reliability rules derived from the monitoring requirements. From the perspective of the shared memory, the monitored program and its monitor are the producer and consumers of the software queue. In this dissertation, the queue is referred to as the communication queue.

To make a multi-core based dynamic program monitoring system flexible, hardware support must be able to supply information needed by a large spectrum of monitoring requirements in real time. This issue is addressed by all research efforts on multi-core based program monitoring including ours [34, 38, 54].

The performance overhead in multi-core-based monitoring systems is caused by two types of stall in the monitored program. First, when the queue becomes full, the monitored program must stall to prevent information in the queue from being overwritten. Secondly, before doing a system call, the monitored program must stall to make sure all queue entries are verified by the monitor. This stall guarantees that the OS kernel is adequately protected by the monitor. In most heavy-load monitoring activities, the first type of stalls occurs far more frequently than the other, and thus is the predominant cause of performance overhead.

The research reported by this dissertation also diverges from other research proposals for multi-core-based program monitoring [34, 38] in its approach to reducing performance overhead caused by frequent queue stalls. While other proposals resort to introducing specialized hardware support to reduce the performance overhead, this dissertation proposes a novel monitor model that reduces communication between the monitored program and the monitor to improve performance. Furthermore, this dissertation reports our efforts on leveraging the emerging parallel architecture General Purpose Graphics Processing Units(GPGPU) to improve the performance of dynamic program monitoring.

1.3 Major Research Efforts and Results in the Dissertation

The goal of this dissertation is to present following research efforts and results:

- The proposal of a generic hardware support that enables efficient and flexible dynamic program monitoring on multi-core platforms.
- The proposal of an efficient multi-core based monitor model: the distill-based monitor model, and the compiler support that implements and optimizes it.
- The proposal of using General Purpose Graphical Processing Units(GPGPU) to parallelize dynamic program monitoring, and the software framework and optimization techniques that implements the proposed idea.
- A comprehensive performance evaluation of proposed technologies on a detailed simulation platform. Results of the performance evaluation show that the proposed technologies significantly reduce the performance overhead of dynamic program monitoring for generic monitoring requirements.

1.4 Organization of the Manuscript

The rest of this dissertation is organized as follows. Chapter 2 presents the details of the generic hardware support used to achieve flexible monitoring on multi-core platforms. Chapter 3 presents two different types of monitor models, referred to as the dispatch-based model and the distill-based model on multi-core based dynamic program monitoring systems, emphasizing the more efficient of the two, the distill-based model. Chapter 4 describes the novel compiler support that implements code generation and the optimization of the distill-based model. Chapter 5 illustrates the design and implementation of parallel dynamic program monitoring with GPGPU

architecture. Chapter 6 lists previous research related to the scope of this dissertation, and Chapter 7 concludes the dissertation.

Chapter 2

Hardware Support for Dynamic Program Monitoring on Multi-core Platforms

This chapter presents the generic hardware support for the multi-core based dynamic program monitoring in the research reported by this dissertation. This hardware support is referred to as *extraction logic*. The chapter begins with an overview of the extraction logic, illustrating its functionality and its two different work modes. It then dedicates two sections to giving a detailed description of the hardware components that implement the two modes of the extraction logic, demonstrating how they accelerate dynamic program monitoring for a large spectrum of monitoring tasks.

2.1 Overview of Hardware Support

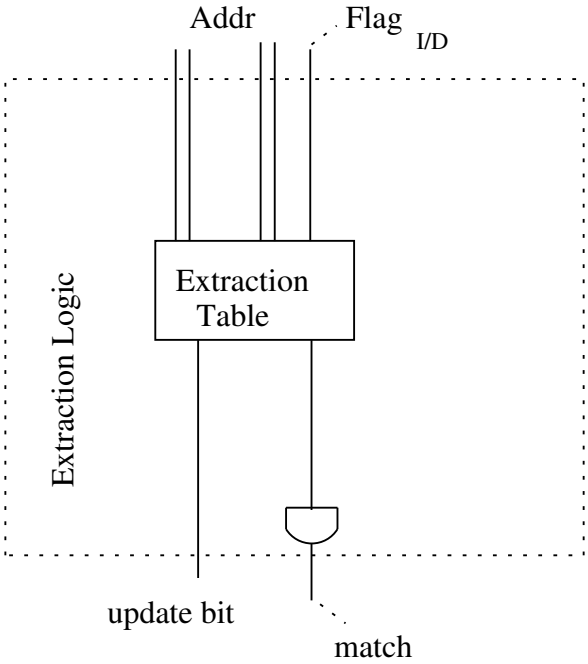
The dynamic monitoring system proposed in this dissertation uses both hardware support and software support to implement multi-core-based monitoring. In this system, the hardware support is referred to as extraction logic, which selectively extracts

information from the monitored process and sends it to the monitor running on a separate core. In the on-chip multi-core system that runs the dynamic monitoring, every core is augmented with this extraction logic so that the monitored process is not bounded to a specific core. For each core, its extraction logic is enabled only when it runs a monitored process.

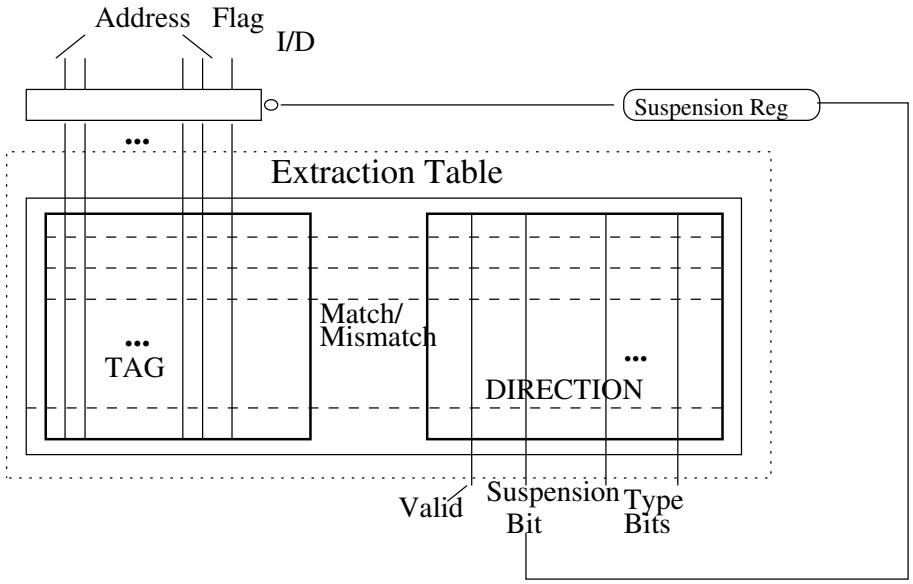
The key feature of extraction logic is its ability to selectively forward execution information for the monitored process. In the context of this dissertation, the dynamic execution information includes instruction-related values such as referred memory addresses, updated destination register values, and instruction addresses(PC). These values are available at the commit stage of each dynamic instruction. Extraction logic obtains this information from the processor pipeline's Reorder Buffer (ROB) and Load-Store Queue(LSQ) where this information is stored when an instruction commits.

In extraction logic, selecting dynamic execution information is essentially extracting values generated when the instructions of interest are executed. For example, the extraction logic can be instructed to extract referred memory addresses for a specified read instruction and ignore all other instructions.

There are two different approaches to direct extraction logic what instructions should have their information extracted: one is driven by a table with a list of instructions and the other by per-instruction flag bit, referred to as the forward bit that indicates whether it is of interest to the monitor. To support the two approaches, extraction logic is configured to work in two different modes. The following two sections illustrate the components of extraction logic that support the two approaches.



(a) Inside the extraction logic: Table-Driven Mode.



(b) Extraction table

Figure 2.1: Extraction Logic in Table-driven Mode

2.2 Table-Driven Extraction Logic

The key component of extraction logic in the table-driven mode is the extraction table, as shown in Figure 2.1(a). Inputs to the extraction table are the PC of the committing instruction or the referred address for committing memory instruction, along with a one-bit I/D flag. For a given input, if the I/D flag is set, it means that the rest of the input is the instruction address of a committing instruction. Otherwise, the rest of the input is the memory address of data referred by a committing memory instruction. Outputs of the extraction table include a one-bit valid signal and a one-bit suspension signal. The valid signal indicates whether the input address has a match on the table; the suspension signal indicates whether the committing instruction corresponds to a monitored event that should suspends the use of the extraction table. We describe the concept of suspending the extraction table later in this section.

The structure of the extraction table is shown in Figure 2.1(b). The extraction table consists of two components. The TAG component and the DIRECTION component. the TAG component is used to specify whether an input address is of interest, and the DIRECTION component stores and sends output bits for all instructions that the TAG component found of interest to the monitor.

The TAG component is implemented with content-addressable-memory (CAM) [58]. Each entry of the CAM corresponds to an address (could be a PC or data address) and a one-bit I/D flag. The DIRECTION component of the extraction table is implemented as a small cache. Each word line of this cache is driven by the corresponding output from the CAM. When an instruction commits, the PC of this instruction is sent to the CAM and the I/D flag is set to 1; if a block in the CAM matches the input, the corresponding word line is set and DIRECTION bits are retrieved from the small cache. If the instruction is a memory instruction, it goes through the same process except the input address to the table is the referred memory address and

the I/D flag is set to D. By checking the valid bit output from the DIRECTION component, the extraction logic is able to determine whether to forward the result of the committing instruction. To support simultaneous look-ups of the table, this structure can be extended with multiple ports.

Since the size of the extraction table is limited and many monitor activities require monitoring contiguous memory addresses, extraction logic leverages the property of ternary CAM [58] to accommodate the address range. In a ternary CAM, a storage cell represents "0","1" or "X" indicating "dont care" [58]. For example, to monitor all access to the elements of an array located between 0x8000a000 and 0x8000afff, only one entry with the address of 0x8000aXXX needs to be entered to the table.

For some segments of execution, the monitor software may require the extraction logic to forward all instructions and thus bypass the extraction table. For example, when the software is updating the extraction logic by loading a new set of interested instruction or memory addresses, the extraction table becomes temporarily inconsistent and its use should be prohibited until the update is done. To implement bypassing of the extraction table, Ex-Mon uses a one-bit suspension register to control the use of the extraction table, and the extraction table is bypassed when the suspension register is set. The mechanism of updating the extraction table is the following: when the monitoring system initializes the extraction logic, the suspension register is initially clear. The monitor software then sets the suspension bit of every extraction table entry that might trigger a table update. During the monitored execution time, each time an extraction table entry is found to match the TAG component, its suspension bit is copied by the DIRECTION component to the suspension register. If a committing instruction matches an entry that has the suspension bit as "1", the extraction logic notifies the monitor software that the monitor should start updating the extraction table. Moreover, from the next committing instruction, the extraction logic bypasses the extraction table and forwards every instruction's

PC and data memory address to the monitor until the monitor resets the suspension register at the end of the update.

The table-driven mode is not efficient when there is a large number of instructions that are of interest to the monitor. In that case, the monitoring system ends up suspending the extraction table so often so that extraction table becomes less efficient. To overcome this disadvantage, we have designed a forward bit mode of extraction logic, as a supplementary or alternative way to select dynamic instruction information. The following section introduces the hardware components used for that mode.

2.3 Forward-Bit-Based Extraction Logic

When the extraction logic operates in forward-bit-based mode, it assumes that the binary of the monitored program is augmented with an annotation. The annotation of the monitored program is an array of forward bits. Each bit corresponds to an instruction of the monitored program, indicating whether the result generated by the instruction should be extracted and forwarded. The extraction logic selects which instructions' results to forward based on their forward bit value.

Two hardware components of the extraction logic are essential to implement the forward bit mode of extraction logic: (1) the fetching component that fetches the forward bits from memory and matches these bits with the corresponding instructions; and (2) the forwarding component that extracts and forwards the results of selected instructions whose forward bits are set. The following two subsections introduce the fetch component and the forward component in detail.

2.3.1 Fetching Component

The fetching component is integrated into the fetching stage of the CPU pipeline. It fetches forward bits of monitored execution from user memory and copies forward

bit of each instruction into the pipeline at the time when the instruction is fetched into the pipeline. Forward bits are generated by the monitoring compiler as a special annotation section in the binary of the monitored program. Every instruction in the code section of the binary of the monitored program has its own corresponding forward bit in the annotation section, indicating whether the instruction should have its result forwarded. When a monitored process is created, the content of the annotation section is mapped into the memory space of the monitored process. As the monitored process runs, the fetching component fetches the forward bit for each instruction that is fetched into the pipeline. Forward bits fetched by the fetching logic are later read by the forwarding component to decide whether the instruction should have its result written into the communication queue on the shared memory upon its commitment.

The key to the fetching component's reading forward bits is to calculate the address of the forward bit for every fetched instruction. To achieve this, the fetching component uses a special register, referred to as the Annotation Base Register (ABR), to keep the physical address of the base of annotation section in the memory space of the monitored process. ABR is initialized with that address when a monitored process is created by the monitoring system, and synchronized with any change in the annotation bases physical address.

When an instruction is fetched, its address is used as the input to a simple mapping function implemented by the hardware to calculate the offset of its corresponding forward bit. The fetching component adds this offset with its ABR value to compute the address of the byte in the annotation that contains the forward bit. Several lower bits of the instruction address are used to select the exact forward bit from the byte of forward bits fetched from annotation .This mechanism of calculating the address of meta-data is similar to that presented in MemTracker [32] with the exception that we calculate the meta-data address for instructions rather than for memory locations. Figure 2.2(a) shows the fetch logic that is implemented in SPARC architecture where

each instruction is 32-bit long. The example instruction has the offset 0x1a004 to the start of the code section, and its forward bit is the bit 1 of the byte fetched from the address from the physical address 0xf0000d00.

Some instructions executed by a monitored process, such as ones in dynamically loaded library codes, do not have forward bits because they do not appear in the monitored program's binary. These instructions are either all ignored or all forwarded, depending on monitoring requirements. If all of them ought to be forwarded, the extraction logic has to switch to the table-driven mode because their instruction body has to be sent to the monitor for interpretation. If they are to be ignored and extraction logic is in the forward bit mode, thee fetching component always assumes that their forward bits are all clear and does not calculate their address. The OS loader is changed to notify the extraction logic when the library code is loaded and unloaded.

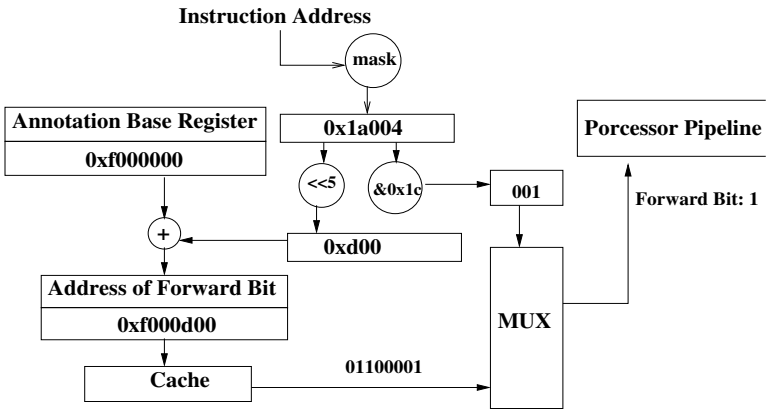
Forward bit fetched by the fetching component are kept in the pipeline until the corresponding instruction retires. In order to keep forward bits in the pipeline, every entry of the reorder buffer (ROB) in the pipeline is augmented with a forward bit flag. The fetching component copies the forward bit of a fetched instruction to that flag when the instruction is inserted into ROB. For architectures such as X86 that translates an instruction into multiple micro operations in ROB, all the micro operations have the same forward bit as the instruction's forward bit.

2.3.2 Forwarding Component

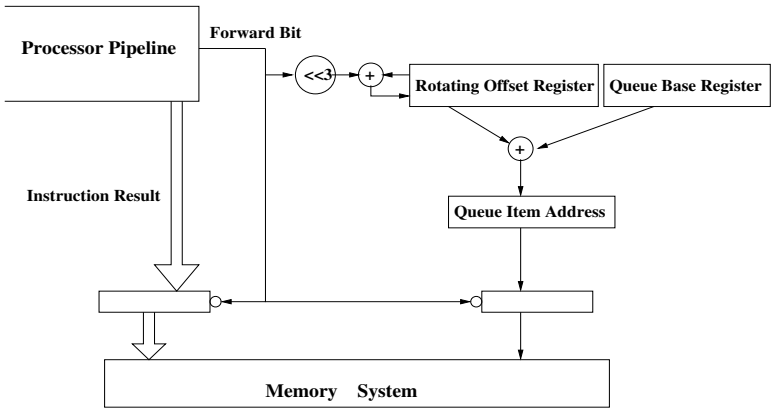
The forwarding component is attached to the commit stage of a CPU pipeline. It uses forward bits fetched by the fetching component to decide whether the result or pertained memory address of an instruction should be forwarded to the monitoring running on another core through the communication queue. The communication queue is part of the shared memory between the monitor process and the monitored

process.

The key to writing the results or referred memory addresses of instructions to the communication queue is calculating the address of the queue entry to write. This is achieved by the forwarding component using two special registers: the queue base register (QBR) and the rotating offset register (RQR). QBR keeps the base physical address of the communication queue and RQR is a saturated counter that indicates the offset of the current available entry to write and is incremented after each forwarding. QBR and RQR are initialized at start-up time. The entries in the communication queue are synchronized at the entry level using full/empty bits. Figure 2.2(b) shows the implementation of the extraction logic in which each queue entry is 8 bytes long.



(a) Fetching Component: For each instruction, the address of its forward bit is calculated using the Annotation-Base Register(ABR) and the offset of the instruction in the code section. The instruction offset is calculated by masking the instruction address. In this example, each instruction is 32 bits in size, and thus the offset of the forward bit is calculated by left shifting the instruction offset by 5, and the highest 3 bits of the lowest 5 bits (32-bit long instruction is 4-byte aligned) of the offset are used to select the forward bit from a byte of forward bits.



(b) Forwarding Component: The forward bit associated with each instruction determines whether the result of this instruction should be forwarded through the communication queue. The address of the queue entry is calculated from the queue-base register (QBR) and the rotating-offset register (RQR). RQR is incremented after each forwarding.

Figure 2.2: Extraction Logic in Forward Bit Mode

Chapter 3

Execution Models of Multi-core-based Program Monitor

This chapter presents two main execution models for implementing the multi-core-based dynamic program monitor: the dispatch-based model and the distill-based model. The program monitors implemented using these two models are significantly different from each other in the utilization of hardware support, the use of extracted information, and the performance. We begin this chapter with an example to give a straightforward illustration of dispatch-based and distill-based models, and show how both of them differ from the instrumentation-based monitor that is widely used on uniprocessor platforms. Then, we describe details of the basic execution model: the dispatch-based model and the more advanced distill-based model. Finally, we conclude the chapter by providing a qualitative analysis of the performance overhead associated with the two models and reasons for why the distill-based model is more efficient than the dispatch-based model.

```
foo(int* p, int y) {
  int* q;
  int i,j,k;
  i = 0;
  do {
    q = p + i;
    VERIFY_LOAD(q);
    k = *q;
    j = k*k + i;
    VERIFY_STORE(q);
    *q = j + y;
    i++;
  } while(i<1024)
}
```

(a) Instrumentation-based monitor: calls to the verifying routines are inserted in the original program.

```
do {
  msg = receive();
  switch(msg.type) {
    ...
    case LOAD:
      VERIFY_LOAD(msg.body);
      break;
    case STORE:
      VERIFY_STORE(msg.body);
      break;
    default :
      break;
  }
} while(...)
```

(b) Dispatch-based monitor: dispatcher invokes the appropriate verification routine based on the forwarded record.

```
foo(int p, int y) {
  int q;
  i = 0;
  do {
    q = receive();
    VERIFY_LOAD(q);
    VERIFY_STORE(q);
    i = receive();
  } while(i<1024)
}
```

(c) Distill-based monitor (unoptimized): the monitor executes a distilled version of the program.

```
foo(int* p, int y) {
  int* q;
  p = receive();
  i = 0;
  do {
    q = p + i;
    VERIFY_LOAD(q);
    VERIFY_STORE(q);
    i++;
  } while(i<1024)
}
```

(d) Distill-based monitor (optimized): the monitor executes an *optimized* distilled version of the program.

Figure 3.1: Four different Implementations of the Memory-bug Detection Monitor.

3.1 Example of Monitor Models

In this section, we use an example to show the differences between the distill-based monitor model, the dispatch-based monitor on multi-core, and the instrumentation-based monitor on uniprocessor.

Figure 3.1(a) shows the sample program with instrumentation for monitoring. The monitoring requirement is to examine all memory access through pointers to ensure that there is no access to unallocated memory and uninitialized memory for load operation. Therefore, instrumentation invokes monitoring functions VERIFY_STORE and VERIFY_LOAD immediately before the examined store and load.

Figure 3.1(b) shows the dispatch-based monitor for the sample program that performs the same check as the instrumentation-based monitor does in Figure 3.1(a). It features a dispatch routine that continuously reads the execution records forwarded from the monitored code, and dispatches them to monitoring functions.

Figure 3.1(c) shows the distill-based monitor for the sample program that performs the same monitoring work. The monitor comprises of invocations to the function *receive* that fetches the value of q and i and the monitoring functions VERIFY_STORE and VERIFY_LOAD with values of q as argument. The control flow statement **while**(i<1024), upon which the invocations to monitoring functions depend, is preserved.

In this example, there is a notable performance advantage of multi-core-based monitor models over single-core-based instrumentation-based monitor. Multi-core-based monitors, both dispatch-based and distill-based, can execute monitoring code in parallel to the monitored execution. This fact indicates that there is no competition with monitored execution for hardware resources in a core, such as the pipeline, physical registers in the latter two monitors.

3.2 Dispatch-Based Monitor

A dispatch-based monitor consists of three essential elements: (i) a dispatching routine that retrieves and decodes messages and dispatches them to appropriate monitoring functions that perform monitoring tasks; (ii) a set of monitoring functions; and (iii) routines to initiate and update the extraction table so that hardware support can capture events that are related to the monitored execution. In this section, we describe these three elements in detail.

```
data declarations:
typedef struct message_struct
    {Type, Address, Value, I/D flag }
message_struct msg;
List of events for initializing and
updating the extraction table.
message_struct Msg_List[];

algorithm DispatchRoutine:
InitializeExtractionTable(Msg_List);
while (true)
 msg = ReadQueue();
 switch (msg.Type))
  case ALLOC:
   verify_alloc(msg);
   break;
  case FREE:
   verity_free(msg);
   break;
  case LOAD:
   verity_load(msg);
   break;
  ...
  case UPDATE:
   UpdateExtractionTable(msg);
   break;
  ...
  case EXIT:
   monitor_exit(msg);
   break;
 end switch
end while
end algorithm
```

(a) Dispatching routine.

```
verify_free(message)
 begin
  if  (AllocatedBlockLookup(message.Address))
   AllocatedBlockRemove(message.Address);
   return;
  else
   ReportIllegalFree(message);
  end if
 end
```

(b) a monitoring function to verify call to *FREE*.

Figure 3.2: Dispatch-based Monitor for Memory-bug Detection

3.2.1 Dispatching Routine

The dispatching routine is usually structured as a while loop whose body is mainly a switch-case structure. Each iteration of the loop decodes a message fetched from the communication queue, and processes it by dispatching it to the appropriate monitoring function. Messages that are not relevant to monitoring purpose are dispatched to the default case. When a special message EXIT is processed, that corresponds to the termination of the monitored program, the loop will end. The algorithm that implements the dispatching routine for memory-bug-detection is shown in Figure 3.2(a). In addition, Figure 3.2(b) shows an example of the monitoring function that checks whether there is a double free bug. The details of the double free bug as one type of memory bug can be found in [62, 63].

3.2.2 Monitoring Functions

Monitoring functions perform the actual monitoring tasks manually written by programmers or automatically generated by compiler according to the monitoring requirements. Monitoring requirements can be specified in the format of mathematical logic rules that unequivocally express the requirements an execution has to satisfy for safety and reliability reasons. For example, in the task of memory-bug-detection, in order to prevent memory leakage, we may have the following requirement statement: an address passed as an argument to a *free* function call must be the return value of a function of allocation. This statement can be formalized as a CTL(Computational Tree Logic) [46, 47] rule as the following: $\mathbf{AG}(free(addr) \rightarrow addr = alloc(\_))$. Here **A** and **G** are operators of CTL-FV, indicating that the formula it follows must be true on all possible execution paths of the program. `addr` is a free variable that can be substituted by all applicable addresses. $alloc(\_)$ can represent any function with any type of parameter that does memory allocation.

3.2.3 Initialization and Update Routines

There are two types of routines in a dispatch-based monitor that are in charge of instructing the hardware support in the table-driven mode to extract information that is relevant to the monitoring purposes. In this dissertation, we refer to these as *InitializeExtractionTable* and *UpdateExtractionTable*, respectively. *InitializeExtractionTable* initializes the extraction table and fills it with addresses or address ranges of PCs and data memory locations that are interested by monitor. *UpdateExtractionTable* is used to update the extraction table as described in Chapter 2.

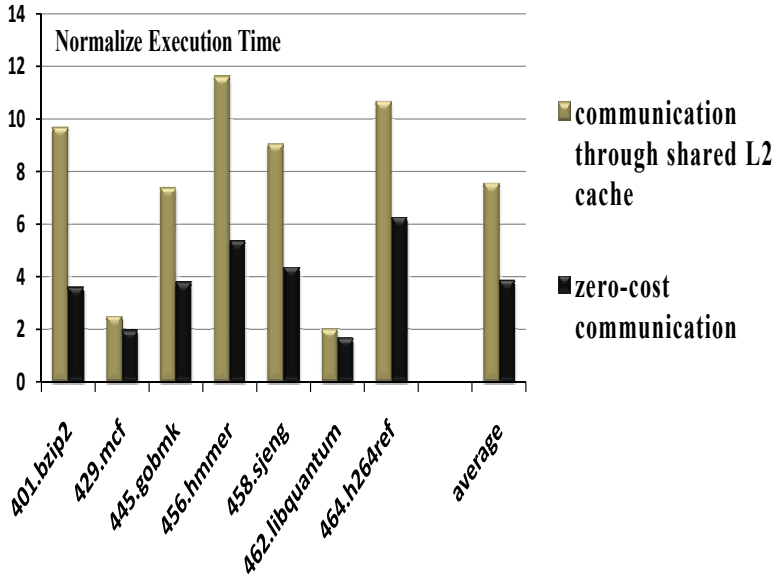


Figure 3.3: **Performance Comparison of Dispatch-based Dynamic Monitors on Multi-core for Taint Propagation**:left bar represents performance with communication through a shared L2 cache, right bars represent performance of zero-cost communication

3.3 Distill-based Monitor

This section presents a different monitor model, referred to as the distill model, that takes full advantage of the extraction logic, described in Chapter 2. Monitors in the distill model, referred to as distill-based monitors, achieve significant performance

improvements, compared to the dispatch-based monitors.

We start this section with an brief introduction to the key idea of the distill-based monitor model, elucidating how it accelerates dynamic program monitoring. After that we demonstrate how to generate distill-based monitor from monitored code.

3.3.1 Key Idea of Distill-based Monitor

The key idea of the distill-based monitor model is that the compiler could generate monitor code targeting individual monitored program. This way, the monitor is customized to the monitored program by the monitoring compiler as it has the knowledge of what values are needed to conduct the monitoring activity during the execution of the monitored code. With this knowledge, the monitoring system can instruct the extraction logic to forward only the values that are relevant to monitoring purpose. As a result, the distill-based monitor model is able to reduce communication between the monitored and monitoring cores, and in turn improve performance significantly for heavy load monitoring tasks that require a large quantity of information.

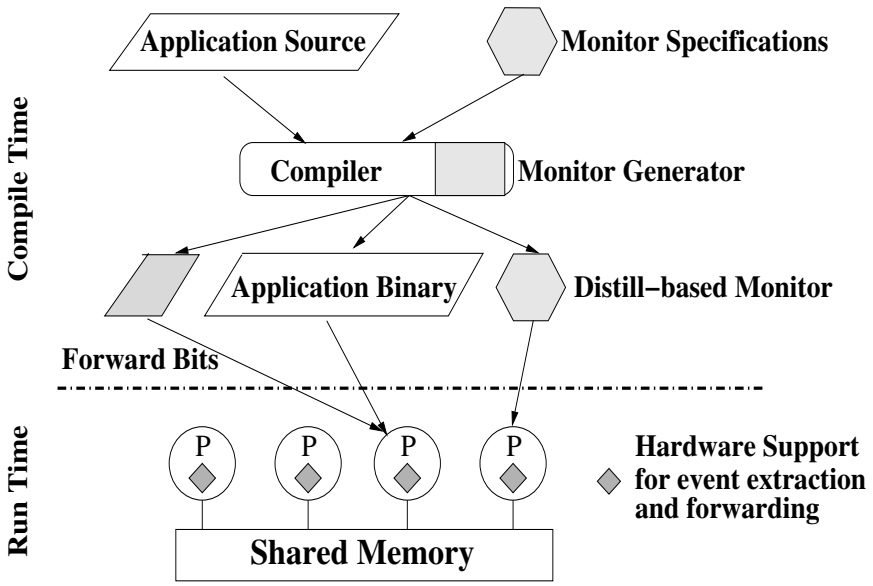


Figure 3.4: **Overview of the Distill-based Monitoring System**: the shaded boxes represent monitor related components. From user-defined *monitor specifications*, the *monitor generator* creates the *distill-based monitor*. The monitored application is annotated with forward bits, that are interpreted by the underlying hardware support to forward results of selected instructions.

3.3.2 Generating Distill-based Monitor

The distill-based monitor is generated by distilling the monitored code. In this section, we focus on how the monitor code generator generates the distill-based monitor without optimization, and we introduce the optimization in the next section.

The workflow of generating the distill-based monitor is illustrated in Figure 3.4. The input to the monitor code generator consists of two parts: the original source code of the monitored program and the monitoring specifications. These specifications can be expressed as annotations to the original monitored program marking where and what instrumentation of the monitoring function would occur if an instrumentation-based monitor was implemented. The output of the monitor code generator is the distill-based monitor and an annotation section in the monitored code's binary where

each instruction has a bit indicating whether the result of the instruction needs forwarding.

The monitor code generator generates code at the procedure level. For each procedure, it identifies invocations to the monitoring functions and all branch instructions on which the invocations depend. For each argument used in the invocation of monitoring functions, if its value cannot be known at compilation time, a fetch from the communication queue operation is inserted as an invocation to *receive()* function in the distill-based monitor. The code generator also marks each instruction that generates a value that should be fetched, indicating that the result of the instruction should be forwarded during the monitored execution. Similarly, for each value used to decide the direction of an identified branch, a *receive()* is generated and the instruction generating the value is marked, as well. Finally, the code is distilled by removing instructions other than the identified branch instructions, the invocations to monitoring functions and the *receive()* operations. This distilled code is the monitor used in the distill-based monitoring system. The marks that are generated by the monitor code generator form an annotation section and is inserted into the binary of monitored program.

The distill-base monitor generated by the code generator (without optimization) consists of two types of operations: invocations of the monitoring functions and invocations of the communication function, a.k.a *receive()* function. The monitoring functions perform monitoring operations and the communication function fetches values for monitoring functions from the communication queue. In addition, a value returned by the communication function could used to decide the outcome of a branch instruction that controls the invocation of monitoring functions.

3.3.2.1 Monitoring Functions

In this chapter, for the purpose of performance evaluation, we report the implementation of monitoring functions for two intensive monitoring tasks: taint-propagation and memory-bug-detection.

The details of taint-propagation can be found in [64, 65]. In our implementation, the distill-based monitor maintains a taint status bit for each byte of the monitored program's user memory space, as well as a taint status bit for each architectural register used in the program. These taint status bits are kept in the monitor's memory space as a shadow memory of the memory space of the monitored execution. A monitored execution may invoke system calls such as *read* or *recv* that writes to user memory with data from untrustworthy sources. When such invocations occur, the monitor invokes the Taint_Init function to set the corresponding taint status bits in the shadow memory. For each dynamic instruction in the monitored execution, the monitor executes a corresponding status propagation operation to propagate taint status around shadow locations accordingly. Taint status bits of indirect jump targets, format strings, and system call arguments are checked by the monitor to detect potential security breaches. Overall, the monitor needs to know all the referred data memory addresses and the directions of all conditional branches in the monitored program to conduct correct verification without false negative and false positive.

It is noteworthy that using the distill-based taint-propagation monitor, a large amount of monitoring function invocations do not need to be forwarded because their parameters are register numbers that are known at compile time. For example, arithmetic operations such as $Add\$R1, \$R2$, can be translated into corresponding taint-propagation monitoring functions such as TaintRegProp(1, 2), that propagates the taint bit of register R1 to the taint bit of register R2. Since the parameters are the register numbers known at compilation time, this type of monitoring function requires

no communication. In contrast, a dispatch-based monitor has to generate communication for these operations. This optimization alone gives a significant performance advantage to the distill-based taint-propagation monitor since a large portion of taint propagation are between registers.

We also implemented monitoring functions for memory-bug-detection. The monitoring functions are similar to those used in the MemTracker [32], except that the state bits in our implementation are maintained at word granularity rather than bit granularity. Moreover, when there is a read to uninitialized data, our monitoring function issues a warning immediately rather than wait until the value read is actually used.

The distill-based program monitoring system enables users to apply their knowledge about the code to make monitors even more efficient. For the example program in Figure 3.1(c), if the user knows in advance that pointer q points to an area whose size is always greater than the size of 512 integers, then half of the monitoring activities and their correspondent communication can just be eliminated.

3.4 A Qualitative Analysis of Performance Overhead

In this section, we start with a qualitative analysis of performance overhead associated with the dispatch-based model, pointing out sources of the performance overhead. Then we show how the distill-based monitor eliminates some of the sources of performance overhead and the distill-based model. The result of the analysis indicates that the performance overhead of the distill-based model should be lower than the dispatch-based model. This conclusion will be vindicated by the experimental results presented in Chapter 4.

3.4.1 Analysis of Performance Overhead of Distill-based Monitor

Performance overhead caused by a dispatch-based monitor is lower than that of instrumentation-based monitor, because a separate processor core is used and thus the monitor runs in parallel to the monitored execution. However, this overhead is still fairly significant. The causes of this high performance overhead are the unnecessary communication and dispatching related to irrelevant messages. In a dispatch-based monitoring system, the monitored process may forward data that are irrelevant by the monitor. In the sample program in Figure 3.1, if we are interested only in verifying memory operations by the pointer, the monitor does not need execution results from other types of operations, thus communication for these results is unnecessary. It is true that a dispatch-based monitor can use the extraction table of the hardware support to select data to be forwarded, but at a cost of possibly frequent updating of the extraction. The more frequent the extraction table is updated, the more the irrelevant information to be forwarded because the extraction logic forwards everything during the period of update. Furthermore, in the dispatch-based monitor model, the type of information of the message has to be communicated, which is solely for dispatching purpose and thus can be considered unnecessary.

Communication is the bottleneck of performance. Figure 3.3 presents the results of a study on the impact of communication on the performance for multi-core-based program monitoring using dispatch-based monitors. Here monitors perform taint propagation [64, 65] and memory-bug-detection [66, 62, 66, 67] for SPEC2006INT [68] benchmarks, respectively. The bars represent the execution time of monitors normalized to that of the benchmarks without monitoring. The communication queue between cores is implemented in shared memory and resides in the shared L2 cache most of time during monitoring. We also assume that each core has a private L1

data cache, and therefore communication of a value may incur a coherence miss that requires a fetch of the data from the shared L2 cache. This causes L2 cache access latency. Using the above assumptions, the slowdown to the benchmarks with the two-process dispatch-based monitoring is represented by the left bars. The right bars represent the slowdown of the dispatch-based monitoring in an ideal situation where the communication is latency-free. The figure suggests that a significant portion of performance overhead is caused by communication.

In addition to the communication cost, dispatching a message consumes a certain number of CPU cycles of the monitor core. It involves decoding a message to get its type, extracting the value from the message, and jumping to the place where the appropriate monitoring is invoked.

3.4.2 How Distill-based Monitor Reduces Performance Overhead

Figure 3.1(c) demonstrates, with an example, how the distill-based monitor reduces communication for dynamic program monitoring. The sample program communicates only values of `q` and `i`. It leverages the forward-bit mode of the hardware support extraction logic to selectively forward values. Communication of values that are not relevant to the monitoring purpose is eliminated in the distill-based monitor. Since most, if not all, monitoring tasks need only a portion of all values generated by the monitored execution, this reduction is significant.

The distill-based monitor is optimized by replacing the communication operation with less expensive computation operations on the monitor side. For example, in Figure 3.1(c), the communication of data value q and i can be replaced by local computations of simple integer add operations. If the monitor compiler initiates only the communication of the value of p, then all subsequent values of q and i can be

computed locally in the monitor code. Therefore, in Figure 3.1(d) the number of invocations to the communication function `receive()` for the while loop is reduced from 2048 to 1. As communication is a notable source of performance overhead, this optimization leads to an even greater performance overhead. This optimization of distill-based monitor is the main focus of the next chapter, Chapter 4.

Chapter 4

Optimization of Distill-based Monitor

The distill-based monitor can be optimized by eliminating the communication of values that can be computed by the monitor locally. In this chapter, we present details of algorithms that implement this optimization of reducing communication by computing values locally. The chapter starts with an introduction to general ideas and concepts about the optimization, then we give an overview of the algorithm that implements the optimization. Following that, we describe the different kinds of information needed by the algorithm and the ways in which they are collected. After that, we describe the details of the algorithm along with discussion of the optimality and complexity of the algorithm. Finally, we present detailed results of the performance evaluation of the optimized distill-based monitor, comparing it with the basic distill-based monitor, the dispatch-based monitor and the instrumentation-based monitor. To illustrate the optimization in a more lucid way, we use a sample program shown in Figure 4.1 that is slightly more sophisticated than the example shown in 3.1. The sample program with instrumentation, the unoptimized distill-based monitor, and optimized version of the distill-based monitor are shown in Figure

```
foo(int* a, int y) {
  int *p;
  int i=0,j=0;
  do {
    p = a + i;
    VERIFY_STORE(p);
    *p = i;
    j = j + i;
    if (i==y) {
      p = a + j;
      VERIFY_STORE(p);
      *p = 0;
    }
    else {
      j = y + i;
      p = a + j;
      VERIFY_STORE(p);
      *p = y;
    }
    i++;
  } while (i<1024)
}
```

(a) Sample monitored code: original code with instrumentation

```
foo(int* a, int y) {
  int *p;
  int i=0,j=0,;
  y = receive();
  do {
    p = receive();
    VERIFY_STORE(p);
    if (i==y) {
      p = receive();
      VERIFY_STORE(p);
    }
    else {
      p = receive();
      VERIFY_STORE(p);
    }
    i = receive();
  } while (i<1024)
}
```

(b) Unoptimized monitoring code

```
foo(int* a, int y) {
  int *p;
  int i=0,j=0;
  a = receive();
  y = receive();
  do {
    p = a + i;
    VERIFY_STORE(p);
    if (i==y) {
      p = receive();
      VERIFY_STORE(p);
    }
    else {
      j = y + i;
      p = a + j;
      VERIFY_STORE(p);
    }
    i++;
  } while (i<1024)
}
```

(c) Optimized monitoring code: communication are replaced by local computations

Figure 4.1: Code Example of Using Compiler to Reduce Communication

4.1(a), 4.1(b), and 4.1(c) ,respectively.

4.1 General Ideas and Concepts of the Optimization

Essentially, the optimization of the distill-based monitor in the context of this thesis reduces overall communication volume between cores by replacing the communication of some values required by monitoring purposes with local computation on the monitor side. In this section, we introduce ideas and concepts that lead to a better

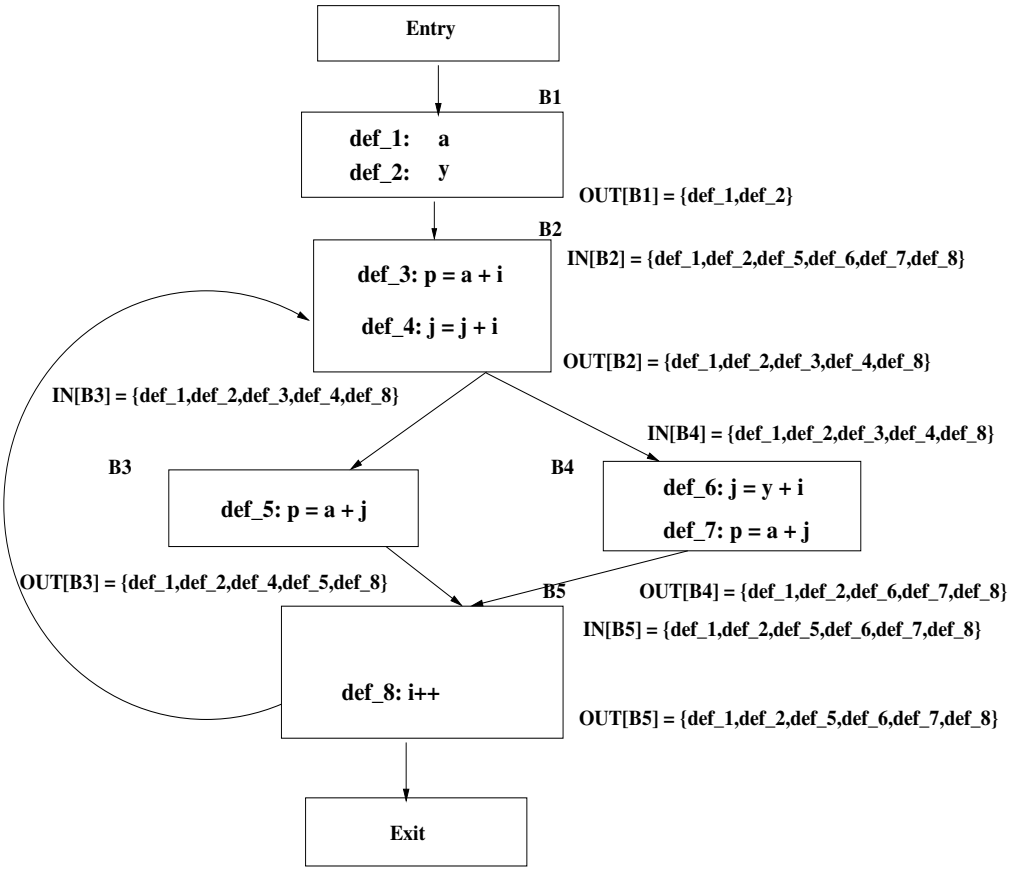


Figure 4.2: Control Flow Graph of the Sample Program and Results of Reaching Definition DFA

understanding of the optimization.

We start with studying the correctness requirement and the effectiveness requirement of the optimization. In the context of dynamic program monitoring, the correctness of the optimization means that the optimized distill-based monitor must perform exactly the same monitoring operations in exactly the same order as the monitor without optimization. Thus, if a value is communicated in the unoptimized monitor, it must be either computed or communicated in its corresponding optimized monitor. In a program, values are generated at points of variable definitions. In this section, we use $BasicCommu$ set to denote the set of variable definitions whose values are communicated in the unoptimized distill-based monitor. For the sample program

in Figure 4.1 with its control flow graph shown in Figure 4.2, the *BasicCommu* set is $\{def\_2, def\_3, def\_5, def\_7, def\_8\}$.

Making the optimization effective means that optimized monitors must outperform their corresponding unoptimized monitors. To guarantee this, two constraints must be met: first, computations introduced in the optimized monitor must compute only those variable definitions that one or more variable definitions in the *BasicCommu* set are directly or indirectly dependent. For example, in Figure 4.2, def_3 and def_8 are in *BasicCommu* and are dependents of def_8, therefore, computing def_8 on the monitor side may eliminate communication for def_3 and def_8. In this dissertation, we use *OptCompute* set to denote the set of these computed definitions. Second, the cost of computing values locally must not exceed the cost of communicating them. This is guaranteed by comparing the cost of computations involved in deducing those values against the cost of communicating them. It is worthy pointing out that the optimization may replace a communicated value with local computations and communication of a value that is not defined in the *BasicCommu* set. This may seem like a performance degradation at the first glance. It is possible for the additional communication to achieve better performance by eliminating other communications. In this dissertation, we refer to the set of the communicated variable definitions in an optimized monitor as the *OptCommu* set.

Thus, in the context of this dissertation, to optimize the distill-based monitor is to find the *OptCompute* set and the the *OptCommu* set that minimize the overall performance overhead associated with the monitor. In the example shown in Figure 4.2, these two sets are:

$OptCompute = \{def\_3, def\_6, def\_7, def\_8\}$

$OptCommu = \{def\_1, def\_2, def\_5\}$

4.2 Algorithm Overview

The algorithm that implements the optimization is based on the data flow analysis (DFA) framework. Inputs of the algorithm are the control flow graph (CFG) of the monitored program and the *BasicCommu* set of the monitored program. The algorithm traverses CFG, visiting each basic block to decide the *OptCompute* and *OptCommu* sets for the entire CFG. While visiting each basic block, the algorithm examines every variable definition in the block, decides whether it should go into the *OptCompute*, the *OptCommu* set, or be ignored if it does not contribute to any value needed by monitoring function.

The algorithm makes decisions for all variable definitions in a basic block corporately rather than individually. It actually evaluates the cost and the benefit of combinations of possible choices for each variable definitions and choose the combination with the highest overall benefit as its decision for the basic block. In the context of the algorithm, the benefit and the cost of the decision are measured in the execution time of the monitor. Since the optimization reduces monitor execution time solely through reducing communication, a decision's benefit is estimated by counting the number of communications that can be eliminated by the decision. The cost is the cost of additional computations and communications the optimization introduces into the monitor.

Through the DFA framework, the decision made for a basic block will be propagated to successor basic blocks to help them make decisions. All the variable definitions on CFG marked for communication form the *OptCommu* set, and those marked for computation form the *OptCompute* set. For all definitions in the *OptCommu* set, the compiler generates communication functions in the optimized monitor and sets the forward bit of the corresponding instructions in the monitored program. Computation instructions that generate definitions in the *OptCompute* are preserved in the

distill-based monitor program, and all other instructions are removed.

4.3 Information Needed by the Algorithm

The key to the algorithm is accurately estimating benefit and cost of decision combinations for basic blocks. In this section, we describe information that is necessary to the estimation and present the algorithm that collects it.

There are three types of information needed for accurate cost-benefit estimation. First, the algorithm needs the reaching definition information. Reaching definition information shows at any given program point [76], for any variable, which definition may reach there. Second, to estimate a decision's benefit, the algorithm needs to identify a subset of $BasicCommu$. This subset includes all variable definitions whose communication might be eliminated for better monitor performance . If the decision is at a basic block n, using v to represent one of the available definitions, this subset is called the Basic Dependent Set of v at the exit of n, or $BDS(n, v)$ in short. Lastly, the final type of information is the estimation of the computation cost associated with a decision, or in other words, the information of what computation operations are indispensable to the elimination of communication of every definition in $BasicCommu$. This type of information is obtained by maintaining a computation stack for each definition in $BasicCommu$.

We have implemented three different Data Flow Analysis (DFA) passes to collect these three types of information, as discussed below.

4.3.1 Reaching Definition

The first pass obtains reaching definition information using the classic Reaching Definitions DFA(Data Flow Analysis) [77]. The result of Reaching Definitions DFA gives a picture of which definitions may reach any given program point in the CFG.

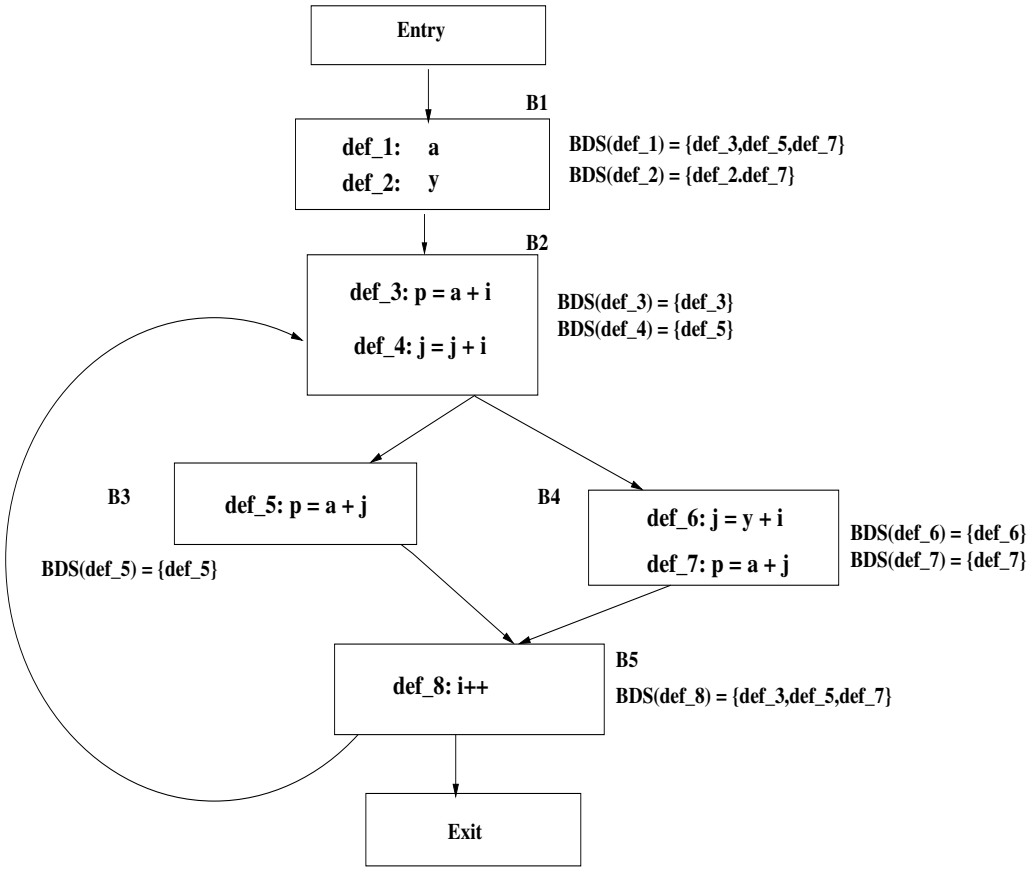


Figure 4.3: Results of Basic Dependence Set DFA Pass for the Sample Program

In the research reported in this dissertation, we store only the information per basic block, that is the set of definitions that may reach to the entry and the exit of a basic block. This saves storage space usage without losing accuracy because the set of reaching definitions at any program point within a basic block can be computed by the reaching definition set at the block entry. The result of reaching definitions DFA for the sample program in Figure 4.1 is shown in Figure 4.2.

Reaching definition DFA can serve as the basic means of computing use-def and def-use chain information. This information depicts the dependence relations among variable definitions on CFG, and which are used by other DFA passes we introduce below.

4.3.2 Basic Dependence Set

After the reaching definition DFA pass, the compiler computes the Basic Dependent Set (BDS) for all reaching definitions using a backward DFA analysis. The purpose of the BDS pass is to collect basic information about the potential benefit of making a variable definition available when making decisions for basic blocks of a CFG.

The domain of the BDS DFA is the power set of the BasicCommu set. The following dataflow equation computes $BDS(n, v)$ at the exit of the basic block n for variable v:

$$BDS(n,v) = \begin{cases} \emptyset; & \text{if } n = exit \\ \prod\limits_{m \in succ(n)} transfer(m, v, BDS(m, v)); & \text{otherwise} \end{cases}$$

The meet operator $\prod$ is set union. Before the *Basic Dependent Set Analysis* begins, all nodes are initialized to $\perp$, which represents the empty set. The transfer function is defined as the following:

- If $v \in Gen(m)$, then $transfer(m, v, BDS(m, v)) = \emptyset$.
- If $v \notin Reach(m)$, then $transfer(m, v, BDS(m, v)) = \emptyset$.
- If a definition w in $Gen(m)$ depends on v, then every element of $BDS(m, w)$ is added into $transfer(m, v, BDS(m, v))$.
- If a definition v is an argument to a monitoring function call in node m, v is added into $transfer(m, v, BDS(m, v))$

Here $Gen(m)$ is the set of definitions generated in the basic block m and $Reach(m)$ is the set of definitions reaching to the entry of m. For the sample program, the BDS computed at the exit of each basic block is shown in 4.3. Due to space limits, the figure shows only the BDS of the definitions generated in a basic block, though in the implementation we maintain BDS for all definitions reaching to the entry and the

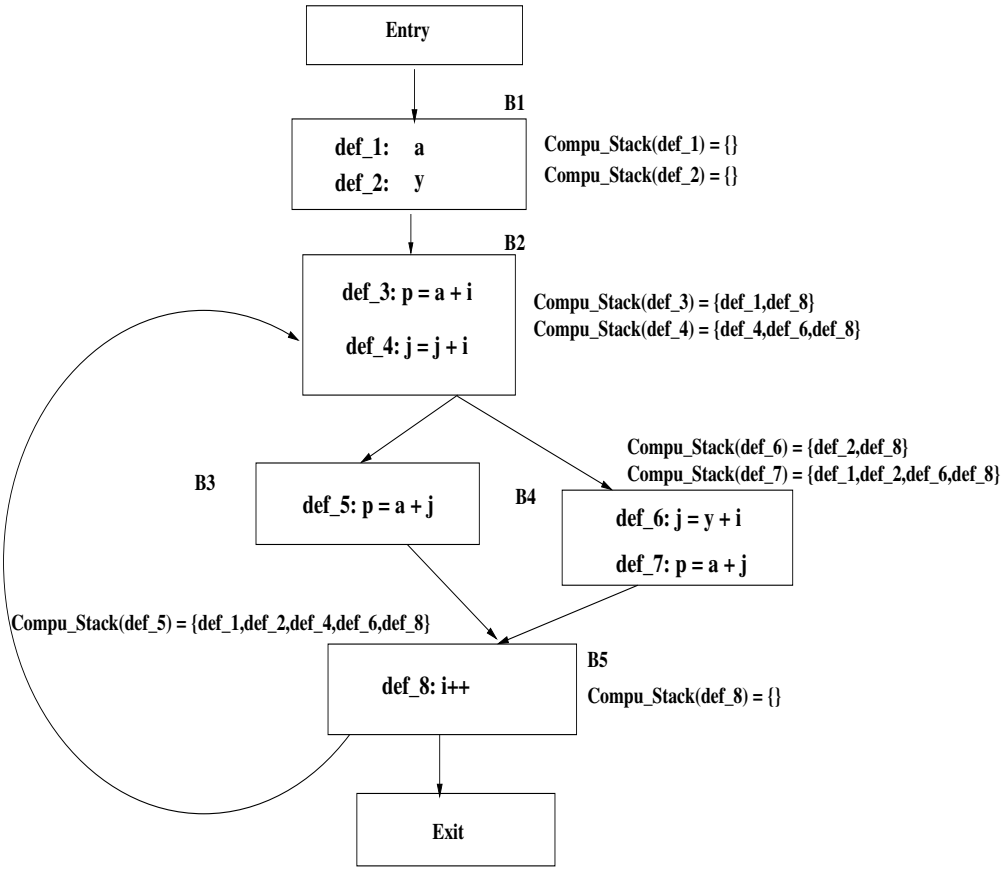


Figure 4.4: Results of Computation Stack DFA Pass for the Sample Program

exit of a basic block.

4.3.3 Computation Stack

A decision for a basic block may choose to compute a variable definition in the optimized monitor. In a program, to compute the value of a variable from a given program point, usually there is a sequence of interdependent computations are necessary. This sequence of computations can be best formatted as a stack of computations, referred to as the computation stack in this dissertation. Therefore, to know the cost of computing a variable definition from a program point, a DFA pass is needed to compute the computation stack for the variable definition at that program point.

The computation stack for variable definition is originally defined in [69]. This

DFA pass is a backward DFA that builds a computation stack at every basic block for every definition in *BasicCommu* that reaches the basic block. The implementation of computation stack DFA is similar that presented in [70], where readers can find details of the algorithm. The result of computation stack DFA is that at any given basic block n, for each definition u in *BasicCommu* set, this DFA pass builds a stack $Stack(n, u)$ that records all the computations needed to get the value of u from the exit of the basic block n. For the sample program, the result of this DFA pass is shown in Figure 4.4.

4.4 Select OptCommu and OptCompute Set: Details of Algorithm

After running all three information-collecting DFA passes, all the necessary information that the algorithm needs is ready. The compiler then runs the algorithm to find the *OptCommu* set and *OptCompu* set. The sets contain values to be communicated and computed in the optimized monitor for the given CFG. In this section, we present details of the algorithm and discuss its complexity.

In this section we first illustrate the whole algorithm using the concepts of DFA as it is a DFA pass. Then, we describe the core part of the algorithm: the estimation of the benefit and cost of a decision for a basic block. Finally, we discuss the complexity and practical use of the algorithm.

4.4.1 Select *OptCommu* and *OptCompute* Set: Forward DFA Pass

The *OptCommu*/*OptCompute* problem is solved as a forward dataflow problem, where the solution at each node corresponds the optimal communication/computation

begin algorithm
$OptCommu_{out}[ENTRY] = \emptyset$;
$OptCompute_{out}[ENTRY] = \emptyset$;
foreach (basic block B other than ENTRY)
  $OptCommu_{out}[ENTRY] = \emptyset$;
  $OptCompute_{out}[ENTRY] = \emptyset$;
end foreach
while (changes to any OutCommu or OutCompute set occur)
  foreach (basic block other than ENTRY)
    $OptCommu_{in}[B] = \bigcup_{P \in B_{pred}} OptCommu_{out}(P)$;
    $OptCompute_{in}[B] = \bigcup_{P \in B_{pred}} OptCompute_{out}(P)$;
    $(OptCommu_{out}[B], OptCompute_{out}[B]) = transfer(OptCommu_{in}[B], OptCompute_{in}[B])$;
  end foreach
end while
end algorithm

(a) Select *OptCommu* and *OptCompute*: Forward DFA

data declarations:
```
typedef struct message_struct
    {Type, Address, Value, I/D flag }
message_struct msg;
List of events for initializing and
updating the extraction table.
message_struct Msg_List[];
```

$transfer_B$($OptCommu_{in}[B]$,$OptCompute_{in}[B]$)
begin
  $DecisionCandidateList$ = $AllValidCombination$($OptCommu_{in}[B]$,$OptCompute_{in}[B]$);
  $CurrentBest$ = $DecisionCandidateList$.first;
  $CurrentNetBenefit$ = $Benefit_{CurrentBest}$ - $Cost_{CurrentBest}$;
  foreach *candi* on $DecisionCandidateList$;
    $NetBenefit_{candi}$ = $Benefit_{candi}$ - $Cost_{candi}$;
    if ($NetBenefit_{candi}$ > $CurrentNetBenefit$)
      $CurrentBest$ = *candi*;
      $CurrentNetBenefit$ = $NetBenefit_{candi}$;
    end if
  end foreach
  $OptCommu_{out}[B] = OptCommu_{in}[B] \bigcup$ (*candi.commu_set*)
  $OptCompute_{out}[B] = OptCompute_{in}[B] \bigcup$ (*candi.compute_set*)
end

(b) Transfer function of the algorithm.

Figure 4.5: Distill-based Monitor Optimization Algorithm

decision that minimizes the performance overhead of the monitor. This optimal decision as well as decisions that are propagated to B from its predecessors, are then propagated to successors of B in the format of two sets of definitions: $OptCommu_{out}(B)$ and $OptCompute_{out}(B)$. $OptCommu_{out}(B)$ is the set of variable definitions decided to be communicated before the exit of B. $OptCompute_{out}(B)$ is the set of definitions decided to be computed before the exit of B. Correspondingly, the dataflow values at the entry of B are denoted as $OptCommu_{in}(B)$ and $OptCompute_{in}(B)$. The meet operator of the DFA is set union which means that $OptCommu_{in}(B)$ and $OptCompute_{in}(B)$ should consider all the definitions reaching to B that are communicated or computed in previously visited basic blocks. A more formal description of the algorithm in the form of standard DFA iterative algorithm can be found in Figure 4.5(a).

As Figure 4.5(a) shows, it is the transfer function that decides which variable definitions in the block should be included in the *OptCompute* set, and what should be included in the *OptCommu* set. To achieve this, the transfer function examines every variable definition in the block. For each variable definition, there are at most three choices: it can be communicated, it can be computed if the correspondent instruction has all the source operands available, or it can simply be ignored as long as it is not in *BasicCommu*. The transfer function enumerates and evaluates the benefit and cost of all valid combinations of choices for variable definitions generated in the basic block. A valid combination is referred to as a decision candidates for the basic block. A formal description of the transfer function can be found in Figure 4.5(b).

4.4.2 Estimation of Benefit and Cost

The key for making the optimal decision for a basic block in this algorithm is to estimate the net benefit of each decision candidate. The net benefit is calculated

by subtracting the cost from the benefit that are measured in CPU cycles. Specifically, benefit is measured as the potentially saved CPU cycles from the reduction of communication, and cost as the additional cycles spent on the introduced local computations and extra communication. In this section, we describe details of how the benefit and the cost for a decision candidate are calculated, respectively.

The benefit of a decision candidate is the product of the number of eliminated invocations to communication functions and the average latency of a communication. The average latency of communication in the monitoring system reported in this dissertation is estimated as the latency to serve a L2 cache request. This is because reading a value from the communication queue incurs a L1 cache coherence miss and is served from L2 Cache. The number of communications eliminated by the decision is the sum of the frequency of the variable definitions whose communication can be eliminated by the decision. The variable definitions whose communication is eliminated by a decision constitutes a subset of $BasicCommu$. The algorithm computes this subset with the following steps. First, the algorithm computes the union set of the BDSs of all the reaching definitions that are decided to be computed or communicated by the decision. Next, the algorithm computes another union set of all the BDSs of all the definitions that are not made available by the decision. The relative complement set of the latter computed set with respect to the former computed set is the set of definitions whose communications can be eliminated by the decision. The frequency of variable definitions can be obtained from either static analysis or dynamic profiling information, which is provided by most modern compilers.

The cost of a decision candidate is estimated as the total number of CPU cycles spent on the local computations and extra communications introduced by the decision. Extra communications are communications of those variable definitions that are not included in the $BasicCommu$ set. In other words, these communications are introduced to potentially improve overall performance better. Therefore, the extra

communication cost can be calculated by simply estimating the cycles spent on the communication of the definitions generated by the decision but not included in the *BasicCommu* set. To compute the cost of introduced local computations, we use the results from the Computation Stack DFA pass. For all *BasicCommu* definitions whose communication maybe eliminated by the decision, the algorithm counts all the operations in their computation stacks in the basic block. The total cycles spent on these operations are the cost of the local computations introduced. The sum of the cost of extra communications and the cost of computations is the cost of the decision candidate. While calculating the cost of a decision candidate, the frequency of variable definitions is considered as it is in calculating the benefit.

The transfer function then tries to finds the best decision for a basic block by comparing the estimated benefit and cost of all decision candidates. The candidate that has the greatest net benefit is considered the best decision for the basic block. The decision indicates which among the variable definitions that are generated in the basic block should go into *OptCompute* and which should go into *OptCommu*. When the DFA converges, the union set of *OptCompute* sets of all basic blocks is the *OptCompute* for the CFG, and the union set of *OptCommu* sets of all basic blocks is the *OptCompute* for the CFG.

For the example code shown in Figure 4.1, the result of the optimization DFA pass is shown in Figure 4.6. $OptCommu_{out}$ and $OptCompute_{out}$ sets of each basic block in the figure show only decisions of each basic block, and thus is only the subset of real $OptCommu_{out}$ and $OptCompute_{out}$ sets used in the algorithm. We illustrate subsets only because the real $OptCommu_{out}$ and $OptCompute_{out}$ sets for most basic blocks after the final iteration of this DFA pass are all the same and thus not clear for the purpose of illustration. In addition, BDS info is added to help readers understand the benefit and cost estimation of decisions.

As Figure 4.6 demonstrates, the final decision made by the optimization pass for

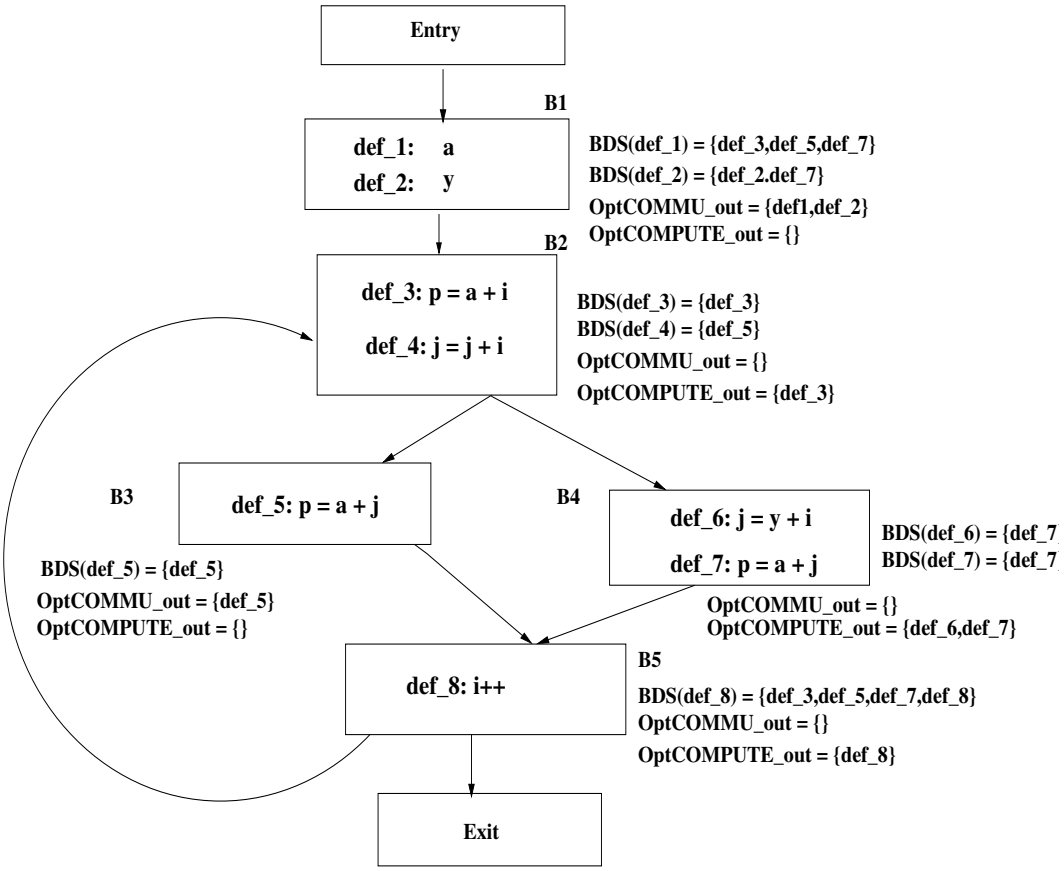


Figure 4.6: **Results of Optimization DFA Pass**: results of the forward DFA pass that selects OptCommu and OptCompute Set, which implements the optimization algorithm. For the purpose of clear presentation, the figure includes only results at end of every basic block, and result of each basic block shows only decisions made for definitions of that basic block. Results of BDS are also included to show the benefit of the decision. The Out-OptCommu set and Out-OptCompute set of a basic block B used in the DFA transfer function should include results shown in the figure and the results that B inherits from its predecessors.

the entire CFG is the following $OptCompute = \{def\_3, def\_6, def\_7, def\_8\}$ $OptCommu = \{def\_1, def\_2, def\_5\}$ It is noteworthy that def_5 is selected to be communicated because it is in the infrequent basic block $B3$, and that to compute it, computation def_4 has to be made available, and def_4 is in the frequent basic block $B2$. Thus, the optimization algorithm finds it more beneficial to communicate def_5, and thus ignore def_4. This example shows the role that frequency plays in the algorithm.

4.5 Discussion of Complexity of the Optimization

The optimization essentially consists of four DFA passes: reaching definition, BDS, computation stack, and the *OptCommu* and *OptCompute* Set Selection algorithm. Therefore, to know the complexity of the optimization, we need to analyze the time and space complexity of each individual DFA pass. In this section, we analyze the time and space complexity of every DFA pass using big O notation.

Throughout this section, we use d as the total number of variable definitions in the CFG, n as the number of basic blocks (nodes) in the CFG, and m as the number of variable definitions in the $BasicCommu$ set.

4.5.1 Complexity of Information Collection DFA Passes

The worst-case space complexity of reaching definition DFA is $O(dn)$ in big O notation, as each basic block may store up to d definitions in its In and Out sets. In the reaching definition DFA framework, the algorithm traverses the CFG to compute reaching definition sets. It keeps traversing until no reaching definition set for any CFG node changes. Based on this fact, the time complexity of the reaching definition is the product of two parts: the number of traversals over the CFG and the time that the algorithm spends in each traversal. In each traversal, the algorithm visits every CFG node once, in each CFG node it processes no more than d definitions, and for each definition it spends constant time to process it. Therefore, the time complexity of each traversal is $O(dn)$. As to the number of traversals of the DFA, using the concepts and the proof presented in [76], it is bounded by the DFA lattice height and n. In the reaching definition DFA, as the DFA only increases In and Out sets as it traverses the CFG, the DFA is a monotone framework. Each definition has only two states (in a set or not), so the height of the lattice is 1. Therefore, the number

of traversals is $O(n)$, and thus the worst-case time complexity of the reaching definition DFA is $O(dn^2)$. However, for most realistic programs, the algorithm settles to a fixed-point solution after a few traversals with the CFG. Thus we bound the number of traversals with a constant, and the time complexity of this DFA is $O(dn)$.

The Basic Dependence Set (BDS) DFA has the maximum domain size as the size of the $BasicCommu$ set, therefore, the worst-case space complexity for the DFA is $O(mn)$. As to the worst-case time complexity, BDS DFA also keeps traversing the CFG until all BDS sets are settled. In each traversal, it also visits every node and processes no more than d definitions, each within constant time. Therefore, we can apply the same analysis that we used for the reaching definition DFA to deduce its worst-case time complexity out of the time on each traversal and the number of traversals. For one traversal, BDS DFA spends $O(dn)$ time. BDS DFA pass is a monotone framework, as traversals only put more elements into BDS sets. In addition, since whether a definition is in a BDS is independent to that of another definition, the lattice height of the BDS DFA is also 1. Therefore, the number of traversals is $O(n)$, and the time complexity of BDS DFA is $O(mn^2)$. Again, if we bound the number of traversals with a constant, the time complexity of this DFA is $O(mn)$.

A similar analysis can be applied to the computation stack DFA pass. The computation stack pass has its domain size as the total number of all computation operations, which is $O(d)$. The framework is also a monotone framework as the algorithm only increases computation stack as it traverses over the CFG. Hence, its worst-case space complexity is $O(dn)$ and the time complexity is $O(dn^2)$ or $O(dn)$ with traversal times bounded with a constant.

4.5.2 Analysis of Complexity of OptCommu and OptCompute Set Selection

The *OptCommu* and *OptCompute* Set Selection DFA pass, a.k.a the optimization algorithm DFA traverses the CFG until *OptCommu* and *OptCompute* sets for all CFG nodes are settled. As the algorithm only increases the *OptCommu* and *OptCompute* sets as it traverses over the CFG, this DFA pass is also a monotone framework. Therefore, for each variable definition, the height of its lattice is 1, which means that the number of traversals is bounded by $O(n)$. However, unlike other DFA passes mentioned above, in each traversal, for each definition in a CFG node, the transfer function spends non-constant time to process it. The time the transfer function spends in each traversal can presented as the following expression.

$$\sum_{i=1}^{n} T(f(B_i)) \tag{4.1}$$

Here, n is the total number of basic blocks in the CFG, $T_{f(B_i)}$ is the time that transfer function spends on ith basic block. The algorithm of the transfer function can be found in Figure 4.5(b). The algorithm enumerates all possible decision combinations, and for each combination, evaluates its net benefit. Each variable definition has at most three options: being communicated, being computed, and being ignored. Therefore, the number of possible combinations, in the worst case is 3^x as x denotes the number of variable definitions defined in the basic block. For each combination, it takes at most $O(d)$ time to evaluate its net benefit because evaluation involves examining all elements in BDS and compuatation stack, and the sizes of BDS and computation stack are bounded by d. Thus, the time complexity of the transfer function for a basic block can be expressed as follows:

$$T(f(B_i)) = O(3^{x_i} d) \tag{4.2}$$

So, we have the time that the transfer function spends in each traversal as follows:

$$\sum_{i=1}^{n} O(3^{x_i} d) \tag{4.3}$$

It can be further deduced to:

$$O(3^{x_{max}} dn) \tag{4.4}$$

Here, x_{max} is the maximum number of definitions a basic block in the CFG holds. The challenge is that x_{max} could be d in the extreme case, which would make the optimization a NP-hard problem if there is no other optimal solution. Moreover, since valid combinations have to be stored during the execution of the transfer function, the extreme case brings space complexity up to the exponential level, as well.

To solve this challenge, we trade some optimality for time and space efficiency. We put a cap on the number of valid combinations to be evaluated for a basic block, thus bringing down the number of valid combinations $3^{x_{max}}$ to a constant. As a result, the worst-case time complexity of the practical implementation of optimization algorithm DFA pass becomes the following:

$$O(dn)O(n) = O(dn^2) \tag{4.5}$$

It is worthwhile to point out that for the benchmarks we have evaluated, the x_{max} never exceeds the cap we set up, which is 3^6. Therefore, we do not lose optimality for the evaluation that is reported in the following section.

As to the space complexity, as we put a cap on the number of valid combinations to be evaluated, the storage requirement of the transfer function becomes constant. On the other hand, the pass has the maximum cardinality of d for each $OutCommu(B)$ and $OutCompute(B)$. Therefore, its worst-case space complexity can be just $O(dn)$.

Table 4.1: Benchmark Descriptions

Benchmark	Description
401.bzip2	Data compression and decompression
429.mcf	Public transportation scheduling. It uses
	a network simplex algorithm
445.gobmk	An application of Artificial Intelligence in game of Go.
456.hmmer	A tool that searches gene sequence. It uses profile
	hidden Markov models.
458.sjeng	Chess play game. It is an application of AI in chess game.
462.libquantum	Quantum computer simulator. It runs Shor's polynomial-time
	factorization algorithm
464.h264ref	Stream video encoder. It implements H.264, a state-of-the-art
	video compression standard.

4.6 Performance Evaluation

This section presents the results of the performance evaluation of the dynamic program monitors on multi-core platforms, including the dispatch-based and the distill-based model. Performance evaluation is conducted using Simics simulation platform [95] with GEMS simulator [96] to simulate memory hierarchy. The performance evaluation adopts SPEC2006INT [68] as monitored programs with taint-propagation and memory-bug-detection as monitoring tasks. We have implemented a binary rewriter to implement code generation and optimization of the distill-based monitors.

In this section we first describe the infrastructure used in the performance evaluation. After that, we demonstrate the performance results of different software-based monitor models, including instrumentation-based monitors, dispatch-based monitors, distill-based monitors without optimization and optimized distill-based monitors. Finally, we highlight the effect of the distill-based model and its optimization by comparing the performance of the optimized distill-based monitor with the performance of the dispatch-based monitor with hardware-based optimization.

Table 4.2: Simulation Parameters

CPU Parameters	
Instruction Set	Sparc-v9
Clock Rate	1.5 GHz
Number of Cores	4
Number of Chips	1
Register File	64 Integer, 64 FP

Memory Parameters	
Cache Line Size	64 Bytes
Instruction Cache	64KB,4-way set-assoc, 4 banks
Data Cache	64KB,4-way set-assoc, 4 banks
Unified L2 Cache	4MB, 4-way set-assoc, 8 banks
Coherency Protocol	MSI(Modified-Shared-Invalid).
L2 Cache latency	12 cycles
Main Memory Size	4 GB
Main Memory Latency	100 cycles.

4.6.1 Infrastructure

The performance evaluation reported in this dissertation is conducted on a full-system simulator built on Simics simulation platform [95]. On top of Simics, GEMS [96] is used to simulate memory hierarchy in details. The simulated machine has a quad-core CMP Sparc-v9 processor [59]. Each core is equipped with a separate 64K L1 data and 64K L1 instruction cache, and an unified inclusive L2 Cache shared between all cores. The MSI(Modified-Shared-Invalid) cache coherence protocol [61] is implemented to keep the caches coherent. Detailed simulation parameters can be found in Table 4.2.

The multi-core-based monitors evaluated are configured to run as follows. The monitored program and its monitor are assigned to separate cores during their executions. Information about the execution of the monitored program is sent from the monitored core via a 64K-entry communication queue to the monitor. The communication queue is implemented in the shared memory. Each queue entry is 64 bytes in size in the dispatch-based model since it has to carry all aspects of information and

only 8 bytes in the distill-based model. To prevent security breaches from tampering with the OS kernel, before the monitored execution switches to the kernel mode, the execution is stalled until items on the queue are completely processed by the monitor. The same technique has been reported by Chen et.al [35].

We have implemented taint propagation and the memory-bug detection monitors that monitor execution of SPEC2006INT benchmarks with test input sets. A brief description of the benchmarks used in the experiment can be found in Table 4.1. A more detailed description can be found in [93] for each benchmark except 429.mcf and 462.libquantum. The first two to ten billion instructions are fast-forwarded as the initialization phase and the following one billion instructions are simulated. 429.mcf and 462.libquantum have a high Cycle Per Instruction (CPI) ratio, and thus we only simulated the 200 million instructions after their initialization.

We have developed a binary rewriter to implement the compiler techniques described in this dissertation. The tool takes the binaries of the monitored benchmark programs as inputs and generates taint propagation and memory-bug detection monitors. Both monitors need information on a large percentage of instructions, and thus serve as a good pressure test for the efficiency of the underlying hardware/software support.

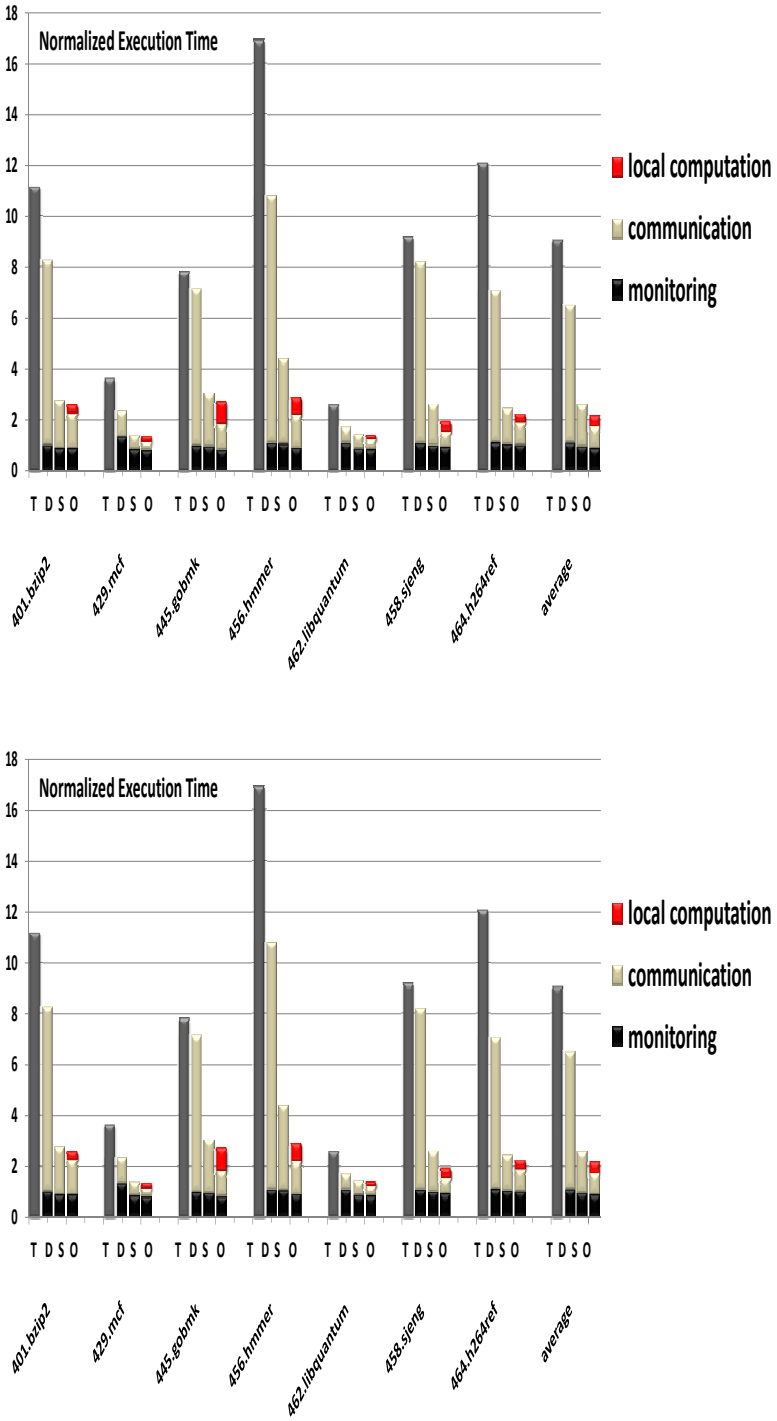
In the performance evaluation reported in this chapter, we have evaluated the performance of four different implementations of the dynamic program monitor: the instrumentation-based, the dispatch-based, the unoptimized distill-based and the optimized distill-based monitors. We have chosen to evaluate the instrumentation-based implementation because it is the dominant approach of dynamic monitoring on single-core systems. The dispatch-based approach and the distill-based approach are built on similar extensions of multi-core architecture and rely on inter-core communication. The dispatch-based approach and the distill-based approach have the same ability to support a large spectrum of monitoring tasks. The dispatch-based approach is the

state-of-the-art implementation of dynamic program monitor on multi-core platforms prior to the distill-based approach. Compared with the distill-based approach, it is a more straightforward approach that requires an excessive amount of communication. The distill-based approach is exclusively proposed by us and is the most efficient among the four.

4.6.2 Performance of Different Monitor Implementations

Performance of four different implementations of dynamic program monitors performing taint propagation and memory-bug detection is demonstrated in Figure 4.7(a) and Figure 4.7(b), respectively. The bars represent the execution time of the monitors normalized to that of the execution without monitoring. T bars represent the instrumentation-based monitor running on a single-core platform, D bars represent the dispatch-based monitor on multi-core, S bars represent the unoptimized distill-based monitor on multi-core platforms, and O bars represents the optimized distill-based monitor on multi-core platforms. D, S and O bars are broken into the time on fetching from the communication queue, time on executing monitoring functions and time on local computations on the monitor side (this is zero for D and S bars).

The results show that using a multi-core platform and assigning monitored code and monitor onto separate cores can significantly improve the performance of monitoring. As T bars indicate, the instrumentation-based monitor causes a 10.6x slowdown for taint-propagation and a 9.0x slowdown for memory-bug-detection on average. In contrast, D bars indicate that the dispatch-based monitor causes only 7.6x and 6.5x slowdowns for those two monitor tasks, respectively. There are two major factors that contribute to this performance improvement. First, separating the monitor and monitored code onto different cores allows them to execute in parallel so that they do not wait for each other all the time. Second, more hardware resources, such as registers and caches, become available in the multi-core environment, so the monitored



(b) Implementations of memory bug detection monitor

Figure 4.7: **Performance of Four Different Implementations of Monitors**: T bars represent the instrumentation-based monitor running on a single-core platform, D bars represent the dispatch-based monitor on a multi-core platform, S bars represent the unoptimized distill-based monitor on a multi-core platform, and O bars represents the optimized distill-based monitor on a multi-core platform. D, S and O bars are broken into the time on fetching from the communication queue, time on executing monitoring functions, and time on local computations.

code does not compete with its monitor.

The performance of dynamic program monitoring is further improved on a multi-core platform when the dispatch-based monitor is replaced by the distill-based monitor. As D bars demonstrate, the unoptimized distill-based monitor causes only a 3.1x and a 2.6x slowdown for the taint propagation and memory-bug detection monitoring tasks. This performance improvement is ascribed mainly to the achievement of communication reduction. The reduction in communication is reflected as the difference in the time breakdown between the D bars and the S bars. Unlike dispatch-based monitors that forward the results of all instructions in the monitored code, distill-based monitors forward only the results that are relevant to the monitoring purposes. Thus, the communication volume is dramatically reduced.

The differences between the O bars and S bars demonstrate the performance impact of the optimization described in this chapter. As the char shows, the optimization of replacing communication with local computations reduces the slowdown factors to 2.36x for taint-propagation, and 2.18x for memory bug detection. The performance improvement is achieved by eliminating communication of the values that are less expensively locally computed. Specifically, results show that for the S bars and the D bars, the percentage of time spent on communication is reduced by 28% and 22% for taint propagation and memory-bug detection, respectively, but local communication increases by 23% and 20%, respectively. Overall, the proposed optimization leads to a 22% speedup for taint-propagation and a 16% speedup for memory-bug-detection.

429.MCF has shown a relatively lower performance overhead. The optimized dispatch-based monitors show only 1.33x and 1.37x slowdowns for taint-propagation and memory bug detection, respectively. This is because 429.MCF by its very nature suffers a high data cache miss rate even without monitoring. In contrast, its monitor does not suffer a heavy cache miss penalty so that it can easily keep up with the monitored execution without frequently stalling it. The reason monitors have less cache

miss penalty is that though the distill-based monitor is distilled from the monitored program, it does not access the memory space of the monitored program. Instead, it maintains only one or two bits for each byte of the monitored program's memory and thus usually has a much smaller memory footprint than its monitored program.

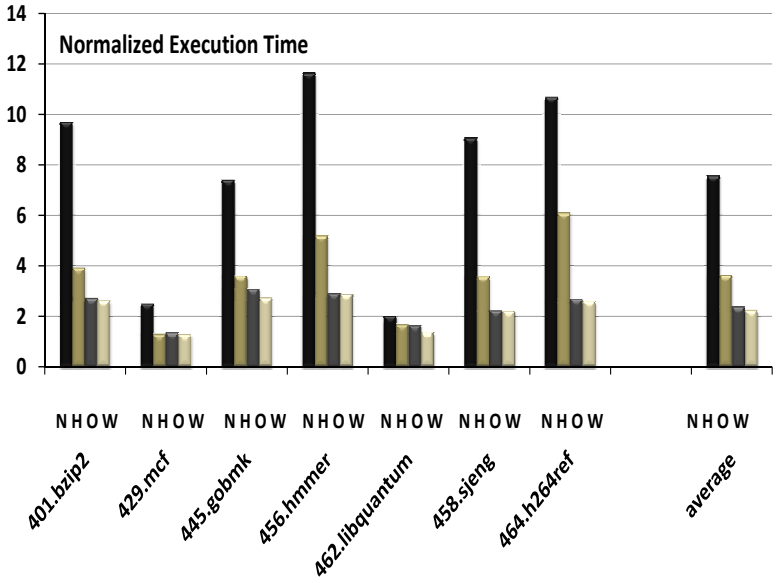


Figure 4.8: **Performance comparison with additional hardware support**:N bars represent the dispatch-based monitor without the two hardware supports; H bars represent the dispatched monitor with the hardware supports; O bars represent the optimized distill-based monitor without the hardware supports; and W bars represent the optimized distill-based monitor with the hardware supports.

4.6.3 Comparison with Hardware-Based Optimizations

Research prior to this dissertation has proposed using specialized hardware supports to optimize dispatch-based dynamic program monitors on multi-core platforms [37, 36, 38]. We have implemented some of those hardware-based optimizations that facilitates communication reduction in our simulation infrastructure and evaluated their performance in comparison to the optimized distill-based implementation. We implemented and evaluated two such hardware supports: inheritance tracking table [34] and communication queue compression [36]. We chose only these two hardware

supports because other proposed hardware supports aim to optimize other aspects of dynamic monitoring systems [38] than communication.

A hardware inheritance tracking table is proposed in [34] to optimize the dispatch-based taint propagation monitor. The key idea is inspired by the fact that the taint status of the register is inherited from the taint status of memory location. Therefore, the taint status of the register does not need to be the actual taint value but the memory address whose taint status the register inherits. Since memory addresses can be known at the monitored code side, the taint status of registers does not need to be forwarded to the monitor if a table is used to keep track of inheritance changes in the taint status of registers. This table is referred to as the inheritance tracking table. The number of entries in the table is the number of architectural registers. Each entry, representing a register's taint status, is actually a memory address. If the new content of a register originates from a memory location, the taint status of that register inherits becomes that memory address, if the new content originates from one register, the taint status of the destination register is copied from the source register when the content of a register is written into the memory. The inheritance tracking table proposed in [34] does not take care of instructions with multiple sources. Since this taint status tracking for registers is done by the table, the communication for the register to register and the memory to register instructions are eliminated. The large percentage of communication eliminated by using the inheritance tracking table for the dispatch-based monitor is just a subset of those eliminated by the distill-based monitor. In the distill-based monitor, register-register taint-propagation is always performed locally by the monitor program without any communication, and since memory addresses are computed with local computations, most of the memory to register instructions do not need communication as well.

Another hardware-based optimization we have implemented is to reduce the cost of communication by compressing forwarded messages. This idea is proposed by Chen

[36]. The core idea is that a hardware logic of compressing/decompressing forwarded messages is incorporated into each CPU core. The communicated messages are compressed on the monitored core, and decompressed on the monitor core so that the average size of messages is reduced. Consequently, the total workload of communication is reduced. Compared to this approach of reducing average message size, the distill-based monitors reduce the size of messages through a more deterministic and effective way: in a distill-based monitor, a message contains only the result produced by the relevant instruction ,while in a dispatch-based monitor a message contains not only the result but also the instruction body.

To evaluate the effects of the two hardware supports, we measured the performance overhead of four different configurations of the taint-propagation monitor. Figure 4.8 shows the execution time of these monitors normalized to that of the original program without monitoring. In this figure, N bars represent the dispatch-based monitor without the two hardware supports; H bars represent the dispatch-based monitor with the hardware supports; O bars represent the optimized distill-based monitor without the hardware supports; and W bars represent the optimized distill-based monitor with the hardware supports. The results show that the hardware supports improve the performance of the dispatch-based monitor significantly, bringing the slowdown factor from 7.6x to 3.6x.

Chapter 5

Parallelize Program Monitoring Using GPGPU

To improve the performance of multi-core-based dynamic program monitoring, the key is to accelerate the monitor running on the monitor core. The hardware support and compiler support reported in previous chapters significantly improve the performance of monitors on the monitor core, thus improving the efficiency of multi-core based dynamic programming. However, as all the monitors presented in previous chapters used only one monitor core, they did not benefit from the potential performance improvement that comes from the parallelization of monitors. In this chapter, we fill this gap by reporting our research on implementing the parallelization of dynamic program monitors in multi-core-based monitoring systems and report the performance improvement gained from this work.

Recently, GPGPU has become an effective architecture for accelerating generic-purpose stream processing programs [80, 81, 82, 83], and the major dynamic program monitors are classic examples of such programs. In a conceptual stream processing program, a small kernel program is applied to many elements of an input stream, and the results of instances of the kernel are independently written to different elements

of the output. In dynamic program monitoring, such as memory-bug detection or taint propagation, the input stream consists of the events of the monitored execution, the kernel program is one of the monitoring functions that performs verification of monitored events, and the output stream is the shadow memory data that record the states of the monitored execution.

The most recent GPGPU design features the integration of GPGPUs and CPU into one die [86], which makes communication latency between CPU and GPGPU cores much smaller. This feature facilitates fine-grained parallelization of dynamic program monitoring that is descried in detail in the rest of this chapter.

The rest of this chapter begins by introducing the particular system architecture that integrates the CPU cores and GPGPU into one die. On this system architecture, we have built the parallelized monitor. We also analyze characteristics of representative monitoring tasks to demonstrate the feasibility of parallelizing dynamic program monitoring. Following that, we present an abstract parallelization framework for dynamic program monitors on GPGPU, as well as its application in implementing GPGPU-based monitors for taint-propagation and memory-bug-detection monitors. After that, we describe how the basic implementation of GPGPU-based monitors can be optimized by customizing them to peculiar characteristics of monitoring tasks, using taint propagation and memory-bug detection as examples. Finally, we conclude this chapter by presenting results of our performance evaluations of optimized GPGPU-based monitors.

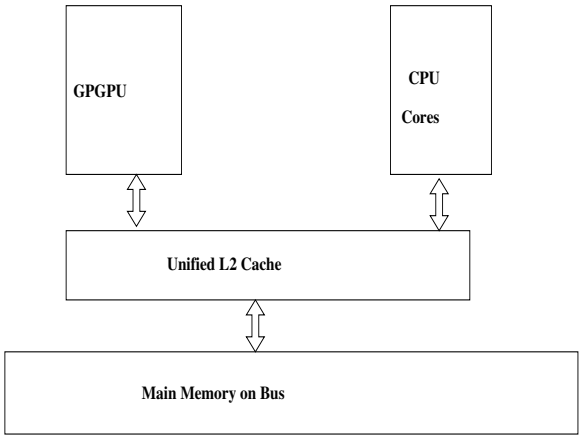


Figure 5.1: **A System Architecture Fusing CPU cores and GPGPU together**: The figure illustrates the system architecture on which we build parallelized dynamic program monitors. All components shown in the figure are integrated into one die. Streaming Processors and CPU cores share a Unified L2 Cache through which CPU cores communicate with the GPGPU.

5.1 GPGPU Architecture to Parallelize Dynamic Program Monitoring

The research work reported in this dissertation simulates a GPGPU architecture. The simulated GPGPU architecture incorporates features of existing GPGPU architectures that effectively serve the need for common applications and are desirable for parallelizing program monitors. In the rest of this section, we begin with an introduction to the architecture, highlighting the features it inherits from Fermi and Fusion. Following that, we discuss characteristics of dynamic program monitoring tasks, showing how we can take advantage of the architectural design presented here to help parallelize them. This discussion uses memory-bug detection and taint propagation as representative monitoring tasks.

5.1.1 Integrating CPU and GPGPU Cores

Figure 5.1 gives an overview of the system architecture. In this system, CPU cores and GPGPU are integrated into the same chip. The idea of fusing CPU cores and GPGPU streaming processors together is proposed and implemented in AMD's APU [86]. The detailed architecture of AMD's Fusion can be found in [86]. However, unlike Fusion, which uses system memory and a special block transfer engine to transfer data from CPU to GPGPU, the architecture proposed here uses a unified L2 cache to transfer data between the GPGPU processors and CPU cores.

The GPGPU side of the architecture contains multiple streaming multiprocessors. A close-up view of a streaming multiprocessor is given in Figure 5.2. The design of the streaming multiprocessor in this architecture refers to Nvidia's Fermi [84]. It contains 8 streaming processor cores that can perform arithmetic and branching operations, and 8 load/store units that perform memory operations. These functional units share a big register file that can have thousands of registers. A streaming multiprocessor also includes 64KB of configurable fast memory, which can be configured as a combination of L1 cache and software-manageable shared memory such as that is in the Fermi architecture.

In the GPGPU architecture described above, CPU cores and GPGPU cores are on the same die area, and thus communication queue entries can be transferred into the GPGPU SMs at a fast rate without much intervention of the CPU core. Moreover, because in the architecture CPU cores are integrated with the GPGPU and its host interface on the same die, thread creation should have a smaller overhead than that in most existing GPGPUs on the market that use a motherboard bus to transfer data from CPU to GPGPU [82].

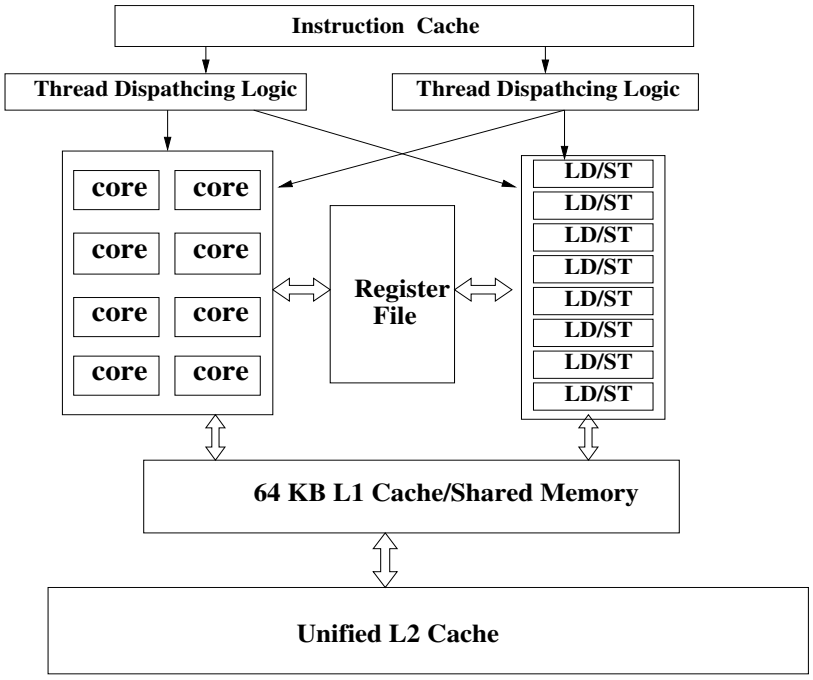


Figure 5.2: Layout of a Stream Multiprocessor: The figure illustrates the system architecture on which we build parallelized dynamic program monitors. All components shown in the figure are integrated into one die. Streaming Processors and CPU cores share a unified L2 cache through which CPU cores communicate with the GPGPU.

5.1.2 Characteristics of Memory-Bug Detection and Taint Propagation

Two common characteristics of taint propagation and memory-bug detection monitoring tasks make them suitable to be parallelized and accelerated using GPGPU architecture. The first is that the core of both tasks are a few small monitoring functions that are invoked repeatedly on different elements of input data, and the second is that invocations of monitoring functions are often parallel. In the rest of this section, we discuss the available parallelism in these two monitoring tasks.

5.1.2.1 Memory-Bug Detection

The monitoring task of memory-bug detection have only two types of monitoring functions to perform all the work of maintaining a shadow of monitored memory space to detect potential memory-related bugs. The first type includes two monitoring functions that set and clear allocation status bits in the shadow memory, which are invoked when library function calls allocate or free heap memory. The other type is the monitoring function that checks the safety of the memory load and stores instructions.

The first type of monitoring function is highly parallel because setting and clearing contents of a shadow memory area is a Single Instruction Multiple Data (SIMD) operation. With many streaming processor cores available in the GPGPU architecture, we can achieve a significant and scalable speedup for this type of monitoring function. Furthermore, as GPGPU cores share register file and L1 cache, reading and writing to shadow memory is much faster than other parallel architectures in which cores share data at L2 cache level.

The second type of monitoring function checks the safety of each memory access and is a good candidate for parallelization. This monitoring function is invoked for each memory access instruction of the monitored execution when the monitor fetches a communication queue entry that represents monitored load or store instruction. Therefore, using GPGPU, the monitor can fetch a number of communication entries simultaneously, and it can create the same number of GPGPU threads running in parallel to verify the corresponding memory access operations.

5.1.2.2 Taint Propagation

Monitoring the task of taint-propagation maintains a shadow memory of the taint status of the monitored core's user memory and architectural registers. Three types of monitoring functions constitute the monitor. The first is the monitoring function

that sets the taint status of affected memory locations whenever a library function call that copies data from untrustworthy sources to user memory during the monitored execution. The second includes the register move, arithmetic and memory instructions of the monitored execution invoke the monitoring function of taint propagation. The propagation function essentially propagates the taint status of one or a few sources of the instruction to the taint status of the destination of the instruction. Finally, there is the monitoring function that checks whether a jump target is tainted whenever an indirect jump instruction is executed in the monitored program.

An invocation to the first type of monitoring function can be implemented as a launch of monitoring threads, each of which sets the shadow taint status bits for a memory word in the monitored execution. In the simulated GPGPU architecture, the thread can be as fast as a cache write. In addition, this type of monitoring function is invoked when memory copy occurs. Therefore, because the copy operation of each memory cell is independent of others, the monitoring thread that sets the taint status of each individual memory word is also independent of each other. In other words, there is no need to check dependence for this type of monitoring function.

The second type of monitoring function does taint propagation work. In monitoring time, it is invoked for each monitored instruction that might propagate taint from its source to its destination. Therefore, naturally, in GPGPU-based monitoring, the propagation monitoring function can be implemented as a kernel, making each invocation to the function a thread of the kernel. As entries in the communication queue of taint propagation monitor correspond to monitored instructions, this design makes sense in that our fused GPGPU architecture for GPGPU threads can fetch entries directly from the communication queue in the shared memory hierarchy, which most of the time is the unified L2 cache.

The last type of monitoring function checks the targets of indirect jumps. To make the check accurate, the invocation to the monitoring function must not occur until

all monitored instructions before the indirect jump instruction have had their taints properly propagated. On the other hand, for the same reason, no taint status change should happen before the invocation to this monitoring function is finished. Failing to meet these requirements leads to inaccurate checking that could be fatal to the monitored execution. Considering that this type of monitoring function essentially does only a comparison, it is neither worthwhile nor safe to make it parallel itself or run it parallel to other type of monitoring function invocations.

5.2 Abstract Framework of GPGPU-based Monitor

In this section, we present a generic abstract framework for implementing a GPGPU-based monitor for various monitoring tasks. We begin with the typical workflow of GPGPU-based monitoring, followed by an overview of GPGPU-based monitoring through three different perspectives. Then, we give a detailed description of the core part of the generic GPGPU-based monitor, the execution of invocations to monitoring functions, referred to in the following text as the monitoring thread. Finally, we give a brief performance analysis to identify the challenges a GPGPU-based monitor might have to solve to make it useful, which serves as a prelude for the next section focusing on implementation of GPGPU-based monitors for memory-bug detection and taint propagation.

5.2.1 GPGPU-based Monitoring

At the top level of the entire monitoring system, the workflow of the GPGPU-based monitoring are not different than other conventional multi-core-based monitoring systems except that the monitoring core offloads the major part of the monitoring job to

the GPGPU. The top-level workflow is illustrated in Figure 5.3. It is noteworthy that on the CPU core side, the GPGPU-based monitor uses exactly the same hardware support (i.e. extraction logic) described in previous chapters as other multi-core-based monitors.

Figure 5.3 demonstrates three important aspects in GPGPU-based monitoring: monitoring information extraction, monitoring thread creation, and monitoring thread execution. These aspects reflect the chronological order of processing a monitored event using GPGPU-based monitoring. The first aspect, information extraction is performed by the same extraction logic as the conventional multi-core-based monitors described in previous chapters. The second aspect, monitoring thread creation, is executed by a CPU core, referred to as the monitor core. The monitor core keeps polling the communication queue; when the number of valid entries in the queue reaches a threshold, the monitor core creates threads running on the GPGPU, and each GPGPU thread, referred to as monitoring thread in the following text, processes a queue entry. The third aspect, execution of monitoring thread, is performed in the GPGPU cores.

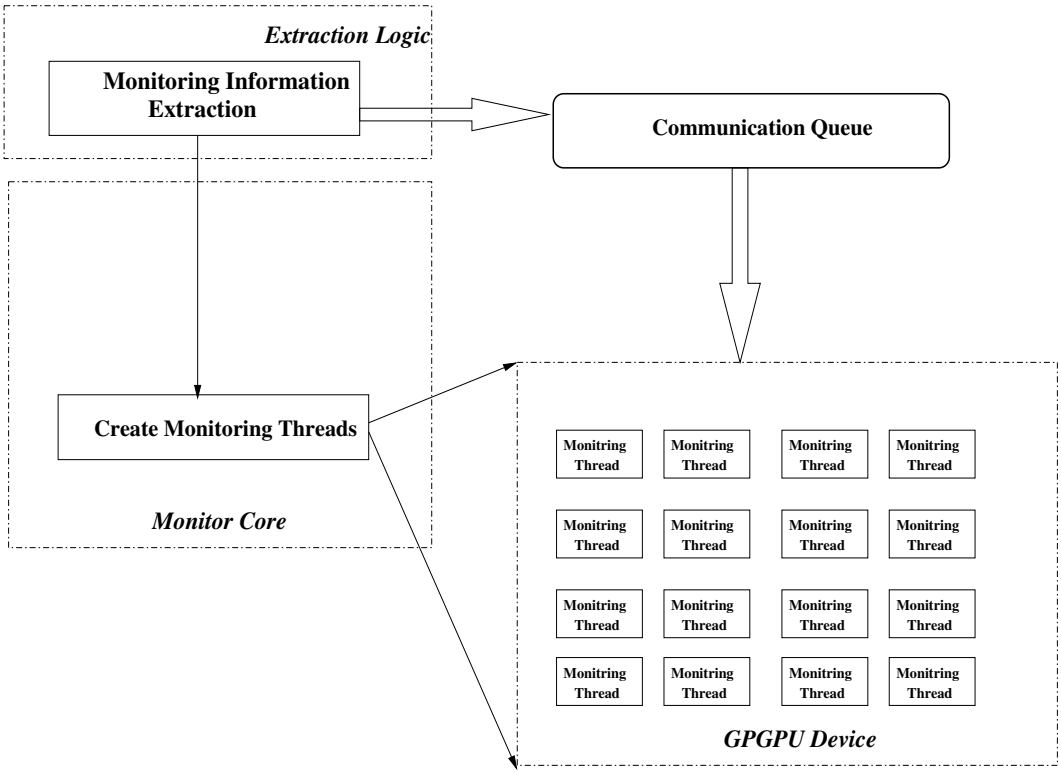


Figure 5.3: GPGPU-based Monitoring: Information extraction is done by extraction logic on the monitored core side. The monitor core fetches only information from the queue, copies it to GPGPU, and launches monitoring threads on the GPGPU to perform checking.

As the Figure 5.3 demonstrates, the role that the monitor core plays in GPGPU-based monitoring is different from that in other multi-core-based monitors mentioned in previous chapters. Here the monitor core works as a bridge between the monitored core and the GPGPU. Although this might introduce extra overhead to GPGPU-based monitoring, it is dispensable to GPGPU-based monitoring because state-of-the-art GPGPU cannot create and launch GPGPU threads by itself.

```
data declarations:
enum MONITOR_FLAG
    {POLL,ERROR...,EXIT }
typedef struct message_struct
    {Type, Address, Value, I/D flag }
\\ messages fetched from the queue
message_struct msg_array [Q_LENGTH/2];
static MONITOR_FLAG mon_flag;
message_struct *specialMsg = NULL;

algorithm GPGPU-based monitor:
\\ Initialize monitor flag
mon_flag = POLL;
while (mon_flag == POLL)
 msg_array = ReadQueue();
 if (msg_array != NULL)
  \\ create and launch monitoring threads
  CudaConfigureCall(launchParameters,...,ShadowMem, msg_array)
  CudaLaunch("mon_func_thread");
  \\ barrier to make sure monitoring threads are all done
  cudaThreadSynchronize();
 end if
 \\ Special message may require additional processing
 if (specialMsg != NULL)
  ProcessSpecialFlag(specialMsg, msg_array);
 end ifend while
ProcessErrorOrExit(mon_flag);
end algorithm
```

Figure 5.4: Algorithm for a GPGPU-based Monitor.

5.2.2 Overview of the GPGPU-based Monitor

The GPGPU-based monitor is the software part of the GPGPU-based monitoring system that implements all steps except the information extraction which is the responsibility of the hardware support. Figure 5.4 gives an algorithm view of the GPGPU-based monitor code running on the monitor core. The core of the monitor is a while loop of which each iteration fetches items of half the communication queue and launches the same number of monitoring threads that each verifies a fetched item. In the following paragraphs, we will introduce the GPGPU-based monitor from the perspective of the hardware usage, the way it is programmed and its monitoring model. In addition, we compare the GPGPU-based monitor with other multi-core based monitors introduced in previous chapters.

The GPGPU-based monitor shown in Figure 5.4 uses both a CPU core and a GPGPU. The pseudo-code shown in Figure 5.4 is the code running on the monitor CPU core. This code creates and launches monitoring threads on a GPGPU. An abstract description of the monitoring thread of the kernel function mon_func_thread can be found in Figure 5.5, which we will describe in detail later.

The GPGPU-based monitor shown here is written by programmers based on monitoring requirements in the pseudo-code of CUDA-C [91]. CUDA-C is an extension of C programming language that provides programming interfaces to implement kernel threads running on the GPGPU and coordinate activities on the CPU core and the GPGPU. CUDA APIs used in the GPGPU-based monitor shown in Figure 5.4 are: cudaThreadSynchronize which sets up a barrier to ensure all previous activities on GPGPU-device are finished; CudaConfigure which configures the parameters of a CUDA launch of monitoring threads; and CudaLaunch, which creates and starts monitoring threads running on the GPGPU. Notice that because we use the fused

GPGPU architecture, we include no concept of device memory and no call to cudaMemcpy. At the stage of research reported by this dissertation, the compiler support described in chapter 4 is not yet used in the GPGPU-based monitor.

The GPGPU-based monitoring model resembles the dispatch-based monitoring model in the way that it uses the table-driven mode of the extraction logic, takes typed items from the communication queue and dispatches them to invoke monitoring functions. However, compared with conventional dispatch-based monitoring, GPGPU-based monitoring is parallel in every step on the monitor side: it fetches queue items en masse, decodes items with monitoring threads that run parallel in the GPGPU, and executes monitoring functions embodied in monitoring threads in parallel. Because of this full-scale parallelization, the GPGPU-based monitor introduces elements that are specific to parallelization, such as barriers. In addition, due to the GPGPU's limits, when a monitoring thread finds an error or detects the end of monitoring, it cannot handle this by itself and thus just ask code on the monitor processor core to handle it.

Table 5.1 summarizes important aspects of GPGPU-based monitoring and compares it to other monitor models mentioned in previous chapters.

Table 5.1: Comparisons of Characteristics of Different Implementations of Monitors

Monitor Model	Hardware Usage	Programming Model	Parallel	Monitor Compiler Support
Instrumentation-based	Single Core	Instrument monitored program	No	No
Dispatch-based	Dual Core with extraction logic	Written separately according to monitoring requirements	No	No
Distill-based	Dual Core with extraction logic	Distilled from the monitored with monitoring requirements	No	Yes
GPGPU-based	Dual Core with extraction logic and GPGPU	Written separately in language supporting GPGPU	Yes	No

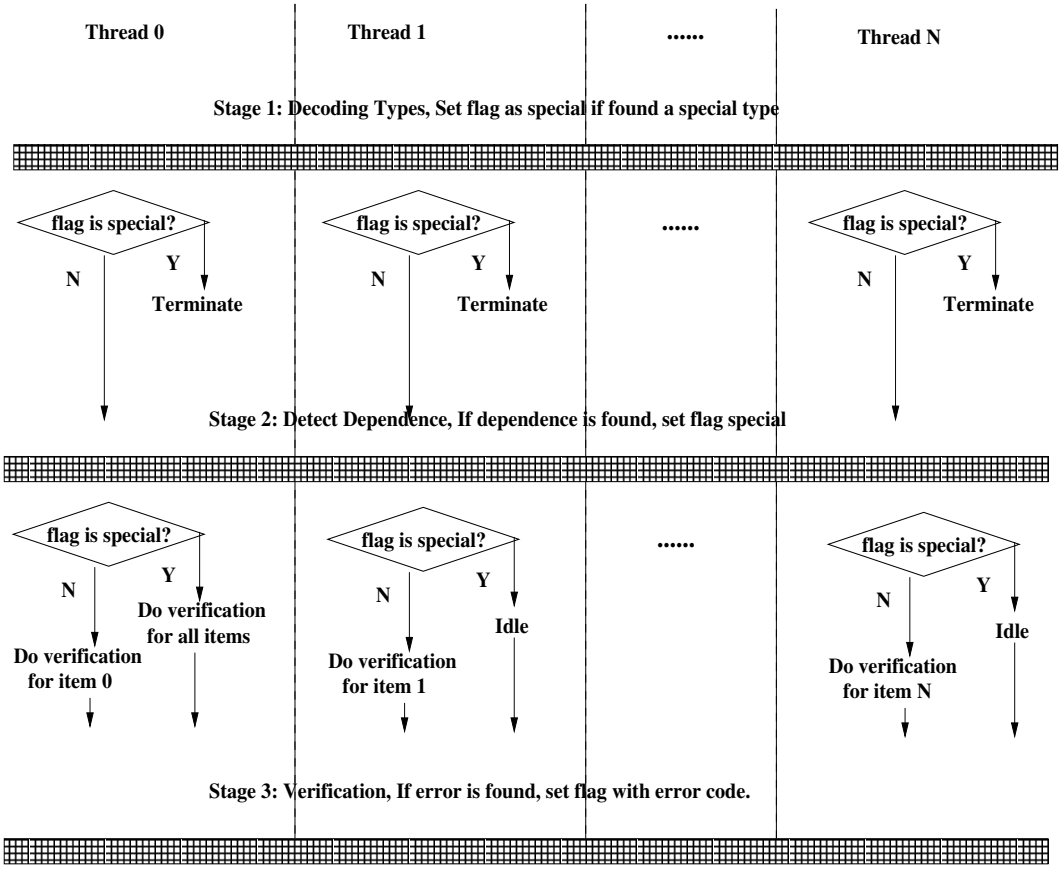


Figure 5.5: **General framework of monitoring thread implementation**. The framework features three phases of execution. Phases are bound by synchronization barriers represented by shaded bars in the diagram. The first phase decodes message items sent from the extraction logic of the monitored core. If there is a message in special type that requires redoing the launch of threads, all threads terminate. The second phase detects dependence among items. If there is no dependence, each thread verifies an item in parallel to one another. Otherwise, thread 0 verifies for all items while other threads become idle. The final phase collects results of the verification. If any error is reported by any thread, the flag will be set with error code and thus return to the GPGPU-based monitor.

5.2.3 Implementation of Monitoring Thread

In the context of this dissertation , the monitoring thread refers to the implementation of the monitoring function that runs on the GPGPU. The monitoring thread is not merely a simple migration of monitoring functions in other multi-core-based monitors

to GPGPU architecture. The implementation of monitoring thread must solve two challenging problems: dependence resolution and parallel decoding. Dependence resolution involves detecting and resolving dependences among monitoring threads that are launched in the same batch, referred to as a grid in CUDA and the following text. Parallel decoding is a challenge because some messages that are initially assigned to one monitoring thread may actually require multiple monitoring threads for better performance. For instance, events in types of *alloc* and $free$ in memory-bug detection changes the shadow memory states of many independent memory locations simultaneously. Therefore, by their nature they should get multiple monitoring threads for each event of these types even though it has just one entry on the communication queue.

The rest of this section is organized as follows: First, we begin with the general framework of a monitoring thread implementation that could be adapted to most monitoring tasks. Then, we describe in detail how this framework can be applied to implement monitoring threads for two representative monitoring tasks: memory-bug detection and taint propagation.

5.2.3.1 Abstract Framework of Monitoring Thread

Though implementation of the monitoring thread could vary across monitoring requirements, it follows a common general framework that features a phase-by-phase procedure. Figure 5.5 demonstrates the common framework of monitoring thread implementation. As the figure shows, a monitoring thread is divided into three chronological phases: type decoding, dependence detection and verification. Phases are divided by thread synchronization primitives to guarantee that all threads are in the same phase all the time. The design of this monitoring thread framework provides an effective solution to the challenges of dependence resolution and parallel decoding. In the following paragraphs, we describe the activities of these three phases in detail.

During the type decoding phase, each monitoring thread decodes a queue item assigned to it. In a general monitoring thread implementation, each queue item is decoded by one and only one monitoring thread in this phase. If a monitoring thread decodes an item and finds that it is a special type message (for instance, an *alloc* in memory-bug detection) that requires multiple monitoring threads of a special type to execute its corresponding monitoring activity, the monitoring thread sets a special word with the address of the queue item. This special flag is shared among all monitoring thread. When multiple special items are detected, the one that has the lower thread ID gets the priority to write its queue item address into the flag. At the end of this stage, after all threads come to the synchronization barrier where the special word is checked. It the word contains a valid queue item address, all threads are terminated, and the address is copied to the GPGPU-based monitor's main function so that the monitor will know which queue item represents a special event and thus redo the monitoring thread creation in a special way.

The second phase is to detect and solve dependences among monitoring threads. The details of dependence detection are specific to monitoring tasks since the definition of dependence varies across monitoring tasks. A monitoring thread should be notified if another thread is accessing the same entry in the shadow memory. The first step of dependence detection for each monitoring thread is to detect if there are multiple threads accessing the same shadow memory location. Once a conflict is detected, a shared flag is set to indicate that there is a conflict among monitoring thread and all the items must be verified sequentially. At the end of the dependence detection phase, all threads are synchronized at a barrier. Therefore, in this phase, we solve the problem of dependence detection.

The last phase verifies items in the queue simultaneously if no dependence is detected; otherwise thread 0 verifies all queue items. By this mechanism, we solved the problem of dependence resolution.

Table 5.2: Dependence in Memory-bug Detection

Dependence Type of Monitored Operations	Is a Dependence of Monitoring Threads	Activities on Shadow Memory
Read After Read	False	Both checks the alloc and the init status
Read After Write	True	Write reads the alloc status and might set the init status, Read reads both.
Write After Read	True	Same as above
Write After Write	False	Both operations read alloc status and might set init status

5.2.3.2 Implementation of Monitoring Thread for Memory-bug Detection

In this section, we describe details of monitoring thread implementation for memory-bug-detection, strictly following the general framework mentioned above without any optimization. We begin the section by introducing data structures used by memory-bug-detection monitoring thread, then discuss implementation details of each phase in memory-bug detection monitoring thread implementation that use and maintain these data structures.

The data structure used by the memory-bug detection monitoring thread consists of three types of shadow memory. Like other memory-bug detection monitors mentioned in previous chapters, the monitoring threads of GPGPU-based memory-bug detection monitor maintain two types of shadow memory: allocation state shadow memory and initialization state shadow memory. Each shadow memory bit of shadow memory records the state of a memory word in the monitored program's user memory space. For the allocation state shadow, these are the allocation status, the initialization state shadow, the initialization state. These two shadow memory areas are designed for verification purposes. In addition, the GPGPU-based memory-bug detection monitor introduces an additional type of shadow memory for the purpose of dependence detection: touch shadow memory. Each bit of touch shadow memory

corresponds to a memory word in the monitored program's user memory space, indicating for a particular grid of monitoring thread, whether other two shadow states of corresponding memory word might be changed or not. We will further explain the use of touch bit in detail when we discuss dependence detection.

We dedicate the rest of this section to three technical details that are particular to the monitoring task of memory-bug detection: events of special type, definition of dependence, and implementation of verification. These details are the technical content the of three phases of monitoring thread for memory-bug detection.

Special Type Events in Memory-bug Detection:Allocation and Free

In memory-bug detection, special type events are invocation of allocation and free functions . When a memory-bug detection monitor detects an allocation event, it sets corresponding allocation bits and clear corresponding initialization bits. For free event, the monitor checks if the corresponding allocation bits are set to detect a double-free bug and clears the bits if no bug is detected. Allocation and free events usually involve multiple memory words and multiple status bits. Therefore, to take full advantage of parallelism, each allocation and free event should be handled by as many monitoring threads as the number of monitored memory words it touches.

Dependence of Monitoring Thread for Memory Read and Write in Memory-bug Detection

In the context of memory-bug detection, dependence detection is solely designed for monitored operations of memory read and write because allocation and free are detected by the earlier phase. Each monitored memory read or write operation in monitored execution has one and only one monitoring thread. Different monitored memory read/write operations may access the same memory address that translated into accesses to same shadow memory location, which might cause dependence violation and thus incorrectness of monitoring threads. However, dependence between monitored operations of memory read and write may not translate into dependence of

their correspondent monitoring thread. Table 5.2 describes the types of dependences for monitored memory read and write in the context of memory-bug detection. The table shows that monitoring threads for memory read or write may change only initialization status bit so that dependence seems to be decided by the order of read and write on the initialization bit. However, unlike conventional dependence definition, WAW is not considered as a true dependence in the context of memory-bug detection because the content to be written to the initialization status bit is the same.

Touch shadow memory is used to detect dependence of a monitoring thread in memory-bug detection. Each bit of touch shadow memory corresponds to a monitored memory word. During the dependence detection phases of the monitoring thread, if a thread changes the initialization shadow bit of a monitored memory word, it sets the corresponding touch shadow bit as well. After the phase of dependence detection, each thread comes to a synchronization barrier that guarantees that every thread finishes deciding whether to set its correspondent touch bit. Right after this barrier, at the beginning of the next phase of verification, every thread checks whether the touch bit of its corresponding monitored memory word is set. If a monitoring thread for monitored memory read finds that the touch bit is set, this means there is an initialization status change by another monitoring thread for a monitored write. As this suggests a possible RAW or WAR hazard among monitoring threads, the verification work of monitored threads should be serialized. At the end of the execution of all monitoring threads, touch bits are cleared to prevent them from affecting the next grid of monitoring threads.

5.2.3.3 Implementation of Monitoring Thread for Taint-Propagation

This section gives details of the implementation of monitoring threads for taint-propagation. As in the previous section, we begin by introducing shadow memories, then focus on the technical details of three different phases of the monitoring thread:

Table 5.3: Behavior and Activities of Monitored Events in Taint Propagation

Event Type	Taint Status Change	Cause Security Breach	Monitoring Activity
area-taint	change taint status of multiple memory words	not directly	set taint bits of affected memory area
indirect jump	no taint status change	may directly cause security breach	check taint status of the jump target area
propagation event	change taint status of single item	not directly	propagate taint

special events, dependence definition, and implementation of verification.

The data structure used by a taint propagation monitoring thread consists of two types of shadow memory: taint shadow memory and touch shadow memory. Taint shadow memory keeps taints of memory words and registers in the monitored execution. A bit of taint shadow memory is set when the corresponding memory word or register is tainted by being written with the content from non-trusted sources. In contrast, a taint bit is cleared when a constant value is written into the correspondent memory word or register. In addition to these cases, taint bits are propagated during a program's execution. For example, an unary instruction copies the taint bit of the source to that of the destination, and a multiple-source instruction should set the taint bit of the destination as the result of *or* of taint bits of all sources. Taint bits are examined for targets of indirect jumps because executing code that originates from untrustworthy sources might pose a security breach. A detailed description about taint-propagation shadow memory use can be found in chapter 3 or other literature [6, 13]. The touch shadow memory in taint-propagation serves for purpose of dependence detection and is similar to the touch shadow memory in memory-bug detection.

Special Type Events in Taint-Propagation: Area-taint and Indirect jump

There are two types of special events in taint-propagation: area-taint and indirect jump. In this context, area-taint event refers to those library calls that receive data

from untrustworthy sources to the monitored execution's memory space. The area-taint event requires the monitor to set taint bits of an area of shadow memory. Indirect jump event refers to monitored jump instructions whose target might be a tainted memory and thus the monitor must check the taint bit of the target for safety.

Area-taint event and indirect jump event are special in taint-propagation because for all other events the monitor propagates one single taint bit. For area-taint event, the GPGPU-based monitor creates one or more grids of monitoring threads to set multiple taint bits in parallel. For indirect-jump event, the monitor creates only one monitoring thread in the grid to check the target address.

Dependence of Monitoring Thread in Taint Propagation

To detect dependence among taint-propagation monitoring threads, the monitor uses a touch shadow memory. Each bit of shadow memory corresponds to a memory word or register in the monitored program. During the dependence detection phase, each monitoring thread sets the touch bit of the destination register or destination memory word of its corresponding taint propagation event, which could be a memory reference instruction, arithmetic instruction or data movement instruction. A synchronization barrier at the end of this phase guarantees that all monitoring threads finish setting their touch bit. Right after the barrier, each thread checks whether the touch bits of any source or the destination of the corresponding monitored instruction is set. If any such touch bit is set, that indicates that the monitoring thread might have a dependence with some other thread in the same grid. In this case, the entire grid of threads has to be executed sequentially by thread 0. At the end of each monitoring thread, each monitoring thread clears the touch bit of the destination of its corresponding monitored instruction to prevent it from confusing the next grid of monitoring threads.

5.3 Optimization of GPGPU-based Monitors

The performance overhead of the basic version of GPGPU-based monitor for taint-propagation and memory bug detection is prohibitively high due to the high volume of dependence detected. Figure 5.6 presents a measurement of the performance overhead of the basic version of GPGPU-based monitors for both memory-bug detection and taint propagation. The performance statistics shown in Figure 5.6 are collected using the same simulation infrastructure reported in chapter 4. The GPGPU integrated with CPU cores is simulated using the GPGPU-Sim simulator [97] with parameters described in Table 5.5. Figure 5.6 compares the performance overhead of the basic version of GPGPU-based monitors for taint propagation and memory-bug detection with other implementations including the instrumentation-based, the dispatch-based, and the optimized distill-based monitor we have described in previous chapters. Bars represent the execution times of the monitors normalized to that of the execution without monitoring. The performance of the basic GPGPU-based monitor is significantly worse than any of the other monitor implementations. For taint-propagation, its reaches 18.97x, and 13.21x for memory-bug-detection.

This huge performance overhead of the basic GPGPU-based monitor is caused by the excessive amount of dependence detected among monitoring threads. Dependence among monitoring threads, once detected, causes all monitoring threads to run sequentially. In sequential mode, the GPGPU-based monitor not only completely loses its advantage of verifying multiple queue items in parallel, but also performs significantly slower than CPU-based monitors in the verification of individual verification job. This is due to several factors. First, in CPU-based monitor, the shadow memory content and queue items can be cached in L1 cache. In contrast, the GPGPU-device has no cache to cache shadow memory content, and thus every access to shadow memory and queue item in a monitoring thread is actually a device memory access

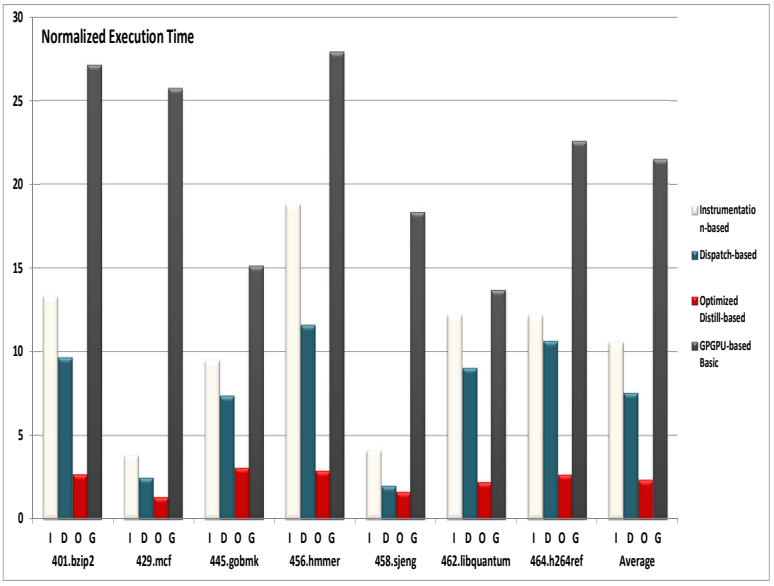
that is much slower than CPU cache access. Second, the state-of-the-art CPU cores usually have a faster clock rate than GPGPUs, in our simulation, the CPU core runs at 1.5GHz, while GPGPU runs at 1.0GHz. Last but not least, another important factor is that CPU core has a super-scalar pipeline with sophisticated features like branch predication, speculative execution, etc. In contrast, GPGPU execution engines usually come with a much simpler design. All these factors make individual monitoring threads much slower than their CPU-based counterparts. Moreover, besides verification, GPGPU-based monitor perform phases of special type detection and dependence detection that are not needed in CPU-based monitors. The performance overhead caused by these phases, though usually lightweight, is an extra burden on the GPGPU-based monitor.

We conducted experiments to measure the amount of dependence in basic GPGPU-based monitors. In these experiments, for each benchmark, we count the number of monitored events that are verified sequentially for each benchmark, along with the total number of monitored events sent to monitors to verify out of 100 millions executed instructions. The results for taint propagation and memory-bug detection monitors are listed in Table 5.4.

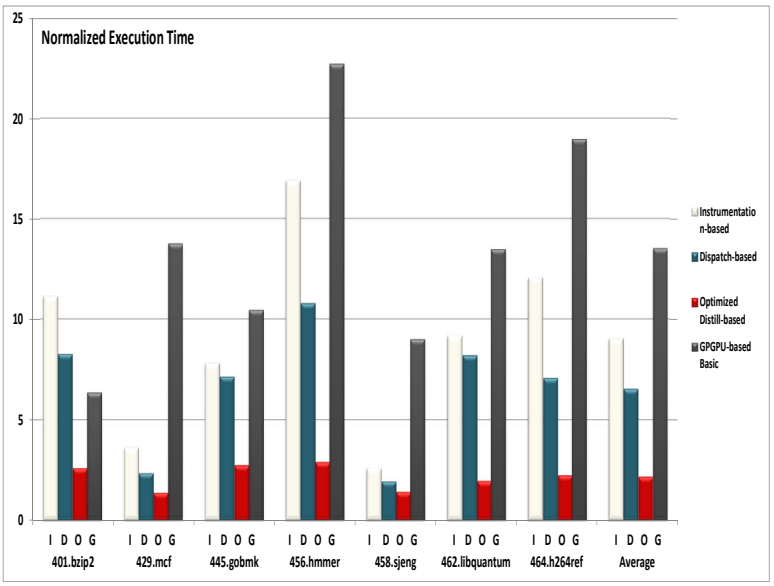
Table 5.4: Performance Statistics of Basic GPGPU-based Monitor

benchmark	taint-propagation: percentage of sequentially executed threads	taint-propagation: Basic GPGPU-based monitor performance overhead	memory-bug detection: percentage of sequentially executed threads	memory-bug detection: Basic GPGPU-based monitor performance overhead
401.bzip2	99.88%	24.69x	99.96%	5.84x
429.mcf	95.90%	23.87x	100.00%	13.28x
445.gobmk	97.98%	14.40x	69.03%	10.45x
456.hmmer	99.80%	23.02x	99.69%	22.48x
458.sjeng	74.41%	16.00x	96.58%	8.90x
462.libquantum	73.55%	13.23x	78.91%	13.22x
464.h264ref	99.98%	15.56x	99.62%	18.34x

Dependence detected by the general monitoring framework is extremely high for most of the benchmarks: for both monitoring tasks, almost all benchmarks have over 70% of monitored events being verified sequentially. Actually, except 458.sjeng and



(a) Performance of Basic Version of GPGPU-based Taint Propagation Monitor



(b) Performance of Basic Version of GPGPU-based Memory-bug Detection Monitor

Figure 5.6: Performance of basic GPGPU-based monitors in Comparison to Other Taint Propagation Monitors: Performance is measured in execution time of monitors normalized to the execution of monitored program without monitoring. For each benchmark, four bars represent normalized execution time of four different types of monitors:I bars represent the instrumentation-based monitor; D bars represent the dispatch-based monitor; O bars represent the distill-based monitor with optimization; and G bars represent basic the GPGPU-based monitor without optimization

462.libquantum in taint propagation and 462.libquantum and 445.gobmk in memory-bug detection, all other benchmarks in both monitoring tasks have 95% of monitored events being sequentially verified.

5.3.1 Techniques of Optimizing GPGPU-based Monitor

Based on the above analysis of performance bottleneck, the key to optimize GPGPU-based monitor is reducing the amount of dependence detected by the monitor. As we cannot change the definition of dependence inherent in monitoring requirements, one effective way to reduce the amount of dependence is to identify monitoring threads whose verification work could be ignored and thus cannot cause true dependence violations.

In the context of this dissertation, verification of a monitored operation includes two parts: verifing the safeness of the operation the thread represents and changing shadow bit(s) of the memory word or register that the operation accesses. Therefore, if an operation is proven to be safe and does not change the status of the memory word or register it accesses, there is no need for it to be verified. In summary, a monitoring thread is considered redundant if it meets two requirements:

- The monitored event represented by the thread is guaranteed to be safe by the existing program state.
- The monitored event represented does not change the shadow status of any monitored memory word or register.

If a monitoring thread meets both requirements, ignoring it does not affect the correctness of the monitor. Monitoring threads that perform redundant verification work should be prevented from entering dependence detection and verification phases, i.e., redundant monitoring threads should terminate earlier. However, whether a monitoring thread meets two requirements is decided solely by the characteristics of the

monitoring task. Hence the abstract framework described in Figure 5.5 does not contain detection of the redundancy, nor other additional ingredients of monitoring threads that exploit specific characteristics of monitoring tasks. In the following text, we analyze the monitoring tasks of memory-bug detection and taint propagation respectively, and propose ideas to reduce the amount of dependence detected.

5.3.1.1 Optimizing Memory-bug Detection Monitor on GPGPU

Optimization of the memory-bug detection monitor can be achieved by detecting redundant monitoring threads. In the context of memory-bug detection, the requirements for redundancy listed above are embodied in the following statements.

- Read from a memory word with a set initialization bit is always safe.
- Read from a memory word does not change any shadow status of the word.
- Store to a memory word that is initialized is always safe.
- Store changes initialization status only if it is not previously set.

To detect redundant monitoring thread in memory-bug detection, we add the concept of redundancy detection into the monitoring thread framework described in Figure 5.5. This redundancy detection checks redundancy according to the four statements listed above. If a monitoring thread is found to be redundant, it should be terminated earlier and not change the shadow touch bit of its corresponding memory word. This way, a redundant monitoring thread will never be considered in any dependence.

However, examining the statements in the monitoring thread individually cannot detect all redundant threads. To be complete, redundancy detection has to consider the chronological order of threads in the same grid. For example, suppose in a monitoring thread launch, there are two monitoring threads m and n with $m < n$, which means that thread m is earlier than thread n; the monitoring thread m represents a

```
data declarations:
enum MONITOR_FLAG
    {NORMAL,ALLOC_ERR, INIT_ERR,EXIT }
typedef struct message_struct
    {Type, Address, Value, I/D flag }
message_struct msg_array[Q_LENGTH/2];
\\ flag used in this phase
MONITOR_FLAG mon_flag = NORMAL;
algorithm
...
\\ check if the memory address is allocated
if (allocShadow[msg_array[thread_id].Address] is not set)
 mon_flag = ALLOC_ERR;
 return
end if
cudaThreadSynchronize();
\\ if an error is detected, all threads stop
if (mon_flag != NORMAL)
 return
end if
\\ reset touch Shadow
touchShadow[msg_array[thread_id].Address] = MAX;
\\ if previously initialized, no need for verification
if (initShadow[msg_array[thread_id].Address] )
 return
end if
cudaThreadSynchronize();
\\ a write operation may change initialization bit
if (msg_array[thread_id].Type == WRITE)
 if (touchShadow[msg_array[thread_id].Address] > thread_id)
  \\ touchShadow should always be the earliest thread ID
  touchShadow[msg_array[thread_id.Address] = thread_id
 end if
 Set initialization bit of initShadow[msg_array[thread_id].Address];
 return
end if
\\ all threads synced to make sure touchShadow has the earliest ID
cudaThreadSynchronize();
if (touchShadow[msg_array[thread_id].Address] < thread_id )
 \\ initialized by an early thread, this thread is redundant
 return
else
 \\ this is the case of uninitialized read
 mon_flag = INIT_ERR;
end if
end algorithm
```

Figure 5.7: Algorithm of Optimized Memory-bug Detection GPGPU-based Monitor.

write operation to a memory word v, and the monitoring thread n represents a read operation to the same memory word v. If the initialization bit of v is not set before the launch of the grid, individually examining the four statements for thread n would not consider n redundant. However, in reality, since the thread n comes after the thread m that sets the initialization bit, the verification of monitoring thread n is redundant. Since this type of redundancy is caused by monitoring threads inside the same launch, we refer to this type of redundancy as in-grid redundancy.

In order to detect redundancy completely, we change the design of the touch shadow memory to incorporate information on the ordering of the monitoring thread. To each monitored memory word, we store a monitoring thread ID instead of a touch bit in the touch shadow memory. This monitoring thread ID is the ID of the earliest monitoring thread that writes to the corresponding memory word in a monitoring thread grid. Furthermore, in the context of memory-bug detection, since the earliest write operation makes all following operations to the same memory word safe regardless of what those operations are, this earliest thread ID information is even sufficient for the monitoring thread to perform verification without dependence detection.

Therefore, based on this new design of touch shadow memory, we redesigned the monitoring thread of memory-bug detection. Now we change all phases after the special-type detection phase with the algorithm shown in Figure 5.7. The algorithm can be divided into four stages: detection of unallocated access (a.k.a dangling pointer), detection of redundancy, setting touch shadow memory, and detection of in-grid redundancy and uninitialized access. These stages are demarcated by synchronization barriers implemented as calls to API cudaThreadSynchronize to make sure all threads are always in the same stage.

The first stage of the algorithm detects the bug of unallocated access, a.k.a dangling pointer. Since the algorithm starts after the special-type detection phase, all

monitoring threads executing this algorithm are of a common type, represent either a write operation or a read operation, and not set the allocation status of the memory word. Therefore, the dangling pointer bug can be detected by looking up the existing allocation shadow bit that corresponds to the memory word that each thread accesses. If there is a thread that accesses a memory word without the allocation bit set, it represents an access to unallocated memory, which causes a dangling pointer bug. The algorithm then assigns the error code $ALLOC\_ERR$ to the monitor flag that corresponds to the dangling pointer bug.

At the beginning of the second stage, the monitor flag is checked to see if a dangling pointer bug is detected by the previous stage. A detected bug should stop execution of all monitoring threads. After this check, the algorithm initializes the touch shadow for the accessed memory word, setting it to a number, referred as MAX ID, that is always greater than any executing thread ID. If an accessed memory word has its touch shadow as the MAX ID, it means that no thread in the current grid has changed the shadow status of it. Finally, this stage also checks the initialization bit in the initialization shadow memory for the accessed memory word. If the initialization bit is set, the thread is redundant because the operation it represents is safe and will not change any shadow status of the accessed word. A redundant thread should return immediately once it is detected.

The third stage sets up touch shadow memory. Accessed memory words will have the ID of the earliest monitoring thread that changes its initialization bit stored in its shadow memory after the stage. Only threads that represent write operations may change the initialization bit, and thus the algorithm compares just the thread ID of each one of those monitoring threads with the ID stored in the touch shadow, and sets the shadow with the smaller of the two IDs. After all threads have finished this comparison and setting, the thread ID in the touch shadow that corresponds to the accessed memory word is the ID of the earliest thread that truly changes its

initialization bit, because operations that access previously initialized memory word have already quit as early detected redundant threads in the previous stage. Also in this stage, the initialization bits of accessed memory words of those write operation threads are all set, and all write monitoring threads return at the end of the stage.

Threads that remain in the last stage all represent read operations. They are either redundant or represent a bug. A remaining read thread is redundant if there is early write thread that initializes the memory word this thread reads; a remaining thread represents a bug if the corresponding initialization bit is not set. Redundant threads safely return with the monitor flag remaining normal, but a buggy monitoring thread sets the monitor flag with the error code $INIT\_ERR$, indicating it has found an uninitialized read.

In summary,the optimized GPGPU-based monitoring of memory-bug detection using the algorithm is fully parallel, and, compared to the generic monitoring thread algorithm using the abstract framework, has no sequential verification in it at all. Moreover, though it has four stages, it is very likely that all monitoring threads will finish as early as the end of the second stage, because it should not be uncommon for a monitored program to have a sequence of read and write instructions that access only initialized memory.

5.3.1.2 Optimizing Taint Propagation Monitor on GPGPU

The performance of basic taint propagation monitoring on GPGPU can be improved by redesigning monitoring threads to exploit the characteristics of the monitoring task. Nevertheless, applying the idea of eliminating redundancy does not work for taint-propagation. Suppose we refer to redundant monitoring threads in the context of taint-propagation as those threads that propagate to their destinations the same taint value as their previous taint value. In taint propagation, it is difficult if not altogether impossible to improve performance by detecting and terminating

```
data declarations:
typedef struct message_struct
    {Operand1, Operand2, Destination }
message_struct msg_array[Q_LENGTH/2];
\\ flag used in this phase
bool taintOP1, taintOP2, taintDest;
algorithm Optimized Taint-propagation GPGPU-based Monitor:
...
touchShadow[msg_array[thread_id].Destination] = MAX;
cudaThreadSynchronize();
while(true)
\\ set touch shadow to the earliest ID that may change it
 if (touchShadow[msg_array[thread_id].Destination] > thread_ID)
  touchShadow[msg_array[thread_id].Destination] = thread_ID;
 end if
 \\ taint status of source operands
 taintOP1 = taintShadow[msg_array[thread_id].Operand1];
 taintOP2 = taintShadow[msg_array[thread_id].Operand1];
 cudaThreadSynchronize();
 \\ test whether this thread depends on others
 \\ if not, this thread can finish now
 if (touchShadow[msg_array[thread_id].Operand1 >= thread_ID)
  && touchShadow[msg_array[thread_id].Operand2] >= thread_ID)
  \\ finish propagation here
  taintShadow[msg_array[thread_id].Destination] = (taintOP1 || taintOP2);
  \\ reset touch Shadow to MAX
  touchShadow[msg_array[thread_id].Destination] = MAX;
  return;
 end if
 cudaThreadSynchronize();
end while
end algorithm
```

Figure 5.8: Algorithm of Optimized Taint Propagation GPGPU-based Monitor.

early redundant monitoring threads. Unlike memory-bug detection monitoring, most redundant monitoring threads in taint-propagation cannot be detected by examining monitoring threads simultaneously. The reason is that for most monitored events (usually instructions), the taint values to be propagated to the destination are calculated from taint values of source operands. As the taint values of sources might be changed by one or more earlier monitoring threads in the same grid, it is hard to even predict the right taint value to propagate, let alone give a definite answer. Moreover, in the context of taint-propagation, knowing a thread is redundant does not benefit performance for we can know only that the thread is redundant until the propagated taint is correctly calculated.

However, we can improve performance of the GPGPU-based taint-propagation without violating dependence by applying CPU design's superscalar pipeline to the GPGPU-based monitor software. Here we use the following monitored instruction sequence to explain the idea:

$$\begin{aligned} I1 &: Add\$R1, \$R2, \$R5 \\ I2 &: Add\$R3, \$R4, \$R6 \\ I3 &: Add\$R3, \$R1, \$R3 \\ I4 &: Mul\$R5, \$R6, \$R1 \\ I5 &: Mul\$R3, \$R4, \$R3 \end{aligned} \tag{5.1}$$

This instruction sequence contains five instructions, that are in the 3-operand format with destination as the last. A GPGPU-based monitor should generate five monitoring threads for these five instructions, performing their correspondending taint propagation. The dependence graph of this instruction sequence, as shown in Figure5.9, also reflects dependence among their corresponding monitoring threads. A superscalar processor would issue these five instructions in just two cycles rather than

five cycles given enough hardware resources: first, I1,I2,I3, second, I4,I5. Similarly, the optimized GPGPU-based monitor could execute five corresponding monitoring threads in two iterations instead of executing them all sequentially.

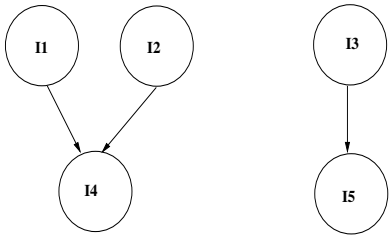


Figure 5.9: Dependence Graph of the Instruction Sequence, Arrows come from dependees to dependents.

Inspired by this observation, we can design monitoring threads to adopt a well scheduled scheme to execute monitoring threads in iterations. Each iteration executes and finishes monitoring threads whose source operand taints are not changed by any other remaining monitoring thread. i.e. we always execute monitoring threads that are independent of all remaining threads. After an iteration of threads finishes, more monitoring threads become ready to execute, executes another iteration, and repeat this process until all monitoring threads in the launch grid finish. Just like a superscalar processor needs a scoreboard to keep track of the readiness of instructions, here we can extend the touch memory to contain thread ID information to decide when to have monitoring threads execute.

To implement this optimization, we change the design of the touch shadow memory in a fashion similar to the optimized memory-bug detection monitor. Now, touch shadow memory records a thread ID for each monitored memory word and each monitored register. That thread ID represents the earliest monitoring thread that may change the taint status of the monitored memory word or register in a monitoring thread launch. Unlike memory-bug detection, where the earliest thread certainly changes its initialization bit, the earliest thread recorded by the touch shadow memory

in taint-propagation may not necessarily change the taint bit, but is just the earliest thread whose destination is the correspondending memory word or register.

Based on this new touch shadow memory design, we replace all phases after the special-type detection phase with the algorithm shown in Figure 5.7. The algorithm features a while loop of which each iteration has some monitoring threads finished. The algorithm terminates when all monitoring threads in the grid are finished. Threads that finish in an iteration are those that do not depend on any other remaining thread to calculate the taint value of its destination. Therefore, the number of iterations of the loop is the length of the longest dependence chain of the instructions that are represented by monitoring threads in the launch. For example, the monitoring threads as the example sequence of instructions should finish in 2 iterations, which exploits full parallelism without making any speculation. In the following text, we introduce the algorithm in detail.

The algorithm operates on a grid of monitoring threads in which, each thread works on a message fetched from the queue, which represents a monitored instruction. The algorithm starts with the initialization before its main body which is a loop. During this initialization, the touch shadows of the destinations of instructions represented by the monitoring threads are all initialized with the MAX ID, which are equal to the biggest thread ID plus one. As the thread ID stored in the shadow memory is replace by the ID of the earliest thread that may change the taint in living threads, this initialization guarantees that all monitoring threads replace the thread ID properly. After this initialization, all threads are synchronized to make sure they enter into the loop at the same time.

The loop body consists of two stages: setting touch memory and calculating taints. These two stages are demarcated by thread synchronization. In the first stage, monitoring threads compare their thread ID with the thread ID stored in the touch shadow that is associated with the destination, and replace the stored ID if the thread ID is

smaller. As a result, at the end of this stage, the touch shadow for the destination has the ID of the earliest thread that may change its taint. Also in this stage, the taint values of source operands are fetched for the preparation for the next stage. The second stage calculates the taint value for those thread whose taints of source operands are ready, which means that both source operand taints are not changed by any remaining thread. Once a ready monitoring thread has its taint value calculated and propagated into taint shadow, it resets the touch shadow of the destination to MAX and returns. The next iteration of the loop will then begin with fewer threads to execute. As the loop iterates, all monitoring threads soon or later become ready to calculate and to propagate taint in this stage, and eventually all threads are guaranteed to finish.

For a monitoring thread grid, the number of iterations of the loop is exactly the length of the longest dependence chain of the instructions represented by monitoring threads in the grid. The most desired case of the algorithm is that all monitoring threads are thus ready to finish in the first iteration of the loop, which happens when all monitoring threads are independent of each other. The worst case of the algorithm is that all monitoring threads are in the same dependence chain and thus one iteration can finish only one thread. However, the performance of this worst case is equal to that of running all monitoring threads sequentially, and happens only when there is absolutely no single bit of instruction level parallelism among instructions represented by threads in a grid. For most real-world programs, this worst case should happen rarely. Therefore, the optimized taint-propagation GPGPU-based monitor should gain significant performance improvement compared to the basic version.

5.4 Performance of Optimized GPGPU-based Monitor

In this section, we present the results of our performance evaluation of GPGPU-based monitors. As the performance of the basic version of the GPGPU-based monitor is reported in Figure 5.6, this section focuses on the performance of optimized GPGPU-based monitors. Since we use an architecture that is not yet on market, the performance evaluation is conducted through simulation. The CPU core part of the system architecture is simulated by using Simics simulation platform [95] with GEMS simulator [96] to simulate the memory hierarchy. The GPGPU is simulated using GPGPU-Sim simulator [97]. Using this infrastructure, we have evaluated the performance of the optimized GPGPU-based monitors for SPEC2006INT [68] benchmarks with memory-bug-detection and taint-propagation monitoring tasks. The measured performance shows a significant improvement compared with the basic GPGPU-based monitors.

This section is organized as follows: We begin with the description of the infrastructure used in the performance evaluation. After that, we demonstrate the performance of optimized GPGPU-based monitors in comparison to basic GPGPU-based monitors, and optimized distill-based monitors. We also analyze the statistics of the optimized GPGPU-based monitors to reveal the source of the improvement.

5.4.1 Infrastructure

The simulation infrastructure used to evaluate the performance of GPGPU-based monitors consists of two parts: CPU core simulation and GPGPU simulation. The CPU core simulation uses Simics simulation platform [95] with GEMS simulator [96] to simulate memory hierarchy; the simulation parameters of CPU and memory are the same as in chapter 4, Table 4.2. The communication queue between the monitored

Table 5.5: GPGPU-based Monitor Simulation Parameters

Shader Parameters	
Number of shader cores	8
Core clock rate	800 MHz
Number of DRAM channels	8
Number of blocks per core	8
Maximum thread per core	1024
Warp size	32
Number of registers per core	16384

Memory Parameters	
Shared memory per Core	16KB, partitioned from L1 Cache
L1 Cache	64KB, 4B lines, 4-way, LRU
L2 Cache	4MB, 4-way set-assoc, 8 banks
DRAM bus bandwidth	4 Bytes per DRAM cycle
DRAM size	4GB
Number of DRAM banks	8

core and the monitor core has a size of 2048 items, each of 32 bytes.

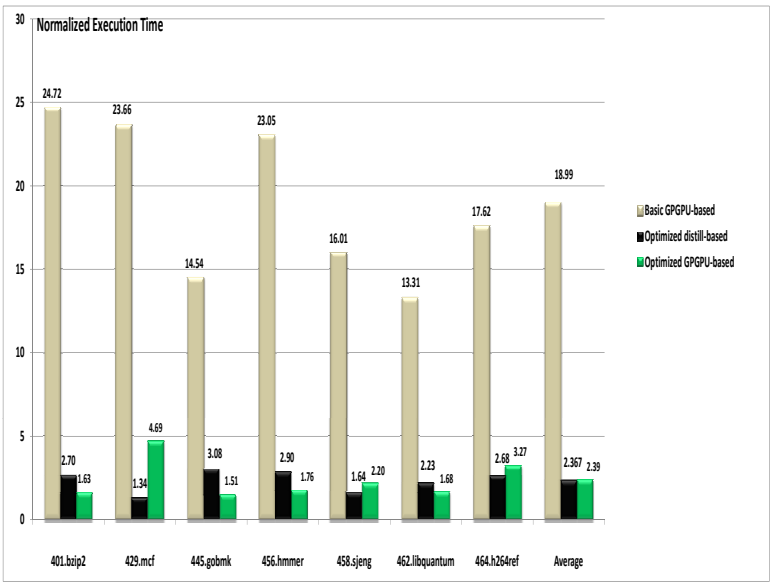
The GPGPU part of the architecture is simulated using GPGPU-Sim simulator [97]. Detailed simulation parameters can be found in Table 5.5. The GPGPU-based monitors are written in CUDA 2.1 [92]. In these monitors, each grid is configured with 1024 monitoring threads, processing 1024 items at one time. Since we configured the communication queue size to 2048 items, which is double the maximum of GPGPU concurrently running threads, the extraction logic on the monitored CPU core side can keep writing to the queue during the execution of monitoring threads.

The performance evaluation adopts SPEC2006INT [68] as monitored programs with taint propagation and memory-bug detection as monitoring tasks. Benchmarks are compiled with Gcc4.2.3 with O3 optimization level. Due to limits in the simulation speed, we simulated 100M instructions for each benchmark, and these instructions were sampled from different phases of the benchmark's execution.

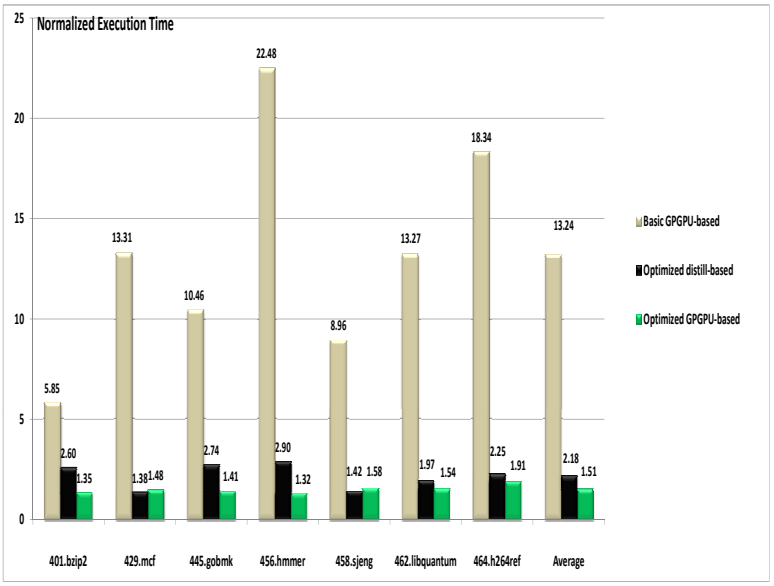
5.4.2 Performance of Optimized GPGPU-based Monitors

Figure 5.10(a) and Figure 5.10(b) show the results of the performance evaluation of optimized GPGPU-based monitors for taint propagation and memory-bug detection, respectively. The bars represent the execution time of the monitors normalized to that of the execution without monitoring. For each benchmark, three bars represent three different versions of GPGPU-based monitors: the left-hand bars represent the basic GPGPU-based version without optimization, the bars in the middle represent the optimized multi-core based monitor using optimized distill-based monitoring model, and the right-hand bars optimized GPGPU-based monitors.

These performance results demonstrate the power of the optimizations for taint propagation and memory-bug detection described in the previous section. For taint-propagation, the optimization is able to bring the slowdown factor from the 18.97x of the basic version down to 2.39x on average, and for memory-bug detection, from 13.21x to 1.51x. In the following, we analyze the reasons behind these performance improvements in taint propagation and memory-bug detection, respectively. The performance results also indicate that even without using the compiler support from which the optimized distill-based monitors benefit, the optimized GPGPU-based monitor can achieve a comparable performance for taint-propagation(2.39x of the GPGPU-based versus 2.367x of the distill-based), and significantly better (1.51x versus 2.18x) for memory-bug detection. Considering that the compiler support is orthogonal to GPGPU-based optimizations and parallelization, the future of GPGPU-based monitoring looks even more promising.



(a) Performance of Optimized GPGPU-based Monitors: Taint Propagation



(b) Performance of Optimized GPGPU-based Monitors: Memory-bug Detection

Figure 5.10: Performance of Optimized GPGPU-based Monitors: Performance is measured in execution time of monitors normalized to the execution of monitored program without monitoring. For each benchmark, three bars represent normalized execution time of three versions of GPGPU-based monitors: left-hand bars represent the basic version of GPGPU based monitor; middle bars represent the optimized distill-based monitors; and the right-hand bars optimized GPGPU-based monitors

5.4.3 Performance Analysis of Optimized GPGPU-based Monitors: Taint Propagation

For taint-propagation, the optimization is effective because it exploits instruction-level parallelism among monitored instructions. In the basic version, since most of the time monitoring threads are dependent on another thread in the same grid, most grids wind up executing monitoring threads sequentially in 1024 iterations at the verification stage. In contrast, an optimized taint-propagation GPGPU-based monitor executes all monitoring threads in fewer iterations, and the number of iterations is the length of the longest dependence chain among monitoring threads. With this optimization technique, for the selected SPEC benchmark programs, the average iteration number is reduced to 154.

The performance charts also reveal a discrepancy in performance across benchmarks. This should not be surprising as the benchmark simulations vary in the total number of monitored events, in its memory behavior, and in the degree of instruction level parallelism (ILP). The total number of monitored events contributes to the communication cost and monitoring cost of the monitor. The memory behavior of the benchmark affects not only the ommunication cost, but also the cache behavior of monitoring threads, as the monitored CPU core, monitor CPU core, and GPGPU cores all share a unified L2 cache. And most of all, the degree of instruction level parallelism (ILP) determines the number of loop iterations to execute all 1024 monitoring threads in a grid of optimized taint propagation monitoring threads, which is the key factor in the effectiveness of the optimization. Table 5.6 presents the statistics of iteration number per grid for each benchmark. The average number of independent instructions that are monitored can be calculated by dividing the number of threads in a grid (1024) by the number of iteration. The table also shows the performance overhead, which reveals a strong correlation between the average number of iterations

in a grid and the performance overhead.

Table 5.6: Statistics of Optimized GPGPU-based Taint Propagation Monitor

benchmark	average number of iterations per grid	performance overhead
401.bzip2	87.37	1.60x
429.mcf	255.57	4.70x
445.gobmk	99.21	1.49x
456.hmmer	102.0	1.74x
458.sjeng	177.60	2.13x
462.libquantum	105.68	1.68x
464.h264ref	253.98	3.25x

429.mcf has the highest performance overhead in these benchmarks. There are two factors in this: low instruction-level parallelism and poor cache behavior in shadow memory. The core of 429.mcf features a linked-list traverse where it tends to have a long data dependence chain, and this characteristic is reflected in the table statistics as a high per grid iteration number. In addition, as the touch shadow memory used in the optimized monitor is one integer (implemented as 16 bytes in our case) per monitored memory word (which is 64 bytes in the simulated architecture), as opposed to one bit per monitored word in the basic version, the cache miss problem starts showing on GPGPU as the monitored program suffer high cache misses.

We have also measured the breakdown of the execution time of the optimized GPGPU-based monitor. The details are shown in Figure 5.11. In the table, the execution time of the optimized GPGPU-based monitor is broken down into the time spent on different activities of the monitor. The statistics show that the percentage of time spent on special-type detection (including time spent on monitoring special-type events) is rather fixed, at 3.74% on average. This should not be a surprise considering that both detection and verification of special-type events are fully parallelized. Moreover, the statistics show that a large percentage of time is spent on

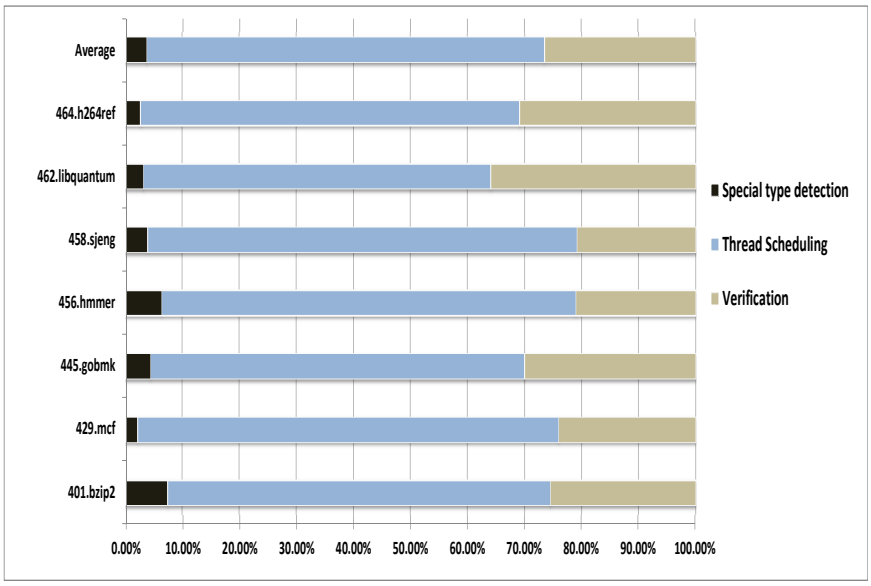


Figure 5.11: Execution Time Breakdown of Optimized GPGPU-based Taint Propagation Monitor

thread scheduling for the optimized GPGPU-based taint propagation monitor. This portion of execution time is additional, but necessary overhead to pay in order to exploit the parallelism of taint propagation monitoring threads.

5.4.4 Performance Analysis of Optimized GPGPU-based Monitors: Memory-bug Detection

Thanks to the characteristics of the memory-bug detection monitoring requirements, the effects of optimization for memory-bug detection GPGPU-based monitors is more significant than for taint-propagation, as it completely eliminates the possibility of sequential execution. Furthermore, by detecting redundant threads, it terminates a large number of monitoring threads early even before they access the touch shadow memory.

The performance overhead of the optimized memory-bug detection GPGPU-based monitor comes from three sources: first, communication cost, which is the time spent

on transferring the communication queue entries from shared memory to GPGPU streaming multiprocessors' private cache; second, time spent on detecting dangling pointer error and testing whether a thread is redundant, which is the cost all monitoring threads have to pay; and finally, time spent on setting the shadow memory, and possibly, actual change in init bits. This cost is paid only by a few threads as most of the threads exit early.

Table 5.7: Statistics of Optimized GPGPU-based Memory-bug Detection Monitor

benchmark	non-redundant percentage	performance overhead
401.bzip2	1.18%	1.18x
429.mcf	1.68%	1.41x
445.gobmk	1.48%	1.41x
456.hmmer	0.26%	1.30x
458.sjeng	2.14%	1.45x
462.libquantum	0.72%	1.48x
464.h264ref	1.941%	1.60x

Table 5.7 gives an indirect quantitative measurement of the influence of these three sources of performance overhead. In this table, for each benchmark, the table gives the total number of monitoring threads, as well as the number of those monitoring threads that are not redundant and thus did not exit early. The percentage of non redundant threads for all benchmarks is negligibly low, which means most of monitoring threads did terminate early. The total number of monitoring threads reflects the communication cost for each monitoring thread processes a communicated message.

The statistics reveal the reason for slight differences in performance overhead across benchmarks. Benchmarks with a high number of monitoring threads such as 462.libquantum and 464.h264ref have higher performance overhead, as they have to pay more for communication, thread creation and launch. On the other hand, benchmarks that have a relatively higher percentage of non-redundant threads such

as 458.sjeng and 464.h264ref have a higher overhead too, because they execute more thread synchronization and have to access touch shadow memory more often than the others.

Chapter 6

Related Work

The demand for architectural and system software support for software reliability and security has significantly increased in the past decade. A growing body of research efforts has appeared to meet this demand [71, 72, 73, 74, 75, 25]. In this chapter, we discuss the currently available research on enhancing software security and reliability by dynamically monitoring a program's execution. These research efforts are closely related to the research reported in this dissertation and have either inspired us or serve as a complements or alternatives to our proposed solutions.

The chronology of these efforts reflects the evolution of proposed methods of software reliability and security monitoring: purely software-based monitors built on instrumentation have evolved into ideas for using hardware support customized to monitoring tasks; the idea of special hardware support has spun off ideas for more generic type of hardware support; proposals based on single-core systems have evolved into multi-core-based systems; proposals using either pure hardware support or software support have inspired hybrid approaches such as the one reported in this dissertation.

Following this evolution, the rest of this chapter is organized as follows. We start with instrumentation-based proposals, introducing representative research efforts taking this approach, as well as some recent efforts in it. Next, we introduce proposals

that implement software security and reliability monitoring based on hardware support specific to monitoring tasks, that target the inefficiency of the instrumentation-based approach. Following this, we briefly illustrate efforts that take advantage of multi-core platforms to facilitate software security and reliability monitoring. Finally, we describe research efforts on parallelizing dynamic program monitoring, and highlight the differences between the GPGPU-based monitoring proposed in this dissertation and earlier efforts.

6.1 Instrumentation-based Monitors

Instrumentation-based monitoring inserts monitoring code into monitored program to monitor their execution at run time. Based on the timing of the instrument monitoring code, there are two categories of instrumentation-based monitors: static instrumentation monitors and dynamic instrumentation monitors. Representative static instrumentation monitors include Eraser [9], which rewrites the monitored program's binary to insert monitoring code before each shared memory reference to detect data race; Ccured [12], which checks type safety of C programs by inserting checking code while the monitored program is being compiled; Purify [5], which insert checking code into a compiled monitored program before its execution to detect memory-related bugs, and a taint analysis tool developed by [6].

Instrumentation-based monitors are usually implemented on the basis of a generic dynamic instrumentation tool. There are a number of well-known dynamic instrumentation tools such as PIN [14, 15], DynamoRIO [16], and Valgrind [7]. Valgrind is the most representative one of those designed to support dynamic program monitoring for security and reliability. What makes Valgrind a good fit for security and reliability monitoring is that it implements the idea of shadow memories for the monitored program's memory. This feature makes implementation of heavy-load security

and reliability monitors [17, 6, 18, 19] much easier. Because of this feature, however, monitors built on Valgrind usually suffer high performance overhead.

A few instrumentation-based monitors adopt various technologies to reduce performance overhead. LIFT [20], which is built on PIN, optimizes the instrumentation code by eliminating unnecessary taint propagation operations, merging taint checking and simplifying the context switch between the instrumentation code and monitored code. A more generic optimization for instrumentation code is proposed by Saxena et al. [21] which uses static binary analysis to enable optimization. The optimization features a fine-grained instrumentation to the monitored program. In addition to these efforts, SHIFT [22] leverages the existing instructions for speculation in Itanium [60] processor to generate efficient instrumentation code for taint analysis purposes. The work done by Ruwase et al. [24] take advantage of multi-core and speculatively run multiple versions of instrumented programs to accelerate dynamic monitoring.

Compared to instrumentation-based monitors most of which are built on single-core systems, the dynamic monitor proposed in this dissertation has a performance advantage as it taps more hardware resources to help monitoring and uses different approach to optimize monitoring. It is true that the optimized GPGPU-based monitor bears a resemblance to LIFT [20] in eliminating redundant taint checking work, but the two approaches are used in significantly different frameworks and have different implementations. Like Speck [23] and the work done by Ruwase et al. [24], the GPGPU-based monitor is also an effort to parallelize monitoring on multi-core platforms. However, the GPGPU-based monitor proposed in this dissertation is fundamentally different from earlier proposals in two aspects: it does not use any speculation; and it uses GPGPU cores instead of CPU cores. The latter difference makes the GPGPU-based monitor a more scalable approach to parallelize monitoring tasks.

6.2 Monitors with Task-specific Hardware Support

Researchers have also proposed hardware supports that are customized to various monitoring tasks to improve the performance of software security and reliability monitoring. Compared to instrumentation-based monitors, monitors built with hardware support are efficient and less intrusive to the monitored program because they require little or no instrumentation of monitoring code. Instead of monitoring code, most of the monitoring work is done by special hardware in those monitors. For heavy-load monitoring tasks, this special hardware support is worth considering because heavy-load instrumentation causes a prohibitively high performance overhead [7, 6].

Among heavy-load monitoring tasks, memory-bug detection and taint propagation have drawn most attention. As a result, most of the representative task-specific hardware support proposals build monitors for these two tasks. Of these proposals, representative memory-bug detection monitors include the following: Memtracker [32] which associates each word of data in memory with state bits, and uses a state transition table to keep track of state changes to detect memory reference violations; Hardbound [40] which proposes a hardware bounded pointer architectural primitive that serves as an under hood implementation pointer arithmetic in programming language constructs so that memory bugs are detected at run time without affecting monitored execution; SafeMem [41], which takes advantage of existing ECC bits in memory to store states and adds additional hardware logic managing them to detect memory bugs; and AccMon [42] which proposes a statistics-based hardware support to detect memory references by outlier PC that likely links to a memory bug. According the statistics reported in these related publications, all these proposals are capable of keeping performance overhead of memory-bug detection low.

Representative proposal of special-hardware-based taint-propagation monitors include the followings: RIFLE [44] that proposes a new instruction set architecture

(ISA) which supports taint-propagation monitoring, and define architectural support that translate binaries in conventional ISA into this new ISA at runtime;Raksha [31] that adds a few tag bits to registers and memory words, and changes definition of every instruction to make them support taint-propagation and check safety of them based on user specified security rules;FlexiTaint [33] proposes an in-order addition to the back-end of the processor pipeline, which computes taints for every committed instructions and store taint values into an aggregated taint bits array for all monitored memory words; and Caisson [43] that is essentially a new design of hardware description language that can be used to design processors with taint-propagation monitoring feature.

Compared to these memory-bug detection and taint propagation monitors mentioned above, the proposed monitors in this dissertation do not use any hardware support customized to any particular monitoring task. The architectural support proposed in this dissertation, referred to as extraction logic, extracts only runtime information of the monitored execution and forwards it into another core or GPGPU device. By its function definition, it is generic and can serve a large spectrum of monitoring tasks. As a result of this flexibility, the performance of the monitors proposed in this dissertation is not as good that of the ones that use special hardware support. However, because of this generality, the monitors proposed in this dissertation are more acceptable to hardware manufacturers.

6.3 Monitors on Multi-core Platforms

Multi-core architecture has in recent years become the mainstream design of processor, and researchers have already started investigating the idea of using multi-core systems to facilitate dynamic monitoring for software security and reliability.

iWatcher [29], one of earliest efforts of this kind proposes a set of architectural supports that monitor the accesses to memory locations and invoke monitoring functions when specified reference patterns are detected. This design by itself offers a way of monitoring functions to be executed in a separate core and iWatcher implemented this by using TLS to accelerate invocation to monitoring functions.

The first full-fledged dynamic program monitor on a multi-core platform was proposed by Chen et al [35]. Although the performance of that research is not appealing enough, it established the idea of assigning the monitored execution and the monitor onto different cores, as well as generic hardware support that extracts information only from the monitored execution. To improve the performance of multi-core based monitoring, recent research efforts can be categorized into two main approaches: using software support and using additional hardware support. In the following, we introduce some representative research efforts in each of these categories.

Among research efforts on software support, those reported in chapter 4 of this dissertation and [53] represent efforts to use compiler support to reduce communication and redundant monitoring work. They achieved performance improvement on multi-core systems without sacrificing the flexibility of monitors; decoupled lifeguards [48] which are built upon the work reported in [35], propose using a dynamic optimizer JIT to optimize monitor functions to eliminate redundant monitoring and shortcut verification. Orthrus [49] provides another perspective on accelerating dynamic program monitoring on multi-core systems. It uses LLVM compiler [79] to generate optimized replicas of the monitored program that are coated with monitoring code, and to run them in different cores to perform verification simultaneously.

Besides efforts on multi-core-based monitoring acceleration by software support, other research ideas have been proposed to use hardware support to accelerate multi-core based monitoring. Chen et al. [34] propose a series of architectural supports

to accelerate the multi-core-based monitor reported in [35]. These architectural supports include an inheritance tracking table for taint propagation, idempotent filters for memory-bug detection, and Metadata-TLBs for reducing latency to access shadow memory a.k.a metadata. Other representative proposals of this kind are as follows: OASES [38, 39] focuses on the common activities of monitoring tasks: for access to shadow memory, it provides architectural support (in the form of ISA support and exposed cache events) and corresponding OS support to support efficient implementation of shadow memory. Work by Hari Kannan [45] proposes a hardware solution that helps monitors keep track of the access ordering of monitored memory in their corresponding shadow memory locations.

6.4 Parallel Monitors

To take full advantage of multi-core platforms to improve the performance of dynamic monitoring, in recent years researchers have proposed various schemes to parallelize dynamic program monitoring. The work by Oplinger et al. [50, 51] is an early representative proposal of slicing the monitored program for parallelization of monitoring. In Oplinger's monitor, each monitoring function call spawns a new thread of the monitored execution to resume the monitored execution in parallel to the monitoring function. The spawned thread may invoke another monitoring function call, and thus another thread of monitored execution, and which goes on so that full parallelism is exploited. The proposal also relies on a Thread Level Speculation (TLS) mechanism to ensure thread safety. A similar idea is later applied in Pulse [52] to monitor concurrency bugs among processes by speculatively creating new slices of them to help the monitor predict buggy behavior. Recent representative work includes Speck [23], which proposes a relay monitor that pairs monitor threads with slices of monitored execution, so that not only can monitored execution go speculatively ahead as a new

slice when the previous slice is still executing, but the new slice also has its own monitor thread.

The GPGPU-based monitor proposed in this dissertation is different from all previous proposals to parallelize dynamic program monitoring. To our best knowledge, it is the first effort to use GPGPU many-core architecture to parallelize dynamic program monitoring for security and reliability. It does not slice monitored programs, but completely separates monitor code from monitored execution, and it does not use special hardware support for the purpose of parallelization. These differences have a profound impact on dynamic program monitoring, as all previous parallelization proposals are based on multiple CPU-core architecture and thus bound by constraints of availability of resources and power consumption, while using GPGPU is a much more scalable and environmentally-friendly approach.

Chapter 7

Conclusions

Dynamic program monitoring is an effective and powerful approach to enforce software security and reliability. As dynamic program monitoring is performed along side with the monitored program, its efficiency is key to its usability and applicability. This dissertation contributes to the research in dynamic program monitoring by proposing novel technologies to improve the efficiency of dynamic program monitoring using state-of-the-art multi-core architecture.

In this dissertation, we have proposed a generic hardware support, referred to as extraction logic, to enable the separation of monitored execution and its monitor. The extraction logic implements a basic functionality of multi-core dynamic program monitoring: extracting runtime information of the monitored program. The implementation is achieved with a desirable design: flexible and configurable to support a broad spectrum of monitoring tasks, it does not require any change to ISA or CPU pipeline design or any instrumentation to the monitored program. Extraction logic serves as the only architectural support for all other techniques reported in this dissertation, and it can be extended to support other research efforts on multi-core-based dynamic program monitoring.

Based on the extraction logic as the architectural support, we have proposed compiler support to implement efficient multi-core-based dynamic program monitoring. The compiler support can automatically generate optimized monitor code from the monitored program's source based on monitoring requirements. The generated monitor is referred to as the distill-based monitor. The distill-based monitor not only reduces the demand for communication between the monitored core and its monitor, but also eliminates the dispatching cost incurred by other types of multi-core-based monitor. The performance evaluation of the compiler support shows that the optimization techniques used in the compiler support can significantly improve the efficiency of dynamic program monitors in multi-core systems. Compared to other types of multi-core-based monitor systems that require information forwarding for all instructions, using the proposed compiler support can improve the performance for memory-bug detection and taint propagation by a factor of 3 on average for SPEC2006 INT programs. The performance achieved by the proposed compiler support is comparable even to that achieved by using dedicated hardware support.

We have also proposed a promising approach for parallelizing dynamic program monitoring using emerging GPGPU architecture. We first proposed a generic software framework on GPGPU that is flexible enough to serve as a guideline to develop GPGPU-based monitoring for various kinds of monitoring tasks. On top of the generic framework, we have also developed customized optimization techniques for memory-bug detection and taint propagation. The performance evaluation shows that the optimized GPGPU-based monitors significantly improve performance of the monitoring tasks taint propagation and memory-bug detection for SPEC2006INT benchmarks. Compared to traditional instrumentation-based monitoring, the performance overhead of monitoring is improved by 3.1 times and 5.2 times for taint propagation and memory-bug detection, respectively.

Bibliography

[1] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In Proc. *2nd USENIX Symposium on Operating System Design and Implementation.*, pages 229-243, 1996.

[2] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation* , pages 320-331, Ottawa, Canada, June 2006.

[3] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jekrey S. Foster. Mixing type checking and symbolic execution. In In *ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, pages 436-447, 2010.

[4] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified Self-Modifying Code. In *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'07), San Diego, CA, pages 66-77, June 2007

[5] R. Hastings, B. Joyce, Purify: Fast detection of memory leaks and access errors, in: *the Winter 1992 USENIX Conference*, San Francisco, California, pp. 125C138.

[6] J. Newsome, D. X. Song, Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software, in: *Network and Distributed Systems Security 2005* (NDSS 2005).

[7] N. Nethercote, J. Seward, Valgrind: A framework for heavyweight dynamic binary instrumentation, in: *ACM SIGPLAN 07 Conference on Programming Language Design and Implementation* (PLDI'07), San Diego, California, USA.

[8] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments* (VEE '07).

[9] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. ACM *Transaction of Computer Systems* 15, 4 (November 1997), 391-411

[10] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN 94 Conference on Programming Language Design and Implementation* (PLDI'94), 1994.

[11] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ACM SIGSOFT 2002 International Conference on Software Engineering* (ICSE'02), Orlando, Florida, 2002.

[12] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. Ccured in the real world. In *ACM SIGPLAN 03 Conference on Programming Language Design and Implementation* (PLDI'03), pages 232-244, San Diego,California, 2003.

[13] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. IEEE *Journal on Selected Areas in Communications*, 21(1), January 2003.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (PLDI '05)

[15] Steven Wallace and Kim Hazelwood. 2007. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. *In Proceedings of the International Symposium on Code Generation and Optimization* (CGO '07).

[16] Derek L. Bruening. 2004. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA

[17] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. 2007. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications* (OOPSLA '07).

[18] I. K. Isaev and D. V. Sidorov. 2010. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. ACM *Journal of Programming and Computing Software*. 36, 4 (July 2010), 225-236

[19] Alex Groce , Rajeev Joshi, Extending model checking with dynamic analysis, In *Proceedings of the 9th international conference on Verification, model checking, and abstract interpretation*, p.142-156, January 07-09, 2008, San Francisco, USA

[20] Feng Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, Y. Wu, Lift: A low-overhead practical information flow tracking system for detecting security attacks, Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on (Dec. 2006)

[21] Prateek Saxena, R Sekar, and Varun Puranik. 2008. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (CGO 2008)

[22] Haibo. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, F. T. Chong, From speculation to security: Practical and efficient information flow tracking using speculative hardware, in *Proceedings of the 35th International Symposium on Computer Architecture* (ISCA 2008)

[23] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. 2008. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (ASPLOS 2008)

[24] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. 2008. Parallelizing dynamic information flow tracking. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures* (SPAA '08)

[25] Hossain Shahriar and Mohammad Zulkernine. 2011. Taxonomy and classification of automatic monitoring of program security vulnerability exploitations. ACM *Journal of System Software* 84, 2 (February 2011), 250-269.

[26] E. Witchel, J. Cates, K. Asanovic, Mondrian memory protection, in: *10th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-X).

[27] J. R. Crandall, F. T. Chong, Minos: Control data attack prevention orthogonal to memory model, in: *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium onMicroarchitecture*, IEEE Computer Society, Washington, DC, USA, 2004, pp. 221-232

[28] G. E. Suh, J. W. Lee, D. Zhang, S. Devadas, Secure program execution via dynamic information flow tracking, in: *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ACM, New York, NY, USA, 2004, pp 85-96.

[29] P. Zhou, F. Qin, W. Liu, Y. Zhou, J. Torrellas, iwatcher: Simple, general architectural support for software debugging, in: *31st Annual International Symposium on Computer Architecture* (ISCA'04).

[30] R. Shetty,M. Kharbutli, Y. Solihin,M. Prvulovic, Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection, *IBM Journal of Research and Development* archive Volume 50 Issue 2/3, March 2006.

[31] M. Dalton, H. Kannan, C. Kozyrakis, Raksha: A flexible information flow architecture for software security, in: *34th Annual International Symposium on Computer Architecture* (ISCA'07).

[32] G. Venkataramani, B. Roemer, Y. Solihin, M. Prvulovic, Memtracker: Efficient and programmable support for memory access monitoring and debugging, in: *13th International Symposium on High-Performance Computer Architecture* (HPCA-13).

[33] G. Venkataramani, I. Doudalis, Y. Solihin, M. Prvulovic, Flexitaint: A programmable accelerator for dynamic taint propagation, in: *14th International Symposium on High-Performance Computer Architecture* (HPCA-14).

[34] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. Mowry, et al., Flexible hardware acceleration for instruction-grain program monitoring, in: *34th Annual International Symposium on Computer Architecture* (ISCA-08).

[35] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. 2006. Log-based architectures for general-purpose monitoring of deployed code. In Proceedings of the 1st workshop on Architectural and system support for improving software dependability (ASID '06). ACM, New York, NY, USA,

[36] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, Evangelos Vlachos. "Flexible Hardware Acceleration for Instruction-Grain Lifeguards". *IEEE Micro*, Jan/Feb 2009 Special Issue

[37] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry. "Log-Based Architectures: Using Multicore to Help Software Behave Correctly". ACM SIGOPS *Operating Systems Review*, Volume 45 Issue 1, January 2011 (OS Review'11).

[38] V. Nagarajan, R. Gupta., Runtime monitoring on multicores via oases, *SIGOPS Operating System Review 43* (2009) 15-24.

[39] Vijay Nagarajan and Rajiv Gupta. 2009. Architectural support for shadow memory in multiprocessors. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS *international conference on Virtual execution environments* (VEE '09)

[40] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (ASPLOS XIII)

[41] Feng Qin, Shan Lu, and Yuanyuan Zhou. 2005. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)

[42] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. 2004. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. In Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)

[43] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: a hardware description language for secure information flow. In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)

[44] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. 2004. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)

[45] Hari Kannan. 2009. Ordering decoupled metadata accesses in multiprocessors. In Proceedings of the 42nd Annual IEEE/ACM *International Symposium on Microarchitecture* (MICRO 42)

[46] Michael Huth and Mark Ryan (2004). Logic in Computer Science (Second Edition). Cambridge University Press. p. 207. ISBN 0-521-54310-X.

[47] E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (April 1986), 244-263

[48] Olatunji Ruwase, Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2010. Decoupled lifeguards: enabling path optimizations for dynamic correctness checking tools. In Proceedings of the 2010 ACM SIGPLAN conference on *Programming language design and implementation* (PLDI '10)

[49] Ruirui Huang, Daniel Y. Deng, and G. Edward Suh. 2010. Orthrus: efficient software integrity protection on multi-cores. In Proceedings of the fifteenth edition of ASPLOS on *Architectural support for programming languages and operating systems* (ASPLOS '10)

[50] Jeffrey Oplinger and Monica S. Lam. 2002. Enhancing software reliability with speculative threads. In Proceedings of the 10th international conference on *Architectural support for programming languages and operating systems* (ASPLOS-X)

[51] Jeffrey Thomas Oplinger. 2004. Enhancing Software Reliability with Speculative Threads. Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Monica S. Lam. AAI3145591

[52] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. 2005. Pulse: a dynamic deadlock detection mechanism using speculative execution. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05).

[53] Improving the Performance of Program Monitors with Compiler Support in Multi-Core Systems Guojin He and Antonia Zhai, in *the Proc. the IEEE International Parallel & Distributed Processing Symposium* (IPDPS), 2010

[54] Guojin He and Antonia Zhai. 2011. Efficient dynamic program monitoring on multi-core systems. *Journal of System Architecture* 57, 1 (January 2011), 121-133

[55] G.Hilton et. Al, The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal* Q1, 2001

[56] R. Varada, M. Sriram, K. Chou, and J. Guzzo. Design and Integration Methods for a Multi-threaded Dual Core 65nm Xeon Processor. In *International Conference on Computer-Aided Design*, Nov 2006

[57] Balaram Sinharoy. 2009. POWER7 multi-core processor design. In Proceedings of the 42nd Annual *IEEE/ACM International Symposium on Microarchitecture* (MICRO 42). ACM, New York, NY, USA

[58] Kostas Pagiamtzis and Ali Sheikholeslami. Contentaddressable memory (CAM) circuits and architectures:A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712-727, March 2006.

[59] The SPARC ArchitectureManual, Version 9 .Sun Microsystems Inc., 1994.

[60] Intel Corporation, Intel Itanium 2 Processor Reference Manual For Software Development and Optimization. ftp://download.intel.com/design/Itanium2/manuals/25111003.pdf, May 2004, Section 11.

[61] James Archibald and Jean-Loup Baer. 1986. Cache coherence protocols: evaluation using a multiprocessor simulation model. ACM *Transactions on Computer Systems*. 4, 4 (September 1986), 273-298.

[62] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In Proceedings of the eighteenth *ACM symposium on Operating systems principles* (SOSP '01). ACM, New York, NY, USA, 73-88

[63] William A. Arbaugh, William L. Fithen, John McHugh, "Windows of Vulnerability: A Case Study Analysis," IEEE Computer, pp. 52-59, December, 2000.

[64] A. Sabelfeld and A. C. Myers, "Language-based information-flow security", IEEE *Journal on Selected Areas in Communications*, 2003.

[65] Gerard Boudol. 2009. Secure Information Flow as a Safety Property. In Formal Aspects in Security and Trust, Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli (Eds.). L*ecture Notes In Computer Science*, Vol. 5491. Springer-Verlag, Berlin, Heidelberg 20-34.

[66] Dinakar Dhurjati and Vikram Adve. 2006. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In Proceedings of the *International Conference on Dependable Systems and Networks* (DSN '06). IEEE Computer Society, Washington, DC, USA, 269-280.

[67] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2009. Efficiently and precisely locating memory leaks and bloat. In Proceedings of the 2009 *ACM SIGPLAN conference on Programming language design and implementation* (PLDI '09).

[68] Cloyce D. Spradling. 2007. SPEC CPU2006 benchmark tools. SIGARCH Comput. Archit. News 35, 1 (March 2007), 130-134.

[69] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. 2002. Compiler optimization of scalar value communication between speculative threads. In Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS-X)

[70] Antonia Zhai, J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. 2008. Compiler and hardware support for reducing the synchronization of speculative threads. ACM Trans. Archit. Code Optim. 5, 1, Article 3 (May 2008)

[71] Nicolas Palix et al., 2011. Faults in linux: ten years later. In Proceedings of the sixteenth international conference on *Architectural support for programming languages and operating systems* (ASPLOS '11). ACM, New York, NY, USA, 305-318.

[72] Gernot Heiser, June Andronick, Kevin Elphinstone, Gerwin Klein, Ihor Kuz, and Leonid Ryzhyk. 2010. The road to trustworthy systems. In Proceedings of the fifth *ACM workshop on Scalable trusted computing* (STC '10). ACM, New York, NY, USA

[73] David Monniaux. 2007. Verification of device drivers and intelligent controllers: a case study. In Proceedings of the 7th ACM & IEEE *international conference on Embedded software* (EMSOFT '07).

[74] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. 2006. Can We Make Operating Systems Reliable and Secure? ACM Computer 39, 5 (May 2006), 44-51.

[75] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. 2010. We crashed, now what?. In Proceedings of the Sixth international conference on Hot topics in system dependability (HotDep'10). USENIX Association, Berkeley, CA, USA, 1-8

[76] A. Aho, M. Lam, R. Sethi, J. Ullman, Compilers: Principles Techniques and Tools. 2007. second ed., Pearson Addison Wesley, 2007

[77] Frances E. Allen. 1970. Control flow analysis. In Proceedings of a symposium on Compiler optimization. ACM, New York, NY, USA

[78] John Ban Nang Kam. 1976. Monotone Data Flow Analysis Frameworks: A Formal Theory of Global Computer Program Optimization.. Ph.D. Dissertation. Princeton University, Princeton, NJ, USA. AAI7622636

[79] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the *international symposium on Code generation and optimization*: feedback-directed and runtime optimization (CGO '04)

[80] Mark Harris. 2005. Mapping computational concepts to GPUs. In ACM SIGGRAPH 2005 Courses (SIGGRAPH '05), John Fujii (Ed.). ACM, New York, NY, USA, , Article 50

[81] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A view of the parallel computing landscape. *Communications of ACM* 52, 10 (October 2009)

[82] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Computer. Architecture News 38, 3 (June 2010)

[83] J. Tolke and M. Krafczyk. 2008. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*. 22, 7 (August 2008), 443-456

[84] Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. 2011. Fermi GF100 GPU Architecture. IEEE *Micro* 31, 2 (March 2011), 50-59. DOI=10.1109/MM.2011.24 http://dx.doi.org/10.1109/MM.2011.24

[85] Nvidia Quadro fx5800 http://www.nvidia.com/object/product_quadro_fx_5800_us.html

[86] AMD Fusion http://sites.amd.com/us/fusion/apu/pages/fusion.aspx

[87] Mayank Daga, Ashwin M. Aji, and Wu-chun Feng. 2011. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing (SAAHPC '11). IEEE Computer Society, Washington, DC, USA, 141-149

[88] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In Proceedings of the 38th annual international symposium on Computer architecture (ISCA '11).

[89] AMD. OpenCL: The Open Standard for Parallel Programming of GPUs and Multi-core CPUs. http://ati.amd.com/technology/streamcomputing/opencl.html.

[90] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. ACM Queue 6, 2 (March 2008), 40-53. DOI=10.1145/1365490.1365500 http://doi.acm.org/10.1145/1365490.1365500

[91] NVIDIA. 2007. CUDA Technology; http://www.nvidia.com/CUDA.

[92] NVIDIA. 2009. http://developer.nvidia.com/content/cuda-toolkit-21-january-2009

[93] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. SIGARCH *Computer. Architecture. News* 34, 4 (September 2006), 1-17. DOI=10.1145/1186736.1186737

[94] John B. Kam and Jeffrey D. Ullman. 1976. Global Data Flow Analysis and Iterative Algorithms. *Journal of ACM* 23, 1 (January 1976), 158-171

[95] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A Full System Simulation Platform. ACM *Computer* 35, 2 (February 2002), 50-58

[96] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. SIGARCH *Computer Architecture News* 33, 4 (November 2005), 92-99

[97] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS), Boston, MA, USA, April 2009.