

University of Massachusetts Amherst
ScholarWorks@UMass Amherst

Masters Theses

Dissertations and Theses

November 2015

Hardware Monitors for Secure Processing in Embedded Operating Systems

Tedy Mammen Thomas
University of Massachusetts Amherst

Follow this and additional works at: https://scholarworks.umass.edu/masters_theses_2

Recommended Citation

Thomas, Tedy Mammen, "Hardware Monitors for Secure Processing in Embedded Operating Systems" (2015). *Masters Theses*. 302.
https://scholarworks.umass.edu/masters_theses_2/302

This Open Access Thesis is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Masters Theses by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

**HARDWARE MONITORS FOR SECURE PROCESSING IN EMBEDDED
OPERATING SYSTEMS**

A Thesis Presented

by

TEDY MAMMEN THOMAS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

SEPTEMBER 2015

ELECTRICAL AND COMPUTER ENGINEERING

© Copyright by Tedy Mammen Thomas 2015

All Rights Reserved

**HARDWARE MONITORS FOR SECURE PROCESSING IN EMBEDDED
OPERATING SYSTEMS**

A Thesis Presented

by

TEDY MAMMEN THOMAS

Approved as to style and content by:

Russell Tessier, Chair

Tilman Wolf, Member

Daniel Holcomb, Member

C. V. Hollot, Department Head
Electrical and Computer Engineering

*To my mother, Anuja
and my father, Thomas.*

ACKNOWLEDGMENTS

I would like to thank Professor Russell Tessier for giving me the opportunity to work on this project and for his constant motivation and guidance. I am also grateful to Professor Tilman Wolf and Professor Daniel Holcomb for agreeing to be on my thesis committee. I would like to extend my gratitude to Arman Pouraghily and Kekai Hu for their contributions in this project. Finally, I would like to thank Justin Lu for his valuable suggestions and support.

ABSTRACT

HARDWARE MONITORS FOR SECURE PROCESSING IN EMBEDDED OPERATING SYSTEMS

SEPTEMBER 2015

TEDY MAMMEN THOMAS

B.TECH. E.C., COCHIN UNIVERSITY OF SCIENCE & TECHNOLOGY

M.S. E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell Tessier

Embedded processors are being increasingly used in our daily life and have become an important part of many types of infrastructure in the world. As people start depending more on embedded systems for personal and business processing operations, the attacks on these systems have also been on a rise. Existing defense mechanisms targeted for desktop and server processors cannot be used to defend embedded systems as these system exhibit constraints on processing performance and processing power and energy. Thus, embedded systems require low overhead security approaches to ensure that they are protected from attacks.

This thesis describes a hardware based approach to monitor the operation of an embedded processor instruction-by-instruction, where deviations from expected program behavior are detected within the time associated with the execution of an instruction. Previous work in this area has focused on monitoring a single task on a CPU while here a novel hardware monitoring system that can monitor multiple active tasks in an operating-system-based platform is presented. This approach doesn't need any change in application binary code. The hardware monitor is able to track context switches that occur in the

operating system and ensure that monitoring is performed continuously, thus ensuring system security.

This thesis describes the design of the system as well as results obtained from a prototype implementation of the system on an Altera DE4 FPGA board. It is demonstrated in hardware that applications can be monitored at instruction level without execution slow-down and buffer overflow attacks can be defeated using this system. When an attack occurs, it is detected within a cycle and the attack task is killed before it can harm the system. The system uses an off-chip DRAM for storing the application binary and the operating system kernel. A centralized graph memory is implemented on-chip to support the storage of all monitoring graphs associated with the system. MiBench benchmarks such as qsort, bitcount, stringmatch, basicmath and dijkstra are used to evaluate the system.

TABLE OF CONTENTS

	Page
ACKNOWLEDGMENTS	v
ABSTRACT.....	vi
LIST OF TABLES	x
LIST OF FIGURES	xi
 CHAPTER	
1. INTRODUCTION	1
1.1. Attacks and Defenses in Embedded Systems	4
1.2. Organization of the document.....	6
2. BACKGROUND AND RELATED WORK	7
2.1. Related work	7
2.1.1. Security Issues in Embedded Processor.....	7
2.1.2. Monitoring Systems	7
2.1.3. Recovery	9
2.1.4. Embedded Processor Context Switch	10
2.2. Single-Task Monitoring System	10
2.3. Summary	14
3. SYSTEM & SECURITY MODEL.....	15
3.1 Secure Processing with Hardware Monitors	15
3.2. Operating System Interaction and Recovery	17
3.3. Security Model.....	18
3.3.1. Attack Example.....	18
3.3.2. Security Requirements	19
3.3.3. Attacker Capabilities.....	19
4. OPERATING SYSTEM MANAGEMENT	21
4.1. Processor Interface.....	21
4.2. Task Creation	22
4.3. Context Switch.....	24
4.4. Task Delete	26
4.5. Summary	27
5. SYSTEM ARCHITECTURE	28
5.1. Task Management in the Operating System	28
5.2. Multi-Task Hardware Monitor System.....	29

5.3. DMA Interface to Centralized Graph Memory.....	31
5.4. OS-to-Monitor Interface for Context Switch.....	34
5.5. OS-to-Monitor Interface for Process Creation.....	35
5.6. OS-to-Monitor Interface for Process Deletion.....	36
5.7. Dual-core Monitoring System Design	36
5.8. Summary	39
6. PROTOTYPE IMPLEMENTATION.....	40
6.1. System Setup.....	40
6.2. Monitor Context Management.....	44
6.3. Monitor Process Creation and DMA interface	45
6.4. Attack Detection and Protection	47
6.5. Monitoring System Resources	51
6.6. Dual-Core Monitoring System.....	53
7. CONCLUSION.....	56
BIBLIOGRAPHY	58

LIST OF TABLES

Table	Page
1: System recovery time.....	49
2: Resource use on DE4 FPGA.....	51
3: Monitoring graph size.....	52
4: Cycle count and time delay for various operations.....	52
5: Resource use for dual-core embedded system.....	55

LIST OF FIGURES

Figure	Page
1: Embedded devices installed base forecast [11]	1
2: Attack on packet processing system in network router data plane [8]	2
3: Embedded processor system with hardware monitor	3
4: State machine generation from processing binary [17]	11
5: NFA state machine [17]	12
6: DFA state machine using powerset construction [17]	12
7: Grouping of states [17]	12
8: Memory representation of DFA monitoring graph [17]	13
9: System architecture of Multi-Task Hardware Monitor system	16
10: Attack detection and system recovery	18
11: Processor interface	21
12: Multiple tasks in μ C/OS-II [18]	23
13: Reporting task create in software	24
14: Context save of suspended task [18]	25
15: Context restore of next task [18]	25
16: Reporting context switch in software	26
17: Reporting task delete in software	27
18: Detailed view of multi-context monitoring system	30
19: System architecture with centralized graph memory	32
20: DMA controller interface	33
21: Dual-core monitoring system	37
22: DDR interface	41

23: NIOS II setup in Qsys.....	41
24: Offline analysis	42
25: Attack code	43
26: Console display during stack smashing	43
27: SignalTap waveforms showing monitor context switch.....	44
28: SignalTap waveforms showing monitor process creation	45
29: SignalTap waveforms showing monitor task creation.....	46
30: SignalTap waveforms showing detection of an attack	47
31: Console Display showing task detection and deletion.....	48
32: SignalTap waveforms showing monitor task delete	49
33: Recovery process in the embedded system.....	50
34: DDR2 address space partition to support dual Nios II core	53
35: SignalTap waveform shwoing graph transfer in dual-core monitoring system.....	54

CHAPTER 1

INTRODUCTION

Embedded processing systems are widely used and are key technology for control systems, the Internet of Things, personal health monitoring, home automation, and many other application domains. Figure 1 illustrates the rise in the usage of embedded devices. Due to their wide use and the importance of their tasks, embedded systems need to be protected from hacking attacks. With an increasing number of embedded systems being connected to networks, one typical attack against embedded systems is through the global Internet.

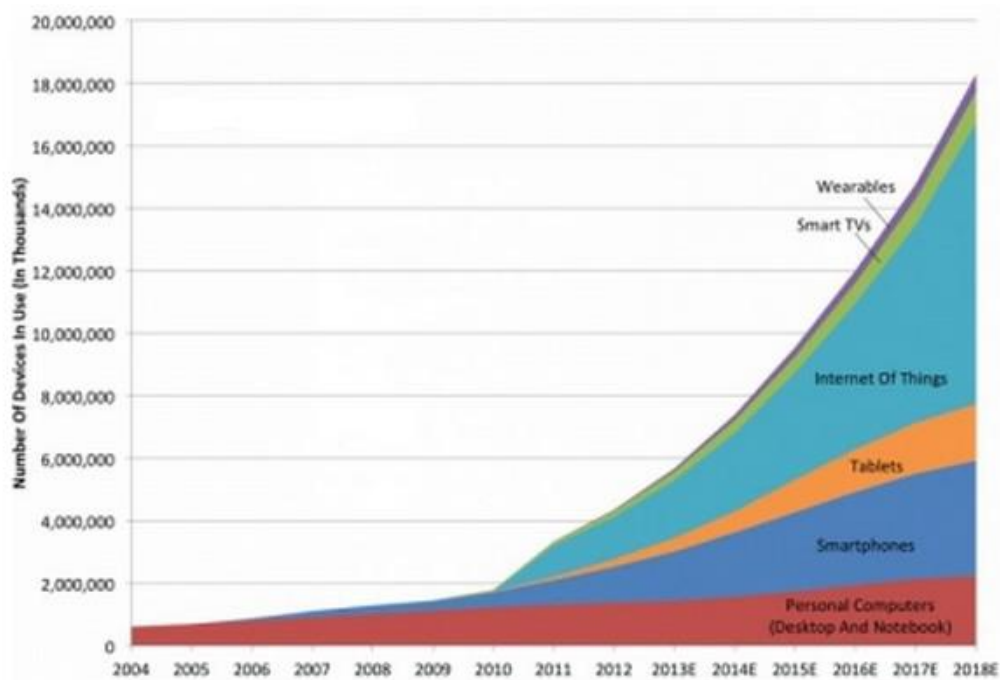


Figure 1: Embedded devices installed base forecast [11]

Many embedded systems are based on general-purpose processing systems that are vulnerable to the same types of attacks as conventional desktop and server computers, albeit for a different set of applications. The National Vulnerability Database (NVD) [24] shows

that around 10% of vulnerabilities (6518 out of 66,399) in systems are related to overflows that can be exploited via a network. Many of these overflows then enable an attacker to execute malicious code. Thus, this thesis focuses on protecting embedded systems from this important type of attack. Figure 2 illustrates a denial of service attack generated in-network by exploiting this inherent vulnerability of embedded network processors [8].

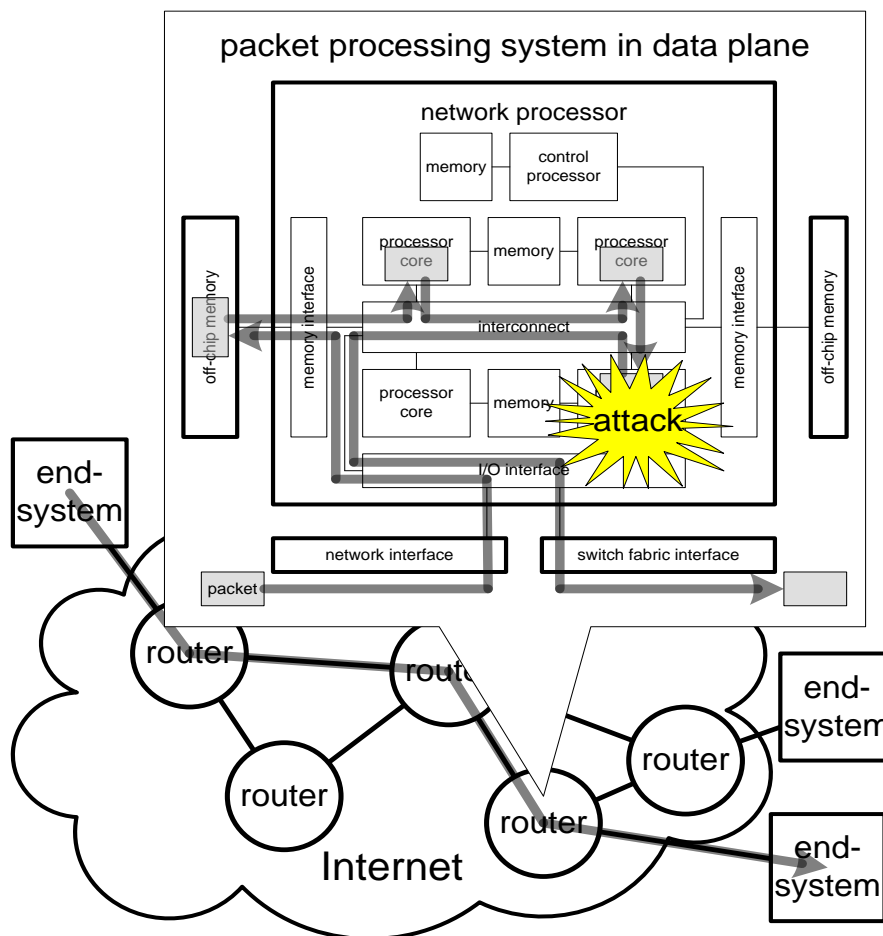


Figure 2: Attack on packet processing system in network router data plane [8]

While desktop and server computers have the processing power to run malware detection software (e.g., virus scanner, intrusion detection software, etc.), embedded systems are typically not able to do so due to resource constraints (e.g., limited power budget, limited processing capacity, etc.). Instead, hardware-based protection mechanisms have been

developed, in particular “hardware monitors”, as illustrated in Figure 3. These monitors look for deviations from expected processor behavior using run time processing information. In case of an attack, the monitor detects deviation from expected behavior and a suitable recovery process is initiated.

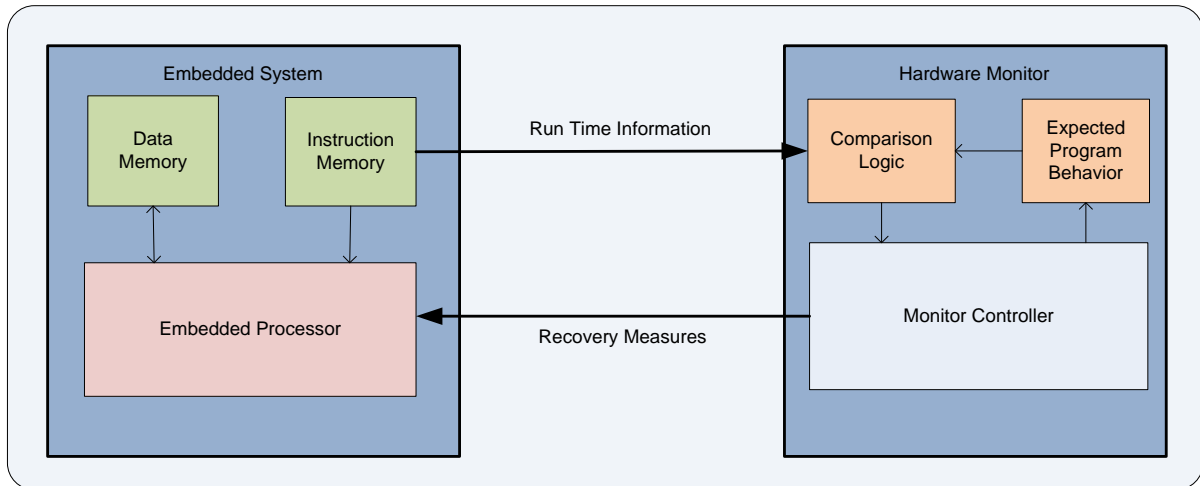


Figure 3: Embedded processor system with hardware monitor

A variety of different hardware-based solutions have been proposed to protect embedded processing systems. In general, there have been two shortcomings in existing work:

- Monitoring on systems with complex workloads is based on coarse indicators (e.g., function call sequence). This approach leaves the system vulnerable to attacks that happen between these indicators (e.g., within a function call).
- Fine-grained monitoring systems do not support multi-task workloads on operating systems. This constraint limits the applicability of this single-task monitoring systems to specialized domains (e.g., embedded control systems, network processors, etc.).

To make hardware monitors an effective protection mechanism for attacks on embedded systems in any application domain, it is critical to develop fine-grained monitoring on multi-task embedded systems. In this work, the design of a hardware monitoring system

that coordinates with the task switching dynamics of an operating system to verify every instruction executed by applications is presented. This approach neither requires any changes to application binary, nor detection software in the operating system. Thus the processor core defense can be implemented efficiently and is backward compactable with the existing embedded system code. The result from this thesis can aid in defending the embedded processing systems from various different domains and make them more enduring against an increasing range of attacks.

1.1. Attacks and Defenses in Embedded Systems

Many important functions in today's global infrastructure are implemented using embedded systems. The value of the data being processed in embedded systems and the operations they perform are enough reasons for attacks to target these systems. Some of the major motivations towards the attacks on embedded systems are information theft, energy drainage, confusion of the sensor, device reprogramming, network intrusion, physical intrusion, etc [25].

The focus of this thesis is on code injection attacks, where malicious code is injected remotely (e.g., via a network). This type of attack is extensively used in practice. Stuxnet [19] is very popular example of this type of attack which led to the physical destruction of embedded controllers because malicious code was injection into them. Although, Stuxnet is a very complex example based on embedded systems with advanced operating systems, there are also examples for low-end embedded systems such as smart cards and RFID chips [32].

The typical characteristics of embedded systems such as limited processing performance and battery power make conventional software-based defenses, such as virus scanners or other intrusion detection system, unsuitable. Particularly in real-time environments, it is not acceptable to account for the unknown processing overhead of

malware detection software. Also, considerable portions of the energy budget of an embedded system may be required by software-based malware detection.

In this work, the design of the hardware-based protection mechanism is such that it doesn't come in the way of the operation of the embedded system, which still being able to reliably detect any attack that changes the operation of the system. The hardware monitoring system, in particular, does not slow down the embedded processor nor does it require any changes to the application binary.

Prior work that has proven effective in detecting code injection attacks [7, 17] forms the basis of this thesis. The focus of prior work was on network processing systems, a certain type of embedded system with single-threaded workload. This thesis focuses on the expansion of this work to make it practical for real-time embedded systems with multiple-threaded workloads that are controlled by an operating system (OS).

The goal of this thesis is to develop a Multi-task Hardware Monitor (MTHM) system that can protect modern embedded processors with real-time performance constraints from attacks that are targeted at vulnerabilities in embedded system. The specific contributions of this thesis are:

- Design of a Multi-task Hardware Monitor system that supports multi-tasking contexts and that operates in sync with an embedded operating system.
- Design of a centralized graph memory circuitry for the storage of all monitoring graphs associated with the embedded processor core and its interface to the Multi-task Hardware Monitor system.
- Embedded system recovery when an attack is detected by the Multi-task Hardware Monitor system.

- Prototype implementation of the system on an Altera DE4 board using DRAM for application storage.
- Evaluation of the prototype and demonstration of system protection from stack smashing attack.

1.2. Organization of the document

The rest of the thesis document is organized as follows. Chapter 2 provides a brief overview of the background and related work. Chapter 3 describes the system and security model that is representative of the embedded system that can be attacked remotely. Chapter 4 describes the operating system management in the embedded system and Chapter 5 discusses the system architecture. Chapter 6 explains the prototype implementation and Chapter 7 concludes the thesis with directions for future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides a brief review of related work that provides context for the research. We will also look into the implementation details of the single-task monitoring system and see how this work needs to be expanded to monitoring of multi-task embedded systems.

2.1. Related work

2.1.1. Security Issues in Embedded Processor

A variety of techniques can be used to attack an embedded processor [25]. Physical security can be achieved by employing tamper resistant designs as described by Ravi et al. [31]. Embedded processors are also susceptible to side-channel attacks (e.g., differential power analysis [16]), although it is not considered in this work. Embedded systems are also exposed to remote attacks in a networked framework as considered by Wood et al. [39]. Most of the time, these attacks take advantage of weakness in user software to take control of embedded processor operation. Repetitive messages are often sent to entice the processor to reveal secret information [31], especially if cryptographic protections are weak. Malicious code can also be inserted if the security is weak, in which case, an embedded processor is re-tasked [25]. Processor debug and test ports are used by other software attacks to extract vital processor information [33].

2.1.2. Monitoring Systems

Constrained programming environments for embedded processors have been proposed to address security concerns [14]. However, a full and diverse programming environment is required by many embedded processors. The usage of these approaches for

security can lead to high overhead in embedded operating systems. Advanced techniques such as multiple independent levels of security (MILS) [26] are used by comprehensive embedded processor software systems to effectively isolate different processes and their data. In low-overhead embedded systems with minimal OS capabilities, this additional security layer may not be appropriate. In this work, security is achieved by monitoring processor execution. Monitoring has been used also in system by Arora et al. [3] and the IMPRES system [29], but a fine-granularity of monitoring is used by our monitor. Information is collected across multiple executed instructions in determining if the operation is valid in SAFE-OPS system by Zambreno et al. [40]. Attacks and errors can be detected by this system at the end of such a sequence, whereas our monitor can immediately detect the first instruction that deviates from expected behavior.

A control flow graph for monitoring program execution is also used by Abadi et al. [1]. Integrity checks were introduced into the micro-architecture by Nakka et al. and the system used special check instructions [23]. In the system by Ragel et al. [30], microinstructions are introduced to monitor fault detection, return address checks and memory boundary checks. Unlike this, our monitor doesn't require any changes in the machine code to implement the necessary checks. A general, hardware-based architecture is developed by Goginat et al. [12] to protect embedded systems against a range of attacks. Chen et al. [9] proposes a log-base architecture (LBA) approach for monitoring using multiple processor cores. An unmodified program is run on one core while special monitoring hardware records all executed instructions in a log that scans for attacks and failures by software on another core. This architecture is very powerful and can detect a range of problems, but requires large log buffers [10]. This may introduce serious limitations in embedded system and also delay attack detection. Our monitor is able to detect deviations in operation within one instruction and is thus more suitable.

Various other security approaches have been directed at embedded processor execution. One of them involves tagging of non-instruction memory pages with NX (No eXecute) or XD (eXecute Disable) bits. By doing so, control flow change to a piece of code that belongs to data memory is prevented. This technique is useful in preventing buffer overflow attacks. A similar purpose is served by pre-set values placed on the stack called stack canaries. Before control flow change, these values are checked to verify that specific locations (e.g., a return address) have not been modified [28]. Shao et al. [34] describes another technique to defend against buffer overflow attacks, where bound checks are used and function pointers are protected by XORing them with a secret key. Although, none of these approaches consider a case where an attacker tries to overwrite instruction memory.

Zhang et al. propose the use of co-processors to monitor operating system kernel data structure [41]. In this approach, a hardware co-processor is implemented separately. The idea of information flow is used to determine if data is authentic or malicious in the system proposed by Suh et al. [37]. Alternatively, embedded operating systems can create a separate task to evaluate the control flow of multiple tasks [28]. However, these types of systems require the design of a complex operating system and its integration with the processor. Advanced security approaches such as virus scanners and trusted execution hardware which are found in general-purpose computers are not appropriate for many low-end embedded systems. The solution presented in this work has simple interfaces to the micro-architecture. Our system uses dedicated hardware monitors to monitor the system and reduce the vulnerability of the system without using any user-level code.

2.1.3. Recovery

A large portion of prior work related to embedded processor recovery has focused on recovery from hardware faults rather than external attacks. Soft errors in the processor datapath are detected and recovered with minimal performance loss using datapath

redundancy in the system proposed by Bournoutian et al. [5]. The concept of checkpoints and rollbacks [27] are used in many embedded processors in the presence of detected faults, which leads to fast recovery. These approaches are effective for the occasional soft errors, but are insufficient for a targeted attack. Unused processors in a multi-core network are used to provide redundancy in the system by Luo and Fan [20]. So even if a specific core fails, processing is moved to an idle core. Although all these techniques are effective, none of them address monitoring and recovery from network attacks.

2.1.4. Embedded Processor Context Switch

The frequent context switching in embedded processors has made the use of task monitoring complicated. Numerous prior approaches have been developed to reduce the impact of context switches on real time behavior. Some of these approaches are applicable to the saving of monitor state. Isolation of possible context switches to points in a task when the live sets of registers are at a minima was proposed by Zhou et al. [42]. This can accelerate context switch time because these points reduce the amount of register information which must be saved. Alternatively, by breaking down larger task into smaller ones, it is possible to reduce the need for pre-emptive context-switching [4]. But the need to schedule many small tasks could be a challenge for an embedded operating system. Multiple threads can be compressed into a single thread as another extreme approach [35]. Although this would increase resource usage as this eliminates thread switching and all tasks must be compiled and monitored together.

2.2. Single-Task Monitoring System

This section describes the working of the single-task monitoring system which has been proven effective in defending network processors from network attacks [17]. Network processors are designed to execute network applications (e.g. IPv4) effectively and consist of a single-threaded workload. Since it is critical to detect deviation in expected behavior of the

application within a single cycle, it is important that the monitor is able to retrieve the next state information of every instruction which is being monitored within a single cycle. The monitoring graph is essentially a state-machine where each instruction is represented by a state and an edge represents the transition from the instruction to its next valid instruction. A 4-bit hash of the instruction is used to label these edges and it helps in reducing the size of the monitoring graph. The monitoring graph is generated using the application binary as shown in Figure 4.

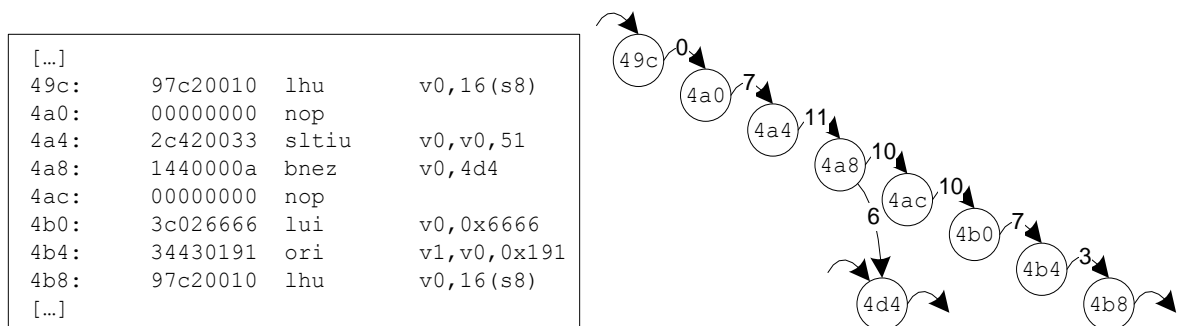


Figure 4: State machine generation from processing binary [17]

However, the next instructions from a control flow instruction share the same hash value and it leads to non-determinism in the monitoring graph. This non-determinism gets multiplied if the control flow instructions continue and it can lead to a complex implementation of the monitoring graph. Figure 5 illustrates such a scenario where state 3 and 5 can be reached from state 2 for input condition c . This problem is addressed by converting the non-deterministic graph into a deterministic graph by using powerset construction. Figure 6 illustrates the deterministic graph for the corresponding non-deterministic graph after powerset construction. It can be seen that states 3 and 5 are combined into a single state $\{3,5\}$ and can be reached from state 2 for input condition c .

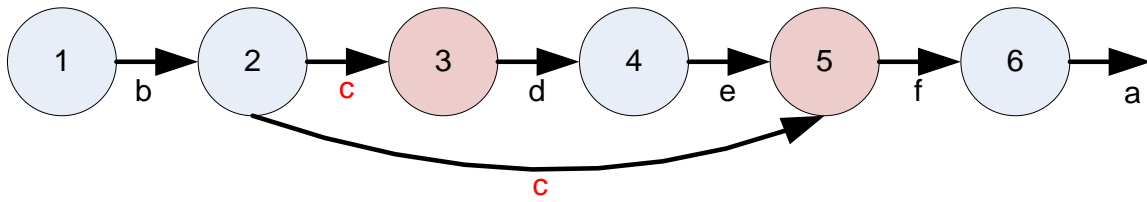


Figure 5: NFA state machine [17]

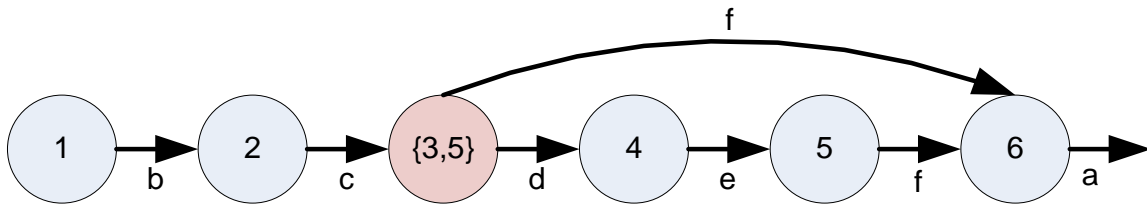


Figure 6: DFA state machine using powerset construction [17]

As mentioned earlier, it is critical that state transitions are implemented with one memory access per instruction and hence all necessary information is encoded in a single table entry and the states are grouped by the number of outgoing edges, i.e. a state belongs to group g if its previous state has g outgoing edges. For example, groups are shown with different colors in Figure 7. It can be seen in the figure that a state can belong to multiple groups (state f belongs to group 2 because a has two outgoing edges and to group 3 because e has three outgoing edges).

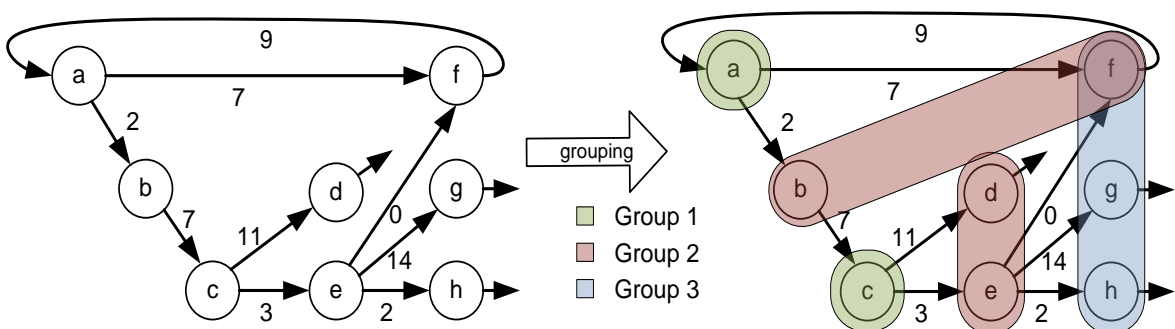


Figure 7: Grouping of states [17]

Figure 8 shows the system architecture of the single-task monitoring system. Each entry in the memory contains a sequence of number of next states, offset in state group, valid hash values on outgoing edges and the memory is logically divided into groups. A register file with 16 entries stores the base address for each group. The hash comparison logic determines if the one-hot coded hash bit of the incoming processor instruction is set in the valid hash values read from the memory and calculates the value of k , which is the position of matching hash among the valid hash values. If the one-hot coded hash bit is not set, then it means an illegal operation has taken place, indicating an attack and recovery measures are initiated. Otherwise, the next state transition is found by multiplying the number of next states and offset in state group and adding it to group base address and the value of k . Thus, by just one memory access, it is possible to determine the state-transition. The representation of the graph memory is compact and ensures high performance implementation of the hardware monitoring system. The single-task hardware monitoring system is able to reliably detect and recover from an attack without reducing the processing performance of the processor core.

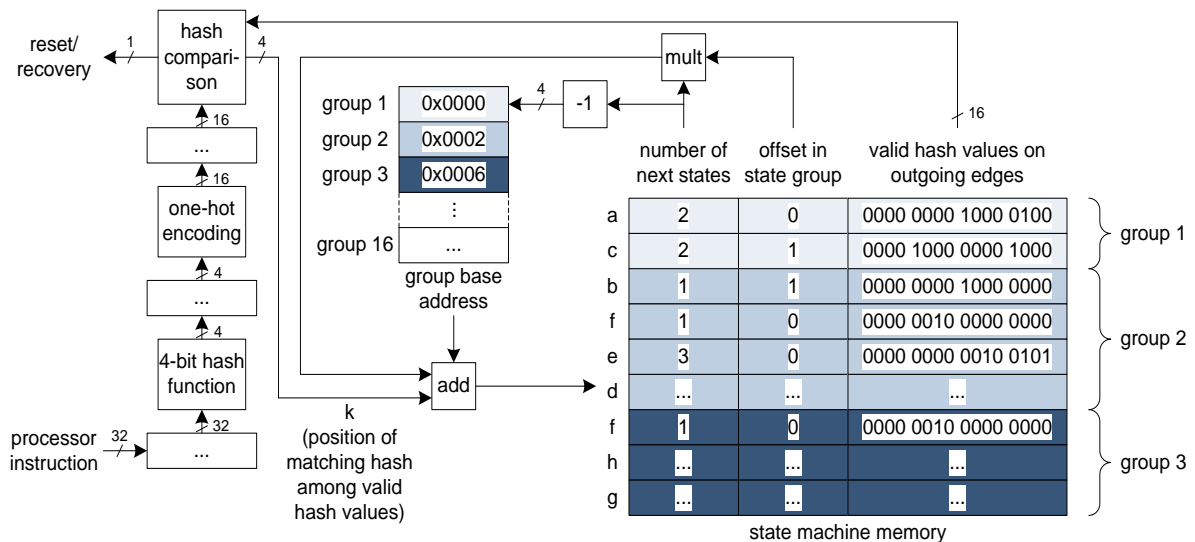


Figure 8: Memory representation of DFA monitoring graph [17]

However, this approach can only be applied for ensuring security in single-thread processor cores and is not sufficient to provide security for a real-time embedded processor with multi-threaded workload. Hence it is necessary to develop a Multi-task Hardware Monitor system that can coordinate with the embedded system to enable dynamic context switches and recovery when attacks are detected.

2.3. Summary

This chapter introduced related work that is essential for understanding the prototype system. Also a brief overview of the background work was provided. The next chapter outlines the system and security model used for this work.

CHAPTER 3

SYSTEM & SECURITY MODEL

To provide the necessary context for the Multi-Task Hardware Monitor system design presented in Chapter 5, the operation of MTHM and the security model for this work is briefly discussed [38].

3.1 Secure Processing with Hardware Monitors

Hardware monitors are components that are co-located with processor cores to track processing of software on that core. The objective is to access the operation of the processor and determine when incorrect behavior is detected (which can be due to faults or malicious attacks). As discussed in related work, there are a number of different approaches to monitoring based on what information is communicated from the processor to the monitor and what information is used to determine if that behavior is normal.

In this work, the hardware monitor receives information about every instruction executed on the processor core and compares it to a monitoring graph that is based on the analysis of the processing binary (similar to [21]). Each instruction is represented by a 4-bit hash value (to reduce the size of the monitoring graph compared to the size of the binary) and the state transitions correspond to possible control flow paths between the instructions. We use a deterministic finite automation (DFA) representation of the monitoring graph as described in Chapter 2.

The system architecture of the Multi-Task Hardware Monitor systems which supports multiple tasks is illustrated in Figure 9. The figure shows that application binaries are analyzed offline. During runtime, the comparison logic in MTHM matches the monitoring graph to the currently active task on the processor. To do the operation, the OS-to-Monitor Interface (OMI) communicates the necessary context information between the processor and

the monitor. When the processor execution does not match the expected behavior reflected in the monitoring graph of the current task, a task reset signal is sent from the monitor to the processor to terminate the current task.

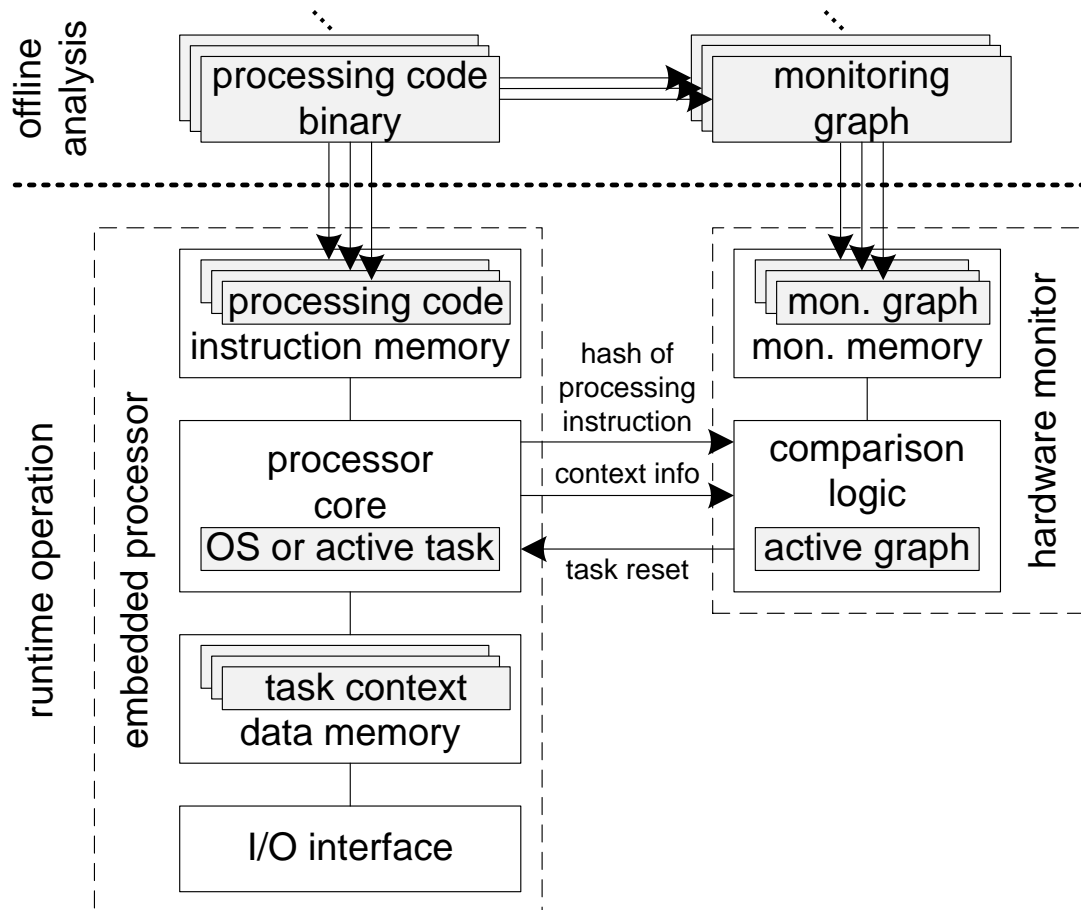


Figure 9: System architecture of Multi-Task Hardware Monitor system

It is important to note that the hardware monitoring system is isolated from the processor and thus cannot be tampered with remotely by the attacker (e.g., to change the monitoring graphs to match an attack). Related work discusses how to achieve such isolation while still enabling dynamic installation of hardware monitoring graphs through the use of cryptographic mechanisms [15].

It is to be noted that this thesis do not consider intrusion detection heuristics, which may be slow and computationally expensive. Instead, a novel multi-task monitoring system

that can detect deviations from normal processing with a single instruction is presented. Such fast detection is important for real-time embedded systems since the processing time for tasks may be only a few microsecond.

3.2. Operating System Interaction and Recovery

Operating systems are used in embedded systems to manage workloads that consist of multiple tasks that dynamically become active on various cores of the system. The system uses operating system to enable the coordination between the processor and hardware monitor as described above. When a context switch happens, the monitoring state needs to be saved, just like processor state needs to be saved in the processor core. Also we need to ensure that this coordination doesn't slow down the processor. The monitor can support multiple tasks running on the core without interfering with the real-time operation of the embedded system. Chapter 4 describes in detail the design of the OS-to-monitor communication mechanism in the system.

In addition, OS also assists in recovery steps when an attack is detected by the monitoring hardware since these recovery operations cannot easily be customized in hardware. Different applications require different levels of recovery (e.g., a simple processor reset for network operations or complex check-pointing for transaction-style processing). It is to be noted that the change in OS to support hardware monitoring and recovery is the only change to any software in the embedded system. In the system, on detection of deviation from expected processing behavior, the hardware monitor notifies the OS to kill the currently active task. The attack code is terminated by executing an Interrupt Service Routine (ISR) to kill the task. After terminating the malicious task, the embedded processor continues its normal operation and starts executing the next high ready task. Figure 10 illustrates the attack detection and recovery process in the embedded OS. Thus, the recovery process in the system doesn't affect other tasks running on the embedded processor core.

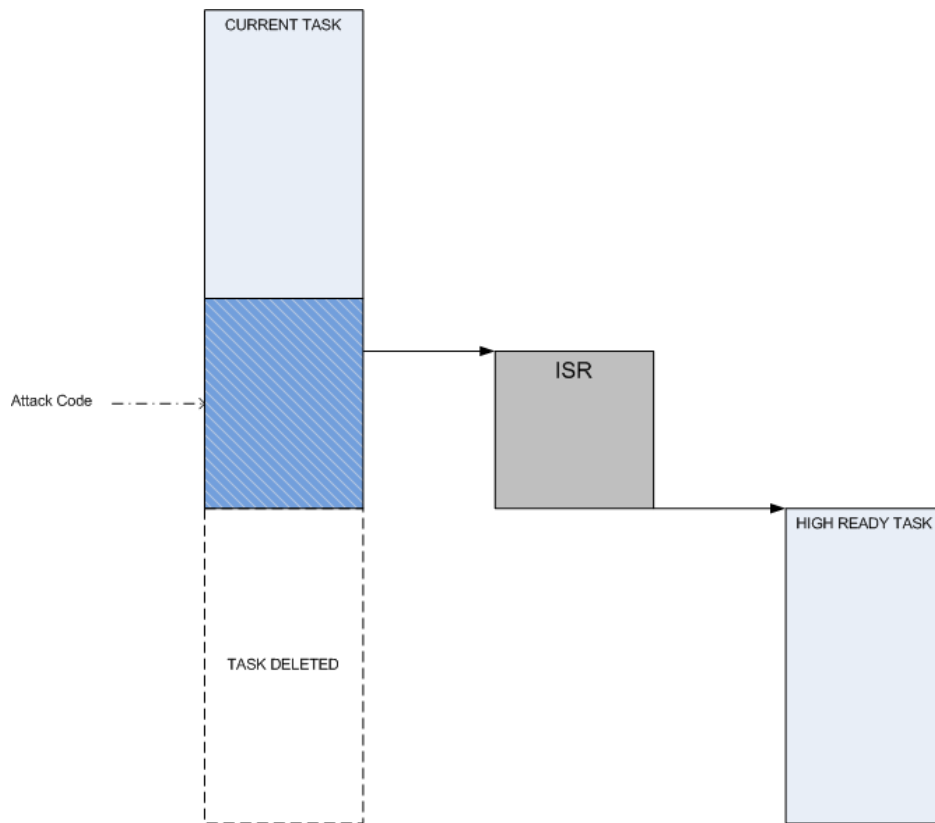


Figure 10: Attack detection and system recovery

3.3. Security Model

To justify how the system provides a secure processing environment, the security model that is basis for this work is discussed briefly.

3.3.1. Attack Example

In this work, the focus is on the class of attacks that aim to change the processing behavior of the embedded system. This class of attack is very board and encompasses a number of specific attacks. Hence one specific example is illustrated, but note that many other attacks are covered by this work.

In this scenario, the I/O functionality of the embedded system is used to cause the embedded processor to misbehave. Intentional modification of processor stack (e.g., stack smashing) is used to generate this type of attack. By making a controlled change in the stack,

it is possible for an attacker to change the control flow such that malicious code is executed. This type of attack is very common and is used in Internet to gain access to end-systems via vulnerable software. Code Red worm is one of the most famous example for this kind of attack that exploited a vulnerability in a service of the Windows operating system and used it to spread itself around the globe [6, 22]. As the growth of embedded systems increases, these types of attacks will continue to spread. In our prototype implementation, a similar attack code is used to test the system.

3.3.2. Security Requirements

It is required that the system meets the following security requirements:

- *SC1*: The system should only allow execution of code as programmed in the executable binaries of each task.
- *SC2*: Secure processing should be provided for multiple, dynamically changing tasks.
- *SC3*: Malicious code execution in one task should not affect other tasks.

In addition to security, these are also practical performance requirements. As it is shown in the results, the hardware monitor does not reduce the performance of the embedded processor in any way. The only overhead is a few instructions in the operating system code when switching tasks, which lead to a negligible reduction in processing speed.

3.3.3. Attacker Capabilities

The following assumptions are made about the capabilities of an attacker that tries to change the operation of the embedded system and/or tries to execute malicious code on the embedded system:

- *AC1*: An attacker can provide any input through input/output interface of the embedded system.

- *AC2*: An attacker can start and stop any task from an installed binary in the embedded system (within the limitation of a maximum number of active tasks).
- *AC3*: An attacker can tamper with any of the binaries.

In order to provide a practical solution for secure processing in an embedded system, we also require some reasonable constraints on attacker capabilities:

- *AC4*: An attacker cannot tamper with the operating system itself.
- *AC5*: An attacker cannot tamper with the hardware monitoring system (e.g., modifying monitoring graphs for installed executables)

As discussed above, this proposal does not discuss the secure installation of monitoring graphs, which has been solved in related work [15], in more detail. Next chapter provides a comprehensive discussion on the operating system management which is one of the key aspects in this research.

CHAPTER 4

OPERATING SYSTEM MANAGEMENT

In this chapter, we discuss about the OS activities which are communicated to the Multi-Task Hardware Monitor for its smooth operation. Task creation, context switch and task delete are the three operations whose information is vital to the Multi-Task Hardware Monitor. This chapter also talks about how the operating system communicates with the hardware monitor and keeps the monitor updated about its activities.

4.1. Processor Interface

The *processor interface* is a set of four registers created for communication between the operating system and the hardware monitor. Whenever *task create* or *context switch* or *task delete* occurs, the operating system writes into these registers regarding the details of the operation. The hardware monitor reads these registers to take necessary action to coordinate with the operating system as explained in Chapter 5. Figure 11 shows a simple block diagram of the processor interface.

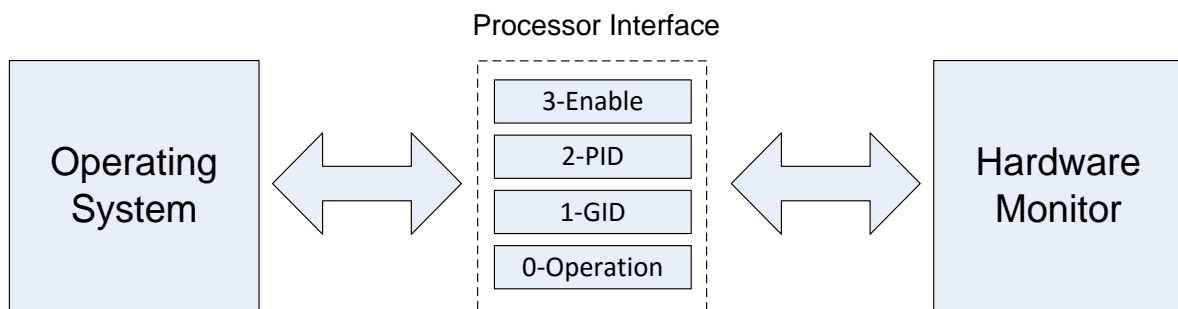


Figure 11: Processor interface

The Operation register is used to indicate what operation is happening in the OS. 0x01 indicates a *task create*, 0x02 indicates a *context switch* and 0x03 indicates a *task delete*. The GID register is used to write the graph ID (GID) of the newly created task in case of a *task*

create operation. The GID of a process is used to identify various graphs located in the hardware monitor. Section 5.3 explains in detail the role of GID in the system architecture. The PID register is used to write the process ID (PID) of the new task in case of a *task create* or PID of the newly scheduled task in case of a *context switch* or PID of the deleted task in case of a *task delete*. Whenever the Operation register is read by the hardware monitor, it flushes its content to acknowledge that it has been read. This also helps in avoiding any confusion when multiple *task create* or *context switch* or *task delete* operations happen consecutively. The Enable register is used to inform the hardware monitor if an OS activity is going on in the operating system or program execution is happening. If an OS operation is being executed on the processor, then this register is set to 0x0 and the hardware monitor need not monitor these instructions since we know that the operating system is secure and cannot be tampered with by an attacker. Once the program execution starts, this register is set to 0x1 and the hardware monitor begins to monitor the processor instructions. Thus, this register is used to enable/disable the hardware monitor.

4.2. Task Creation

A task/process is a simple program running on the CPU. In $\mu\text{C}/\text{OS-II}$, each task is given a unique priority and has its own set of CPU registers and stack. When a new task is created in $\mu\text{C}/\text{OS-II}$, it is assigned a task control block (TCB), which is a data structure used by $\mu\text{C}/\text{OS-II}$ to maintain the state of a task when it is preempted. Figure 12 illustrate how multiple tasks are handled in $\mu\text{C}/\text{OS-II}$ [18]. Multiple tasks reside in the memory and when a task is scheduled to run next, its TCB contents are moved to the CPU registers (context) for its execution.

The user creates a new task by calling *OSTaskCreate()*. This function requires four arguments: a pointer to the task code, a pointer to the argument that is passed to the task, pointer to the top of the stack that is assigned to task and the desired priority of the task. In

$\mu\text{C}/\text{OS-II}$, the priority of the task acts as the process ID since each task has a unique priority. The *OSTaskCreate()* is modified such that whenever a *task create* occurs, the hardware monitor is notified about it. The *OSTaskCreate()* also reports the *PID* and *GID* to the hardware monitor. Since *task create* is an operating system activity, the Enable register is set to 0x0 before entering this function.

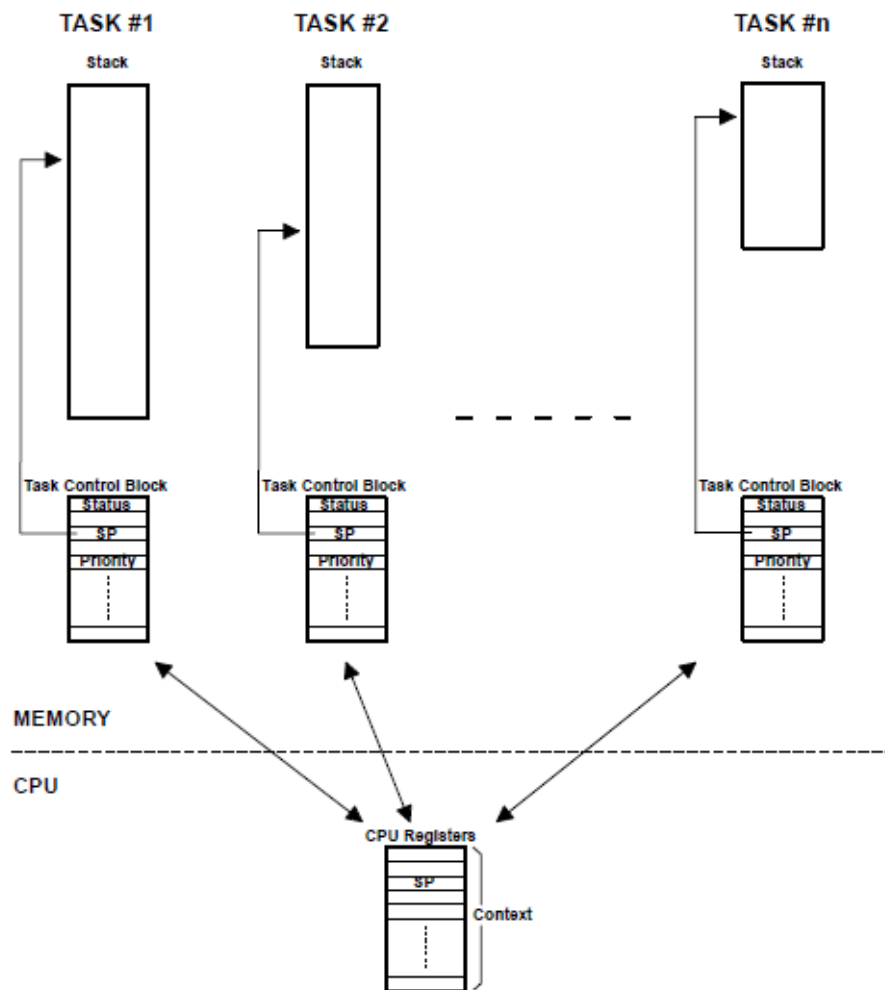


Figure 12: Multiple tasks in $\mu\text{C}/\text{OS-II}$ [18]

Figure 13 illustrates how the hardware monitor is notified of task creation. In this figure, it can be seen that the operation register is set to 0x1 to indicate a *task create*. Corresponding *GID* and *PID* are also written into the respective registers (in this case, *GID* and *PID* are both the priority of the task).

```

IINT8U OSTaskCreate (void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT8U prio)
{
    .....
    .....

    IOWR(PROCESSOR_INTERFACE, 1, prio);      // Report GID of new task
    IOWR(PROCESSOR_INTERFACE, 2, prio);      // Report PID of new task
    IOWR(PROCESSOR_INTERFACE, 0, 1);         // Report a task create operation

    .....
    .....
}

```

Figure 13: Reporting task create in software

4.3. Context Switch

A context switch happens when the OS kernel decides to run a different process on the CPU. A context switch is triggered by an OS event or a timer. Before this can happen, the OS needs to save the current task's context (CPU registers) in its stack and restore the new task's context from its stack. The priority of each task decides which task needs to run next. Once the next task for execution is determined, the operating system initiates the context switch process. The scheduler decides which task to run next. Like most commercial real-time OS, $\mu\text{C}/\text{OS-II}$ uses a preemptive type of kernel, which means whenever a higher priority task is ready to run, the current active task is suspended and the higher priority task is given control of the CPU [18]. Figure 14 shows the context of the CPU registers being saved into the TCB of the current task (*OSTCBCur*) during a context switch. Next, the higher priority task (*OSTCBHighRdy*) is made the current task (*OSTCBCur*) and the states of this task are restored into the CPU registers as shown in Figure 15.

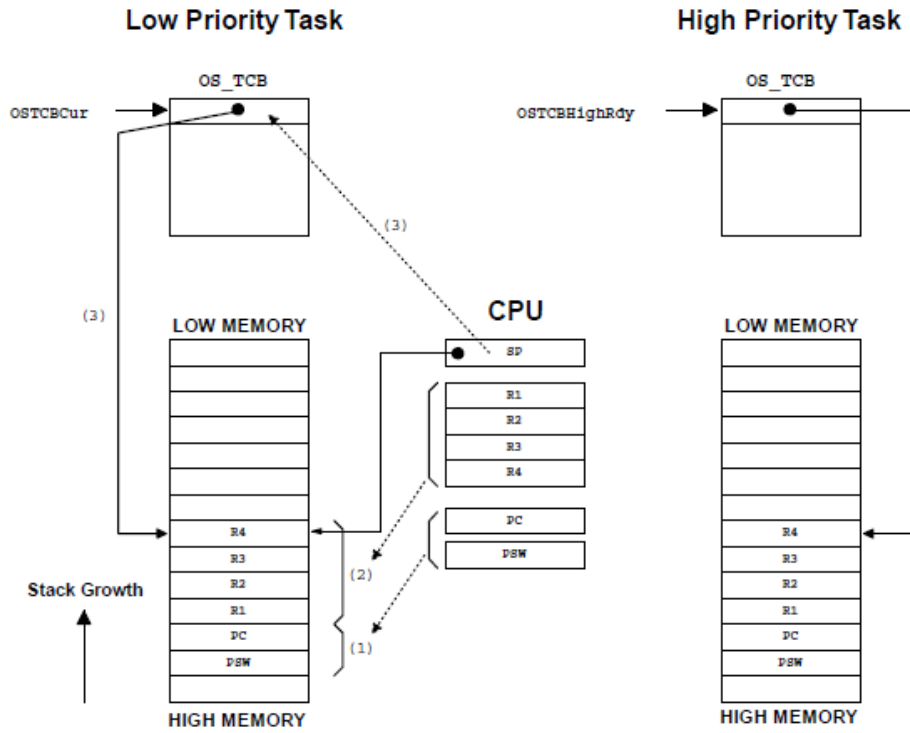


Figure 14: Context save of suspended task [18]

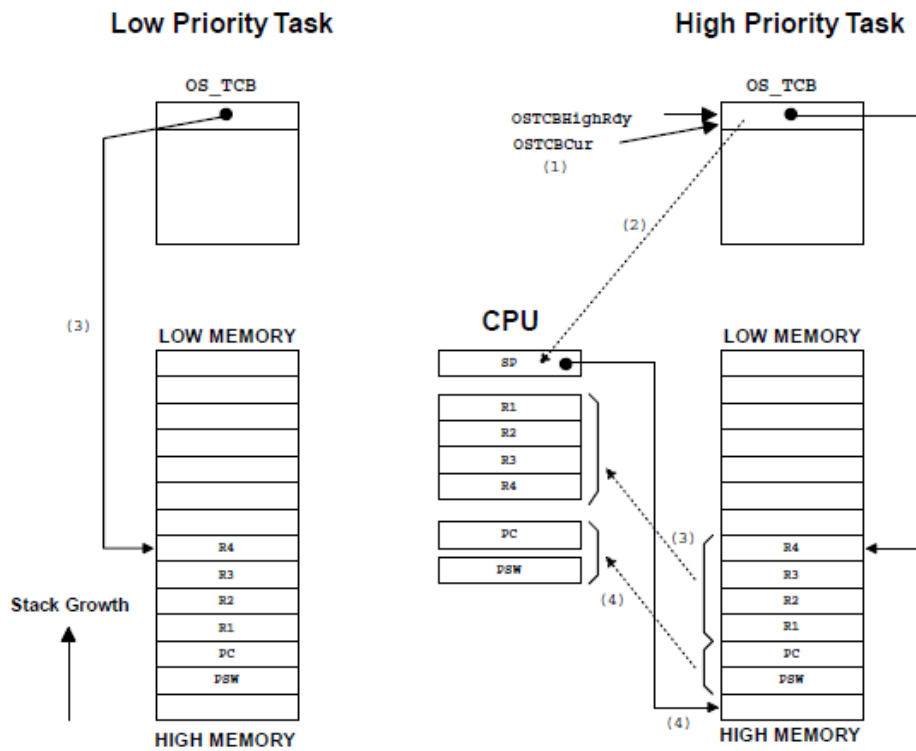


Figure 15: Context restore of next task [18]

In $\mu\text{C}/\text{OS-II}$, *OS_Sched()* is used to determine the next task which is ready to run and this function is modified to notify the hardware monitor about *context switch* and *PID* of the next task. As before, a *context switch* is an operating system activity and hence the Enable register is set to 0x0 before entering this function. In Figure 16, it can be seen that the *OS_Sched()* writes 0x2 into the Operation register to indicate a *context switch* and also writes the *PID* of the next task into the PID register (which in this case is *OSPrioHighRdy*).

```
void OS_Sched (void)
{
    .....
    .....

    IOWR(PROCESSOR_INTERFACE, 2,OSPrioHighRdy);    // Report PID of next task
    IOWR(PROCESSOR_INTERFACE, 0, 2);                // Report a context switch operation

    .....
    .....
}
```

Figure 16: Reporting context switch in software

4.4. Task Delete

Sometimes we need to delete a task because it is no longer required to run on the processor or because it is identified as a malicious task. Deleting a task returns it to a dormant state and the task is no longer scheduled by $\mu\text{C}/\text{OS-II}$ [18]. A task can be deleted by calling *OSTaskDel()*. *PID* of the task to be deleted is given as the argument to this function. Once the task is deleted, its associated *OS_TCB* is freed and can be used by another task to be created.

```

void OSTaskDel (INT8U prio)
{
    .....
    .....

    IOWR(PROCESSOR_INTERFACE, 2, prio);      // Report PID of deleted task
    IOWR(PROCESSOR_INTERFACE, 0, 3);        // Report a task delete operation

    .....
    .....
}

```

Figure 17: Reporting task delete in software

It is necessary to inform the monitor about a *task delete* as it helps in removing redundant graph information from the monitor and creates space for another monitoring graph. It can be seen in Figure 17 that the *OSTaskDel()* is modified to communicate the necessary *task delete* information to the monitor. 0x3 is written into the Operation register to indicate a *task delete* and the *PID* (prio) of the deleted task is written into the PID register. The Enable register is set to 0x0 before entering *OSTaskDel()* since *task delete* is an OS activity.

4.5. Summary

This chapter talked about how the OS communicates with the hardware monitor. Also the role of processor interface in signaling the monitor about the OS activity was discussed. Next chapter explains in detail the system architecture and how the monitor behaves to signals received from OS.

CHAPTER 5

SYSTEM ARCHITECTURE

In this chapter, the system architecture for the system is presented. This chapter talks about how the activities in the operating system are coordinated with the Multi-Task Hardware Monitor to ensure continuous tracking of processor operation [38].

5.1. Task Management in the Operating System

A key aspect of our monitoring system is its ability to fit seamlessly within the context switch operations of a typical operating system. As noted in Chapter 6, the time required to switch monitoring graphs for different task is significantly less than the typical time required for other activities in a context switch. In this implementation, graph switching is synchronized with other OS actions (e.g., register file save and restore) that occur during a context switch so that user tasks are protected at all times, Typical context switch activities for embedded operating systems, such as μ C/OS-II used for this work include:

1. A timer or other OS event generates an interrupt triggering a context switch.
2. The OS scheduler determines the next process for execution. This implementation uses a priority based scheme, although round-robin or other schedulers would also be appropriate.
3. The OS provides the PID of the next process to the monitoring system, triggering a monitoring graph switch in the monitor. This switch includes monitor state saving for the process currently being monitored, and a restoration of monitoring state for the next process.
4. Concurrently, the OS saves process state (registers, program counter, etc.) for the current task to main memory.

5. The OS retrieve process state for the next process from main memory and restores it to processor registers.
6. The OS checks the status of the monitoring system to confirm that the monitor for the next process is ready to use.
7. The OS sends a trigger to the monitoring system to start monitoring for the newly-loaded processes.

After the context switch is completed, the processor sends every instruction executed for the process to the monitoring system. In the next section, we provide a detailed view of the monitoring system and how it interacts with the processor for steps 3, 5, and 6 above.

5.2. Multi-Task Hardware Monitor System

A detailed view of our monitoring subsystem is shown in Figure 18. The portions of the monitoring system can be split into *monitoring hardware* (three boxes in upper left corner of the figure), which checks the per-instruction operation of the companion processor, *graph memory*, which stores states information about monitoring of each process, *controller and processor interface*. Design of the *controller* was done by Arman Pouraghily.

The monitoring hardware checks each processor instruction using information from the monitoring graphs stored in graph memory. In the figure, graphs for four separate applications are stored in *slots* in the graph memory. Each graph includes one row per instruction, effectively representing expected program control flow as a state machine [17]. A *read address* pointer indicates the entry in the graph that corresponds to the instruction that has just completed execution. During the execution of an instruction, a multi-bit (in this case 4-bit) hash value of the instruction is generated and converted to a one-hot representation. This one-hot encoding is compared against expected next-instruction hash values (*valid hash*) that are stored in the graph entry for the previously executed instruction. Since branch instructions may have several possible next instructions, and, consequently, several possible

valid hashes, multiple one-hot valid hash bits may be set per entry. A match of any of these hashes indicates a valid instruction. If no match occurs, an illegal instruction has been executed, leading to the generation of a recovery signal which is used by the processor for process termination.

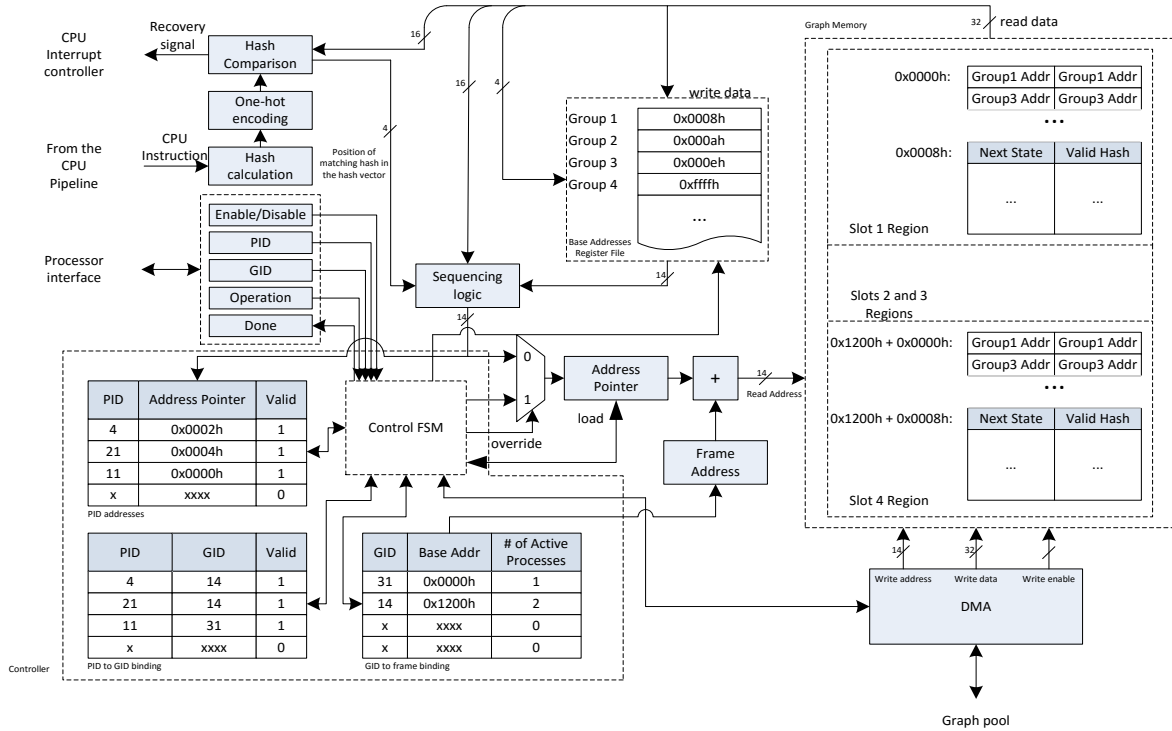


Figure 18: Detailed view of multi-context monitoring system

The next *read address* (memory row) in the monitoring graph is determined using next state information stored in the current entry, the matched hash value, and information stored in *base address registers* which group states based on fanin count [17]. These values are combined via addition in the *sequencing logic box* in the figure. The resulting address is stored in the *address pointer* and subsequently added to the start address for the appropriate graph slot for the application (*frame address*). The implemented monitor requires only one memory lookup per instruction.

Effectively, the monitoring information for each process at any given point in execution is defined by the contents of the *address pointer*, the monitoring graph for the process and the contents of the *base address registers*. If a context switch is requested, these values must be updated to use values for the requested next process.

5.3. DMA Interface to Centralized Graph Memory

The graph memory shown in Figure 18 stores the monitoring graph information for all the active processes in the embedded system. However, there are other processes which could become active at some point later in time whose monitoring graph information is not present in the graph memory. Hence a centralized graph memory is used to store the monitoring graph information of all the processes and the monitoring system copies the necessary graph information from the centralized graph memory when a new process is executed on the embedded processor system whose graph information is not available in the monitor graph memory. A new monitoring graph can be securely downloaded into the centralized graph memory by using the approach mentioned in related work [15]. There are two main reasons why it is advantageous to have a centralized graph memory instead of storing all the monitoring graphs in the graph memory of the monitoring system:

1. Downloading new monitoring graphs into the centralized graph memory doesn't get in the way of the operation of the hardware monitoring system. New graphs can be downloaded securely into the system even when the monitoring system is tracking the embedded processor operation for any deviations.
2. The centralized graph memory can be shared between multiple hardware monitoring systems to monitor multi-core embedded systems. This can help in reducing the resource usage of the system. Monitoring systems can avoid storing redundant graph information and copy new graph information from the centralized graph memory when needed.

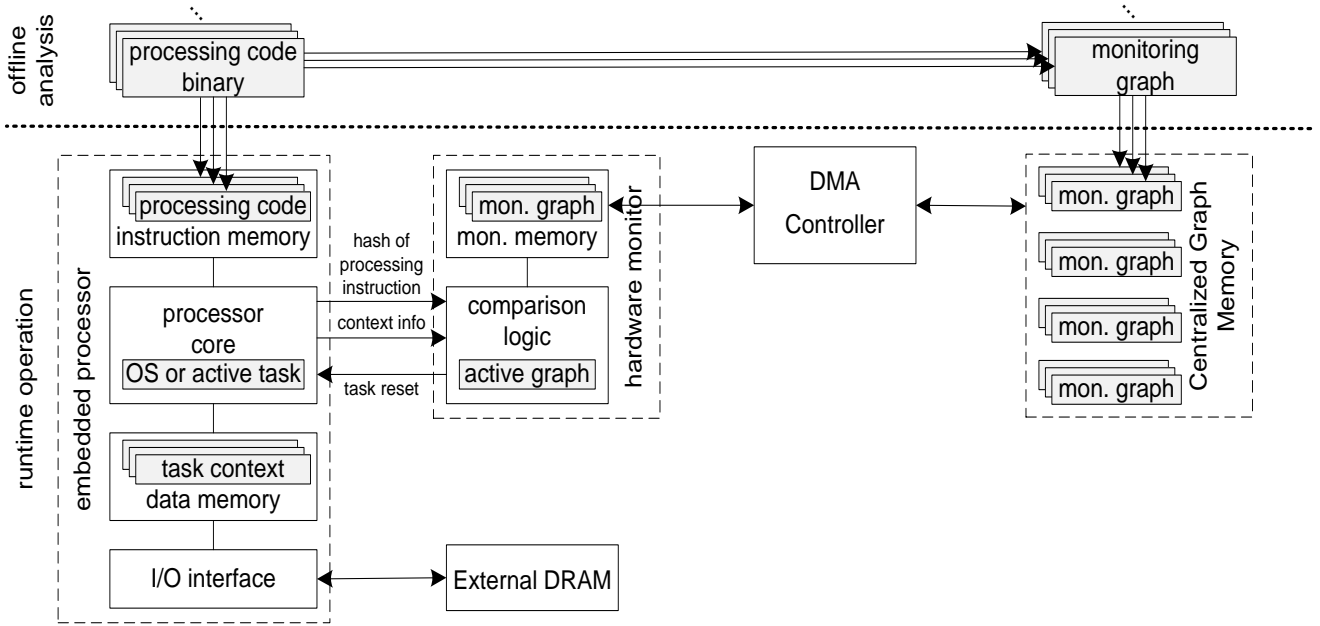


Figure 19: System architecture with centralized graph memory

Figure 19 illustrates the system architecture with centralized graph memory. The monitoring graphs of the applications are stored in the centralized graph memory. When the hardware monitor requires a new graph to be loaded into its monitor memory, it informs the DMA controller and the DMA controller facilitates the graph transfer from the centralized graph memory to the monitor memory.

A DMA controller as shown in the Figure 20 is used to interface the centralized graph memory and the monitoring system. When a new process is created in the embedded processor, the OS reports the *GID* and the *PID* of the new process to the hardware monitor. The hardware monitor checks whether the graph information for this new process is available in the monitor graph memory. This can be done by checking the *GID* to *frame binding* storage. If not, it sends the *GID* of the newly created process which is obtained from the *processor interface* to the DMA controller. The DMA controller uses this information to locate the address of the graph in the centralized graph memory using a look-up table in the

DMA controller. After locating the address of the graph, the DMA controller initiates the transfer of the graph information from the centralized graph memory to the monitor graph memory.

The DMA operation begins by setting the *DMA_start* signal high. The *GID* of the graph to be copied is passed to the DMA controller, which is used to locate the address of the graph in the centralized graph memory. Once the graph is located, the graph transfer operation is initiated by setting the *Write* signal high. The address and data of the new graph are loaded through the *DMA_address* and *DMA_data* ports respectively. The time required to load a new graph from the centralized graph memory to the monitor graph memory depends on the number of entries in the monitoring graph of the new process since it takes 1 cycle to copy one memory entry from the centralized graph memory to the monitor graph memory. Once the graph transfer operation is complete, the *DMA_done* signal is set high.

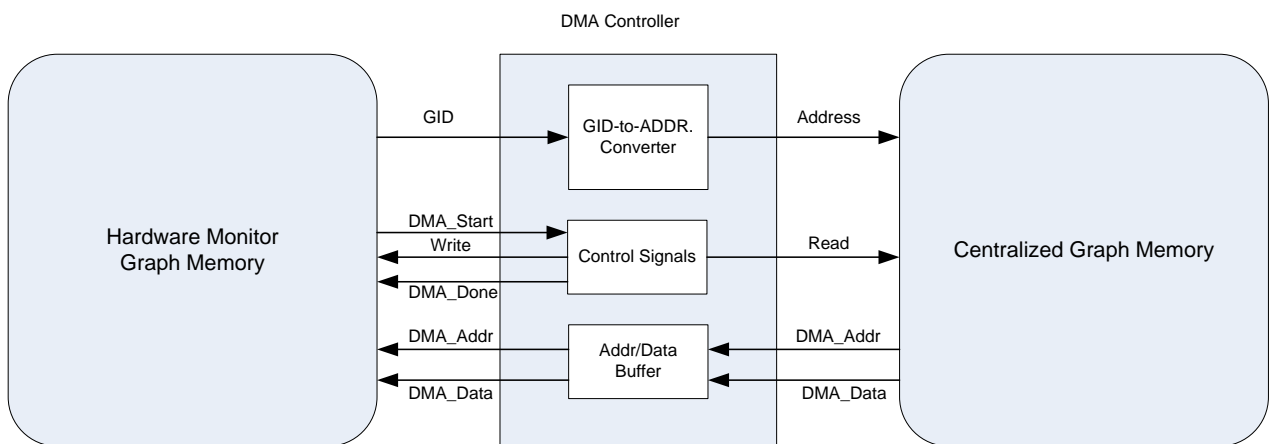


Figure 20: DMA controller interface

5.4. OS-to-Monitor Interface for Context Switch

In case of a context switch, control information is exchanged between the processor and the monitoring system. The exchange of monitoring information starts when the processor writes the *PID* of the next process into the *PID* register in the *processor interface* of the monitoring system and sets a bit in the Operation register. The monitoring system *control FSM* then performs the following actions:

1. The *address pointer* for the currently executing process is saved in the *PID* address storage so that it can be restored for the next invocation of the process.
2. The *GID* associated with the next process is located in the *PID* to *GID* binding storage using the *PID* written to the processor interface.
3. If the *GID* of the next process differs from the ID of the previous one, the *base address registers* are loaded with values for the graphs of the next process. These values are loaded from the graph memory (e.g., Group0 Addr, etc).
4. The *GID* is used to determine the *frame address* for the start of the appropriate monitoring graph in graph memory for the process. This information is stored in the *GID* to *frame binding* storage.
5. The *address pointer* value for the next process is restored from the *PID* addresses storage.
6. The *Done* bit is set in the *processor interface* indicating that the monitoring system is now ready to monitor the next process. This bit can be read by the processor.
7. Once all other context switch activity for the next process has concluded (e.g., processor registers are loaded), the processor sets an Enable bit in the *processor interface*, restarting monitoring. The processor waits until this bit set is successfully made, ensuring synchronization. Instructions of the newly-loaded process are then monitored.

In chapter 6 it is shown that these steps can be performed in 18 clock cycles for our prototype system.

5.5. OS-to-Monitor Interface for Process Creation

When a new process is being created by the OS, it is assigned a unique *PID* and *GID* by the operating system. Since many processes of the same application may exist, the *GID* may not be unique. The following steps are used to initialize the security monitor for the new process.

1. The two identifiers (*GID* and *PID*) are passed to the monitor via the *processor interface*.

The monitor first searches for an empty slot in the *PID addresses* storage and *PID to GID binding* storage to insert the new bindings.

2. While making these associations, the *GID to frame* binding storage is searched to determine if the appropriate graph is already loaded. If it is available, the next step is skipped.
3. If the *GID* is not found in the *GID to frame* binding storage, the *GID* is inserted into the table. The new graph is then loaded from the centralized graph memory using the DMA interface. Following graph loading, the base addresses are updated.
4. The *Done* bit is set in the *processor interface* indicating that the monitoring system is now ready to monitor the next process. This bit can be read by the processor.

During system startup, monitoring graphs are loaded from an external centralized graph memory for the new processes that will be executed by the processor. If a new process replaces an existing one, the *PID addresses*, *PID to GID binding*, *GID to frame binding*, and *graph memory* are updated to include information about the new process. This update is made via the *Control FSM*. Concurrently, the processor performs a series of process creation operations including initialization of the process stack and control block (registers, etc.). In Chapter 6, it is noted that while process creation can require hundreds of cycles for the processor, if the appropriate monitoring graph is already in the monitoring graphs is already

in the monitoring system, monitoring information update for process creation requires 20 cycles for the monitoring system.

5.6. OS-to-Monitor Interface for Process Deletion

When a process is deleted by the OS, the processor writes the *PID* of the deleted process into the *PID* register in the *processor interface* of the monitoring system and sets a bit in the *Operation* register. The following steps are used to remove the corresponding monitoring graph information from the monitoring system.

1. The *PID* of the deleted process is passed to the monitor via the *processor interface*. The monitor first searches for a matching entry in *PID addresses* storage and *PID to GID binding* storage. If found, the *valid* pointer for that corresponding *PID* is set to 0.
2. The *GID* associated with the deleted process is located in the *PID to GID binding* storage using the *PID* written to the processor interface. The *number of active processes* for that *GID* is decremented by 1 in the *GID to frame binding* storage
3. The *Done* bit is set in the *processor interface* indicating that the monitoring system is now ready to monitor the next process. This bit can be read by the processor.

While loading new graph information from the centralized graph memory, *PID addresses* storage and *PID to GID binding* storage rows with 0 set for *valid* pointer are considered as empty rows and can be used by the new graph for updating its information. Thus, at any given moment, the hardware monitoring system can support monitoring of four active processes in the system.

5.7. Dual-core Monitoring System Design

In this section, we explain how the Multi-Task Hardware Monitor system can be extended to monitor dual-core embedded processors. The architecture of the dual-core monitoring system is illustrated in Figure 21. Two MTHM systems are co-located with the dual-core embedded processor to track the processing of the software on individual cores as

in the case of a single-core monitoring system. The centralized graph memory is shared among both the MTHM systems and graphs can be loaded into the monitoring systems when required without slowing down the embedded system.

An arbiter is used to access the centralized graph memory. When a monitoring system needs to copy a new graph from the centralized graph memory, it requests the arbiter to grant access to the centralized graph memory. If the centralized graph memory is free, then the arbiter grants access to the requesting monitoring system. If the centralized graph memory is busy, then the monitoring system requesting access waits until the centralized graph memory becomes free. If both the monitors make a request at the same time, priority is given to Monitor1 by default.

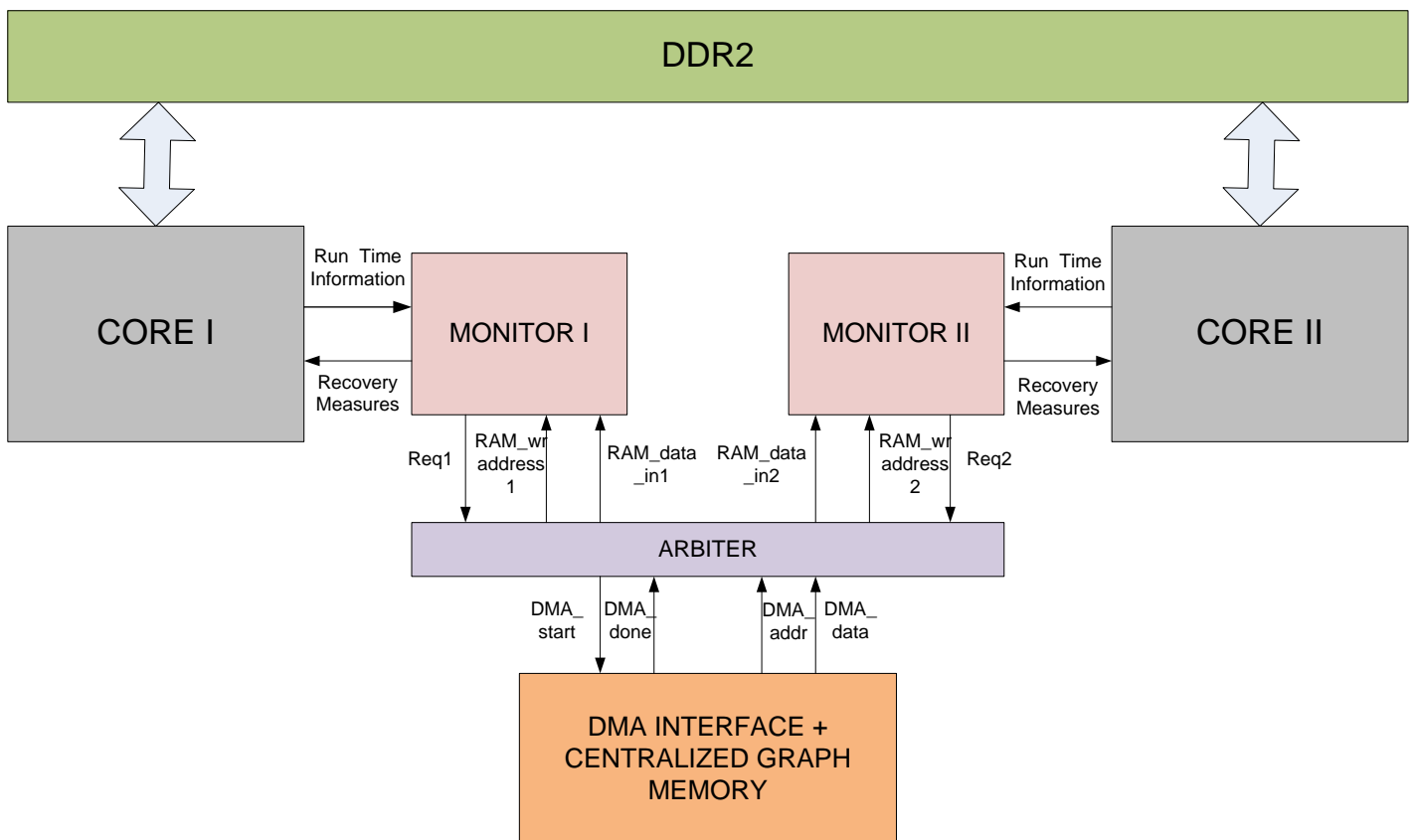


Figure 21: Dual-core monitoring system

When Monitor1 needs to copy a new graph from the centralized graph memory, it sets the *Req1* signal high. The arbiter on receiving this request from Monitor1, checks whether the centralized graph memory is used by Monitor2. The centralized graph memory is currently busy if the *DMA_Done* port in Figure 20 is low. If not, then the arbiter establishes a connection between the DMA interface and Monitor1 and the necessary graph can be transferred from the centralized graph memory to Monitor1. Now if suppose, Monitor2 also needs to copy a new graph from the centralized graph memory, it sets the *Req2* signal high. The arbiter receives this request and on checking the *DMA_Done* port, it finds out that the centralized graph memory is currently being used by Monitor1. The arbiter waits until the centralized graph memory is free and then establishes a connection between the DMA interface and Monitor2.

In Chapter 6, it is noted that process creation requires 600 cycles in the embedded processor while process creation in monitor requires on an average 140 cycles. Hence, even if one monitoring system has to wait till the other monitoring system finishes copying the graph, the waiting monitoring system will still have enough time to copy the necessary graph before process creation is finished in the embedded processor.

All task management activities such as process creation and context management and its interface to the monitoring system remains the same as in the case of the single-core monitoring system. This approach can also be extended to monitor multi-core embedded systems as the arbiter can manage multiple requests to the centralized graph memory. Priority of the process can be used to determine which monitor gets access to the centralized graph memory when multiple requests are made at the same time.

5.8. Summary

This chapter explained the system architecture for the monitoring system and the coordination of the operating system activities with the Multi-Task Hardware Monitor. The graph transfer operation from the centralized graph memory to the monitoring system was also discussed in this chapter. Finally, this chapter showed how the monitoring system can be extended to support dual-core embedded systems.

CHAPTER 6

PROTOTYPE IMPLEMENTATION

This chapter describes the system setup for the prototype implementation and shows how monitor context management and task management happens in the system. In the later sections, the system is verified by an attack code and shows how the monitor can detect the attack to defend the embedded processor system [38].

6.1. System Setup

To verify the functionality of our monitoring system, we implemented an embedded NIOS II processor plus monitoring system using a Stratix IV GX230 FPGA located on an Altera DE4 board. A single-core NIOS executing a μ C/OS-II operating system was used for testing. Monitoring logic and memory were implemented in on-chip resources. Since modern day embedded processing systems have numerous applications being executed on them, we have implemented an external DRAM interface to the system and use off-chip DRAM to support the storage of multiple application binaries and OS kernel. DDR2 SDRAM available on the Altera DE4 board is used for storing the application binaries and the OS kernel. DDR2 is the second generation DDR and offers a maximum transfer rate of 3200 MB/s. Figure 22 illustrates the DDR2 SDRAM interface system level diagram. An Altera DDR2 controller is used to handle the complex aspects of using DDR2 SDRAM which includes initializing the memory devices, managing SDRAM banks and keeping the devices refreshed at appropriate intervals [2]. Altera DDR2 SDRAM controller can be up to 90% efficient and hence offers a maximum transfer rate of 2880 MB/s. Figure 23 shows NIOS II setup in Qsys with DRAM controller interface.

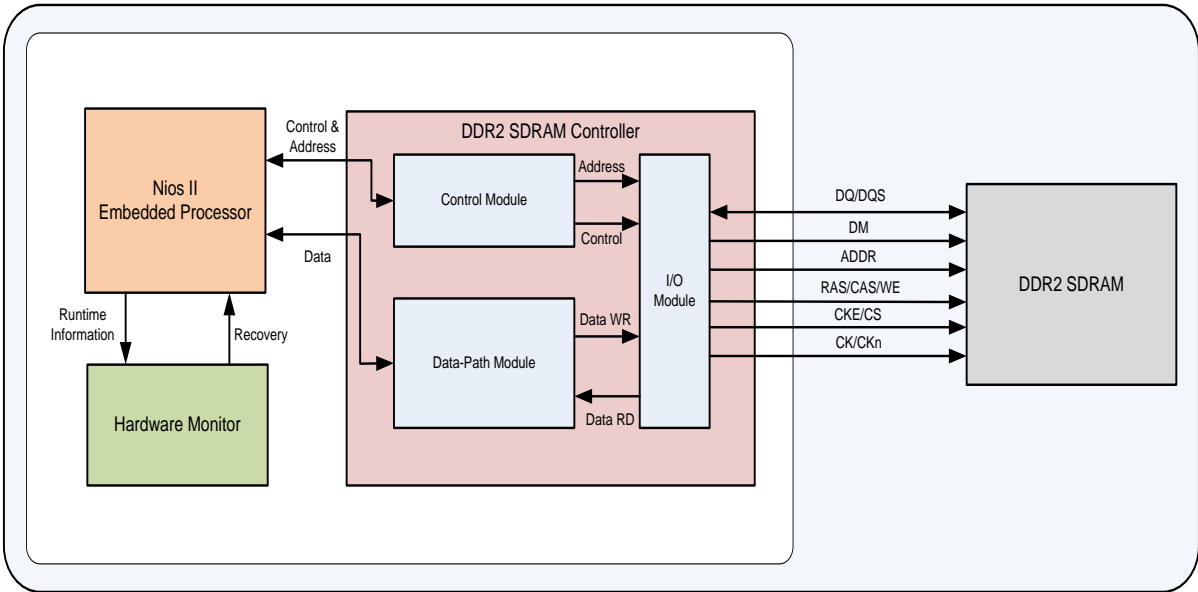


Figure 22: DDR interface

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
<input checked="" type="checkbox"/>		clk_0	Clock Source	clk	exported			
<input checked="" type="checkbox"/>		dk_in	Clock Input	reset	clk_0			
<input checked="" type="checkbox"/>		dk_in_reset	Reset Input					
<input checked="" type="checkbox"/>		dk	Clock Output					
<input checked="" type="checkbox"/>		dk_reset	Reset Output					
<input checked="" type="checkbox"/>		mem_if_ddr2_emif_0	DDR2 SDRAM Controller with UniPHY					
<input checked="" type="checkbox"/>		pll_ref_clk	Clock Input		clk_0			
<input checked="" type="checkbox"/>		global_reset	Reset Input					
<input checked="" type="checkbox"/>		soft_reset	Reset Input					
<input checked="" type="checkbox"/>		afn_clk	Clock Output		mem_if_ddr...			
<input checked="" type="checkbox"/>		afn_half_clk	Clock Output		mem_if_ddr...			
<input checked="" type="checkbox"/>		afn_reset	Reset Output					
<input checked="" type="checkbox"/>		afn_reset_export	Reset Output					
<input checked="" type="checkbox"/>		memory	Conduit	memory	mem_if_ddr...	0x0000_0000	0x3fff_ffff	
<input checked="" type="checkbox"/>		avl	Avalon Memory Mapped Slave	status				
<input checked="" type="checkbox"/>		status	Conduit	status				
<input checked="" type="checkbox"/>		oct	Conduit	oct				
<input checked="" type="checkbox"/>		nios2_qsys_0	Nios II Processor					
<input checked="" type="checkbox"/>		clk	Clock Input		clk_0			
<input checked="" type="checkbox"/>		reset_n	Reset Input		[clk]			
<input checked="" type="checkbox"/>		data_master	Avalon Memory Mapped Master		[clk]			
<input checked="" type="checkbox"/>		instruction_master	Avalon Memory Mapped Master		[clk]			
<input checked="" type="checkbox"/>		d_irq	Interrupt Receiver		[clk]			IRQ 0
<input checked="" type="checkbox"/>		jtag_debug_module_...	Reset Output		[clk]			IRQ 31
<input checked="" type="checkbox"/>		jtag_debug_module	Avalon Memory Mapped Slave		[clk]	0x4000_0800	0x4000_0FFF	
<input checked="" type="checkbox"/>		custom_instruction_m...	Custom Instruction Master		[clk]			
<input checked="" type="checkbox"/>		timer_0	Interval Timer		clk_0			
<input checked="" type="checkbox"/>		clk	Clock Input		[clk]			
<input checked="" type="checkbox"/>		reset	Reset Input		[clk]			
<input checked="" type="checkbox"/>		s1	Avalon Memory Mapped Slave		[clk]	0x4000_1000	0x4000_101F	
<input checked="" type="checkbox"/>		irq	Interrupt Sender		[clk]			
<input checked="" type="checkbox"/>		jtag_uart_0	JTAG UART		clk_0			
<input checked="" type="checkbox"/>		clk	Clock Input		[clk]			
<input checked="" type="checkbox"/>		reset	Reset Input		[clk]			
<input checked="" type="checkbox"/>		avalon_jtag_slave	Avalon Memory Mapped Slave		[clk]	0x4000_1030	0x4000_1037	
<input checked="" type="checkbox"/>		irq	Interrupt Sender		[clk]			

Figure 23: NIOS II setup in Qsys

Monitoring graphs were generated by passing code through a standard compiler flow to generate assembly-level instructions [17].

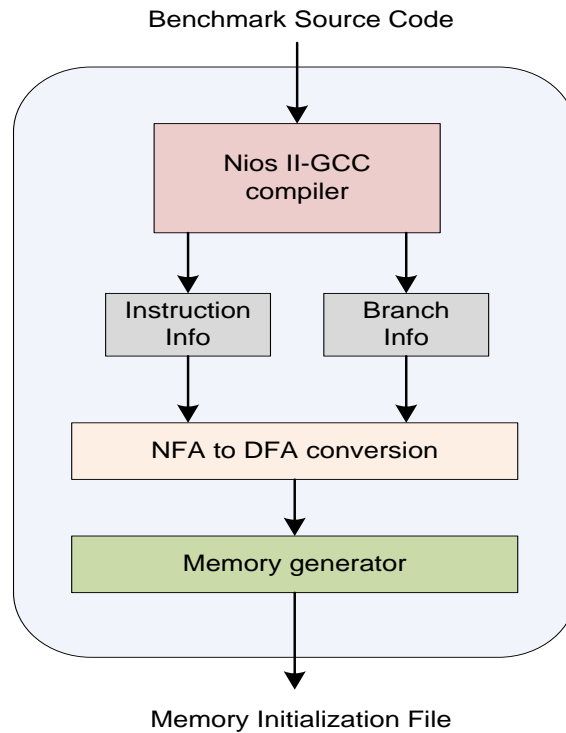


Figure 24: Offline analysis

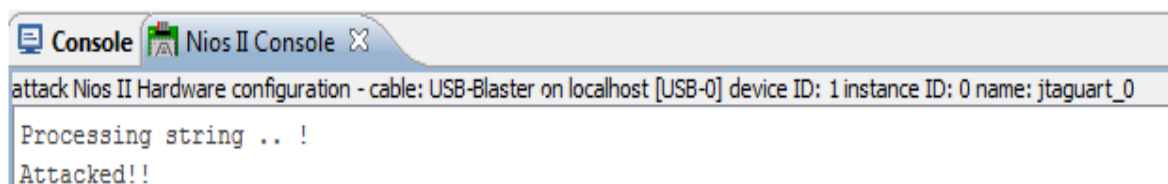
The output of the compiler allows for the identification of branch instructions and their target address. This information was used to generate monitoring graphs as shown in Figure 24 for five MiBench [13] applications (bitcount, qsort, stringmatch, basicmath and dijkstra) and malicious stack-smashing attack code. Monitoring graph information for all Mibench applications can be generated using offline analysis and our examination of all MiBench benchmarks determined that the target for all dynamic branches could be determined at compile time.

The attack code we use for our system is a simple C function which accepts a character string from an I/O port and copies it to a buffer located on the processor stack [36], as shown in Figure 24. The attack code was generated by Kekai Hu.

```
void process_input(char *stringpassed) {
    char name[90];
    strcpy(name,stringpassed);
    printf("Processing string .. !\n");
    return;
}
```

Figure 25: Attack code

In this poorly designed code, no check is made to determine if the string *stringpassed* is longer than the target buffer, so the return address of the function can be over-written with an address which points into the user-provided input string. Instead of characters, this “string” can contain processor instructions which repetitively print out “Attacked!!” on a terminal in a loop, although much more malicious behavior could be imagined. A monitor for the code is able to detect the unplanned control flow jump and kill the process before the attack can perform this activity. The hash value stored in the monitoring graph for the application will not match the values for the malicious instructions as they are executed during the attack. As shown in the Figure 26, we have confirmed that this attack will lead to unexpected results (an attack message) if monitoring is not used.

A screenshot of a console window titled "Nios II Console". The window shows the following text: "attack Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtaguart_0", "Processing string .. !", and "Attacked!!".

```
attack Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtaguart_0
Processing string .. !
Attacked!!
```

Figure 26: Console display during stack smashing

6.2. Monitor Context Management

The ability to perform numerous context switches between multiple processes of the two monitored MiBench benchmarks have been verified both via simulation and in emulation hardware. This switch includes both standard process state used by the processor (e.g., register information, stack) and monitoring information using the mechanism outlined in Chapter 5. Altera SignalTap, a hardware debugger, was used to generate the waveforms shown in Figure 27.

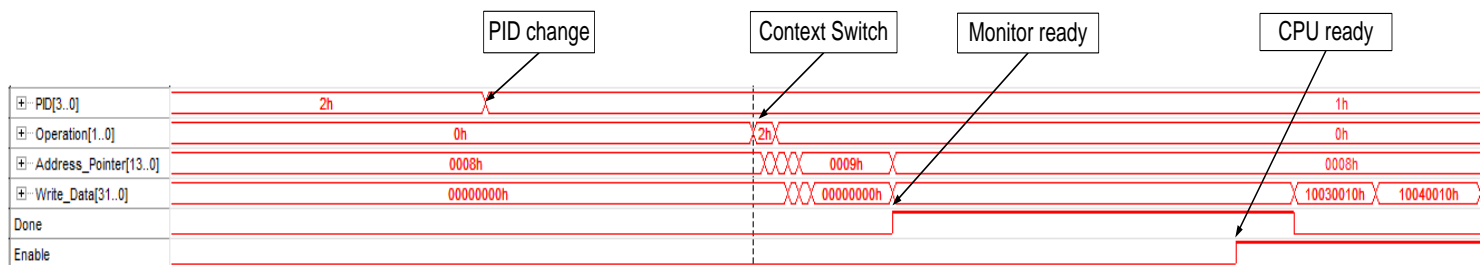


Figure 27: SignalTap waveforms showing monitor context switch

The waveforms show the synchronization between the processor and the monitor as a result of the context switch. First, the processor notifies the monitoring system of the switch by writing the *PID* of the next process into the processor interface. The monitor switch is started by the processor writing into the *Operation* register of the interface. The value of the *address_pointer* for the old process is stored and the value for the new process is restored to/from *PID* address storage immediately after this trigger. The *base address registers* are then configured using the *write_data* port shown in Figure 18. After the *control FSM* performs the monitor update, the *Done* signal is set in the processor interface indicating the monitor context switch is finished. Finally, after the processor finishes other context switch operations, it sets the *Enable* signal in the processor interface to restart monitoring. The

processor waits a cycle until this write is complete. Monitoring for the new process starts with the first instruction received from the process.

Experiments in simulation and in lab on FPGA hardware showed that the processor is able to process data for the MiBench benchmarks equally fast both with and without monitoring (e.g., no slowdown for monitoring). Context switch time is extended by 5 cycles versus no monitoring to allow for monitor context switches. This overhead accounts for the data exchanges between the processor and the monitoring system for synchronization. Overall, we found that the number of cycles needed to perform a monitor context switch is 18 versus the 34 cycles needed for the processor to save and restore registers (note that monitor and processor context switch occur in parallel).

6.3. Monitor Process Creation and DMA interface

The system has been verified for its ability to load new graph information into the hardware monitor when a new process is created in the embedded processor. If the graph information is not available in the graph memory of the monitoring system, then the required graph information is copied from the centralized graph memory through the DMA interface.

Figure 28 illustrates SignalTap waveforms for process creation activity in the monitor.

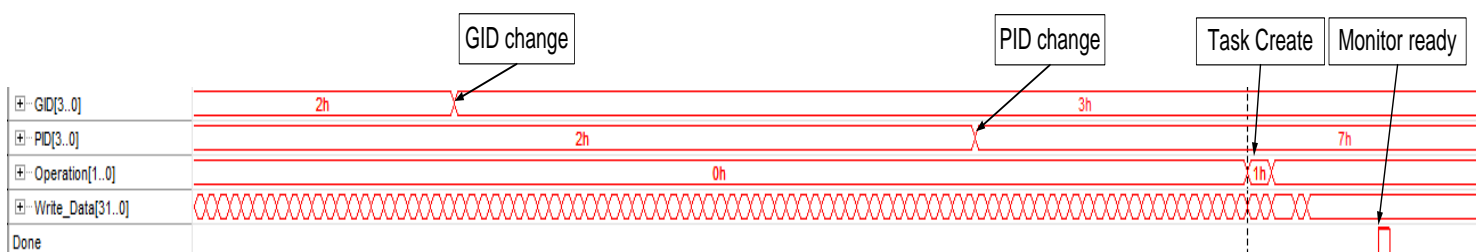


Figure 28: SignalTap waveforms showing monitor process creation

The processor first writes the identifiers (GID and PID) of the new process into the *processor interface*. The monitor process creation begins when the processor sets a bit in the

Operation register of the *processor interface*. If the graph information is present in the monitoring system, the new base addresses are configured using the *write_data* port shown in Figure 18. After the *control FSM* is complete, the *Done* signal is set in the processor interface to indicate the monitor task creation is finished.

If the GID of the new process is not found in the *GID to frame binding* storage, then the graph information for the process needs to be copied from the centralized graph memory. Figure 29 illustrates SignalTap waveforms during DMA operation when a graph is copied from the centralized graph memory to the monitoring system. It can be seen from the waveform that the DMA operation begins when *DMA_start* signal is set high. The GID of the graph to be copied is written into the *GID* port and this information is used to locate the graph in the centralized graph memory. Once the graph is located, the graph copy operation begins by setting the *DMA_wren* signal high. The address and data of the new graph are transferred using the *DMA_address* and *DMA_data* port shown in Figure 20. Once the DMA operation is complete, the *DMA_done* signal goes high.

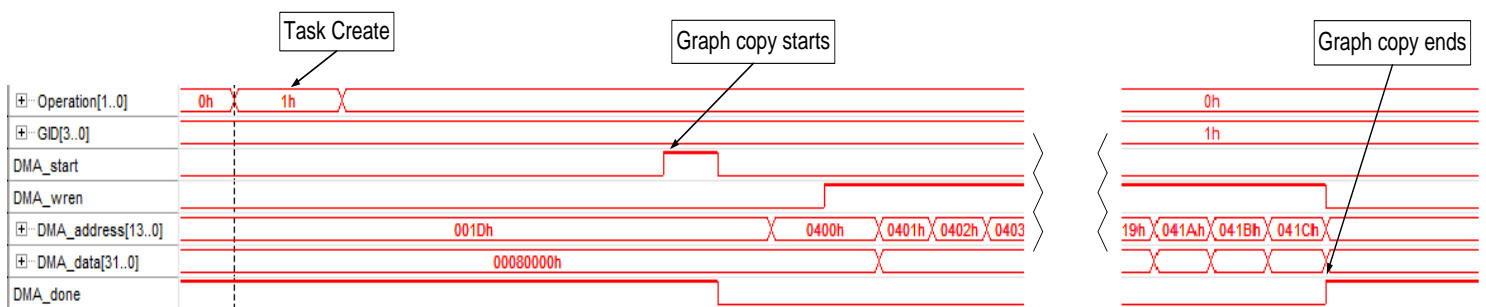


Figure 29: SignalTap waveforms showing monitor task creation

The amount of time needed to create a new process in the OS is about 600 clock cycles versus 20 to create process information in the monitor. If a monitoring graph is loaded from main memory, the cycle count required for the monitor increases to include reading the

number of rows in the monitoring graph for the new process into graph memory. It is noted that the number of rows in the monitoring graph is about 120 for each of our applications.

6.4. Attack Detection and Protection

It has been verified in both simulation and in hardware that the monitoring system is able to detect the stack smashing attack described above and notify the processor so that the malicious process can be terminated. SignalTap waveforms derived from observing hardware operation in system are shown in Figure 30. As described in Chapter 5, an attack is detected when the hash of the CPU instruction does not match the expected value stored in the monitoring graph for the application. In the implemented system, the hash function counts the number of ones in the instruction to form a four-bit hash value. The figure shows the four bit hash value, a one-hot version of the hash value, and the retrieved, expected hash value for the instruction from the monitoring graph (*read_data[15:0]*). In the waveform, it can be seen that the correct hash value is matched twice, but the third hash value is incorrect, indicating a branch to an unexpected section of code. As a result of this detection, a recovery signal is generated, notifying the processor that the process should be terminated.

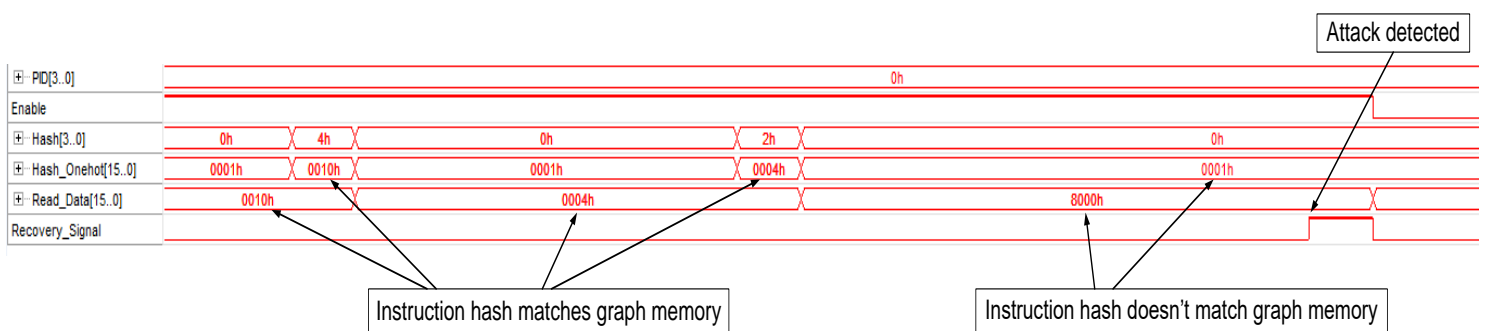


Figure 30: SignalTap waveforms showing detection of an attack

This recovery signal is used to generate an interrupt to the embedded processor. The interrupt causes the current process (which is also the malicious process) to halt and OS takes control over the processor. The OS saves the CPU registers and starts executing the interrupt handler. The interrupt handler determines what caused the interrupt to occur and starts executing ISR for that interrupt. In the ISR, the attack process is found by checking the PID of the process from the PID register of the *processor interface*. This process is then deleted and the ISR is exited. Now the OS scheduler determines the next ready task to run and starts executing it on the processor. Figure 31 shows the attack process being detected and deleted in NIOS II processor. It can also be seen from the figure that after the deletion of the attack process, the processor starts the execution of the next process.

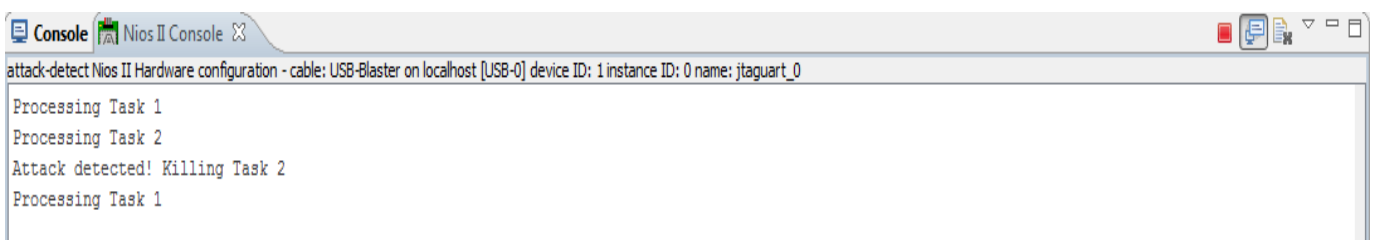


Figure 31: Console Display showing task detection and deletion

While the attack process is being deleted, its corresponding graph information is also removed from the monitoring system. SignalTap waveforms for graph deletion activity in the monitor are illustrated in Figure 32. In the waveform, it can be seen that the graph removal process starts when the processor writes into the Operation register informing about the task delete operation. The *Kill_PID* signal is made high to remove the corresponding PID entry from the *PID addresses* storage and the *PID to GID binding* storage. At the same time, *Update_GID* signal is made high to update the *GID to frame binding* storage. In the waveform it can be seen that the monitor removes PID-1 from the valid array of *PID addresses* storage and updates the valid array entry for GID-1 in *GID to frame binding*

storage by decrementing it by one. We can see that only the graph information for PID-0 and GID-0 exists in the monitor which corresponds to the Task 1 running on the processor as shown in Figure 31.

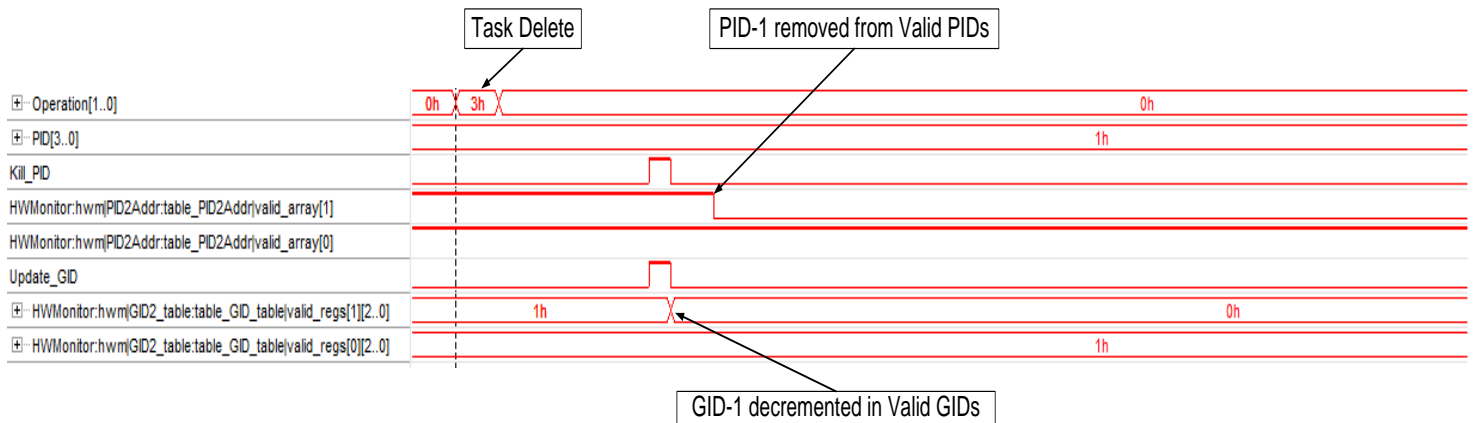


Figure 32: SignalTap waveforms showing monitor task delete

The time it takes for the system to recover after an attack is detected is reported in

Table 1.

Operation	# Cycles
Interrupt Latency	1
Saving CPU registers	25
Interrupt Handler	129
ISR Code	30
Task Delete	126
Total	311

Table 1: System recovery time

From the table, we can see that the *interrupt latency* is 1 cycle. Thus only 2 instructions of the attack code are executed on the processor and the attack task is killed before it can harm the system. Once the interrupt occurs, the OS takes control and *saves the CPU registers*. After this, the *interrupt handler* is called to determine what caused the interrupt to occur. The *interrupt handler* also disables certain OS features like context switching. Later the *ISR code* is executed, which determines the attack task by reading the PID register of the *processor interface* and deletes this task. The *task delete* operation takes 126 cycles. Overall, it takes 311 cycles to recover from the attack and to continue normal operation. Figure 33 illustrates the recovery process in the embedded system.

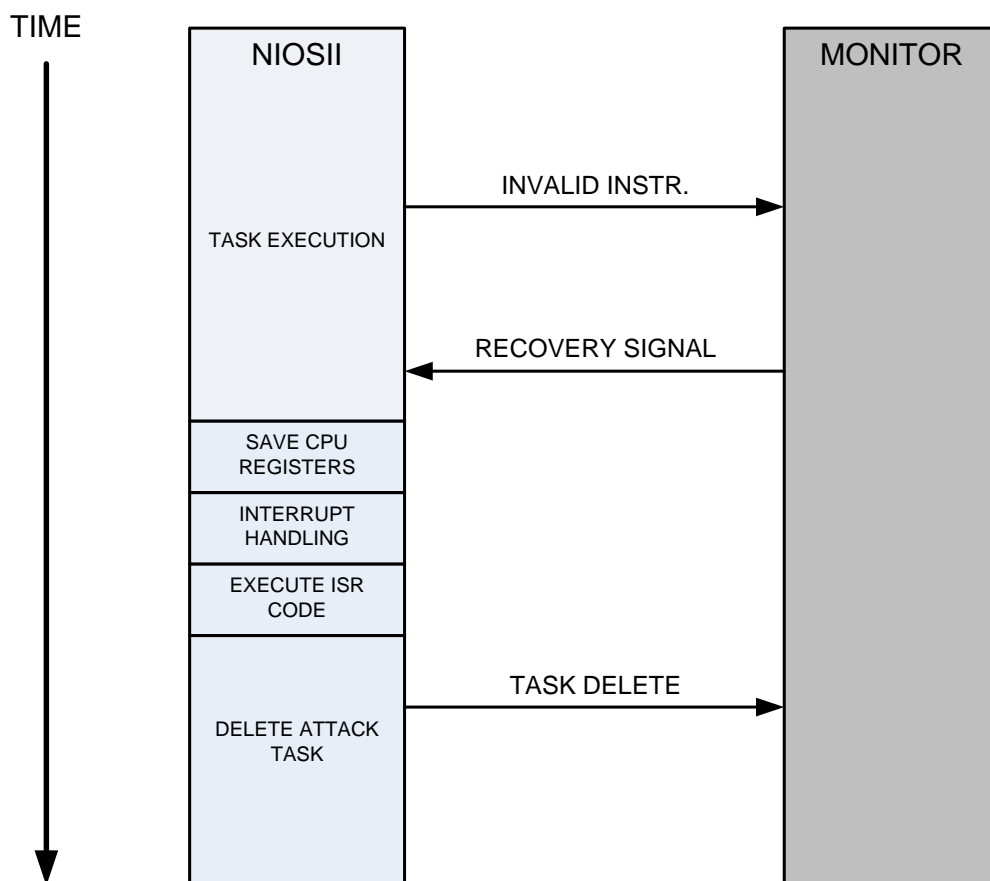


Figure 33: Recovery process in the embedded system

6.5. Monitoring System Resources

To provide some context regarding the amount of overhead required by the monitoring system relative to the processor, hardware results of the system reported by the Altera Quartus II tool are shown in Table 2. The lookup table (LUT), flip-flop (FF), and memory resources required for the monitor are appropriate compared to the processor core. Dynamic power values are also shown in the table. These power numbers were generated using Altera PowerPlay. Toggle rate of 12.5 was used for all the signals during power analysis.

Resources	Hardware Monitor	Nios II Processor	DDR2 Controller	Available
LUTs	516	1,348	5,989	182,400
FFs	525	1,235	8,003	182,400
Mem. bits	131,296	44,032	250,368	14,625,792
Pwr (mW)	46.83	105.97	672.74	-

Table 2: Resource use on DE4 FPGA

Table 3 shows the graph size for the Mibench applications used for evaluating the system. The graph size determines the time it takes to copy a new graph from the centralized graph memory to the monitoring system. It can be seen from the table that the average number of memory entries is 120. In the table, the number of memory entries for an application is higher than the application instructions because we use DFA representation of graphs and hence control instructions have multiple entries in the graph based on their target addresses.

Application	# Application Instr.	# Mem. entry
qsort	96	111
bitcount	60	74
basicmath	107	132
stringmatch	77	97
dijkstra	166	188

Table 3: Monitoring graph size

Table 4 reports the cycle count and time delay for each operation in the embedded processor and the hardware monitor. Both the embedded processor and the hardware monitor operate at a clock frequency of 100 MHz. It is clear from the table that the hardware monitor requires less time than the embedded processor to perform the same operation and hence doesn't slowdown the embedded system.

Operation	# Cycles		Time (us)	
	Nios II	Monitor	Nios II	Monitor
Task Create	600	~140	6	1.4
Context Switch	34	18	0.34	0.18
Task Delete	126	8	1.26	0.08
System Recovery	311	8	3.11	0.08

Table 4: Cycle count and time delay for various operations

6.6. Dual-Core Monitoring System

The functionality of our dual-core monitoring system was verified by using two NIOS II processors. Both the processors use the DDR2 available on the DE4 board to store application binaries and OS kernel. The DDR2 address space was partitioned in two to enable the storage of both processor binaries without data corruption as shown in Figure 34. As earlier, the monitoring logic and centralized graph memory were implemented in on-chip resources.

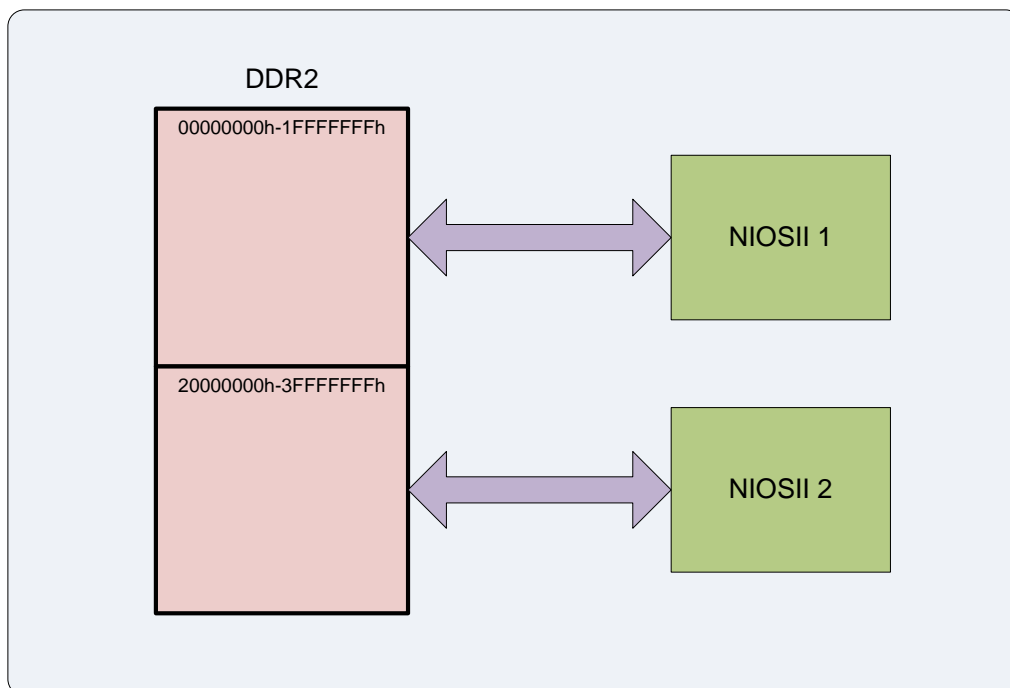


Figure 34: DDR2 address space partition to support dual Nios II core

To verify our dual-core monitoring system, a collection of Mibench applications were assigned to each processor. Both the monitoring systems were able to successfully load the required graph information from the centralized graph memory. Altera SignalTap waveforms illustrated in Figure 35 shows graphs being transferred from the centralized graph memory to both the monitoring systems upon request.

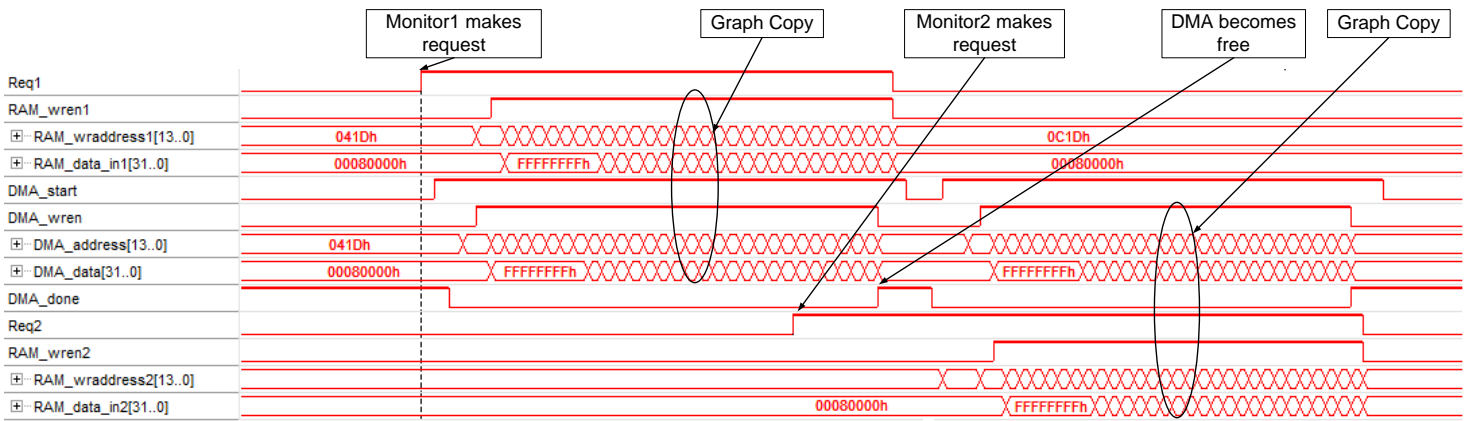


Figure 35: SignalTap waveform showing graph transfer in dual-core monitoring system

In the waveform, Monitor1 requests to access the centralized graph memory by setting *Req1* high. Since *DMA_done* signal is low, a connection is established between Monitor1 and the centralized graph memory. *DMA_start* signal is set high to indicate the start of the DMA transfer operation. We can see in the waveform that graph data and the graph address are being loaded from the DMA controller to the RAM data and RAM address of Monitor1. During this time, it is seen that Monitor2 makes a request to access the centralized graph memory by setting *Req2* high, but since *DMA_done* signal is low, it is made to wait till the *DMA_done* signal goes high to indicate that the centralized graph memory is free to use. Once *DMA_done* signal is high again, a connection is established between Monitor2 and the centralized graph memory and similar operation is followed to transfer the required graph from the centralized graph memory to Monitor2.

After all the required graphs are loaded into the individual monitoring systems, operations such as process creation and context management occur in the same manner as in the case of a single core monitoring system. We have also verified that both the monitoring systems can track the progress of instruction flow on the respective cores without any performance slowdown. If any attack occurs on either core, its respective monitoring system

is able to detect the attack and the attack process is killed without disturbing the other core and the other tasks running on the same core.

Table 5 shows the hardware results reported by the Altera Quartus II tool for the dual-core embedded system with monitoring hardware. Toggle rate of 12.5 was used for all signals during power analysis using Altera PowerPlay.

Resources	Dual-Hardware Monitor	Dual-Nios II Processor	DDR2 Controller	Available
LUTs	1,071	2,477	5,989	182,400
FFs	1,129	2,354	8,003	182,400
Mem. bits	262,592	88,064	250,368	14,625,792
Pwr (mW)	74.54	187.78	672.74	-

Table 5: Resource use for dual-core embedded system

CHAPTER 7

CONCLUSION

The system that has been designed and prototyped achieves the security requirements that are put forth in Chapter 3. The key observation is that the hardware monitor can detect when a specific task executes code that is different from the binary. In such a case, the hash value that is reported from the processor core to the monitor does not match. There is a chance that the attacker is lucky and the hash matches by coincidence or the attacker is clever and aims to construct code that matches. This action however is very difficult to achieve in practice and can be defeated by hiding the hash function [15]. If the monitor detects deviation from the binary, then the processor is signaled to stop execution of the attacked task. Thus, SC1 (no execution of attack code) is achieved.

The system supports multiple tasks that are switched dynamically by the operating system. The hardware monitor follows along in sync and associates the current task on the processor core with the correct monitoring graph. Thus, we achieve SC2 (secure processing for multiple tasks).

Finally, when an attack occurs, the hardware monitor informs the operating system about the attack and the targeted tasks are stopped using a conventional task termination mechanism (similar to the kill command). This mechanism is specially designed to not affect other tasks. Thus, SC3 (isolation of attacked task) is achieved.

We rely on the limitations of attacker capabilities, such as AC4 and AC5 (no tampering of operating system or hardware monitor), to ensure that an attacker cannot circumvent the security mechanism we have put in place.

To conclude, in this thesis, an important security extension for embedded processors that execute multiple processes under the control of an operating system is presented. This

monitoring approach allows the operation of each process to be tracked at the instruction execution level. If a deviation from the expected instruction execution sequence is detected, the monitor can quickly identify and notify the processor to initiate process termination. A significant contribution of the work is the inclusion of multi-context support in the monitoring system. Monitoring state for each process can be quickly saved during a process context switch and previously stored state can be reloaded. Using prototyping, it was seen that the system is effective for multiple processes managed by an embedded OS. A stack smashing attack is identified and suppressed. The monitoring system is modest in size and does not impact the application execution time.

In the future, we plan to extend our monitoring approach for a multi-core embedded processor. We also plan to look into the possibility of monitoring the operating system along with application monitoring. A more powerful operating system like μ Clinux could be used for future works.

BIBLIOGRAPHY

- [1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementation and applications. In *ACM Conference on Computer and Communication Security (CCS)*, pages 340-353, Alexandria, VA, Nov. 2005.
- [2] Altera DDR and DDR2 SDRAM Controller Compiler User Guide http://www.altera.com/literature/ug/ug_ddr_sdram.pdf.
- [3] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted runtime monitoring," in *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, pages 178-183, Munich, Germany, Mar. 2005.
- [4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(4):563-579, Aug. 2005.
- [5] G. Bournoutian and A. Orailoglu. Dynamic transient fault detection and recovery for embedded processor datapaths. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 43-52, 2012
- [6] CERT Coordination Centre, Carnegie Mellon University, Pittsburgh, PA. CERT Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow in IIS Indexing Service DLL, July 2001.
- [7] D. Chasaki and T. Wolf. Attacks and defences in the data plane of networks. *IEE Transactions on Dependable and Secure Computing*, 9(6):798-810, Nov. 2012.
- [8] D. Chasaki, W. Qiang and T. Wolf, "Attacks on Network Infrastructure," in *Proc. of 20th International Conference on Computer Communications and Networks (ICCCN)*, vol., no., pp.1-8, July 31 2011-Aug. 4 2011.
- [9] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T.C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general purpose monitoring of deployed code. In *Proc. of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 63-65, San Jose, CA, Oct. 2006.
- [10] S. Chen, M. Kozuch, P. B. Gibbons, M. Ryan, T. Strigkos, O. Ruwase, E. Vlachos, B. Falsafi, and V. Ramachandran. Flexible hardware acceleration for instruction-grain lifeguards. *IEEE Micro*, 29(1):62-72, Jan. 2009.
- [11] Decoding smartphone industry jargon. Business Insider, Nov. 2013. <http://www.businessinsider.com/decoding-smartphone-industry-jargon-2013-11>.

- [12] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguët, L. Bossuet, and R. Vaslin. Reconfigurable hardware for high-security/high performance embedded systems: the SAFES perspective. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2):144-155, Feb. 2008.
- [13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001*.
- [14] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In Proc. of the Third ACM SIGPLAN International Conference on Functional Programming Languages, pages 86-93. ACM, 1998.
- [15] K. Hu, T. Teixeira, T. Wolf, and R. Tessier, System-level security for network processors with hardware monitors. In *Proc. of 51st Design Automation Conference (DAC)*, San Francisco, CA, June 2014.
- [16] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proc. of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '99)*, volume 1666 of Lecture Notes in Computer Science, pages 388-397, London, United Kingdom, 1999. Springer-Verlag.
- [17] H. Kumarapillai Chandrikakutty, D. Unnikrishnan, R. Tessier, and T. Wolf. High-performance hardware monitors to protect network processors from data plane attacks. In *Proc. of 50th Design Automation Conference (DAC)*, Austin, TX, June 2013.
- [18] J. J. Labrosse. *MicroC/OS-II - The Real-Time Kernel*. Second Edition. CMP Books.
- [19] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security Privacy*, 9(3):49-51, May 2011.
- [20] Y. Luo and J. Fan. Fault tolerant practices on network processors for dependable network processing. In *Proc. of IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, Miami, FL, April. 2008.
- [21] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," *IEEE Transactions on Computers*, 59(6):847–854, Jun. 2010.
- [22] D. Moore, C. Shannon, and J. Brown, "Code-Red: a case study on the spread and victims of an Internet worm," in *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement*, pages 273-284, Marseille, France, Nov. 2002.

- [23] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu. An architectural framework for providing reliability and security support. In *Proc. of the 2004 International Conference on Dependable Systems and Networks (DSN)*, pages 585-594, Florence, Italy, June 2004.
- [24] National Institute of Standards and Technology. *National Vulnerability Database*. <http://nvd.nist.gov>.
- [25] S. Parameswaran and T. Wolf. Embedded system security – an overview. *Design Automation for Embedded Systems*, 12(3):173-183, Sept. 2008.
- [26] P. Parkinson and A. Baker. Mils architecture simplifies design of high assurance systems. In *EE Design*, Aug. 2011.
- [27] M. Pflanz. *On-line error detection and fast recover techniques for dependable embedded processors*. Springer-Verlag, 1st edition, 2002.
- [28] L. Pike, P. Hickey, T. Elliott, A. Tomb, E. Mertens, D. Kapp, and D. Koranek. TrackOS: a Security- Aware RTOS. Experimental, Galois Corporations, 2012.
- [29] R. G. Ragel and S. Parameswaran. IMPRES: integrated monitoring for processor reliability and security. In *Proc. of the 43rd Annual Conference on Design Automation (DAC)*, pages 502-505, San Francisco, CA, USA, July. 2006.
- [30] R. G. Ragel, S. Parameswaran, S. M. Kia. Micro embedded monitoring for security in application specific instruction-set processors. In *Proc. of the 2005 international conference on Compilers, architectures and synthesis for embedded systems (CASES)*, pages 304-314, San Francisco, CA, Sept. 2005.
- [31] S. Ravi, A. Raghunathan, and S. Chakradhar. Tamper resistance mechanisms for secure, embedded systems. In *Proc. of 17th International Conference on VLSI Design (VLSI Design 2004)*, pages 605-611, Mumbai, India, Jan. 2004.
- [32] M. R. Rieback, P. N. Simpson, B. Crispo, and A. S. Tanenbaum. RFID malware: Design principles and examples. *Pervasive and Mobile Computing*, 2(4):405-426, 2006.
- [33] K. Rosenfeld and R. Karri. Attacks and defences for jtag. *IEEE Design and Test of Computers*, 27(1):36-47, Jan. 2010.
- [34] Z. Shao, Q. Zhuge, Y. He, and E. H. –M. Sha. Defending embedded systems against buffer overflow via hardware/software. In *Proc. of the 19th Annual Computer Security Applications Conference (ACSAC)*, pages 352-363, Las Vegas, NV, Dec. 2003.

- [35] S. Shivshankar, S. Vangara, and A. Dean. Balancing register pressure and context-switching delays in asti systems. In *Proc. of International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 286-294, 2005.
- [36] A. B. Sikiligiri. Buffer Overflow Attack and Prevention for Embedded Systems, PhD thesis, University of Cincinnati, 2011.
- [37] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85-96, Oct. 2004.
- [38] T. Thomas, A. Pouraghily, K. Hu, R. Tessier, and T. Wolf. Multi-Task support for Security-Enabled Embedded Processors. In *Proc. of 26th IEEE Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Toronto, Canada, July. 2015.
- [39] A. Wood and J. A. Stankovic. Denial of service in sensor networks. *IEEE Computer*, 35(10):54-62, Oct. 2012.
- [40] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, and N. Memon. SAFE-OPS: An approach to embedded software security. *Transactions on Embedded Computing Sys.*, 4(1)189-210, Feb. 2005.
- [41] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure coprocessor-based intrusion detection. In *Proc. of European Sigops*, pages 239-242, 2002.
- [42] X. Zhou and P. Petrov. Rapid and low-cost context switch through embedded processor customization for real-time and control applications. In *Proc. of IEEE/ACM Design Automation Conference*, pages 352-357, June 2006.