Nile: A Programmable Monitoring Coprocessor

Leila Delshadtehrani, Schuyler Eldridge, Sadullah Canakci, Manuel Egele, and Ajay Joshi
Department of Electrical and Computer Engineering, Boston University
{delshad, schuye, scanakci, megele, joshi}@bu.edu

Abstract—Researchers widely employ hardware performance counters (HPCs) as well as debugging and profiling tools in processors for monitoring different events such as cache hits, cache misses, and branch prediction statistics during the execution of programs. The collected information can be used for power, performance, and thermal management of the system as well as detecting anomalies or malicious behavior in the software. However, monitoring new or complex events using HPCs and existing tools is a challenging task because HPCs only provide a fixed pool of raw events to monitor. To address this challenge, we propose the implementation of a programmable hardware monitor in a complete system framework including the hardware monitor architecture and its interface with an in-order single-issue RISC-V processor as well as an operating system. As a proof of concept, we demonstrate how to programmatically implement a shadow stack using our hardware monitor and how the programmed shadow stack detects stack buffer overflow attacks. Our hardware monitor design incurs a 26% power overhead and a 15% area overhead over an unmodified RISC-V processor. Our programmed shadow stack has less than 3% performance overhead in the worst case.

Index Terms—Hardware Coprocessor, Programmable Hardware, Shadow Stack, Stack Buffer Overflow Attack

1 Introduction

Microprocessor hardware usage information, gleaned during software execution, provides valuable information that can be utilized for optimizing and analyzing processor behavior, for example, this information could be used for power/thermal management or malicious program detection. Software tools that capture hardware usage information, like Intel Processor Trace [11] using dedicated hardware or VTune [14] and PERF [6] that rely on Hardware Performance Counters (HPCs), do exist. Dedicated hardware or HPCs enable the monitoring of a variety of events, but, due to their fixed nature, limit the tracking of new, complex events selected to elide specific information necessary for new optimizations or analyses.

This limited view of run-time hardware information is starkly contrasted with the old system simulator SimOS [16]. SimOS allowed system designers to expose arbitrary low-level hardware *events* and compose these into high-level software *actions*. The utility of such an *event–action* model for the composition of hardware events that drive run-time optimization and analysis is evident. However, SimOS is a simulation environment. Flexible, software-based techniques, like DynamoRIO [3], enable similar introspection but at extreme cost. Nonetheless, such introspection must, for certain applications like security analysis, be transparent and out-of-band to the microprocessor.

Towards a model like SimOS, that enables flexible and programmable event monitoring and action taking at low performance and power overheads, we propose the implementation of a new hardware monitoring coprocessor, *Nile*.[1] Nile provides a collection of programmable event/action units that can be composed to track complex semantic events. We demonstrate an implementation of Nile as a coprocessor of a RISC-V microprocessor. We minimally modify the RISC-V core to expose a design-time specified *commit log* that exposes information about instruction execution. Internally, Nile contains a number of configurable *Match Units* (MUs) that process the commit log. We provide a software library consisting of a set of functions for configuring and programming Nile and, additionally, modify the Linux Operating System (OS) to support Nile. A user can then program each of the MUs to monitor a separate event, count the number of event occurrences, and take a corresponding action. Furthermore, MUs can communicate with each other through a shared memory structure to allow for more complex event definition. Overall, we make the following contributions:

- **Design**: We propose a novel, flexible and programmable hardware mechanism as well as the software and OS support for monitoring the execution of programs in real time.
- **Application**: We demonstrate the flexibility and programmability of our mechanism to track complex events via a security case study—detecting stack buffer overflows with a shadow stack.
- **Implementation**: We implement a prototype of our hardware monitor as a coprocessor of RISC-V processor and evaluate it running Linux on an FPGA. Our programmed shadow stack incurs overheads of less than 3% in performance, 26% in power, and 15% in area.

2 Programmable Hardware Monitors

In this section, we describe the microarchitectural design of Nile, the software library for accessing it and the required changes in the Linux OS for communicating with Nile.

2.1 Nile Microarchitecture

We implement Nile as a coprocessor that interfaces with the RISC-V Rocket processor [1]. Figure 1 illustrates the communication between the Rocket processor and Nile through the Rocket Custom Coprocessor (RoCC) interface. We have extended this interface to carry instruction execution information in the form of a *commit log*. We collect the commit

1. The *Nile monitor* is a member of the monitor lizard genus.

Manuscript received 03 Nov. 2017; accepted 06 Dec. 2017.
Date of publication 0 . 0000; date of current version 0 . 0000.

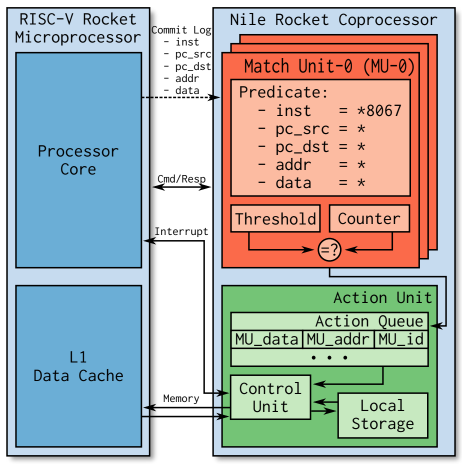

Fig. 1. Nile as a RISC-V Rocket Custom Coprocessor (RoCC). Nile consists of several Match Units (MUs) and an Action Unit. Each MU monitors for an *event* while the Action Unit manages *actions* via interrupts or memory reads/writes.

log from the write-back stage of the processor, which allows us to decouple the program execution from our monitoring process. Note that for an out-of-order processor, we can collect the commit log information in parallel with the commit stage to reduce the performance overhead. Additionally, we expect Nile to incur less area and power overheads when integrated with an out-of-order core.

The commit log consists of five separate entries including the undecoded instruction (`inst`), the current Program Counter (PC) (`pc_src`), the next PC (`pc_dst`), the memory/register address used in the current instruction (`addr`), and the data accessed by the current instruction (`data`). The `inst` entry is the length of an instruction (32-bit normally, 16-bit for compressed instructions) while all the other entries of the commit log are the processor's word length (usually 64-bit). Nile receives the incoming commit log trace from the processor and broadcasts this to all MUs.

Each of the MUs is programmed to monitor a distinct event indicated by a wildcard match on the commit log. Whenever the matching condition evaluates to true, a counter in the corresponding MU increases. Once the counter reaches a programmed threshold, it triggers an activation signal. The MU sends an activation packet to the Nile Action Unit (discussed below). This packet consists of an address (`MU_addr`), some data (`MU_data`), and an MU identification number (`MU_id`). The `MU_data` is programmable and can contain any of the contents of the commit log while `MU_addr` contains the current value of PC (`pc_src`).

An MU may be programmed by a user process (to monitor only its own execution) or by a supervisor/hypervisor to monitor processes with lower permissions. MU configuration then becomes part of a process' context and is preserved across context switches by the OS. There is a trade-off between the number of MUs and the performance, power, and area overheads of Nile. However, the user can monitor more events than the available number of MUs through time-multiplexing the MUs (similar to HPCs).

The Nile Action Unit consists of control, storage, and an activation queue (see Figure 1). The activation queue stores the incoming activation packets from MUs. As long as the activation queue is not empty, the control unit dequeues a packet and takes a corresponding action in two different ways:

1) Nile triggers an interrupt over the RoCC interface and a supervisor handles it.
2) Nile performs a configurable matching operation, which can trigger an interrupt, between the `MU_data` of the dequeued packet and a data stored in a shared memory space. More than one MU can take an action by reading/writing over the shared memory space, where the address is specified by `MU_addr` of the packet or an address stored in the Action Unit local storage.

For the purposes of concrete examples, we briefly describe the usage of these actions to monitor different high-level events:

- **Sensitive Data Protection**: Using the first action type, any access to a specific memory region results in an interrupt.
- **Stack Buffer Overflow Protection**: Using the second action type, a shadow stack is built in memory from one MU writing data on function calls and another MU checking data on function returns.

Note: these examples are intended to be instructional and not definitively encompassing of the capabilities of the Nile event–action model.

The second example highlights the intended capability of communication between two or more MUs through a shared memory space. In this case, the OS allocates a shared memory space (as a user-space memory, which is protected by one of the MUs as a sensitive data region) and provides base/bounds information of this region to Nile. The usage of this shared memory region is up to the designer, e.g., multiple MUs may all write to the shared memory. Additionally, the designer can configure the control unit to perform a matching operation based on a `difference` value, e.g., the `difference` between a read request data and a value written in the shared memory space. In Section 3, we elaborate on the shadow stack example and describe the programming of Nile to achieve this result.

2.2 Nile Software Interface

We provide a set of functions for configuring the Nile MUs and communicating with them. These functions utilize "custom" instructions in the RISC-V ISA, which are intended to interface with accelerators/coprocessors. Table 1 lists these functions and specifies whether each of them is accessible through a user-space program, OS, or both. A user configures the matching pattern of a specific MU using the `MU_id` and a matching input for a specific process or all processes. The matching input defines the matching conditions for each of the commit log entries. A matching condition consists of matching and wildcard masking bits. As an example, according to the RISC-V ISA, a `ret` instruction is a pseudo-instruction defined by `JALR` when `rd=x0`

TABLE 1
Nile Application Programming Interface (API)

Function	Description	Accessibility
set_pattern(MU_id, *mask, pid)	Set the MU's matching pattern	User
comm(MU_id1, MU_id2, *comm)	Set the communication type of two MUs	User
reset(MU_id)	Reset the match counts for a specific MU	User/OS
enable(MU_id)	Enable monitoring for an MU	User/OS
disable(MU_id)	Disable monitoring for an MU	User/OS
set_thresh(MU_id, count)	Set the match threshold to trigger an action	User/OS
count = rd_count(MU_id)	Read the value of an MU	User/OS
wr_count(MU_id, count)	Write the value of an MU	OS
wr_sm_base(*addr)	Write the base address of the shared memory	OS
wr_sm_offset*addr)	Write the offset of the shared memory	OS
wr_sm_size(size)	Write the size of the shared memory	OS
base_sm = rd_sm_base	Read the base address of the shared memory	OS
offset_sm = rd_sm_offset	Read the offset of the shared memory	OS
size_sm = rd_sm_size	Read the size of the shared memory	OS

and `rs1=x1`. Subsequently, we can monitor a `ret` ("`jalr x0,x1,0`") using the following matching condition:

ret : inst = 0x00008067; mask = 0x00000000

Accordingly, the matching condition for `inst` evaluates to true when the current instruction is an exact match with the value of a `ret` instruction. For all the other entries of the commit log (`pc_src`, `pc_dst`, `addr`, and `data`), we set the masking value to `0xffffffffffffffff`, indicating all these fields are "don't cares".

The OS utilizes the last seven functions listed in Table 1 for managing the shared memory allocated for communication between MUs. In the next section, we will emphasize the importance of these functions and describe how we employ them to program our sample use case.

2.3 Operating System Support for Nile

We extend Linux to support Nile and provide a full system configuration. This configuration offers the flexibility of monitoring different processes through programming Nile MUs. However, this integration demands equipping the OS with specific support for communicating with the hardware, which we provide at the process level. To this end, we alter the `task_struct` in the Linux Kernel to save/restore the Nile configuration of each process.

We modify the Linux kernel to initialize the Nile information before the process starts its execution. Once the user configures Nile for monitoring a process, we set a flag for that process. As mentioned earlier, the OS allocates a shared memory space for communication between MUs. After allocation, the OS stores the base address, the offset, and the size of the shared memory as part of the Nile information for the process stored in the `task_struct`. During a context switch, the OS reads the MU information (counter and threshold values) from Nile and stores them as the Nile information of the previous process. Before the OS context switches to a monitored process, it reads the MU information of the next process and writes it to Nile registers using the functions provided in the Nile software library.

The OS is responsible for handling an incoming interrupt triggered by one of the MUs. We configure our RISC-V processor to delegate the interrupt to the OS. In our current implementation, the OS terminates the process that caused the interrupt based on the assumption that an anomaly or violation has triggered the interrupt.

3 Example Application: Shadow Stack

Due to the programmable design of Nile, a user can employ it in diverse applications such as attack detection, anomaly detection, and online profiling. In this section, we discuss our implementation of one possible use case for Nile: a shadow stack for detecting stack buffer overflow attacks.

A shadow stack is a secondary stack, dedicated for storing the return addresses to protect them from being tampered with by an attacker. A stack buffer overflow attack occurs when a program writes data to a memory address on the program's call stack, such that the data is larger than an allocated buffer on the stack. A number of schemes, including shadow stack, have been developed for preventing stack buffer overflow and return-oriented programming attacks.

We use Nile for implementing a hardware shadow stack for protecting against stack buffer overflow attacks. To this end, we program one of the MUs for monitoring `call` and another one for monitoring `ret` instructions. We specify the communication method between these two MUs through the shared memory: the first MU writes the `pc_src` of the `call` while the second MU reads the written value in the shared memory and compares it with the `pc_dst` of the `ret`. The shared memory address is determined by the base/offset information programmed to act as a shadow stack pointer. We configure the `difference` for the matching operation to be equal to 4 (note that `call` and `ret` are 32-bit instructions in RISC-V ISA). If the difference between the `pc_src` and `pc_dst` values is not equal to 4, the control unit triggers a RoCC interrupt. The OS handles the interrupt by terminating the process that caused the interrupt.

Recently, Intel has implemented a hardware shadow stack as part of the control-flow enforcement technology. As opposed to Intel's rigid shadow stack implementation, we can *program* Nile to achieve the same functionality with moderate performance overhead. Furthermore, Nile's flexibility and the genericity of the event–action model allows the user to perform a variety of *new* monitoring tasks without new dedicated hardware, e.g., Nile can act as a code coverage engine by monitoring all taken branches.

4 Evaluation

In this section, we discuss our approach to validate the functionality of Nile as well as our evaluation of Nile using performance, power, and area metrics.

4.1 Experimental setup

We implement Nile as a RoCC (using Chisel HDL [2]) and connect it to the RISC-V Rocket processor [1]. We use the Xilinx Zynq Zedboard evaluation board [15] for prototyping. We use a modified RISC-V Linux port of the Linux kernel 4.6.2 in all of our experiments.

We compare the Nile design with a baseline implementation of the Rocket processor and program Nile to act as a shadow stack using two of the available MUs in our design. For measuring the performance, we calculate the run time of 12 applications from MiBench [10] benchmark suites running in the Linux OS. We use Cadence ASIC toolflow for 45nm NanGate process [13] to design Nile to operate at 1 GHz. We then measure the average power consumption and the area of our system (post place and route).

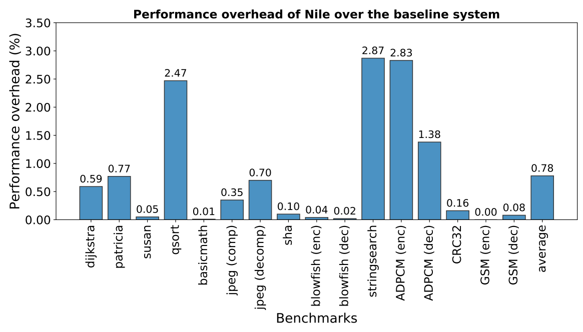

Fig. 2. The performance overhead of Nile for MiBench applications.

4.2 Results

We verify the correct functionality of our programmed shadow stack using benchmarks and buffer overflow vulnerable programs. All of our benchmark programs run successfully with shadow stack, thus Nile caused no false positives. We develop simple buffer overflow vulnerable programs using `strcpy` and `memcpy` functions. We exploit this vulnerability by choosing specific inputs for our developed programs with the intention of gaining the control for performing arbitrary computations. However, Nile is able to detect the mismatches between `calls` and `rets`, triggering an interrupt, and terminating the process.

Figure 2 shows the performance overhead of Nile over our baseline RISC-V Rocket processor. Adding Nile incurs 0.78% performance overhead on average for our evaluated MiBench benchmark applications while our worst case performance overhead (for Stringsearch) is less than 3%. The main sources of performance overhead for Nile are increasing the number of cache requests and storing/recovering the Nile information to/from OS. Cache latency can be reduced by writing to higher levels in the memory hierarchy. The power consumption of Nile is 0.009 mW/MHz while its area is 0.06 mm^2. In comparison to our baseline RISC-V processor [12], Nile incurs about 26% power and 15% area overheads.[2]

5 Related work

A wide range of dedicated hardware monitors have been proposed for different use cases such as performance evaluation of real-time systems [17], measuring cache eviction information [4], and secure program execution [5], [9], [7], [8]. Among monitoring mechanisms for security, HAFIX [5] provides hardware support against ROP and buffer overflow attacks. Our programmed shadow stack is capable of detecting ROP attacks, which we did not discuss due to the limited space. PUMP [9] extends an in-order processor with programmable software policies for tag-based monitoring; however, it brings about invasive and drastic changes to the processor pipeline. Nile provides flexible monitoring by only applying minimal non-invasive changes to the processor. FlexCore [7] architecture is a re-configurable fabric decoupled from the processor, which provides a range of monitoring and bookkeeping techniques for detecting security and reliability errors. Similar to Nile, Harmoni [8] employs a co-processor designed for monitoring and analyzing the instruction trace from the processor. Harmoni can be utilized for applying different run-time tagging-based monitoring techniques. Note that unlike these monitoring mechanisms, Nile is not restricted to providing security, and has other usage including application fuzzing and power management.

2. Our baseline Rocket processor has 0.034 mW/MHz power consumption and its area is 0.39 mm^2 using TSMC 40GPLUS technology [12]. Not having access to this process, we measured the power and area of Nile using the 45nm NanGate technology process.

6 Conclusion

In this work, we presented the design and implementation of Nile, a programmable hardware monitor coprocessor capable of tracking complex semantic events. Nile has broad usage including security, application fuzzing, and power management. For security use case, when Nile is programmed to behave as a shadow stack, it incurs less than 3% performance overhead. For the future work, we plan to provide more possible actions upon detecting a match, e.g. executing a function specified by the user. We will also develop programmable features for monitoring memory and cache accesses.

7 Acknowledgments

We thank Prof. Jonathan Appavoo for providing invaluable help in designing the OS support and the software interface for Nile. This work was supported in part by NSF grant CCF-1533663.

References

[1] K. Asanovic et al. The rocket chip generator. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.
[2] J. Bachrach et al. Chisel: constructing hardware in a scala embedded language. *Proc. DAC*, pp. 1216–1225, 2012.
[3] D. Bruening et al. An infrastructure for adaptive dynamic optimization. *Intl. Symp. CGO*, pp. 265–275, 2003.
[4] B . R. Buck and J . K. Hollingsworth. A new hardware monitor design to measure data structure-specific cache eviction information. *J. High Perf. Comp. App.*, 20(3):353–363, 2006.
[5] L. Davi et al. Hafix: Hardware-assisted flow integrity extension. *Proc. DAC*, p. 74, 2015.
[6] A . C. de Melo. The new linux 'perf' tools. *Slides from Linux Kongress*, volume 18, 2010.
[7] D . Y. Deng et al. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. *Intl. Symp. Microarchitecture*, pp. 137–148, 2010.
[8] D . Y. Deng and G . E. Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. *Proc. DSN*, pp. 1–12, 2012.
[9] U. Dhawan et al. Architectural support for software-defined metadata processing. *Proc. ASPLOS*, pp. 487–502, 2015.
[10] M . R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. *Proc. WWC*, pp. 3–14, 2001.
[11] Intel®. Intel® 64 and ia-32 architectures software developers manual. *Sys. Prog. Guide, Part 3C*, Sept. 2016.
[12] Y. Lee. Risc-v "rocket chip" soc generator in chisel. Online slides: "https://riscv.org/wp-content/uploads/2015/01/riscv-rocket-chip-generator-workshop-jan2015.pdf", 2015.
[13] Nangate. The nangate 45nm open cell library. http://www.nangate.com.
[14] J. Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005.
[15] Digilents ZedBoard Zynq FPGA. Dev. board documentation.
[16] M. Rosenblum et al. Using the simos machine simulator to study complex computer systems. *ACM TOMACS*, 7(1):78–103, 1997.
[17] J . J . P. Tsai et al. A noninvasive architecture to monitor real-time distributed systems. *Computer*, 23(3):11–23, 1990.