Varanus: An Infrastructure for Programmable Hardware Monitoring Units

Leila Delshadtehrani, Jonathan Appavoo, Manual Egele, and Ajay Joshi
Boston University
{delshad, jappavoo, megele, joshi}@bu.edu

Schuyler Eldridge
IBM Research
schuyler.eldridge@ibm.com

***Abstract*—The information collected from hardware performance counters in a typical processor is commonly employed for power management, thermal management, and malware detection. However, these counters cannot be programmed at the hardware level for monitoring complex events. To address this shortcoming, we propose the implementation of programmable hardware monitors and explore the scope of their programmability by discussing some of their practical applications.**

I. Introduction

Hardware performance counters are special counters available in a processor for profiling the hardware usage of a program. Each processor provides a fixed number of hardware performance counters. Each of these individual hardware counters can be configured for monitoring different events from a pre-defined pool of "raw" events. Several software tools such as VTune [8], Processor Trace (PT) [4], and PERF [2] are available for accessing hardware performance counters. Researchers utilize the extracted profiles from these counters for power management [7], thermal management [6] and malware detection [3]. However, it should be noted that the available events might not always satisfy the user's requirement and at the same time the user cannot build a new or complex event based on his/her specific requirements. For example, counting the number of call and ret instructions or matching their corresponding addresses is not possible using hardware performance counters.

When a hardware performance counter reaches its upper limit, it will either *overflow* or generate a *performance monitor interrupt*. However, hardware performance counters do not allow a user to trigger an action when a specific counter reaches a pre-defined threshold without considerably changing the execution of the underlying application or incurring considerable performance overhead.

As an alternative, for monitoring and profiling a system at hardware-level system designers exploit full system simulation environments. For example, SimOS [9] provides an environment for studying the full system through monitoring low-level hardware events and triggering high-level software actions. Consequently, system designers have access to flexible hardware monitors that can be mapped to higher-level software concepts at simulation-level. This flexible access empowers system designers to profile and evaluate the software interaction with the underlying hardware. Despite all the advantages of full system simulators, we need to run applications on real hardware. For an actual system, the available event monitoring tools at hardware-level do not provide the flexible monitoring facilities similar to the existing ones in full system simulators.

To provide flexibility in building events and taking actions with low performance and power overheads for a real system, we propose the implementation of programmable hardware monitors. The main goal of these programmable monitors is to enable operating system and overlaying software to track complex events (which can be programmed) at a fine granularity and then take appropriate action. The most notable aspect of our programmable hardware monitors is that a user can build and monitor increasingly more complex events by monitoring simple events. As an example, the user can monitor function calls and taken branches by tracking *call*/*ret* instructions and indirect/direct *jump* instructions, respectively. Then, the user can monitor a complex event such as the control flow of the program by keeping track of all the monitored function calls and taken branches.

Our programmable hardware monitors are developed on an infrastructure named Varanus[1]. In this paper, we develop and implement one programmable hardware monitor, called Komodo[2], using the Varanus. The user can program Komodo for monitoring different events and employ it in diverse applications.

II. Programmable Hardware Monitors

In this section, we describe the architectural implementation of Varanus as well as our programmable hardware monitor, present the software interface for accessing Komodo and discuss a potential application for employing Komodo.

A. Hardware implementation

We have implemented the Varanus infrastructure as an accelerator that works with the Rocket microprocessor [1]. Rocket is an in-order core based on the RISC-V Instruction Set Architecture (ISA) [11]. As depicted in Figure 1, Varanus communicates with the Rocket microprocessor through the Rocket Custom Coprocessor (RoCC) interface. RoCC is an interface that facilitates decoupled communication (based on handshaking protocol) between a Rocket processor and the attached accelerators/coprocessors [1]. We have modified the RoCC interface by adding a bundle called "commit log"[3] for communication with Varanus. Figure 2 depicts the information carried in the form of commit log. This information is collected from the write back stage of the pipeline in Rocket and routed to all of the Komodo monitoring units.

Komodo is capable of monitoring and taking actions based on events defined by the user. To be specific, Komodo monitors a programmable predicate specified by the user. The predicate is a Boolean logic evaluated based on a set of programmable matching conditions. Whenever the predicate evaluates to true,

[1] Varanus is the common genus among the family of "monitor" lizards.

[2] The Komodo monitor belongs to the monitor lizard family.

[3] Commit log is a log of committed instructions and their writeback values provided in the Rocket microprocessor. We modify and exploit this log according to our monitoring requirements.

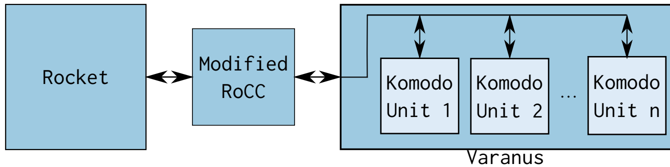

Fig. 1. Varanus infrastructure and Komodo monitoring units.

pc_src	pc_dst	inst	addr	data

Fig. 2. Commit log entries. *pc_src* specifies the current PC value while *pc_dst* determines the next PC value. The current instruction, i.e., the *Instruction Register*, is stored in *inst*. *addr* is the memory address/destination register accessed in the current instruction and *data* is the corresponding data value stored in that address/register. *inst* is a 32-bit value while all the other entries of the commit log are 64-bit values.

Komodo takes action by activating a "doorbell"[4] and executing an action function defined by the user. In Section II-B, we will explain about configuring the predicate in more detail.

In addition to this, the user can define a threshold for the number of observed matches (or partial matches) before taking an action. As a simple example, the user can monitor a specific function call. In this case, the user should program Komodo by specifying a full match for the *pc_dst* entry and a partial match for the *inst* entry of the commit log (we only need to match the *opcode* part of the *inst*).

As depicted in Figure 1, our hardware implementation supports more than one Komodo monitoring unit. Each Komodo monitoring unit acts as a separate comparator unit and has a unique identification number called "Komodo Unit Number". The maximum number of Komodo monitoring units is a design knob.

B. Software interface for accessing Komodo

We provide a set of functions for configuring the Komodo monitoring units and communicating with them. For example, *komodo_enable* and *komodo_disable* functions will enable and disable, respectively, a specific Komodo unit. Similarly, the user configures the matching pattern of a specific Komodo unit using the Komodo Unit Number and a matching input. The matching input defines the matching conditions for each of the commit log entries. The matching condition consists of *matching* and *masking* bits. The predicate is a match with the *matching* value (for the bits that are not masked based on the *masking* value) on the contents of a single commit log entry. For example, the matching input for the *ret* instruction on the *inst* entity of the commit log can be set as following:

```
match_input.match_value.inst  = 0x00008067;
match_input.mask_value.inst = 0x00000000;
```

This indicates that the predicate for *inst* evaluates to true when the current instruction is an exact match with the value of *ret* instruction (0x0008067 according to the RISC-V ISA). For all the other entries of the commit log (*pc_src*, *pc_dst*, *addr*, and *data*), the *masking* value should be set to '0xffffffff', because regardless of the values of these entries we will match all the *ret* instructions. Note that similar to full matches, the user can define a partial match using the *matching* and *masking* values.

[4]Doorbell is a dormant process waiting to be activated when its programmed condition becomes true.

C. Potential Usage of Komodo

Detecting Return Oriented Programming (ROP) attacks [10] is a practical application for Komodo. ROP is a code reuse attack based on gaining the control of the stack and directing the control of the program to a pre-existing gadget. A gadget is a short sequence of instructions ending with *ret* instruction. One gadget only performs part of the desired functionality of the attacker; however, chaining several gadgets may allow the attacker to perform Turing-complete computation.

Recently, Intel has released a control flow enforcement technology [5] to protect against ROP attacks. In particular, Intel has implemented a shadow stack in hardware for the x86/64 architecture. A shadow stack is a second stack, which is used exclusively for monitoring the control flow of the program. The software implementation of a shadow stack incurs considerable performance overhead while its hardware implementation mitigates this overhead. We can employ Komodo monitor for implementing a hardware structure similar to that of shadow stack for protecting against ROP attacks. To this end, the user should program Komodo for monitoring *call* and *ret* instructions and match the *pc_dst* of each *ret* instruction with *pc_src* of its corresponding *call*. When a mismatch between these two values is monitored, Komodo takes action by activating a doorbell and interrupting the execution of the program. Note that we can implement a shadow stack by programming Komodo in a specific way instead of adding specific hardware only for this purpose.

III. Conclusion

In this work, we proposed Varanus as a common infrastructure for implementing programmable hardware monitors and discussed the implementation of an example programmable hardware monitor, called Komodo, on this infrastructure. The main advantage of our programmable hardware monitor is that the user can program it to monitor substantially complex events. We discussed one potential usage of the Komodo monitor, where it can be programmed to behave like a shadow stack to detect ROP attacks.

References

[1] K. Asanovi, R. Avizienis *et al.*, "The rocket chip generator," 2016.

[2] A. C. de Melo, "The new linux 'perf' tools," in *Slides from Linux Kongress*, vol. 18, 2010.

[3] J. Demme, M. Maycock *et al.*, "On the feasibility of online malware detection with performance counters," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, 2013, pp. 559–570.

[4] Intel, "Intel® 64 and IA-32 architectures software developer's manual, volume 3C," December 2016.

[5] ——, "Control-flow enforcement technology preview," June 2016, Revision 1.0.

[6] A. Kumar, L. Shang *et al.*, "Hybdtm: a coordinated hardware-software approach for dynamic thermal management," in *Proc. DAC*, 2006, pp. 548–553.

[7] M. Moeng and R. Melhem, "Applying statistical machine learning to multicore voltage & frequency scaling," in *Proc. CF*. ACM, 2010, pp. 277–286.

[8] J. Reinders, *VTune performance analyzer essentials*. Intel Press, 2005.

[9] M. Rosenblum, E. Bugnion *et al.*, "Using the simos machine simulator to study complex computer systems," *ACM TOMACS*, vol. 7, no. 1, pp. 78–103, 1997.

[10] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. CCS*, 2007, pp. 552–561.

[11] A. Waterman, Y. Lee *et al.*, "The risc-v instruction set manual," 2014.