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. In this paper, we develop and implement one programmable hardware monitor, called Komodo 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” 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,
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, \texttt{komodo\_enable} and \texttt{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 \texttt{matching} and \texttt{masking} bits. The predicate is a match with the \texttt{matching} value (for the bits that are not masked based on the \texttt{masking} value) on the contents of a single commit log entry. For example, the matching input for the \texttt{ret} instruction on the \texttt{inst} entity of the commit log can be set as following:

\[
\text{match\_input.match\_value.} \text{inst} = 0x00000000; \\
\text{match\_input.mask\_value.} \text{inst} = 0x00000000;
\]

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

---

\(^4\)Doorbell is a dormant process waiting to be activated when its programmed condition becomes true.