A Hardware/Software Infrastructure for Performance Monitoring on Leon3 Multicore Platforms

authors omitted for blind review

***Abstract*—Monitoring applications at run-time and evaluating the recorded statistical data of the underlying micro architecture is one of the key aspects required by many hardware architects and system designers as well as high-performance software developers. To fulfill this requirement, most modern CPUs for High Performance Computing (HPC) have been equipped with Performance Monitoring Units (PMU) including a set of hardware counters, which can be configured to monitor a rich set of events. Unfortunately, embedded and reconfigurable systems are mostly lacking this feature. Towards rapid exploration of High Performance Embedded Computing in near future, we believe that supporting PMU for these systems is necessary. In this paper, we propose a PMU infrastructure, which supports monitoring of up to seven concurrent events. The PMU infrastructure is implemented on an FPGA and is integrated into a Leon3 platform. We show also the integration of our PMU infrastructure with the `perf_event`, which is the standard PMU architecture of the Linux kernel.**

I. Introduction

For many decades, computer architects have been using simulators to expose and analyze performance metrics by running workloads on the simulated architecture. The results collected from simulation may be inaccurate in some cases due to the workloads running on top of an operating system or the simulators just considering not all the relevant micro-architectural aspects. More recently, so-called full system simulators such as gem5 [1] are being used by many researchers and system architects in order to accurately gather full statistical data at the system level. In case the investigated architecture is already available as an implementation, performance data can also be collected at runtime with high accuracy and often with higher speed than simulation [2]. To that end, many modern high-performance processors feature a performance monitoring unit (PMU) that allows to collect performance data. A PMU is essentially a set of counters and registers inside the processor that can be programmed to capture the events happening during the application execution. At the end of a measurement, performance monitoring software reads out the PMU counter values and aggregates the results.

Performance monitoring is not only useful in high-performance computing but also in embedded and reconfigurable computing. Especially the exploration of reconfigurable computing has led many researchers to propose ideas for system optimization and adaptation at runtime, such as self-tuning caches [3]. Runtime adaptation techniques demand for a hardware/software infrastructure capable of system performance measurements in real-time. While PMUs have recently been added to some well known embedded processors such as ARM [4], Blackfin [5], and SuperH [6], as well as to the ARM cores in the Xilinx Zynq [7], a performance monitoring feature is mostly lacking for soft cores embedded into FPGAs.

The main contribution of this paper is the presentation of a hardware/software infrastructure for performance monitoring for the Leon3 platform, which is a widely-used open source soft core based on the Sparc-v8 architecture [8]. In the remainder of the paper, we first discuss the background of PMUs in Section II and then describe the hardware and software implementation of our PMU for Leon3 multicores in Section III. In Section IV we present experimental results for MiBench workloads and show the overhead incurred by performance monitoring. Finally, Section V concludes the paper.

II. Background – Performance Monitoring Units

Performance analysis based on performance monitoring units (PMUs) requires both, hardware and software infrastructure. The hardware infrastructure for recording statistical data at the micro-architectural level during program execution basically includes sets of control registers and counters. The control registers can be programmed to specific events that should be captured which are then counted. The configuration written to control registers also determines whether and which interrupts are generated on a counter overflow, whether data is collected only for user mode or also for kernel mode execution, and generally to enable or disable data collection. While most modern processors include some form of PMU [9], the number of measurable events and hardware counters varies in different processors [4], [10]. Events commonly available for monitoring include the number of CPU cycles, miss rates for different levels of instruction, data or unified caches, for TLBs, or IPC values.

The software infrastructure for a PMU needs to set the configuration for monitoring, start and stop data collection, and finally read out and aggregate counter values to performance measures. There exist several tools and system interfaces supporting the collection of statistical data from PMUs. Among them, PAPI [11] and the `perf tool` [12] are commonly used in high-performance computing. These tools rely on a system interface running in kernel mode to access the PMU hardware counters. Running the system interface in kernel mode is advantageous since the hardware counters can easily be saved and restored during a context switch, allowing for per-thread performance monitoring. In the Linux operating system, two patches for performance monitoring are widely used: perfctr [13] and perfmon2 [14]. More recently, the `perf_event` interface [15] and the `perf tool` have been included into the main Linux kernel source code. The `perf tool` tightly works with the `perf_event` interface and makes performance monitoring straight-forward for users since there is no need to compile and add patches.

III. PMU Design and Integration

Leon3 [8] is a 32-bit open source processor platform from Gaisler Research implementing the Sparc-v8 architecture [16]. The platform supports multiple cores with a shared memory and is popular among researchers due to the rich set of supported tools, operating systems, and its extendability. In this chapter, we describe the architecture of our performance measurement units and how they integrate into the Leon3 platform as well as into the standard Linux performance measurement infrastructure.

A. The Architecture

We have been tailoring PMUs for Leon3 multi-core architectures. To be able to handle properly the performance counters for workloads migrating between the processors, the PMUs have been replicated for each processor core and the performance counters are stored to and are loaded from the context of an application by the OS kernel during each context switch. Fig. 1 shows the PMU placement in our current implementation, which is tailored for processor monitoring. The PMUs are connected to the signal pulses driven out from the Integer Units (IU), from L1 instruction and data cache controllers, and from the I-TLB as well as the D-TLB modules of the MMU. The standard open-source Leon3 architecture does not include L2 caches or floating-point units. However, our PMU architecture can easily be extended to monitor and aggregate events from such sources or from custom hardware modules and reconfiguration controllers in reconfigurable multiprocessor-on-chip systems.

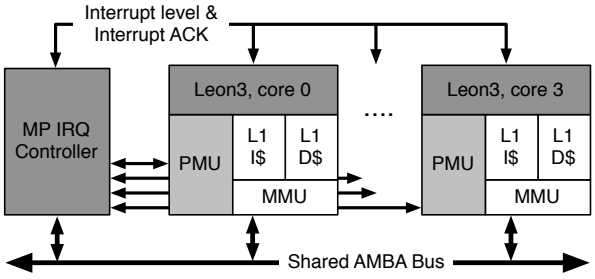

Fig. 1: PMU modules are located close to the monitored event sources. The interrupt signals for overflowed events are handled by the MP IRQ controller module.

Fig. 2 illustrates the hardware implementation of the overall PMU system. Each event counter subsystem consists of an event source multiplexer, a control logic block, a counter with according overflow logic and a common logic for the overflow interrupt generation. The heart of a performance counter subsystem is its control block. The control block takes input from a global and a local control register as well as the interrupt acknowledge from the interrupt controller. The functionality of the global register follows the ARM Cortex-A9 PMU architecture [4] and allows us to reset and enable all event counters by a single register access. This feature is of use for the Linux `perf_event` performance monitoring infrastructure.

Through the local control register a performance counter subsystem can be cleared (clr) and enabled (en), and counting can be started even if the measured processor core is entering the super user mode (su). Furthermore, the local control register determines whether an overflow interrupt should triggered and which event to measure. Currently defined event sources are the CPU clock cycle count, the number of executed instructions, the number of instruction and data cache read accesses, the number of instruction and date cache read misses, and the number of data cache write accesses as well as misses. The signal input of the first counter subsystem is hardwired to monitor the execution time to allow for an accurate measurement basis to which other measurement counters can be normalized to.

The interrupt acknowledge input signal of the control block depends on the situation of the associated counter. If the counter reaches its overflow threshold and the interrupt generation is enabled for this measurement subsystem, an interrupt is generated and a status bit in PMU's global interrupt overflow register is set. The overflow comparator signals the overflow to the control logic which, in turn, clears and sets its counter inactive. The activated software interrupt handler checks, which event counter has triggered an interrupt and updates the according perf_even counter variables. Afterwards, the interrupt handler releases the event counter by pulsing an interrupt acknowledgment signal through MP IRQ to the control logic blocks.

The presence of an interrupt logic is the reason for selecting 32 bit wide event counter registers. While using wider register widths, for instance 64 bits, would make a counter overflow less likely, there are cases where it is required to generate an interrupt after counting some specified amount of events. Additionally, even counting events at 100MHz will cause an interrupt request roughly every two minutes. Since the time overhead for the interrupt handler needed to serve an PMU interrupt is negligible, we avoided wider event counting registers for the sake of a compact architecture and a smaller memory map. However, the counter widths and the memory map can be easily adopted in our design if wider counters are required.

B. PMU Registers - Address Mapping and Access

Table I shows the address mapping for the overall PMU system. Instead of defining new instructions for reading and writing PMU registers, the extended Address Space Identifier (ASI) lda/sta instructions of the Sparc architecture are used [8]. These instructions are available only in system mode. The ASI = 0x02 is reserved for system control registers and has an unused address range from 0xC0 to 0xFC, which is used to clamp the PMU registers. The following sample code demonstrates how these registers can be accessed:

```
u32 val;
asm volatile ("lda [%1] %2, %0":
    "=r"(val): "r"(0xC0), i"(0x02));
```

In this sample code, ASI value 0x02 encoded in the instruction lda makes the IU load the current value in the global control register at address 0xC0, storing it in the variable val.

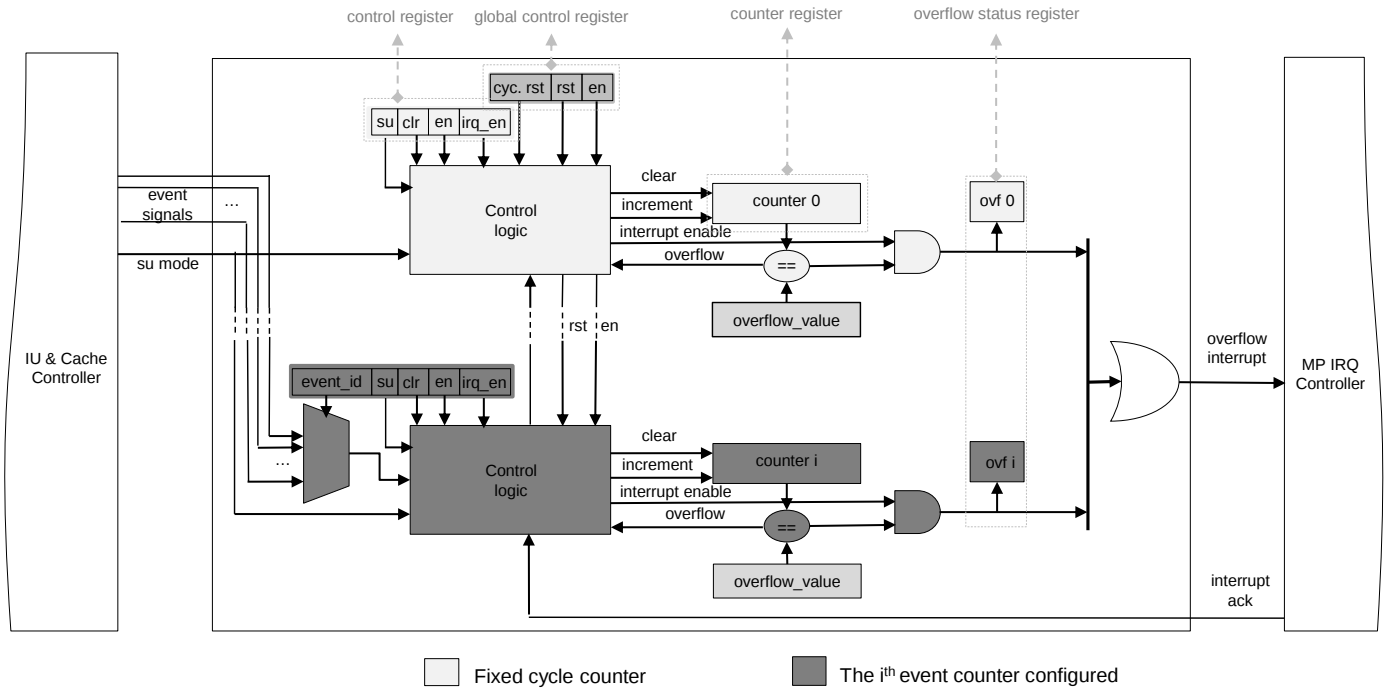

Fig. 2: The design for PMU module per core. The PMU monitors the event signals and manages the counters, depending on a set of control registers. The signals for the overflow interrupt are handled by MP IRQ Controller.

TABLE I: Memory map of the PMU system. The maximum number of event counters supported is limited to 7 due to the current limitations of our system design. The maximum number of monitored events can be up to 256. Currently, cycles, instructions, L1:I / L1:D access and read misses as well as L1:D write accesses and misses, and ITLB as well as DTLB misses are supported.

Registers accessed via ASI = 0x02	
Register—32 bits	ASI Address Mapping
Global control (RW): - Bit[0]: enable all event counters (en) - Bit[1]: reset/clear all event counters (rst) - Bit[2]: reset/clear cycles countered (cyc.rst) - Bit[7..3]: number of event counters supported - Bit[31]: reset/clear IRQ pending	$0xC0$
overflow status (RW): - Bit[0]: overflow cycle counter - Bit[n..1]: overflow for event counter - n..1 - Bit[31]: indication for IRQ pending	$0xC4$
Cycle counter (RW): - Bit[31..0]: counter value is being monitored	$0xC8$
Cycle counter control (RW): - Bit[7..0]: reserved - Bit[8]: enable the counter (en) - Bit[9]: reset/clear the counter (clr) - Bit[10]: counting kernel/user mode (su) - Bit[11]: interrupt enable (irq_en)	$0xCC$
The ith event counter (RW): - Bit[31]: counter value is being monitored	$0xD0 + 8 \cdot (\text{i}^{\text{th}})$
The ith event counter control (RW): - Bit[7..0]: event identifier (event_id) - Bit[8]: enable the counter (en) - Bit[9]: reset/clear the counter (clr) - Bit[10]: counting kernel/user mode (su) - Bit[11]: interrupt enable (irq_en)	$0xD4 + 8 \cdot (\text{i}^{\text{th}})$

C. Handling Overflow Interrupt

There are two basic ways of introducing interrupt sources to Leon3 processors. First, peripheral devices that are connected to the AMBA bus can use the AMBA interrupt lines. The AMBA bus interrupt controller has then to prioritize and relay the interrupt requests to the Leon3 MP IRQ controller. A PMU unit using this method of interrupt generation would need to implement an AMBA bus slave controller and accept the temporal overhead of the AMBA bus interrupt controller. Additionally, interrupts from other peripheral devices may have an impact on the measurement precision.

The second option, and the one we have chosen in this work, for interfacing to the interrupt logic of the Leon3 processor is to directly connect interrupt sources to the internal logic of the MP IRQ controller. To that end, all PMU interrupt request lines are aggregated by an OR gate and sourced into the external interrupt (EIRQ) handling circuitry of the MP IRQ controller. This is shown in Fig. 3. This method has also the that the number of AMBA bus devices is not increased.

D. System Integration - The Software Stack

Fig. 4 presents the integration of our PMUs into the standard Linux `perf_event` infrastructure. Instead of extending the standard Sparc-64 PMU code, we have adopted the perf_event.c code from the ARM PMU implementation due to similarities in the interrupt handling mechanism.

From the user space perspective, the `perf_event` interface and the `perf tool` work together as follows: The `perf tool` invokes an application for measurement. Depending on the input parameters, the `perf tool` provides the event sources to monitor to kernels `perf_event` measurement

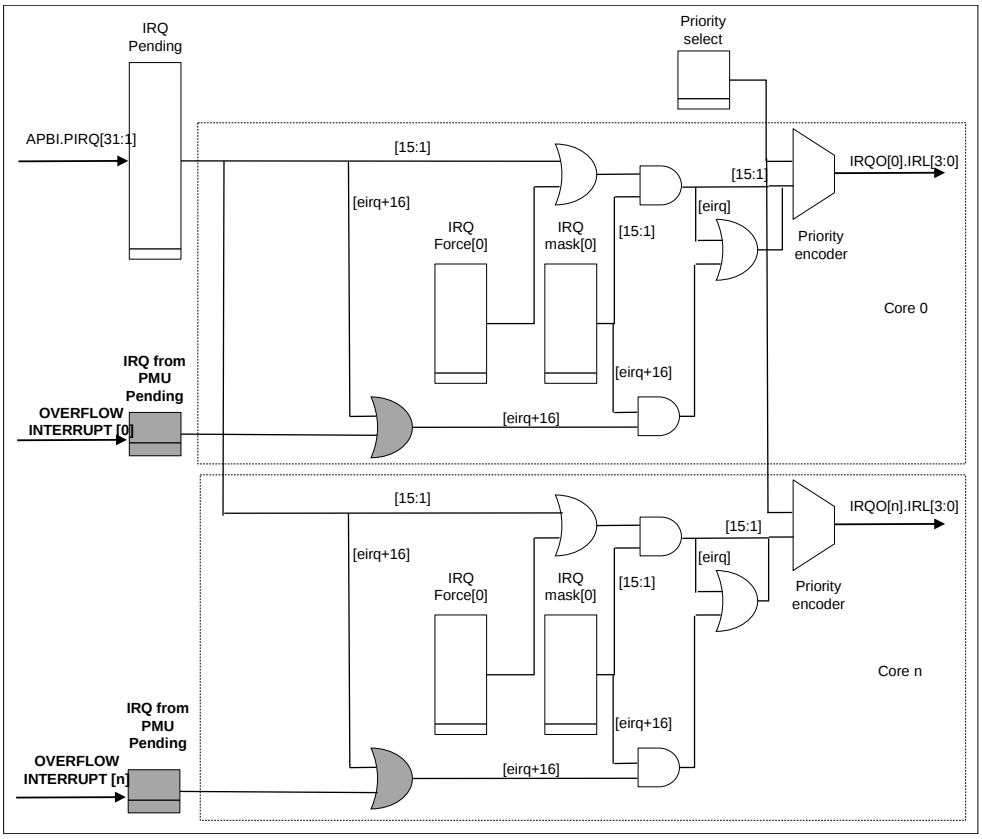

Fig. 3: The additional logic gates inside the MP IRQ Controller of the Leon3 platform support handling PMU interrupts for overflowed event counters.

infrastructure. The `perf_event` infrastructure in turn configures the PMU infrastructure and starts profiling. When the application finishes its execution, the `perf tool` reads out the event counters and aggregates the final results via the `perf_event` interface. An example for the output of the `perf tool` is given in Fig. 5.

IV. Experiments and Results

This section reports on our experiments and results. The used system configuration for the Leon3 platform is shown Table III. We have synthesized the Leon3 platform and programmed it to a Xilinx ML605 Virtex-6 board. The root file system is located on a CF card. Since our goal was to focus on collecting and analyzing statistical data for embedded system workloads, we have chosen a subset of MiBench [17], a free benchmark suite, for experimentation.

A. Hardware resource usage

Before presenting the benchmark results, Tab. II sums up the hardware overhead for the PMU subsystem when implemented for a single, dual and quad-core Leon3 platform. For the single-core variant the overhead for flop flops and look-up tables amount for 2 and 4.4 per cent, respectively. When doubling the core number of a Leon system, the integer units get duplicated. Busses and peripherals are typically not replicated. Therefore, the size of a Leon system grows linear with the number of cores but the hardware effort for the busses and peripherals stays almost unchanged. This explains, why when doubling and quadrupling the core number and the according PM units the overhead for the PM units compared to the total size of an Leon3 platform increases form 2 over 3.2 to 4.4 per cent for the number of flip flops and from 4.4 over 6.0 to 9.0 per cent for the number of look-up tables.

TABLE II: Hardware resource usage for implementing a Leon3 PMU and the total overhead in % compared to a PMU-less Leon3 system.

	1 core		2 cores		4 cores	
	FF	LUT	FF	LUT	FF	LUT
PMU	303	886	606	1641	1212	3956
MP IRQ without PMU	101	205	173	354	285	661
MP IRQ with PMU	102	210	175	368	289	694
Leon3 with PMU	15371	20956	19879	29413	28848	47142
Increase [%]	2.0	4.4	3.2	6.0	4.4	9.2

B. Measurement Output

Table IV displays all our measurements for the selected benchmarks. We have monitored eight events: CPU cycles, instructions, L1:I load misses, L1:I loads, L1:D loads, L1:D load misses, L1:D stores, and L1:D store misses. The presented results cover events captured during user mode, but exclude kernel mode execution. Since in our prototypical implementation the maximum number of event counters is

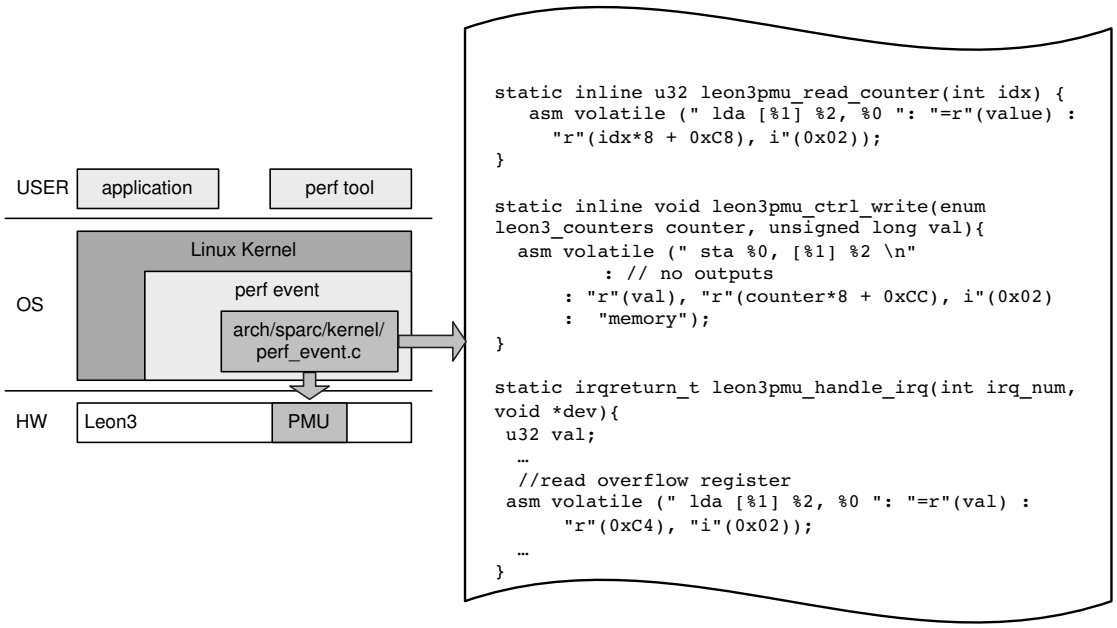

Fig. 4: System integration with `perf_event`.

```
# perf stat -e cycles,instructions,
L1-dcache-loads,L1-dcache-load-misses,
L1-dcache-stores,L1-dcache-store-misses
./queens -c 14

  14 queens on a 14x14 board...
  ...there are 365596 solutions

Performance counter stats for './queens -c 14':
8295258591     cycles
5309555048     instructions     # 0.640 IPC
965961200      L1-dcache-loads
50932          L1-dcache-load-misses
191890081      L1-dcache-stores
27365021       L1-dcache-store-misses
115.170000000  seconds time elapsed
# _
```

Fig. 5: Example: Launch and output of the `perf tool`.

seven, we had to invoke the `perf tool` twice in order to avoid multiplexing events, which can lead to inaccurate results during measurement. The first invocation of the `perf tool` is for counting the number of CPU cycles and instructions, the second for gathering the remaining events. We have repeated each measurement for 10 times and computed the coefficient variation (CV=σ/μ), i.e., the standard deviation divided by the mean average.

The chosen benchmark programs are deterministic in the sense that for a given input data set they execute exactly the same instructions. The reason that the collected values for the number of instructions is not deterministic is the `perf tool`, which also runs in user space. Once the `perf tool` receives a signal indicating that the application stops, the `perf tool` has to spend time for disabling the counters in the event control registers. This leads to a certain deviation in consecutive measurements., which are however, quite below 1%.

TABLE III: Leon3 platform and system configuration

Clock Frequency	75Mhz
Integer Unit	Yes
Floating Point	Software
Instruction Cache	2-way associative, 8KB, 32bytes/line, LRU
ITLB	8 entries
Data Cache	2-way associative, 4KB, 16bytes/line, LRU
DTLB	8 entries
MMU	Yes
MP IRQ Controller	External IRQ (EIRQ) = 14
PMU	New feature
Linux Kernel	2.6.36.4 patch from Gaisler
Compiler	Pre-built Linux toolchain from Gaisler

The variations in the number of cycles, L1:I read, L1:D read, and L1:D store miss events can be explained by initially varying states of the cache lines. Thus, for more accurate results the presented PMU infrastructure should be extended by a cache flush and a preheat procedure, to unify the starting conditions for all benchmarks.

V. Conclusion and Discussion

In this paper, we present a performance measurement infrastructure for the single- and multicore Leon3 processing platform. The infrastructure integrates seamlessly into the standard Linux performance measurement architecture `perf_event` and allow for a comfortable and accurate analysis of microarchitecture measurements using the standard Linux profiling tools. From the reconfigurable system perspective, the presented performance measurement infrastructure support also monitoring events generated by the reconfigurable platform, such as partial reconfiguration times.

TABLE IV: Statistical data collected for a subset workloads of MiBench. Coefficient of variation (CV= σ/μ) compared by running `perf tool` 10 times on the same application.

	1 core				2 cores				4 cores			
Benchmark	Cycles [%]	Inst. [%]	Time [s]	IPC	Cycles [%]	Inst. [%]	Time [s]	IPC	Cycles [%]	Inst. [%]	Time [s]	IPC
basicmath	0.006	0.000	23.005	0.596	0.052	0.000	23.176	0.594	0.040	0.000	23.434	0.594
bitcounts	0.003	0.000	0.821	0.758	0.017	0.000	0.888	0.757	0.056	0.000	0.893	0.755
qsort	0.017	0.001	0.978	0.373	0.081	0.000	1.019	0.373	0.173	0.000	1.048	0.371
jpeg	0.063	0.002	1.085	0.567	0.082	0.001	1.147	0.563	0.169	0.000	1.161	0.564
dijstra	0.052	0.001	1.767	0.486	0.007	0.000	1.81	0.486	0.123	0.056	1.819	0.485
patricia	0.007	0.001	5.421	0.207	0.040	0.001	5.479	0.206	0.145	0.000	5.617	0.206
stringsearch	0.023	0.004	0.077	0.255	0.136	0.000	0.147	0.251	0.109	0.000	0.152	0.251
FFT	0.007	0.000	27.131	0.599	0.008	0.000	27.232	0.598	0.042	0.003	25.419	0.597
CV	0.022	0.001			0.053	0.000			0.107	0.007		

	1 core			2 cores			4 cores		
Benchmark	L1:I load misses [%]	L1:D load misses [%]	L1:D store misses [%]	L1:I load misses [%]	L1:D load misses [%]	L1:D store misses [%]	L1:I load misses [%]	L1:D load misses [%]	L1:D store misses [%]
basicmath	0.040	0.247	0.346	0.078	0.546	1.474	0.140	0.463	1.232
bitcounts	0.136	0.176	0.382	0.102	0.102	0.318	0.256	0.106	0.472
qsort	0.027	0.018	0.160	0.041	0.046	0.154	0.139	0.030	0.092
jpeg	0.125	0.024	0.076	0.537	0.045	0.066	2.213	0.079	0.055
dijstra	0.052	0.025	0.104	0.052	0.018	0.024	0.601	0.187	0.539
patricia	0.006	0.150	0.184	0.004	0.150	0.124	0.012	0.251	0.232
stringsearch	0.108	0.010	0.009	0.189	0.040	0.117	0.170	0.053	0.039
FFT	0.057	0.166	0.393	0.062	0.298	2.558	0.298	0.659	2.243
CV	0.069	0.102	0.207	0.133	0.156	0.604	0.479	0.229	0.613

REFERENCES

[1] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[2] D. Zaparanuks, M. Jovic, and M. Hauswirth, "Accuracy of Performance Counter Measurements," in *Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 23—32.

[3] A. Gordon-Ross and F. Vahid, "A self-tuning configurable cache," in *Design Automation Conference (DAC), 44th ACM/IEEE*, 2007, pp. 234 – 237.

[4] ARM, *ARM Cortex-A9, Technical Reference Manual*. [Online]. Available: http://infocenter.arm.com

[5] *ADSP-BF535 Blackfin Processor Hardware Reference*. [Online]. Available: http://www.analog.com/static/imported-files/processor_manuals/ADSP-BF535_hwr_rev3.3.pdf

[6] [Online]. Available: http://documentation.renesas.com/doc/products/tool/rej10b0110_sh7750.pdf

[7] Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual*, UG585 (v1.7) ed., Xilinx Inc., Feb. 2014.

[8] Aeroflex Gaisler. [Online]. Available: http://www.gaisler.com/products/grlib/grlib.pdf

[9] B. Sprunt, "The Basics of Performance Monitoring Hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64—71, 2002.

[10] SUN Microsystems, *OpenSPARC T2 Core Microarchitecture Specification*. [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc

[11] J. Dongarra, K. London, M. S., P. Mucci, H. Terpstra D.and You, and M. Zhou, "Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters," in *PADTAD Workshop, IPDPS*, 2003, p. 289.

[12] *The linux perf tool*. [Online]. Available: http://www.perf.wiki.kernel.org/index.php

[13] M. Pettersson, *perfctr*. [Online]. Available: http://user.it.uu.se/~mikpe/linux/perfctr

[14] *Perfmon2: A flexible performance monitoring interface for Linux*. [Online]. Available: https://www.kernel.org/doc/ols/2006/ols2006v1-pages-269-288.pdf

[15] V. M. Weaver, "Linux perf event features and overhead," in *FastPath Workshop*, 2013.

[16] SUN Microsystems, *SPARC V8 Manual*. [Online]. Available: http://www.gaisler.com/doc/sparcv8.pdf

[17] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE 4th Annual Workshop on Workload Characterization*, 2001, pp. 3–14.