An embedded operating system API for monitoring hardware events in multicore processors

Giovani Gracioli and Antônio Augusto Fröhlich

Software/Hardware Integration Lab
Federal University of Santa Catarina
88040-900 - Florianópolis, SC, Brazil
{giovani,guto}@lisha.ufsc.br

***Abstract*—This paper presents an operating system API for monitoring hardware events specifically designed for embedded systems that use multicore processors. The proposed API uses the concepts from the Application-Driven Embedded System Design (ADESD) to construct a simple and lightweight interface for handling the complexity of today's Performance Monitoring Units (PMUs). In order to demonstrate the API usage, we monitored an event associated with bus snoops in a real processor. Based on the experience learned, we propose a set of guidelines, such as features for monitoring address space intervals and OS trap generation, that can help hardware designers to improve the PMU capabilities in the future, considering the embedded operating system's point of view.**

***Keywords*-Hardware performance counters, embedded systems, operating system, multicore architectures**

I. Introduction

Hardware Performance Counters (HPCs) are special registers available in the most modern microprocessors through the hardware Performance Monitoring Unit (PMU). HPCs offer support for counting or sampling several microarchitectural events, such as cache misses and instructions counting, in real-time [1]. However, they are difficult to use due to the limited hardware resources (for example Intel Nehalem supports event counting with seven event counters and AMD Opteron provides four HPCs to measure hardware events) and complex interface (e.g., low-level and specific to microarchitecture implementation) [2].

Nevertheless, it is possible to use multiplexing techniques in order to overcome the limitation in the number of HPCs [3], [1] or specific libraries that make the use of HPCs easier [4], while adding a low overhead to the application. Thus, HPCs can be used together with OS techniques, such as scheduling and memory management, to monitor and identify performance bottlenecks in order to perform dynamic optimizations [5]. In multicore systems, for instance, it is possible to count the number of snoop requests, last-level cache misses, and evicted cache lines.

As the utilization of multicore processors in the embedded system domain is increasing, an API for handling the complexity of HPCs specifically designed to this domain is desirable. The current HPCs APIs, initially proposed to general purpose computing, such as PAPI [4], are not the most suitable for embedded systems because they can use a substantial amount of extra code, which makes the communication between software and hardware PMUs slower, or require the initialization and creation of lists or event sets that decrease the performance. Moreover, these APIs provide several functionalities that cannot be of interest in an embedded application, such as user-defined events or performance modeling metrics. Hence, an API specifically tailored for the application's needs can be useful and provide a better performance for such applications.

This paper proposes a HPC API for monitoring hardware events specifically designed for embedded multicore systems. The API is designed following the Application-Driven Embedded System Design (ADESD) concepts [6] and it is able to provide to operating systems a simple and lightweight interface for handling the communication between applications and PMUs. Through an API usage example, in which a hardware event is used by the OS to help scheduling decisions, we were able to identify the main drawbacks of the current PMUs. As a consequence, we propose a set of guidelines, such as features for monitoring address space intervals and OS trap generation, that can help hardware designers to improve the PMU capabilities in the future, considering the embedded operating system's point of view.

The rest of this paper is organized as follows. Section II presents the related work. Section III provides an overview of the ADESD methodology and the Embedded Parallel Operating System (EPOS). Section IV presents the proposed PMU API. Section V discusses how the current PMUs could be improved in order to provide more detailed information to OSs. Finally, Section VI concludes the paper.

II. Related Work

The Performance API (PAPI) is the most used open source cross-platform interface for hardware performance counters [4], [7]. PAPI consists of a machine-dependent layer, a high level interface that provides access to read, start, and stop counters, and a low-level interface that provides additional features. It is not our target, however, to make a comparison between PAPI and our implementation. PAPI supports a wide range of platforms, was designed to general-purpose computing systems, and has been under development for more than 10 years. Instead, our interface is designed to embedded applications, which usually have their requirements

known at design time. Thus, it is possible to generate only the needed code for the application and nothing else.

Linux abstracts the usage of HPCs through a performance counter subsystem. A performance monitoring tool (*perf*) uses this subsystem and allows developers to obtain and analyze the performance of their applications. The tool also supports a command to record data into files which can be later analyzed using a *report* command. Other tools such as Intel Vtune [8] and AMD CodeAnalyst [9] offer a "friendly" interface to monitor performance of processors and applications through the use of HPCs. However, embedded systems usually do not have a control interface.

Considering HPCs as an alternative to easily detect sharing pattern among threads and help scheduling decisions in multicore processors, Bellosa and Steckermeier were the first to suggest using HPCs to dynamically co-locate threads onto the same processor [10]. Tam et al. use HPCs to monitor the addresses of cache lines that are invalidated due to cache-coherence activities and to construct a summary data structure for each thread, which contains the addresses of each thread that are fetching from the cache [11]. Based on the information from this data structure, the scheduler mounts a cluster composed of a set of threads, and allocates a cluster to a specific core. West et al. [12] propose an online technique based on a statistical model to estimate per-thread cache occupancies online through the use of HPCs. However, data sharing among cores is not considered by the authors.

Distributed Intensity Online (DI) is a multicore scheduler that reads the number of cache misses online and distributes threads across caches such that the miss rates are distributed as evenly as possible [13]. Calandrino and Anderson have proposed a cache-aware scheduling algorithm [14]. The algorithm uses HPCs to estimate the working set of each thread and to schedule them in order to avoid cache thrashing and provide real-time guarantees for soft real-time applications. However, to correctly estimate the working set, the threads must not share data and and the data size of the running threads must be less than the cache size.

In general, the above related work neither use nor propose an efficient API to handle the complexity of hardware performance counters in embedded systems. In this paper, we propose a simple and lightweight API for a common PMU family available in today's multicore processors (the Intel family of PMUs).

III. Application-Driven Embedded System Design

Application-Driven Embedded System Design (ADESD) is a methodology to guide the development of application-oriented operating system from domain analysis to implementation [6]. ADESD is based on the well-known domain decomposition strategies found in *Family-Based Design* (FSB) [15], *Object-Oriented Design* (OOD) [16], *Aspect-Oriented Programming* (AOP) [17], and *Component-Based Design* (CBD) [18] in order to define components that represent significant entities in different domains.

Figure 1 shows an overview of the ADESD methodology. The problem domain is analyzed and decomposed into independent abstractions[1] that are organized as members of a family, as defined in the FBD. To reduce environment dependences and to increase abstractions re-usability, ADESD aggregates the aspects separation (from AOP) to the decomposition process. Thus, it is possible to identify scenario variations and non-functional properties and to model them as *scenario aspects* that crosscut the entire system. The *scenario adapter* wraps an abstraction and apply into it a corresponding set of aspects that are enabled to that abstraction [19].

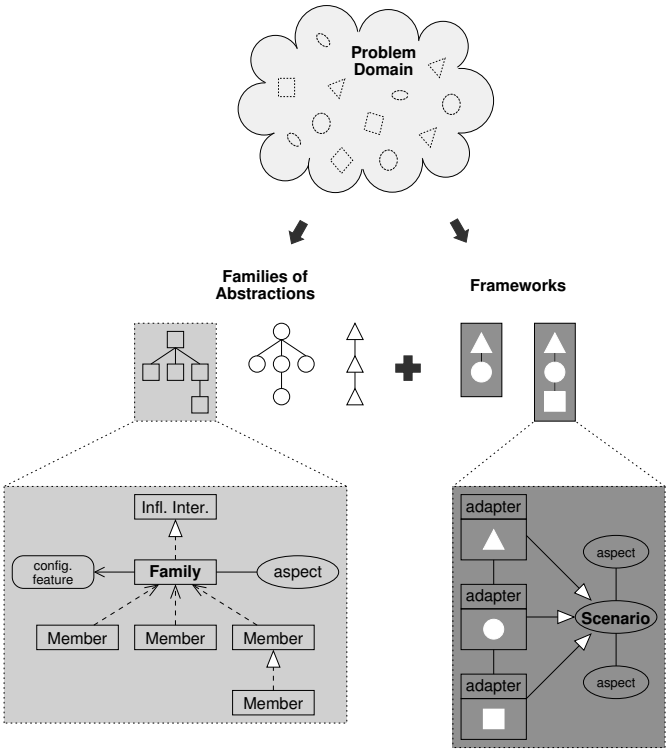

Figure 1. ADESD methodology overview.

Families of abstractions are visible to application developers through *Inflated Interfaces* that export their members as an unique "super component". These *inflated interfaces* allow developers to postpone the decision about which component should be used until enough configuration knowledge is acquired. An automatic configuration tool is responsible for binding an *inflated interface* to one of the family members, choosing the appropriate member that realizes the required interface.

A. Embedded Parallel Operating System

Embedded Parallel Operating System (EPOS) [2] is the first practical case study of ADESD. EPOS is a multi-platform, component-based, embedded system framework in which traditional OS services are implemented through adaptable,

[1] An abstraction is a class (component) with a well-defined interface and behavior.

[2] EPOS is available online at http://epos.lisha.ufsc.br.

platform-independent *System Abstractions*. Platform-specific support is implemented through *Hardware Mediators* [20], which are functionally equivalent to device drivers in UNIX, but do not build a traditional HAL. Instead, they sustain the interface contract between abstractions and hardware components by means of static metaprogramming and method inlining techniques that "dilute" mediator code into abstractions at compile-time (no calls, no layers, no messages; mostly embedded assembly).

Figure 2 presents an example of hardware mediators: the CPU hardware mediators family. This family handles the most dependencies of process management. The class `CPU::Context` defines the execution context for each architecture. The method `CPU::switch_context` is responsible for the context switching, receiving the old and new contexts. The CPU mediators also implement several functionalities as interrupt enabling and disabling and test and set lock operations. Each architecture defines a set of registers and specific address, but the same interface remains. Thus, it is possible to keep the same operations for platform-independent components, such as threads, synchronizers, and timers.

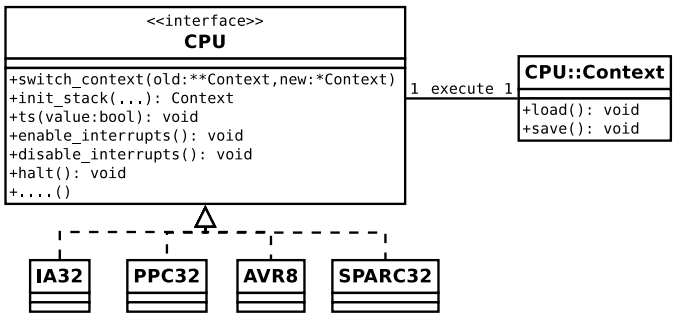

Figure 2. CPU hardware mediators.

The proposed API was implemented as a PMU hardware mediator family and a component in the EPOS operating system and will be presented in the next section.

IV. THE PROPOSED API

The proposed performance monitoring API was designed following the ADESD methodology. The monitoring infrastructure is composed by a PMU hardware mediator family and a platform-independent component. The interface of this component is used by application developers. Moreover, the component uses the hardware mediators in order to configure and read the HPCs. In the next subsections we present the hardware mediator family, the OS component, an example of how to use the API, and a practical use (OS scheduling).

A. PMU Hardware Mediator Family

We have designed a hardware mediator interface for the Intel PMU family. Figure 3 shows the UML class diagram of the interface. The Intel processors, depending on the microarchitecture (e.g., Nehalem, Core, Atom, etc), have different PMU versions. Each version provides different features and variable number of HPCs. For example, the PMU version 2 has two performance counters that can be configured with general events and three fixed performance counters that count specific events, while the PMU version 3 extends the version 2 and provides support for simultaneous multi-threading, up to 8 general-purpose performance counters, and precise event based sampling [22]. Yet, pre-defined architectural events, such as unhalted core cycles, last-level cache misses, and branch instruction retired, are shared by all the three versions.

Configuring an event involves programming performance event select registers (IA32_PERFEVTSELx) corresponding to a specific physical performance counter (IA32_PMCx). In order to select an event, the PERFEVTSELx register must be written with the selected event, unit, and several control masks. The unit mask qualifies the condition that the selected event is detected. For example, to configure the PMC0 to count the number of snoop responses to bus transactions, the PERFEVTSEL0 must be written with the EXT_SNOOP event mask (0x77) and two unit masks that define the conditions when the event should be counted.

The designed PMU hardware mediator family represents the described Intel PMU organization. A base class IA32_PMU implements the Intel PMU version 1 and common services for both version 2 and 3, including the pre-defined architectural events. Also, this class declares memory mapped registers and PMCs. The IA32_PMU_Version2 and IA32_PMU_Version3 extends the base class and implement specific services only available on that version. Finally, available hardware events are listed by specific microarchitecture classes. For instance, Intel_Core_Micro_PMU and Intel_Nehalem_PMU list all available events masks for the Intel Core and Intel Nehalem microarchitectures, respectively.

The hardware mediator interface could be used by a platform-independent component. This component is the one responsible for implementing "intelligent" logic by using the mediators. For instance, event ratios such as Cycles per Retired Instruction (CPI), parallelization ratio, modified data sharing ratio, and bus utilization ratio, combine two or more hardware events in order to provide useful insight into the application performance issues [8]. Moreover, a platform-independent component could also multiplex the hardware counters in order to overcome the limitation on the number of hardware counters. Multiplexing techniques divide the usage of counters over the time, providing to users a view that there exists more hardware counters than processors really support [4]. We also propose a performance monitoring component, which is described below.

B. Performance Monitoring Component

Figure 4 shows the performance monitoring component (`Perf_Mon`). It provides a set of methods to configure and read several event ratios and specific hardware events, such as last-level cache misses and L1 data cache snooped. The component hides from the users all the complexity of configuring, writing, and reading the hardware counters. Moreover, it also provides means for handling possible overflow in the counters.

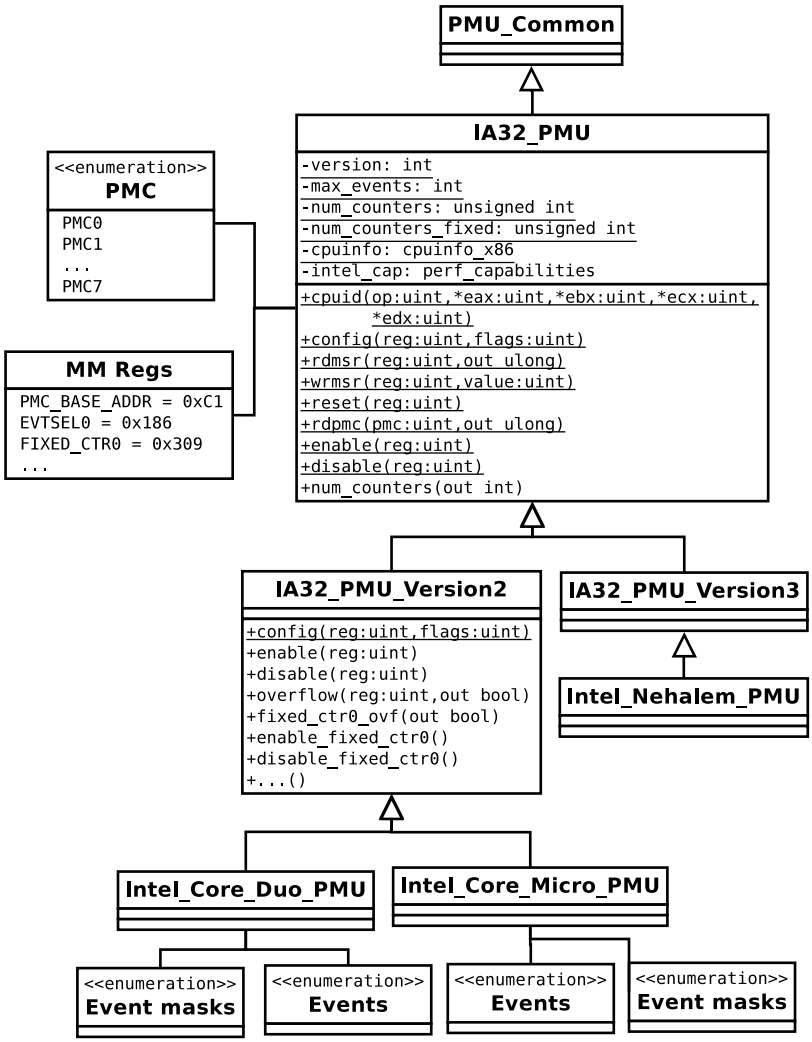

Figure 3. UML class diagram for the proposed PMU hardware mediator API.

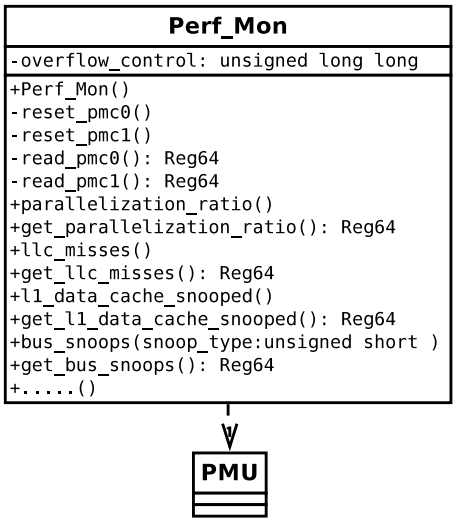

Figure 4. Performance monitoring OS component.

The performance monitoring component uses the presented hardware mediator family. However, due to function inlining, the code of the mediators is dissolved into the component. For example, consider the code in Figure 5. There is a method for configuring the event CPU_CLK_UNHALTED_BUS and another method for reading the event. The code of the mediator `Intel_Core_Micro_PMU::config()` is diluted into the `cpu_clk_unhalted_bus()` method, thus there is no overhead associated to method calls.

At the same way, the code for reading the performance counter 0 is diluted into the `get_cpu_clk_unhalted_bus()` and no method calling or argument passing is generated in the final system image. Consequently, the generated image only aggregates the code needed by the application and nothing else. In the future, we plan to add into the component and the hardware mediator family a support for other PMU families, as those found in the ARM and PowerPC processors. To this end, we plan to make a complete domain analysis and extracted the common events of each PMU family. Thus, it will be possible

to add a platform-independent layer for all PMUs. Moreover, we want to improve the PMU infrastructure, adding support for multiplexing and interrupt generation.

```
1   ...
2   class Perf_Mon
3   {
4   public:
5    ...
6   void cpu_clk_unhalted_bus(void) {
7             //configure PMC0 to monitor the
                    CPU_CLK_UNHALTED_BUS event
8             Intel_Core_Micro_PMU::config(PMU::EVTSEL0, (
                    Intel_Core_Micro_PMU::
                    CPU_CLK_UNHALTED_BUS | PMU::USR |
                    PMU::OS | PMU::ENABLE));
9   }
10
11  Reg64 get_cpu_clk_unhalted_bus(void) {
12          return read_pmc0();
13  }
14   ...
```

Figure 5. Perf_Mon using the hardware mediator. The hardware mediator code is "dissolved" into the component at compilation time.

C. API Usage

In order to exemplify the API usage, we have designed a benchmark to generate shared cache memory invalidations in a multicore processor (Intel Core 2 Q9550). The benchmark is composed by two versions of an application: a sequential and a parallel. Both applications execute the same code (2 functions), but the parallel runs the two functions (in two different threads) in different cores at the same time and share two arrays of data. We also implemented a third version (best-case), in which both functions execute in parallel but do not share data. The objective is to demonstrate the utility of the proposed API in a multicore processor.

Figure 6 shows how the API is used by the parallel and best-case applications. At the beginning of a thread (func0), the performance monitoring component is created and the method for monitoring the number of snoops in the L1 data cache is started. At the end of the function, the hardware event is read and printed in the screen. For the sequential version, the performance monitoring component is created in the main function, since the two functions are executed in a sequential order in the same core. We choose this event because it represents memory coherency activities between the cores.

The benchmark was also implemented on top of EPOS. Each application was executed 10 times in the Intel Core 2 Q9550 processor, then we extracted the average number of snoops for each of them. The arrays' size was set to 8 MB (4 MB each). Each function has a loop with 10000 repetitions in which math operations are executed using the data in the arrays. Figure 7 shows the measured values. We can clearly see the difference among the three applications. The sequential and best-case applications have obtained almost

```
1   ...
2   Semaphore s;
3   ...
4   int func0(void)
5   {
6           #ifndef __SEQUENTIAL
7           Perf_Mon perf0;
8           perf0.l1_data_cache_snooped();
9           #endif
10          register unsigned int sum0;
11           ....
12
13          #ifndef __SEQUENTIAL
14          s.p();
15          cout << "\nL1 data cache snooped func0 = " <<
                  perf0.get_l1_data_cache_snooped() << "\n";
16          s.v();
17          #endif
18  }
19  ...
```

Figure 6. An example of how to use the proposed API.

the same number of events, about 100.000. The parallel one obtained about 3 orders of magnitude more events and was slower than the sequential one. This confirms the software behavior – each function in parallel application frequently reads/writes the same cache lines, generating snoops in the bus and cache line invalidations. The explanation for snoops in the sequential and best-case applications is the natural implementation of a multicore OS, where shared variables are used to guarantee mutual exclusion in some data structures. The hardware event correctly measures the bus activities and can be used by the OS to improve performance. The standard deviation for the sequential, parallel, and best-case application was 4.83%, 0.13%, and 5.05% respectively.

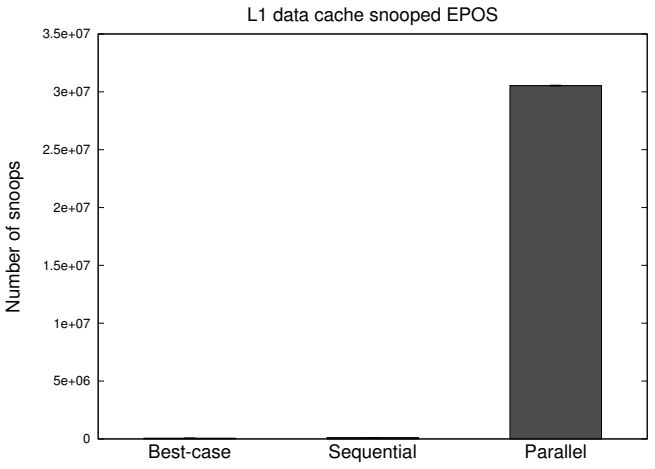

Figure 7. API usage example: number of snoops in the L1 data cache for the three benchmark applications.

In order to compare the obtained values in terms of correctness, we ran the same three benchmark applications in the Linux 2.6.32 and used the *perf* Linux tool to read the

number of snoops. We also ran each application for 10 times and extracted the average value. The evaluation was executed in the same Intel Q9550 processor. Figure 8 shows the obtained values. Linux has obtained in average 30% more snoops than EPOS and the standard deviation was also higher. The standard deviation for the sequential, parallel, and best-case application was 8.27%, 2.85%, and 9.96% respectively. As EPOS generates a system image composed by only the needed code, the influence of other OS parts in the execution of an application decreases. Consequently, the measured hardware events in EPOS are more precise than those obtained in Linux.

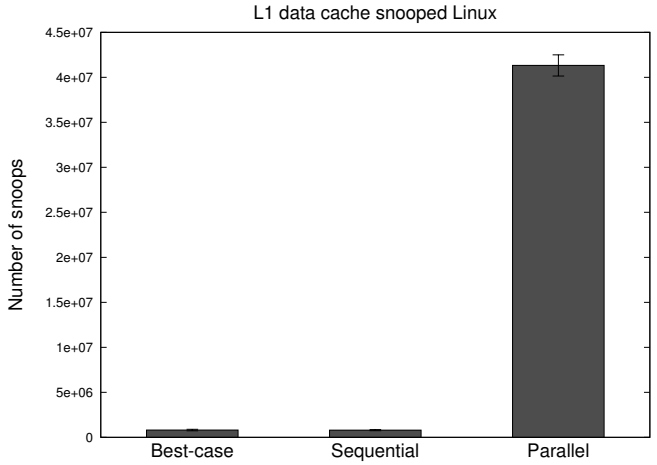

Figure 8. Number of snoops in the L1 data cache for the three benchmark applications running in Linux.

D. Helping the OS Scheduling

As an example to demonstrate the efficiency of HPCs to operating systems, we have used the same event (l1 data cache snooped) to monitor the number of snoops at run-time and help the OS scheduling decisions. We have added in the EPOS reschedule method a call to read the hardware event during a scheduling quantum (10 ms). We observed in the previous graph that during a quantum, the sequential and best-case applications obtained up to 100 events. By setting a threshold value (1000 in this case) we can detect when there is frequent snoops for cache lines and thus take a decision. Figure 9 shows the code exemplifying the changes. When the threshold value is reached, it is possible to move a thread to a core closer to the data (in case of a ccNUMA processor) and thus improving the application performance as demonstrated by Tam et al. [11].

```
1  void Thread::reschedule(bool preempt)
2  {
3       ...
4       Thread * prev = running();
5       Thread * next = _scheduler.choose();
6
7       unsigned long long n_l1_data_snooped = _perf.
            l1_data_cache_snooped();
8
9       if (n_l1_data_snooped > 1000) {
10           // move a thread to another core
11           ...
12      }
13
14      dispatch(prev, next);
15       ...
16 }
```

Figure 9. Modifying the OS scheduling.

V. DISCUSSION

In this section we provide a discussion about the API overhead and possible features that could be added in the future PMUs in order to improve their support particularly for embedded systems.

A. API Overhead

In order to demonstrate the performance of the proposed API, we have measured the memory overhead introduced by the API methods. The method for configuring the hardware counter (the same number of snoops hardware event as used above) occupied 32 bytes and 11 instructions and the method for reading the counter occupied 100 bytes and 40 instructions with no method calls, only inlined assembly code (Section IV-C provides a code example of how these methods were implemented). Other APIs designed to general-purpose computing, such as PAPI [7], require the initialization and creation of lists or event sets that decrease the performance. Moreover, these APIs provide several functionalities that cannot be of interesting of an embedded application, such as user-defined events or performance modeling metrics. Thus, an interface specific tailored for the application's needs can provide a better performance. Polpeta and Fröhlich have compared EPOS hardware mediators to HALs implemented on eCos and uClinux in terms of performance and memory consumption [21]. The authors have shown that hardware mediators have better results in both metrics.

The proposed interface could also be easily implemented to represent other PMU families, such those from the PowerPC and ARM processors. The API was implemented in C++ using the ADESD concepts in the EPOS operating system. EPOS is a component-based operating system, thus the same proposed PMU infrastructure could be used by other component-based operating systems without much implementation effort.

B. Improving the PMU Support

Initially, hardware designers have added PMU capabilities into processors to collect information for their own use [11]. However, PMUs features have become useful for other performance measurements, such as energy consumption management, memory partitioning, and scheduling decision. In consequence, the hardware designers are now adding more functionalities to PMUs, which can certainly help the software developer even more. Below we provide a discussion about desired features that could help hardware designers to improve PMU features in multicore processors:

- **Data address registers:** storing data addresses that generated an event could certainly provide to the OS a powerful mean to perform optimizations. The IBM Power5 processor has a similar feature, but it could be improved. In this processor, the last data address accessed is stored into a special register. Thus, the last monitored event can be associated to the last address accessed. The work proposed by Tam et al. used this feature to estimate the memory consumption of the system threads and to help a cache memory partition mechanism, improving the system performance [23]. It would be interesting if data address registers were associated to events that are representative to an OS, such as last-level cache misses, bus snoops, and bus transactions.
  The recent Intel processors have support for precise event-based sampling (PEBS) that identifies instructions that cause key performance events and allows the developers to allocate a PEBS buffer in memory to hold the samples (e.g., program counter and general-purpose registers values) collected during the program execution, which is extremely important for debugging mechanisms, such as tracing and replay [24].
- **Monitoring address space intervals:** in a multicore processor, several threads run in different cores and share the same address space. From the OS point of view, monitoring only the address spaces that are used by specific threads would allow a more precise view of the software behavior and consequently a more correct action could be taken by the OS.
- **Processing cycles spent in specific events:** especially for embedded hard real-time applications, where deadlines must be always met, bus cycles spent in specific events are extremely important for estimating the influence of shared resources in the threads execution time. In the Intel processors, for example, there are events for measuring the bus cycles spent accessing the shared cache, bus cycles when data is sent on the front-side bus, bus cycles when the HIT and HITM pins are asserted in the bus, and so on. However, it is difficult to get the cycles for a specific event, as a cache miss or a bus snoop. Improving the PMU capabilities for providing the precise number of cycles in an event could ease the OS task of guaranteeing the deadlines for real-time applications running in multicore processors.
- **OS trap generation:** PMUs could generate traps for the OS according to pre-defined numbers associated to events. This feature would allow the OS to be interrupted only when the number of hardware events is reached. Therefore, the OS could handle the exception and take a decision based on the the event that generated the trap.

VI. Conclusion

This paper presented an API for monitoring hardware events in multicore processors considering the embedded system domain. The API was designed following the concepts from the Application-Driven Embedded System Design (ADESD), thus it is possible to generate only the needed code for the application and nothing else, achieving a good performance. The method for configuring a hardware counter occupied 32 bytes and 11 instructions and the method for reading the counter occupied 100 bytes and 40 instructions with no calls, only inlined assembly code.

Based on the experience learned during the implementation and tests, we proposed a set of guidelines that can help the hardware designers to improve the PMU capabilities in the future considering the embedded operating system's point of view: data address registers available to the OS, features for monitoring address space intervals, events for measuring processing cycles spent in specific events, and OS trap generation according to pre-configured events. As future work, we plan to use the proposed API together with a real-time scheduling in order to provide hard real-time guarantees for real-time applications running on multicore processors.

Acknowledgments

This work was supported by the Coordination for Improvement of Higher Level Personnel (CAPES) grant, project RH-TVD 006/2008.

References

[1] B. Sprunt, "Pentium 4 performance-monitoring features," *IEEE Micro*, vol. 22, no. 4, pp. 72–82, Jul/Aug 2002.

[2] R. Azimi, M. Stumm, and R. W. Wisniewski, "Online performance analysis by statistical sampling of microprocessor performance counters," in *Proceedings of the 19th annual international conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, pp. 101–110.

[3] J. May, "Mpx: Software for multiplexing hardware performance counters in multithreaded programs," in *Proceedings of the 15th International Parallel and Distributed Processing Symposium.*, Apr. 2001, p. 8 pp.

[4] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou, "Experiences and lessons learned with a portable interface to hardware performance counters," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 289.2–.

[5] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *SIGOPS Operating System Review*, vol. 43, pp. 56–65, April 2009.

[6] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[7] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[8] R. K. Malladi, *Using Intel® VTuneTM Performance Analyzer Events/Ratios Optimizing Applications*, no. Intel® White Paper.

[9] P. J. Drongowski, *An introduction to analysis and optimization with AMD CodeAnalyst Performance Analyzer*, September 2008.

[10] F. Bellosa and M. Steckermeier, "The performance implications of locality information usage in shared-memory multiprocessors," *Journal of Parallel Distributed Computing*, vol. 37, pp. 113–121, August 1996.

[11] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," *SIGOPS Operating System Review*, vol. 41, pp. 47–58, March 2007.

[12] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang, "Online cache modeling for commodity multicore processors," *SIGOPS Operating System Review*, vol. 44, pp. 19–29, December 2010.

[13] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS '10, 2010, pp. 129–142.

[14] J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 194–204.

[15] D. Parnas, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, pp. 1 – 9, march 1976.

[16] G. Booch, *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming." in *ECOOP'97*. SpringerVerlag, 1997, pp. 220–242.

[18] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design & Test*, vol. 18, pp. 23–33, November 2001.

[19] A. A. Fröhlich and W. Schröder-Preikschat, "Scenario Adapters: Efficiently Adapting Components," in *4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, USA, Jul 2000.

[20] F. V. Polpeta and A. A. Fröhlich, "Hardware mediators: a portability artifact for component-based systems," in *In Proceedings of the International Conference on Embedded and Ubiquitous Computing*, ser. Lecture Notes in Computer Science, vol. 3207. Aizu, Japan: Springer Berlin / Heidelberg, 2004, pp. 45–53.

[21] F. V. Polpeta and A. A. Fröhlich, "On the automatic generation of soc-based embedded systems," in *In Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005.

[22] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, January 2011, no. 253668-037US.

[23] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '09. New York, NY, USA: ACM, 2009, pp. 121–132.

[24] G. Gracioli and S. Fischmeister, "Tracing interrupts in embedded software," in *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, ser. LCTES '09. New York, NY, USA: ACM, 2009, pp. 137–146.