Accelerating and benchmarking operating system functions in a “soft” system

Using reconfigurable technology to provide reliable performance characteristics

Péter Molnár, Ádám Kicsák
PhD School of Informatics
University of Debrecen
Debrecen, Hungary
pmolnar@lib.unideb.hu, pixels@vipmail.hu

János Végh
Institute of Informatics
University of Miskolc
Miskolc, Hungary
jvegh@mazsola.iit.uni-miskolc.hu

***Abstract*—The todays computing technology provokes serious debates whether the operating system functions are implemented in the best possible way. The suggestions range from accelerating only certain functions through providing complete real-time operating systems as coprocessors to using simultaneously hardware and software implemented threads in the operating system. The performance gain in such systems depends on many factors, so its quantification is not a simple task at all. In addition to the subtleties of operating systems, the hardware accelerators in modern processors may considerably affect the results of such measurements. The reconfigurable systems offer a platform, where even end users can carry out reliable and accurate measurements. The paper presents a hardware acceleration idea for speeding up a simple OS service, its verification setup and the measurement results.**

Keywords—soft-processor, operating system, performance measurement, hardware accelerator, reconfigurable systems

I. Motivation

A decade ago, the impressive development of the single-processor computing performance stalled, see Fig. 1 in [1]. On one side, the reason is in electronic technology: the physical limits of hardware implementation of sequential computing seem to be reached, in clock rate [1], dissipation [2], computational density [3], etc. Even since that time "*Processor and network architectures are making rapid progress with more and more cores being integrated into single processors and more and more machines getting connected with increasing bandwidth. Processors become heterogeneous and reconfigurable* …" [5]. However, using out the available hardware resources is not simple even with right hardware support: for example, the hyper-threading "*generally improves processor resource utilization efficiency, but does not necessarily translate into overall application performance gain*" [5].

The software side is also not more hopeful: "*parallel programs . . . are notoriously difficult to write, test, analyze, debug, and verify, much more so than the sequential versions*" [7]. The conclusion is that “*No current programming model is able to cope with this development, though, as they essentially still follow the classical van Neumann model”[5]*. This conclusion caused starting researches in several directions, including those to speed up operating system functionality. To draw quantitative conclusions, the performance of the modified operations must be measured, both in hardware and software.

The performance measurements in computer systems are a field, where solid background knowledge and carefully designed measuring conditions are required, if one wants to derive reasonable performance metrics. The performance can be described from different points of view [8]. Functions of the operating systems – from programmer’s point of view – can be considered as formally short but functionally rather complex machine instructions, and might consume typically many thousands of clock cycles in modern operating systems [9]. The modern processors are constructed with many hardware-accelerating solutions [10], the operation of which can change the execution times in a considerable and nondeterministic way. On top of this nondeterministic operation is superimposed the operation of multitasking operation systems, where the scheduling of the operating system can insert foreign code parts – from the point of view of execution time – into the tested code. Because all of this, maximum care must take place when choosing platform and method for the measurements.

II. Unit under test

In modern computing, the services of the operating systems are frequently used, because of comfort and safety. The performance of their implementation may have serious impact on the performance of the applications. It is especially so for simple services, where the overhead needed to reach the operating system is disproportionally large. This is the reason why researchers attempt to accelerate reaching OS services, using various methods. The methods include complete real-time operating system coprocessor 0 or mixing software-implemented and hardware-implemented threads [11] in the operating system. Although the lack of synchronization was early identified as one of the two fundamental issues of computing [12], even today most of synchronization functionality is done by software, in the operating systems.

In our case the idea was using an easy to understand, implement and handle service: the binary semaphore. As pointed out 0, in this particular case the real goal is to keep the semaphore information in a place where it is not directly reachable for the applications of the operating system. From different reasons, this functionality is traditionally implemented as one of the services of the operating system. This of course needs the usual frame of using the OS services, with the disadvantage of causing a disproportionally high overhead. Our idea was to store this information in and handle by an independent non "stored program" hardware accelerator unit, which was implemented in a reconfigurable device and linked to the processor implemented in the same reconfigurable device. In this way any of the processes of the OS running on the processor can reach the semaphore information only through using the semaphore hardware interface 0.

III. RELATED WORK

In similar measurements several factors, influencing performance measurements, have been scrutinized. Bershad et al [13] have shown, that cache and write buffer have a crucial role in system performance in writing and interpreting micro-benchmarks. Jones et al. [14] provide cache behavior statistics during program execution on a modified LEON processor. Rajagopalan et al. [15] emphasize the high time consumption of kernel bound crossing. Shannon and Chow[16] focus on software performance measuring with gprof integration. The importance of fine-granularity of performance measuring tools is emphasized [17] not only in embedded systems, but also for example in virtualized environments.

One of the fine points of the cooperation between CPU and an independent hardware accelerator, like our HW-implemented semaphore unit, is the method how they are linked. Because most hardware accelerators are prepared to assist commercially available processors, which cannot be changed, the usual method is to link the accelerator as one of the I/O peripherals.

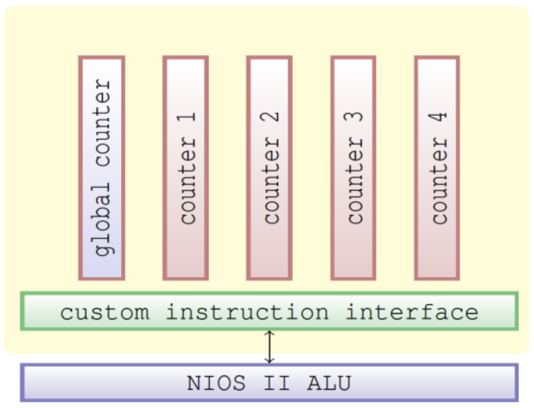

Fig. 1. Performance counter custom instruction block diagram.

While this method of linking is simple, and in this way some characteristics of the operation, like determinism, can be improved 0, it is not performant. In modern operating systems the I/O operations must run under the protection of the operating system, the mandatory usage of which causes considerable overhead. In our case this overhead would take away most part of the expected speedup, so we took a different approach.

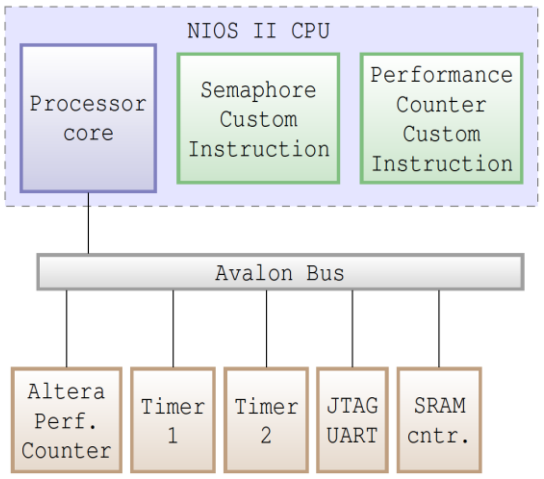

Fig. 2. The SoPC setup used for the benchmarking.

IV. PLATFORM AND METHOD

The commercially available processors are not modifiable and also many of the details of their internal operation are not publicly known. Similarly, the commercial operating systems are not available in open source form. Fortunately, reconfigurable devices reached a high level of functionality, and high quality processors with modern architecture, so called "soft processors" can be implemented in their fabrics. Also, complete open source operating systems can run on those processors, allowing preparing complete soft "System on Chip" applications. In order to avoid the dangers in timing that may occur in modern computer systems, the possible simplest measurement setup shall be assembled on such platform.

A. *The hardware architecture*

For our experiments the NIOS II [18] processor was chosen, mainly because of its easy customizability of its machine instructions. We employed our own developed semaphore and performance counter 0 (see Fig. 1) modules for measurements. We connected to the NIOS II platform the manufacturers performance counter, Avalon bus, SRAM controller, and JTAG debugger (Fig. 2). The minimum amount of devices on the Avalon bus was used, in order to avoid unpredictable overhead. Any unexpected interrupt on the bus could modify the results.

Communication with the host computer is possible through JTAG UART component and USB Blaster Download cable. All the desired modules can be customized and configured for the measurements. We used NIOS II economy and fast core types of the processor during the benchmarking tasks. We modified only the core type; other options remained at their default values.

B. *The software architecture*

Similarly, mainly because of its easy customizability, the Altera Hardware Abstraction Layer and µC/OS-II operating system [19] was used and modified for our goals. The idea was to implement semaphore handling as a (custom) machine instruction, then implement semaphore handling in the OS both the traditional way, though operating system service and in our preferred way, implementing it through using a custom instruction directly. The Nios II Economy core is optimized for small area utilization instead of performance. Even if semaphore operations ends up in one clock cycle, processor instruction execution and data transfers needs additional clock cycles ([18] pp 5-11).

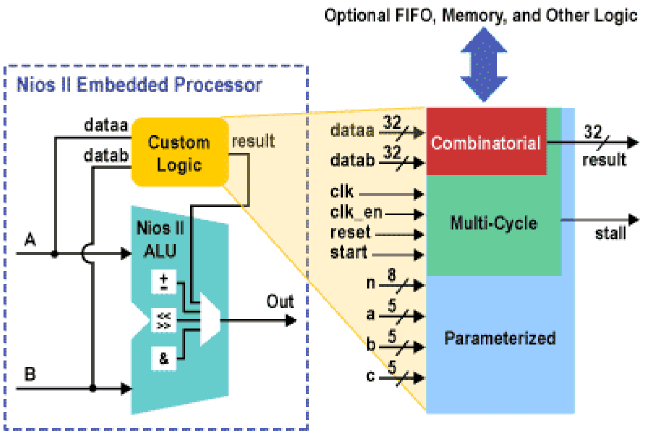

Fig. 3. The Nios II custom instruction.

C. *Linking the hardware semaphore to the processor*

In our case we also had to solve to link a subsystem, implemented in non "stored program" way, to the operating system. This was accomplished in two steps. In the first step the semaphore module 0 was implemented as custom instruction, see Fig. 3. In the second step the semaphore handling was re-implemented, using the custom instruction. Another custom instruction was implemented to measure the overhead of the measurement.

The semaphore hardware implementation is connected to the NIOS II core through extended type custom instruction which determines the calling syntax. The assembler syntax: custom N, xC, xA, xB, where xA, xB are input parameters, xC an output register and N selects the desired custom instruction ([18], pp 8-47). At system generation time, C macros and functions are generated for use in operating system. The generated C macro syntax is the following:

ALT_CI_<name>_CUSTOM_INSTRUCTION_0(<N>,<A>,<B>);

This macro can return custom instruction generated value.

D. *The testbench*

Two software projects were used: one for measuring performance counter custom instruction overhead and another one for measuring operating system semaphore performance with semaphore custom instruction. The overhead computations were realized with assembly instructions; the semaphore measurements were implemented in C.

Table contains the result of one of the most important validating steps: the measurement results of our performance counter custom instruction overhead. To start or stop a counter, two processor instructions need to be executed: a movi and a custom. The former loads in the register the chosen counter identifier and the latter instructs the processor to execute the counter handling custom instruction. Reading the counter values is a separate step, and it can be accomplished at any time. Overhead tests are executed without using the operating system. As shown, even in this simple case serious indeterminism is introduced: the hardware accelerators implemented in the full core make this overhead shorter, typically 22 clock ticks; but – depending on the system state, in a not predictable way – this time can also be 30 ticks.

TABLE I. PERFORMANCE COUNTER CUSTOM INSTRUCTION OVERHEAD (IN CLOCK TICKS)

Custom instruction	**Nios II core type**	
	Economy	***Full***
Overhead	37	22; max 30

V. RESULTS

We employed our own developed semaphore and performance counter modules for measurements. We connected to the NIOS II platform the manufacturers performance counter, Avalon bus, SRAM controller, JTAG debugger, see Fig. 2. We measured the processor in the smallest economy and the fastest full version configurations. While the first one contains only the most necessary components, the last one contains cache and branch predictor as well. We used the minimum amount of devices on the Avalon bus, in order to avoid unpredictable overhead. Any unexpected interrupt on the bus could modify the results, so the measurements run with disabled interrupts.

As an example, the measurement results for the operation creating a semaphore is shown in Table II. Using hardware acceleration in the full core changes the speedup both for the hardware and software implemented versions, and makes the execution time rather unpredictable.

TABLE II. MEASUREMENT SUMMARY OF SEMAPHORE FUNCTION CREATE (IN CLOCK TICKS)

Time to create	**Nios II core type**	
	Economy	***Full***
In software	2726	851-1195
In hardware	77	43-75

Notice also how huge reserves in using resources are still available in the operating mode of the stored program computers: the semaphore operation itself needs only 1 clock cycle, to use it as a single machine instruction takes 19 clock cycles, as a C instruction 77 cycles. If the same functionality is implemented in the operating system in the traditional way, it takes 2726 cycles. Note that in our case the operating system uses supervisor mode only, so there is no need to cross the kernel bound. In the case of operating systems like Linux or Windows the same operation needs about 20,000 cycles [9].

Orders of magnitude in performance are lost for the comfort of using stored program processors and an operating system.

CONCLUSIONS

To measure performance either of a modern processor or the operating systems running on it, or both, is a real challenge. To separate the different contributions to the measurement time from each other is an unsolvable task, if one has a commercial processor and operating system. It is not guaranteed at all, that the hardware platform and/or software services are really comparable, if we do not know all of the relevant details of the implementation. The completely “soft” systems provide a platform, where hardware and/or software features are under complete control, and those individual features can be virtually switched on and off, through generating the ‘unit under test = a specific HW/SW ensemble’ in some otherwise identical environment.

Because of the complexity of the HW/SW interactions, even in such controlled environment special care must be exercised, partly because some consequences of the switched features interfere, partly because the measurement device overlaps with the measured system. When using open-source operating system on open source hardware processor, everything is under complete control. As the presented results show, with carefully designed measurements accurate values can be derived, where the nature of the studied process is deterministic, and range of execution times can be determined where it is not. In the simple case presented the results completely match our expectations: the execution times received verify that using our ‘single-shot’ measurements both processor and operating system service functionalities can be studied with clock cycle accuracy. The prepared measurement setup and environment settings are authentic to carry out such measurements, even in such a complex environment. The measurement setup and methodology allow qualifying hardware and/or software acceleration solutions. The measurements also proved that with proper changes in the functionality of OS services, considerable performance enhancement could be reached, while maintaining compatibility with the traditional solution.

References

[1] S. H. Fuller, L. I. Millett, Computing Performance: Game Over or Next Level?, Computer 44 (2011) 31-38.

[2] V. Agarwal et al, Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures, in: Proc. of the 27th Annual Internat. Symp. on Computer Architecture, 2000.

[3] H. Esmaeilzadeh et al., Dark Silicon and the End of Multicore Scaling, IEEE Micro 32 (3) (2012) 122-134.

[4] J. Williams et al., Computational density of fixed and reconfigurable multi-core devices for application acceleration, Proceedings of Reconfigurable Systems Summer Institute, Urbana, IL, Jul. 2008.

[5] S(o)OS project, Resource-independent execution support on exa-scale systems, http://www.soos-project.eu/index.php/related-initiatives, 2010.

[6] S. Saini et al, The impact of hyper-threading on processor resource utilization in production applications, Proceedings of 18th International Conference on High Performance Computing (HiPC), 2011. Bangalore, India. pp 1-10, DOI: 10.1109/HiPC.2011.6152743

[7] J. Yang, et al, Making Parallel Programs Reliable with Stable Multithreading, Communications of the ACM 57 (3) (2014) 58-69. doi:10.1145/2500875.

[8] D. J. Lilja: Measuring computer performance. Cambridge university Press, 2004. ISBN 0-511-03627-2.

[9] B. E. Randal and D. R. O'Hallaron: Computer Systems: A Programmer's Perspective, Pearson 2010

[10] J. L. Hennessy and D. A. Patterson: Computer Architecture: A Quantitative Approach. Morgan Kaufmann/Elsevier, 2006. ISBN 9780080475028

C. M. Ferreira and S. R. Oliveira: RTOS Hardware Coprocessor Implementation in VHDL. 2009. http://www.academia.edu/200563/RTOS_Hardware_Coprocessor_Implementatio_in_VHDL

[11] H. K-H. So, BORPH: An Operating System for FPGA-Based Reconfigurable Computers, PhD Thesis 2000, University of California, Berkeley.

[12] Arvind and Iannucci, Robert A.: Two Fundamental Issues in Multiprocessing. 1988 4th Internat. DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering, pp. 61–68.

J. Végh, Á. Kicsák, Zs. Bagoly and P Molnár: An Alternative Implementation for Accelerating Some Functions of Operating System. Proceedings of the 9th International Conference on Software Engineering and Applications, Vienna, Austria, 29-31 August, 2014. pp 494-499, http://dx.doi.org/10.5220/ 0005104704940499

[13] B. Bershad, R. P. Draves, and A. Forin, Using microbenchmarks to evaluate system performance. In IEEE Proceedings of the Third Workshop on Workstation Operating Systems 1992, pp 148–153.

[14] P. Jones, P., et al.: Extracting and improving microarchitecture performance on reconfigurable architectures. International Journal of Parallel Programming, 2005, 33:136.

[15] M. Rajagopalan, et al. (2003). System call clustering: A profile directed optimization technique. doi:10.1.1.127.4691.

[16] L. Shannon, L. and P. Chow, P. (2004). Using reconfigurability to achieve real-time profiling for hardware/ software codesign. In Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, FPGA ’04, pages 190–199, New York, NY, USA. ACM.

[17] Anand, A., et al.: Resource usage monitoring for kvm based virtual machines. 2012 18th Internat. Conference on Advanced Computing and Communications (ADCOM), 2012, pp. 66–70.

[18] Altera Corporation, NIOS II Processor Reference Handbook, 2014. http://www.altera.com/

[19] Micrium Inc., μC/OS-II, 2014, http://www.micrium.com/

[20] V.M. Weaver, D. Terpstra, D. and S. Moore, Non-determinism and overcount on modern hardware performance counter implementations. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) 2013, pp 215-224 http://dx.doi.org/10.1109/ISPASS.2013.65571