Chip-Size Evaluation of a Multithreaded Processor Enhanced With a PID Controller

Michael Bauer, Mathias Pacher, and Uwe Brinkschulte

Embedded Systems
Goethe-University Frankfurt am Main, Germany
{mbauer, pacher, brinks}@es.cs.uni-frankfurt.de

Abstract. In this paper the additional chip size of a Proportional/Integral/Differential (PID) controller in a multithreaded processor is evaluated. The task of the PID unit is to stabilize a thread's throughput, the instruction per cycle rate (IPC rate). The stabilization of the IPC rate allocated to the main thread increases the efficiency of the processor and also the execution time remaining for other threads. The overhead introduced by the PID controller implementation in the VHDL model of an embedded Java real-time-system is examined.

Keywords: Multithreaded processors, real-time processors, control loops for processors

1 Introduction

Today it can be observed that electronic devices are getting more and more omnipresent in everybody's life. These embedded systems are small and often have only limited power supply. Processors in embedded systems have to meet different requirements then CPUs in personal computers. The microcontrollers used in such devices have to be fast and energy efficient. Furthermore, the applications running on embedded devices often have real-time requirements.
The most simple solution to handle many threads and to ensure on-time execution for real-time applications would be a fast high-end processor. However, high-end processors consume too much energy for most embedded applications. Furthermore, due to techniques like caches and speculation their real-time performance is hard to predict.
Our solution is to optimize the performance and utilization of a smaller and simpler processor core. We use a PID controller to stabilize the IPC rate for a real time thread. This additional controller reacts adaptively to unpredictable latencies caused by load/store instructions, branches or locks. By reducing the allocated time for a real-time thread to the required rate, the lower priority threads can be executed faster. The smaller consumption of the execution time allows the use of smaller processors.
However, the use of a PID controller increases the energy consumption and needed chip space. In this paper we analyze the additional space needed for the hardware implementation of a PID controller based on VHDL. The logic cells used in a FPGA permit a good estimation of energy and space consumption.
The core we used for our evaluation is an enhancement of the multithreaded Java microcontroller named Komodo [6]. Komodo offers the required capabilities for real time

applications and is also multithreaded. Therefore, the mentioned goals of the PID controller can be tested on this processor.
Furthermore, basic settings and requirements have been researched via a PID controller implementation in a software processor simulator [6,5] using different benchmarks. The resulting controller parameters (e.g. width of the controlling window) are now used as a basis for the chip space evaluation.

2 The Komodo Microcontroller

The Komodo microcontroller consists of a processor core attached to typical devices such as a timer/counter, capture/compare, serial and parallel interfaces via an I/O bus [28]. In the following we explain details about the processor core which is kept at a simple hardware level because of its real-time applications.

2.1 The Pipeline

Figure (1) shows the pipeline enhanced by the priority manager and the signal unit. The pipeline consists of the following four stages: instruction fetch (IF), instruction window and decode (ID), operand fetch (OF) and execute, memory and I/O-access (EXE). These stages perform the following tasks as described in [34]:

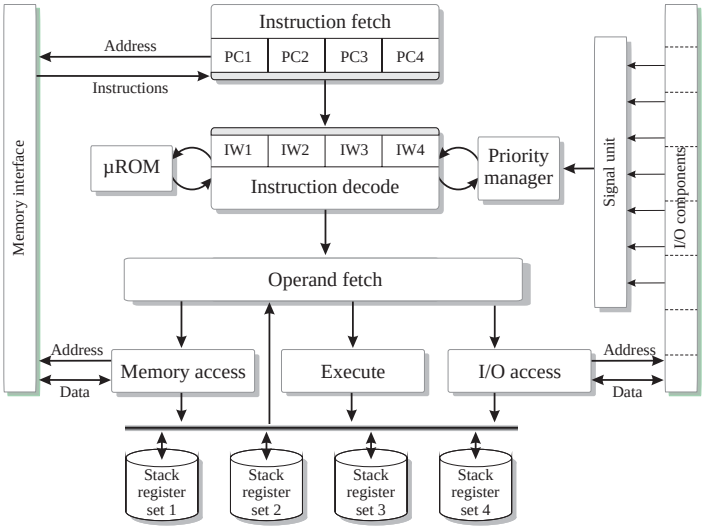

Fig. 1. The pipeline of the Komodo microcontroller

Instruction fetch: The instruction fetch unit tries to fetch a new instruction package from the memory interface in each clock cycle. If there is a branch executed in EX

the internal program counter is set and the instruction package is fetched from the new address.

Instruction window and decode: The decoding of an instruction starts after writing a received instruction package to the correct instruction window. Hereby, the priority manager decides which thread will be decoded by ID in every clock cycle.

Operand fetch: In this pipeline stage operands needed for the actual operation are read from the stack.

Execution, memory and I/O-access: There are three units in the execution stage (ALU, memory and I/O). All instructions but load/store are executed by the ALU. The result is sent to the stack and to OF for forwarding. In case of load/store instructions the memory is addressed by one of the operands. An I/O-access is handled in the same manner as a memory access [29,34].

2.2 Guaranteed Percentage Scheduling

The priority manager implements several real-time scheduling schemes [9,18,34]. For the concern of controlling, only GP scheduling is important. In GP scheduling, the priority manager assigns a requested number of clock cycles to each thread. This assignment is done within a 100 clock cycle period. Figure (2) gives an example of two threads with 20% and 80%, i.e. thread A gets 20 clock cycles and thread B gets 80 clock cycles within the 100 clock cycle interval. Of course, these clock cycles may contain latencies. So thread A might not be able to execute 20 instructions in its 20 clock cycles and this is where our approach starts. By monitoring the real IPC rate, the GP value is adjusted in a closed feedback loop. If there are e.g. 3 latency clock cycles within the 20 clock cycles of thread A, its percentage needs to be adjusted to achieve the desired 20 instructions in the 100 clock cycle interval.

2.3 Measurement of the IPC Rate

In general, the IPC rate of a thread is defined as:

$$IPC = \frac{Number\ of\ instructions\ executed\ within\ time\ slice}{Duration\ of\ time\ slice\ (in\ clock\ cycles)} \tag{1}$$

Since Komodo is single-scalar, each instruction takes one clock cycle, unless there is a latency. If we distinguish between active clock cycles, where an instruction of a thread is really executed, and latency clock cycles, we can refine the definition of the IPC rate for the Komodo microcontroller:

$$IPC = \frac{Number\ of\ active\ clock\ cycles\ within\ time\ slice}{Duration\ of\ time\ slice\ (in\ clock\ cycles)} \tag{2}$$

The number of active clock cycles of a thread is the number of clock cycles executed by a thread without the clock cycles used by latencies. It is obvious that both latencies and locks interfere with the IPC rate. Therefore, the function of the controller is to minimize these interferences with the IPC rate.

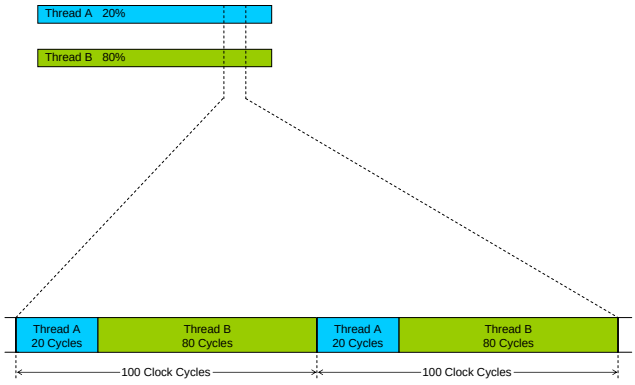

Fig. 2. Example for GP scheduling

We use this formula for different kinds of measures: We compute short- and long-term IPC rates. Short-term IPC rates are measured over a constant time slice of e.g. 400 clock cycles. The cumulative IPC rate is used for the longtime IPC rate. We compute the IPC rate from the beginning of the thread execution up to now, the duration of the time slice is increasing. The short-term IPC rate gives information about variations in the current IPC rate, the cumulative IPC rate shows the overall behavior.

2.4 Use of a PID Controller in a Microprocessor

In system design, the relationship of input and output values of a system is of great importance. Often, this relationship is not exactly known or influenced by unknown disturbances. Here, control theory comes into play. The system input is controlled by observing the output and comparing it with the desired values in order to minimize their difference by varying the input [12,20]. The controller in this closed feedback loop is responsible for generating the control signal y_n from the difference signal x_n. A well known and popular controller is the PID controller. The functional equation for a discrete PID controller is as follows:

$$y_n = K_P * x_n + K_I * \sum_{\nu=1}^{n} x_\nu * \Delta t + K_D * \frac{x_n - x_{n-1}}{\Delta t} \qquad \text{n} = 2, 3, 4... \quad (3)$$

Hereby x_n is the difference at time n and y_n is the controller signal at time n. K_P, K_I and K_D are the controller constants while Δt is the duration between the measurement of x_n and x_{n+1}.

The PID controller works in the processor as is shown in Figure 3. The current IPC rate is computed by a measurement unit. The PID controller compares this value with

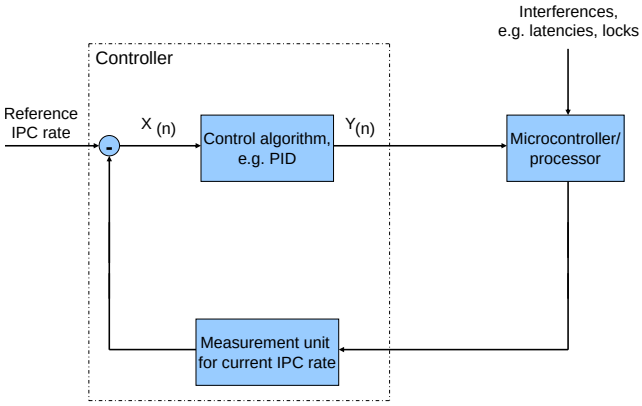

Fig. 3. Use of a controller to stabilize the throughput

the reference IPC value by computing their difference. Then it uses the PID algorithm from equation (3) to compute the new GP value for a thread to be controlled.

As shown in the related work section 4 the control approach as described above works really well. Therefore, it is important to evaluate the size of the controller relative to the size of the microprocessor to estimate the practicability of the use of such a controller in a FPGA or even an ASIC implementation; this work is described in the next section.

3 Evaluation of the Processor Size

3.1 Preliminary Work for the PID Controller Implementation in VHDL

Previous work on a PID controller for IPC rate stabilization was already carried out via a software processor simulator, based on the Komodo architecture [6,5]. The simulator was written in Java code. This allowed easy and quick changes to check different configurations of the PID controller. As a result, tests values for the following essential parameters for the succeeding VHDL implementation have been determined.

- The values of K_P, K_I and K_D depend on the current running applications and the kind of appearing latencies. It was not part of this work to find proper strategies to select these constants. So values of prior simulation runs were taken for the PID controller's VHDL implementation.
- The PID controller has to generate the new signal every 100 cycles. The current IPC rate is also sampled every 100 cycles.
- For stable controller operation, the IPC rate should be calculated as an average value over a longer period of time. However, if this period gets too long, the controller will react lazy. Simulation results have shown that 1000 clock cycles (= 10 controller cycles) are a very good value as a length of the controlling window.

These boundaries define the structure and required space for the VHDL design.
Another approach to the previous work on the simulator was the reuse of the GP scheduler. The PID controller signal y_n directs the IPC rate via this scheduler. Therefore, the period which the PID controller needs to generate a new signal is the same as the GP scheduling window. So, the PID controller simply regulates the allocated GP value within a range of 0 to 100 to reach the desired IPC rate. The former input of the GP scheduler is used as a set value for the PID controller and the signal y_n forms the new input of the scheduler.

3.2 Implementation of the PID Controller in VHDL

The PID controller has been integrated in an enhanced Komodo IP core. The GP scheduler, which was already implemented in the IP core, has a scheduling window of 100 cycles. This dictates the rate of the PID controller. The upcoming operations of one PID controlling window of 100 cycles are:

- Calculating the average difference value x_n for the last 1000 cycles. This period corresponds to the last 10 GP scheduling windows (see figure 4). The overlap between the current and the prior PID controlling windows consist of the previous 9 GP scheduling windows. Regarding this overlap the new value of x_n can be calculated by modifying the average difference value of the prior PID controlling windows x_{n-1}. Therefore the difference value of 10th oldest GP scheduling window has to be subtracted and the difference value of the current GP scheduling window has to be added to x_{n-1}.
  So, the history containing the differences between actual and set values of the last 10 GP scheduling windows has to be stored.

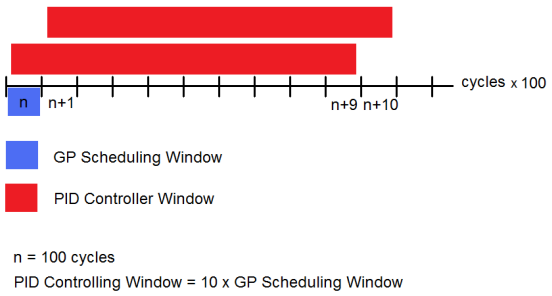

Fig. 4. Relation of PID Controlling Window and GP Scheduling Window

- To calculate the P component of the PID controller equation, $K_P * x_n$, one multiplication has to be carried out.
- To calculate the D component of the PID controller equation, $K_D * \frac{x_n - x_{n-1}}{\Delta t}$, one multiplication of the constant value $\frac{K_D}{\Delta t}$ by the result of $x_n - x_{n-1}$ has to be carries

out. Regarding the way x_n was calculated, the difference of $x_n - x_{n-1}$ can be calculated by simply subtracting the difference value of the 10th oldest GP scheduling window from the value of the current GP scheduling window.
- To calculate the I component of the PID controller equation, $K_I * \sum_{\nu=1}^{n} x_\nu * \Delta t$, one multiplication of the constant value of $(K_I * \Delta t)$ by the result of $\sum_{\nu=1}^{n} x_\nu$ has to be carried out. This sum is calculated recursively by adding the current x_n to the sum $\sum_{\nu=1}^{n-1} x_\nu$ calculated in the PID controlling window before.

The implementation of the PID controller on an Altera Cyclon II FPGA board showed the following result: The required "dedicated logic register" increased by 3 percent and the number of "combinational functions" by 1.4 percent. The number of total logic elements increased by 2 percent overall.

IP Core:	without PID Controller	with PID Controller
Dedicated Logic Registers	4688	4829
Total Combinational Functions	12696	12873
Total Logic Elements	12895	13147
Space used on the Cyclon II FPGA	39%	40%

Table 1. Results of the PID Controller Space Evaluation

This is very promising result gives reason to spend more research on space requirements for various controller and thread settings.

4 Related Work

Hardware multithreading has made its way from pure research to a state-of-the-art processor technique.
Early examples for multithreaded processors are Cray MTA (a high performance multithreaded VLIW processor [2]) and Cray MTA2 (supports up to 128 RISC-like hardware threads [11]). The MIT Sparcle processor is a classical example based on the SPARC RISC processor supporting up to 4 threads and context switching on long memory latencies [1]. The MSparc processor supporting up to 4 threads [24] and real-time properties [19,23] can be cited in this context as well.
In the embedded field, multithreading has been introduced by the two-threaded Infineon TriCore II [27] or the MIPS32-34K [25] with up to nine thread contexts. Research-oriented small processors like the Komodo microcontroller [17,16] with a scalar four-threaded Java processor core and the CAR-SoC [15,26] with a SMT processor core investigate the use of multithreading for real-time embedded applications. Real-time scheduling for SMT processors is also researched in [10]. Here, a resource allocator is used to control shared resources (e.g. registers, issue bandwidth, buffers, etc.) of threads to control the throughput. Another multithreaded processor featuring real-time scheduling is the Jamuth IP core [35], which is an enhancement of the Komodo microcontroller.

Furthermore, multithreading is often combined with *multi-core architectures*. Multi-core processors combine several processor cores on a single chip. In many cases the core is a multithreaded core.
Commercial examples for multi-core multithreaded processors are e.g the Sun Niagara (Sun UltraSPARC T1 [31]) with up to eight four-threaded processor cores or the IBM Power series. IBM Power5 and Power6 [13,14] contain 2 SMT processor cores with shared secondary level cache supporting dynamic resource scheduling for threads [3]. Furthermore, multi-chip modules containing several processor and third level cache chips are available. The upcoming Power7 architecture is announced to contain up to eight processor cores [30]. Intel processors like the Core 2 Duo, Core 2 Quad or their successors Core i3, i5, i7 and i9 are also equipped with multiple cores supporting multithreading. The Intel variant of multithreading is called hyperthreading [21].
Research multi-core processors are e.g. MIT RAW with 16 MIPS-style cores and a scalable instruction set [32] or AsAP (*As*ynchronous *A*rray of simple *P*rocessors) containing 36 (AsAP 1 [36]) or 167 (AsAP 2 [33]) cores dedicated to signal processing. Researching real-time support on such processors, the MERASA project has to be cited [22]. This project combines execution time analysis like worst case execution time or longest execution time with multi-core and SMT microarchitectures for hard real-time systems.
Our approach is completely different from the methods mentioned above where no control theory is used. Some of the approaches mentioned above (e.g. [10,3,13,14]) support dynamic adaptation of thread parameters and resources. However, according to our best knowledge our approach is the only one using control theory as a basis. The improvements achieved by this approach have been shown in several publications [6,5,8,4]. As a major advantage, mathematical models of control theory can be used to guarantee real-time boundaries. Furthermore, we tested the parameters of the PID controller's algorithm on a simulator of the Komodo microcontroller [7].

5 Conclusion and Further Work

Controlling the throughput of a thread by a PID controller has already been successfully tested in the preceding work via a software simulator. In this paper the focus is on the additional required space for the PID controller. Therefore the parameters determined by the simulator were used to build and configure a VHDL model of the controller. As a result of the evaluation, only 2 percent of additional overhead has been created.
An interesting future approach will be to test the configurations in the VHDL model under various thread settings. Depending on the results obtained, optimizations of the VHDL code could reduce the required space. Many parameters like the execution period of the PID controller, the period of calculating the average IPC rate and the history size to calculate the integral part of the PID controller influence the size of the controller.
Also the combination of different types of controlling strategies for threads is of interest with respect to accuracy, speed and real time constraints. The use of a combination of controlling strategies regarding different thread scenarios is an interesting future approach.

References

1. Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B.H., Yeoung, D., D'Souza, G., Parkin, M.: Sparcle: An Evolutionary Processor Design for Large-scale Multiprocessors. IEEE Micro Vol. 13, 3 pp. 48–61 (1993)
2. Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., Smith, B.J.: The Tera Computer System. In: 4th International Conference on Supercomputing. Amsterdam, The Netherlands (1990)
3. Borkenhagen, J.M., Eickemeyer, R.J., Kalla, R.N., Kunkel, S.R.: A Multithreaded PowerPC Processor for Commercial Servers. IBM Journal Research and Development, Vol. 44 pp. 885–898 (2000)
4. Brinkschulte, U., Lohn, D., Pacher, M.: Towards a statistical model of a microprocessor's throughput by analyzing pipeline stalls. In: SEUS. pp. 82–90 (2009)
5. Brinkschulte, U., Pacher, M.: Implementing control algorithms within a multithreaded java microcontroller. In: ARCS. pp. 33–49 (2005)
6. Brinkschulte, U., Pacher, M.: Improving the real-time behaviour of a multithreaded java microcontroller by control theory and model based latency prediction. In: WORDS. pp. 82–96 (2005)
7. Brinkschulte, U., Pacher, M.: Improving the Real-time Behaviour of a Multithreaded Java Microcontroller by Control Theory and Model Based Latency Prediction. WORDS 2005 , Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems, Sedona, Arizona, USA (2005)
8. Brinkschulte, U., Pacher, M.: A control theory approach to improve the real-time capability of multi-threaded microprocessors. In: ISORC. pp. 399–404 (2008)
9. Brinkschulte, U., Ungerer, T.: Mikrocontroller und Mikroprozessoren, vol. 2. Springer-Verlag (2007)
10. Cazorla, F., Fernandez, E., Knijnenburg, P., Ramirez, A., Sakellariou, R., Valero, M.: Architectural Support for Real-Time Task Scheduling in SMT-Processors. In: CASES'05. San Francisco, California, USA (September 2004)
11. Cray: Ceay MTA-2 System (2001), `http://www.netlib.org/utk/papers/advanced-computers/mta-2.html`
12. Dorf, R., Bishop, R.: Modern Control Systems. Addison-Wesley (2002)
13. IBM: Power6 Specifications (2007), `http://www-03.ibm.com/press/us/en/attachment/21546.wss?fileId=ATTACH_FILE1&fileName=POWER6%20Specs.pdf`
14. IBM: IBM POWER systems (2009), `http://www-03.ibm.com/systems/power/`
15. Kluge, F., Mische, J., Uhrig, S., Ungerer, T.: CAR-SoC - Towards and Autonomic SoC Node. In: Second International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2006). L'Aquila, Italy (July 2006)
16. Kreuzinger, J., Brinkschulte, U., Pfeffer, M., Uhrig, S., Ungerer, T.: Real-time Event-handling and Scheduling on a Multithreaded Java Microcontroller. Microprocessors and Microsystems Journal, 27, 1, Elsevier pp. 19–31 (2003)
17. Kreuzinger, J., Pfeffer, M., Ungerer, T., Brinkschulte, U., Krakowski, C.: The Komodo Project: Real-Time Java Based on a Multithreaded Java Microcontroller. In: PDPTA 2000 International Conference on Parallel and Distributed Processing Techniques and Applications. Las Vegas, USA (2000)
18. Kreuzinger, J.: Echtzeitfähige Ereignisbehandlung mit Hilfe eines mehrfädigen Java-Mikrocontrollers. Ph.D. thesis, Logos Verlag Berlin (2001)
19. Lüth, K., Metzner, A., Piekenkamp, T., Risu, J.: The EVENTS Approach to Rapid Prototyping for Embedded Control System. In: Workshop Zielarchitekturen eingebetteter Systeme. pp. 45–54. Rostock, Germany (1997)

20. Lutz, H., Wendt, W.: Taschenbuch der Regelungstechnik. Verlag Harri Deutsch (2002)
21. Marr, D., Binns, F., Hill, D., Hinton, G., Koufaty, D., Miller, J., Upton, M.: Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History. Intel Technology Journal Vol. 6 (2002)
22. MERASA: Multi-Core Execution of Hard Real-Time Applications Supporting Analysability (2009), http://ginkgo.informatik.uni-augsburg.de/merasa-web/
23. Metzner, A., Niehaus, J.: MSparc: Multithreading in Real-time Architectures. J. Universal Comput. Sci. Vol. 6, 10 pp. 1034–1051 (2000)
24. Mikschl, A., Damm, W.: MSparc: a Multithreaded Sparc. Lecture Notes in Computer Science, Vol. 1123, Springer-Verlag, Heidelberg pp. 461–469 (1996)
25. MIPS Technologies, Inc: The MIPS32-34K Core Family: Powering Next-Generation Embedded SoCs. Research Report pp. 1034–1051 (September 2005)
26. Nickschas, M., Brinkschulte, U.: Guiding Organic Management in a Service-Oriented Real-Time Middleware Architecture. In: 6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2008). Capri, Italy (October 2008)
27. Norden, E.: A Multithreaded RISC/DSP Processor with High Speed Interconnect. Hot Chips 15 (2003)
28. Pfeffer, M., Ungerer, T., Uhrig, S., Brinkschulte, U.: Connecting peripherial interfaces to a multi-threaded java microcontroller. Workshop on java in embedded systems, ARCS 2002 (2002)
29. Pfeffer, M.: Ein echtzeitfähiges Javasystem für einen mehrfädigen Java-Mikrocontroller. Ph.D. thesis, Logos Verlag Berlin (2004)
30. Stokes, J.: IBM's 8-core POWER7: twice the muscle, half the transistors. Ars Technica (September 2009), http://arstechnica.com/hardware/news/2009/09/ibms-8-core-power7-twice-the-muscle-half-the-transistors.ars
31. Sun: UltraSPARC T1 Processor (2005), http://www.sun.com/processors/UltraSPARC-T1/
32. Taylor, M., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, J., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., Agarwal, A.: The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. IEEE Micro (March/April 2002)
33. Truong, D.N., Cheng, W.H., Mohsenin, T., Yu, Z., Jacobson, A.T., Landge, G., Meeuwsen, M.J., Tran, A.T., Xiao, Z., Work, E.W., Webb, J.W., Mejia, P., Baas, B.M.: A 167-processor computational platform in 65 nm cmos. IEEE Journal of Solid-State Circuits (JSSC) 44(4), 1130–1144 (Apr 2009)
34. Uhrig, S., Liemke, C., Pfeffer, M., Becker, J., Brinkschulte, U., Ungerer, T.: Implementing real-time scheduling within a multithreaded java microcontroller. 6th Workshop on Multithreaded Execution, Architecture, and Compilation MTEAC-6, Istanbul, Nov. 2002, in conjunction with 35th International Symposium on Microarchitecture MICRO-35 (2002)
35. Uhrig, S., Wiese, J.: jamuth: an ip processor core for embedded java real-time systems. In: JTRES. pp. 230–237 (2007)
36. Yu, Z., Meeuwsen, M., Apperson, R., Sattari, O., Lai, M., Webb, J., Work, E., Mohsenin, T., Singh, M., Baas, B.M.: An asynchronous array of simple processors for dsp applications. In: IEEE International Solid-State Circuits Conference, (ISSCC '06). pp. 428–429 (Feb 2006)