Improving Performance of Multithreaded Scalar Architectures for Embedded Microcontrollers

Horia V. Căpriţă
Department of Computer and Electronic Engineering
"Lucian Blaga" University of Sibiu
Sibiu, Romania
e-mail: horia.caprita@ulbsibiu.ro

Mircea Popa
Faculty of Automation and Computers
"Politehnica" University of Timişoara
Timişoara, Romania
e-mail: mircea.popa@ac.upt.ro

Ioan Z. Mihu
Department of Computer and Electronic Engineering
"Lucian Blaga" University of Sibiu
Sibiu, Romania
e-mail: ioan.z.mihu@ulbsibiu.ro

Abstract – **The primary aim followed in the development of computing systems is increasing the overall performance. The market always requires faster, more efficient and powerful products regardless of the applications that are used: high-end applications, telecommunications, automotive, low-power embedded applications, etc. Regardless of the type of application processed, the products based on simple single core systems have already shown their limitations in obtaining the desired performance. Multicore processing is currently a way to improve the performance of a computing system. Multicore devices have become ubiquitous in everyday life and are used in all areas. In this paper we present and evaluate an interleaved multithreaded scalar architecture having limited resources which supports hardware scouting technique. We will show that the implementation of hardware scouting is viable and efficient on scalar multithreaded systems, systems that have the advantage that use limited hardware resources for efficient processing. This scalar architecture will be used in our future research as a basic processing element (Base Core Equivalent) in developing of new multicore microcontrollers that will be efficient in terms of energy consumption and processing rate.**

Keywords–multithreaded architecture; multicore microcontroller; embedded systems; energy-efficient systems.

I. INTRODUCTION

The nowadays diversity of the end-user equipments that rely on a processing unit (e.g., personal computers, mobile or automotive devices) was made possible by the integration of architectural innovative solutions. The software applications for these devices require more and more computing power regardless of hardware platform that are used. As a result, more and more companies have adopted multicore technology in order to develop more efficient processors, leading to the development of more efficient end-user equipment. This is a life cycle that could be maintained by developing new architectural types to support the next generation of software applications.

The Amdahl law shows us that the fraction of sequential code within the program limits the performance of parallel machines [1] [9]. Reducing this negative effect due to portion of sequential code of a program can be done by improving the core's performance which can affect the parallel execution performance [9]. Despite this, improvement of overall performance can be done using multicore systems. The performance of a multicore system is $\sqrt{n}$, where n is the number of cores [2].

In embedded systems design it applies the same laws used in designing of general purpose systems. Fulminant development of embedded applications which demand massive calculations, led to the concept of high-performance embedded computing (HPEC) [10]. Moreover, embedded applications must comply with more stringent rules than HPC applications on supercomputers. The efficiency of energy of embedded applications is the first criterion used in evaluation. This efficiency can be achieved in different ways: by reducing the operating frequency, by implementing dynamic reconfigurable architectures like drowsy cache [7] [10], or by creating multicore processors based on simple processing elements having limited resources [5]. These techniques can be combined to obtain an efficient processor in terms of energy consumption and performance. These designing constraints lead to the development of energy-efficient high performance embedded computing applications (HPEEC) [10] in which, unlike the field of supercomputers (HPC), not only the raw performance is important, but the amount of energy consumed to achieve this performance.

The profiles of nowadays embedded applications are getting closer to those of general purpose applications. These applications can claim hardware capability to provide support for massive calculations (e.g., multimedia applications), can be parallel or distributed, can have realtime features (working with an RTOS), must be reliable (reliably constrained) and last, but not least, must be energy efficient.

Multicore embedded systems represent a solution to implement HPEEC applications. By combining a multicore system with multithreaded technology we can achieve more efficient systems in terms of processing rate and the energy consumption [6] [10].

In this paper, we will introduce a multithreaded scalar architecture that uses the hardware scouting technique [6]. This scalar architecture could be used as a basic processing element (Base Core Equivalent - BCE) for developing multicore microcontrollers that are efficient in terms of energy consumption and processing rate.

In Section II, we will present a short state of the art of the multithreaded architectures. Section III explains the principles of our proposed multithreaded architecture. In Section IV and V, we will present the results and the conclusions of our research.

II. Related Work

Multithreaded architectures (MT) allow execution of instructions fetched from multiple threads at a time. This paradigm is based on resource sharing when more threads compete for these resources. This thread overlapping influences in a positive way the overall processing performance [15]. Threads competition also leads to unfair resources sharing between them which can affect the processing rate of a single thread at a time. A worst case occurs when we run threads with a low hit rate in cache. Such a thread can block the reorder buffer because the instructions that follow after a data cache miss event will not be issued or will not be able to complete due the stalls imposed by the memory operations with high latencies. Moreover, there may be critical resources assigned to this thread, leading to "starvation" and stalling of other independent threads of current thread. The thread stalling during a memory access limits the exploitation of memory level parallelism (MLP). As a result, the overall performance of multithreaded processor may be affected.

The literature contains references to methods that try to reduce these negative effects. Hardware scouting [6] consists of launching a hardware thread (invisible in software) that runs in front of the main application thread. The main role of this thread is to bring data and instructions, which are necessary for the execution, in internal caches. A "load miss" event will start this hardware thread. In this case, the multithreaded architecture will create a checkpoint with the current state of the thread and then will continue to execute instructions that follow after the load, until the requested data is brought from system memory. At this point, the processor will restore the program state based on the last saved checkpoint and will rerun instructions that follow after the load using new conditions. The advantage of this method is that the instructions re-launched in the second step will be already available in the primary caches because were extracted from memory by hardware thread. Chaudry *et al.* [6] show an increase of 40% of the CPU performance when using an L2 cache of 512 KB. Using the hardware thread can be considered as a sophisticated mechanism of prefetching.

Another version of hardware scouting is presented in [12]. Ramirez *et al.* propose a method to exploit the memory level parallelism (MLP) to increase the performance of Simultaneous Multithreaded processors (SMT): Run-ahead Threads (RaT). Run-ahead execution is a method in which data and instructions are speculatively mapped in caches [11]. Ramirez *et al.* have developed the RaT method which is a strategy of fetching used to increase the performance of memory-bound threads without affecting the quantity of instruction level parallelism exploited in those threads. This method is applied to threads that could stall due to high latency memory operations. The appearance of a high-latency load instruction determines the owner thread to become a runner-ahead thread. This thread will become speculative and will use for a short time some hardware resources, so that other threads will not be limited to getting access to the CPU. At the same time, the prefetching operations from other threads will increase the degree of memory level parallelism, too. The SMT model used in simulations allows resource sharing [12]. More threads coexist in the pipeline and share structures like instruction queues, reorder buffer, physical registers, functional units and caches. This method has the advantage that will increase the performance of a simultaneous multithreaded processor by speculating load-miss events coming from different threads.

In our previous paper [4] we showed that the multithreaded model can be effective when it is implemented on a scalar processor. Our model was inspired by the models presented in [13] and [14]. It was adapted to be used in embedded multicore systems [5] in which energy savings can be made by simplifying the underlying architectural model of a BCE's.

This research focuses on scalar multithreaded architectures (having limited resources) that are capable to adapt the hardware scouting method used by others on superscalar multithreaded processors [6]. We will show that the implementation of this technology is viable and efficient on multithreaded scalar systems, systems that have the advantage that use limited hardware resources for efficient processing.

III. HSSS-IMT Multithreaded Scalar Architecture and Hardware Scouting

In this paper, we present and evaluate an interleaved multithreaded scalar architecture having limited resources which supports hardware scouting technique (HSSS-IMT architecture) [6]. This scalar architecture can be used as a basic processing element (Base Core Equivalent - BCE) in multicore microcontroller's development process; this basic processor could be efficient in terms of energy consumption and processing rate.

The HSSS-IMT architecture is based on the SS - IMT architecture presented in [4]. SS-IMT is a multithreaded architecture based on a scalar processor [3]. SS-IMT is modified to interleave instructions that are coming from different threads in order to execute them using the same pipeline.

Interleaved Multithreading technique (IMT) is often called fine-grain multithreading [15]. The processor switches to another thread after each instruction fetch (Fig. 1). An instruction feeds the pipeline after the previous statement issued. IMT eliminates hazards control and data dependencies between instructions. Memory latencies are hidden by the scheduler. The thread that has generated a latency of memory will be stalled by the scheduler; the interleaving through the pipeline will continue just for instructions that becomes from other threads (Fig. 2). The instructions on the stalled thread will be scheduled again for execution when the memory transaction was done.

One way to increase the performance of this architecture is the hardware scouting [6]. The study presented in this paper is focused on hardware implementation and adaptation of hardware scouting technique to the scalar multithreaded architecture SS-IMT. When a load-miss (long latency) event occur, the new architecture, called Hardware Scouting SS-IMT (HSSS-IMT), continues to use the fetch algorithm that applies Round-Robin on all available threads, including the thread that generated the long-latency event.

Moreover, when a long - latency event occurs, HSSS-IMT will create a checkpoint that will contain the status of all architectural registers of the thread that generated the memory latency. Unlike SS-IMT, HSSS-IMT processor will not block the fetching of instructions belonging to the thread which generates the latency. These instructions (following load-miss instruction) become pseudo-executed instructions and will continue to be scheduled for execution under Round-Robin algorithm (Fig. 3). Thus, in the HSSS-IMT processor there are no stalled threads.

When a memory transaction ends, the instructions which follow after the Load and were pseudo-executed will be flushed from pipeline, while the thread context will be restored from the last saved checkpoint. Check pointing mechanism is implemented in "Check pointing and Flush" block (Fig. 4). Starting from now, the instructions of the thread that generated the long latency event will be rerun through all phases of the instruction cycle (instruction fetch, decode, execute, write back) and using the right context. The advantage of the hardware scouting method will be that, this time, the rescheduled instructions have been already in the instruction cache and their operands could be already loaded into the data cache. Such an instruction will be executed with maximum speed allowed by this architecture.

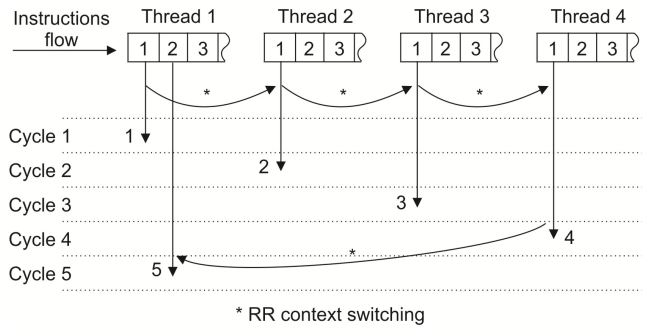

Figure 1. Round Robin context switching in SS-IMT.

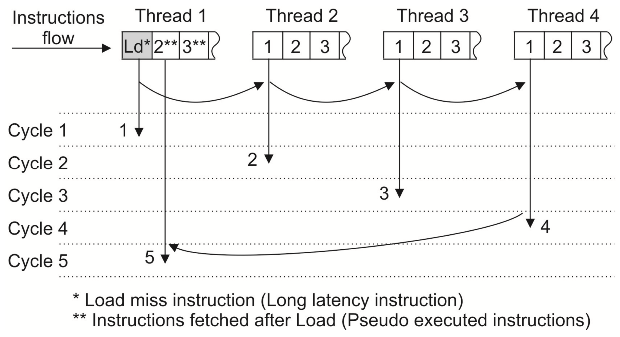
Instructions flow
Thread 1
Ld*
2**
3**
Thread 2
Thread 3
Thread 4
Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
* Load miss instruction (Long latency instruction)
** Instructions fetched after Load (Pseudo executed instructions)

Figure 3. HSSS-IMT: when a load misses, the next instructions will be fetched, and a new checkpoint is created.

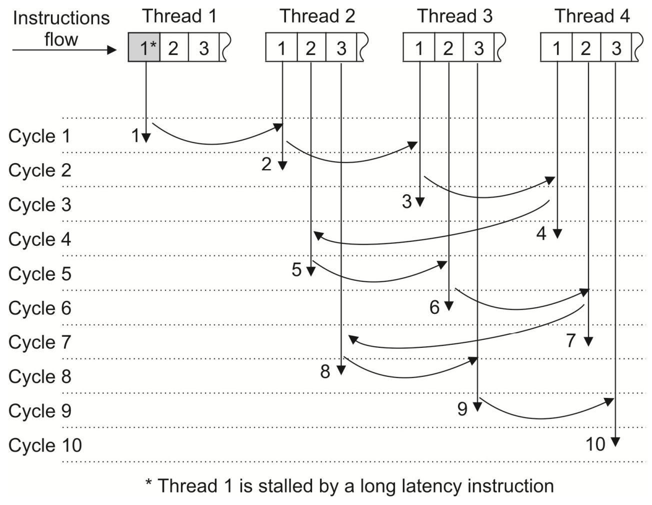

Figure 2. Stalled thread in SS-IMT.

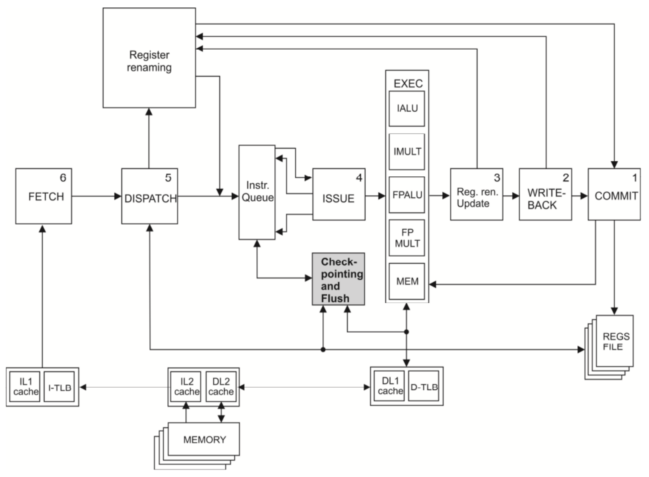

Figure 4. HSSS-IMT architecture.

IV. Results

Being a multithreaded environment, we used benchmarks as threads for our HSSS-IMT evaluation. By porting SPEC CPU2000 [8] suite we obtained 7 integer benchmarks and 9 floating point benchmarks that we have used in simulations (Table I). These benchmarks represent the threads concurrently processed in our architectures (e.g., the evaluation of 8-contexts HSSS-IMT processor was done using 8 concurrent benchmarks as inputs).

TABLE I. SPEC CPU2000 BENCHMARKS PORTED TO HSSS-IMT

SPEC CPU2000 Benchmark type	Benchmark name
Integer	bzip2.ss, gcc.ss, gzip.ss, mcf.ss, parser.ss, twolf.ss, vortex.ss
Floating point	ammp.ss, applu.ss, apsi.ss, art.ss, equake.ss, mesa.ss, mgrid.ss, swim.ss, wupwise.ss

We created some groups of benchmarks named “thread sets”. For example, Gr-2TH-1 thread set contains bzip2.ss and ammp.ss benchmarks, while Gr-8Th-1 thread set contains bzip2.ss, gcc.ss, gzip.ss, mcf.ss, ammp.ss, applu.ss, apsi.ss and art.ss benchmarks. We defined three versions of HSSS-IMT architectures: two hardware contexts (2 threads), four hardware contexts (4 threads) end eight hardware contexts (8 threads). We extended the number of sets for D-cache and I-cache accordingly with the number of threads that we’ve used in simulations. We evaluated the performance of HSS-IMT related to the performance of the basic architecture SS-IMT. Each result was obtained running 100 millions of instructions/benchmark on SS-IMT and HSSS-IMT architectures.

In [4], we proposed a multithreaded model based on Simple Scalar (SS) architecture [3]. Our model, named Simple Scalar Interleaved Multithreaded architecture (SS-IMT), was inspired by multithreaded architectures presented in [13] and [14]. It was adapted to be used in embedded multicore systems [5]. In Figure 5 and Figure 6 we depict the average performance of the SS-IMT architectures with 2, 4 and 8 contexts (threads). Each average value has been obtained using instruction and data caches with 32, 64, 128 and 256 KB per context.

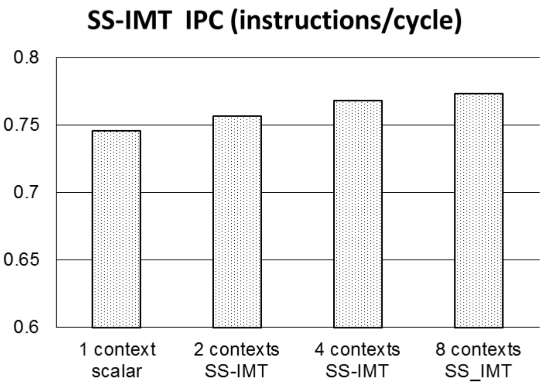

Figure 5. The overall performance (IPC) of SS-IMT architectures [4].

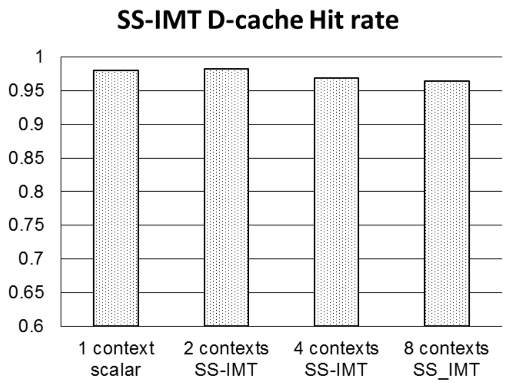

Figure 6. The overall performance (D-cache hit rate) of SS-IMT architectures [4].

As we can see from the previous figures, we note that IPC is growing together with the number of contexts (threads), while D-cache performance decreases. This decrease is due to the number of threads increasing, threads which concurrently coexist in D-cache [4].

HSSS-IMT evaluation has been done using data caches having 32, 64, 128 and 256 KB per thread. In Figure 7 we represent the performance in terms of IPC for HSSS-IMT architecture with 2 hardware contexts. For evaluation we have used 18 groups of 2 benchmarks. These groups contain benchmarks that have an integer or floating point profile or could contain benchmarks from these two categories. It could be observed that the maximum average performance is obtained using the thread set number 2 (0.8361804 IPC), while the worst average performance is given by the thread set 11 (0.784226 IPC). The difference between these two values is given by the content of the thread set (the profiles of the SPEC CPU2000 benchmarks). The average value of the performance for this HSSS-IMT configuration is 0.8130545 IPC.

Figure 8 depicts the HSSS-IMT performance having 4 hardware contexts. For evaluation we have used 22 groups of 4 benchmarks. Like in previous case these groups contain mixed benchmarks and the performance grows together with the dimension of caches.

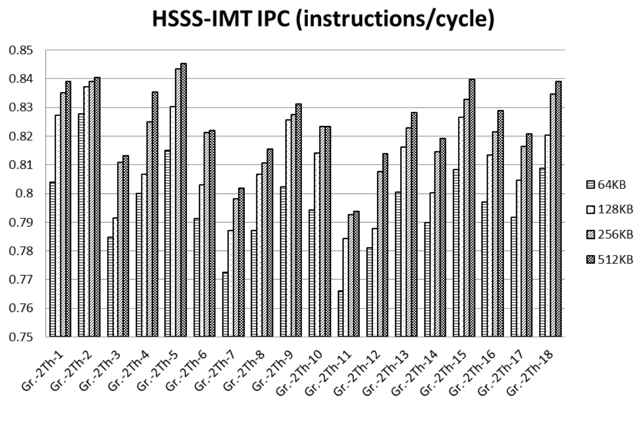

Figure 7. The performance of 2-contexts HSSS-IMT architecture.

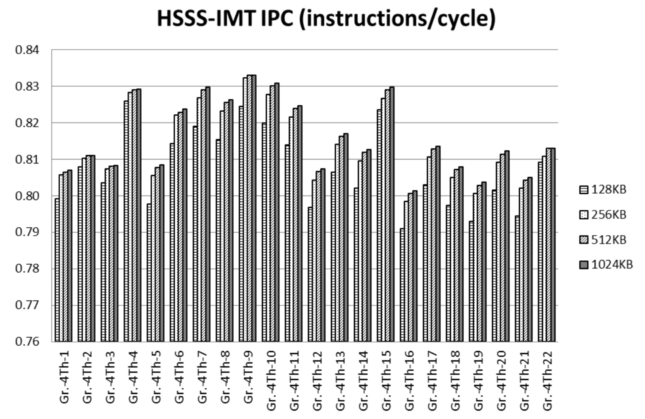

Figure 8. The performance 4-contexts HSSS-IMT architecture.

It can observe that the maximum average performance is obtained using the thread set number 9 (0.8492511 IPC), while the worst average performance is given by the thread set 19 (0.8001177 IPC). The average value of the performance for this HSSS-IMT configuration is 0.817238679 IPC.

In Figure 9, we represent the performance of HSSS-IMT architecture having 8 hardware contexts obtained with a worst case combination of benchmarks. The average value of the performance for this HSSS-IMT configuration is 0.820006152 IPC.

By comparing these results we can see that the performance of HSSS-IMT architecture is superior to the performance of SS-IMT regardless of the number of hardware contexts that are implemented.

From Figure 10, we can observe that while the number of HSSS-IMT contexts is growing, the average performance of this architecture is growing too. The global performance of HSSS-IMT architecture (0.816766446 IPC) is greater than the global performance of the SS-IMT architecture (0.766081157 IPC). In Figure 11 we represent the data cache hit rate for SS-IMT and HSSS-IMT architectures. While the number of contexts is growing we obtain a better performance that is in opposite to the results obtained on SS-IMT (Fig. 6).

This additional performance comes from the implementation of the hardware scouting technique that ameliorates the negative effect of a load-miss event [6]. The number of scouting instructions after a load-miss event will vary and depends on the memory latency.

The global performance of HSSS-IMT architecture related to the SS-IMT performance is depicted in Figure 12 and Figure 13.

Figure 12 represents a synthesis of Figure 5 and Figure 10 and depicts the IPC parameter of SS-IMT and HSSS-IMT architectures. Also, Figure 13 represents a synthesis of Figure 6 and Figure 11 and depicts the hit rate of data caches for these two architectures. The best results are obtained when we increase the number of the hardware contexts (number of threads). Average performance of this new architecture (approx. 0.816 IPC) is greater than average performance of SS-IMT (approx. 0.766 IPC) with approximately 5%. Data cache hit rate grows from approximately 0.97% in SS-IMT to 0.99% in HSSS-IMT.

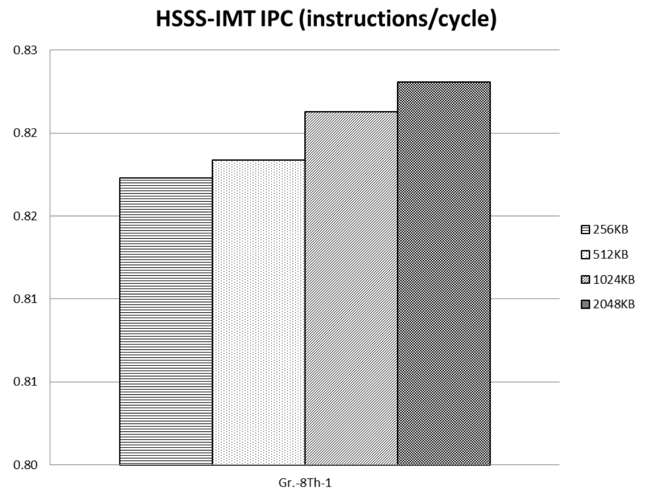

Figure 9. The performance 8-contexts HSSS-IMT architecture.

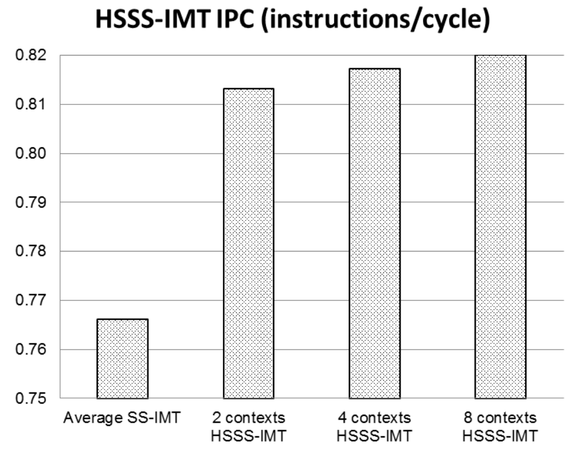

Figure 10. The overall performance (IPC) of HSSS-IMT architecture.

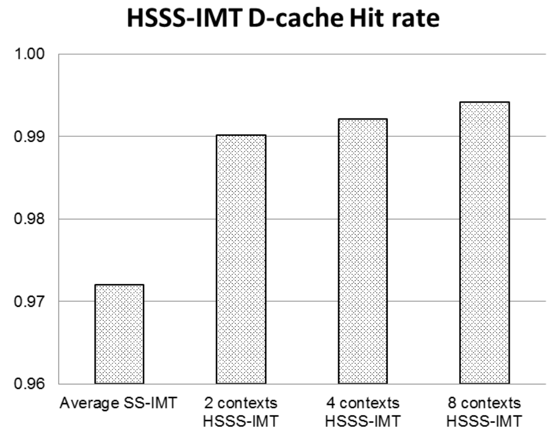

Figure 11. The overall performance (D-cache hit rate) of HSSS-IMT architecture.

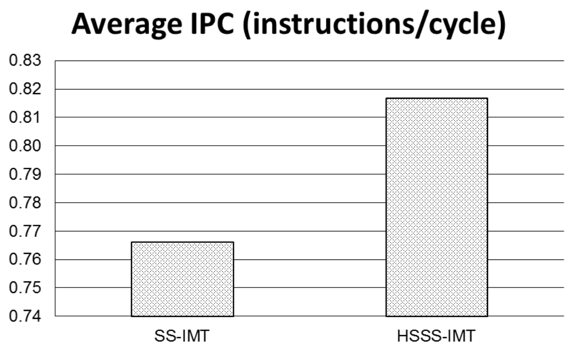

Figure 12. Average performance of HSSS-IMT vs. SS-IMT.

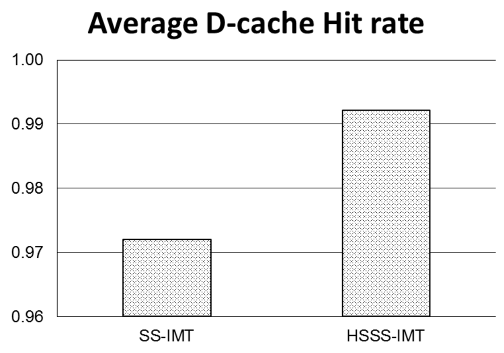

Figure 13. Average D-cache performance for HSSS-IMT vs. SS-IMT.

Implementing the hardware scouting method on our basic scalar multithreaded processor (SS-IMT) led us to a new multithreaded architecture (HSSS-IMT) that has a better performance than previous one presented in [4].

V. Conclusion And Future Work

HSSS-IMT architecture benefits of full of multithreading techniques and simultaneously of hardware scout. Moreover, this check pointing technique offers a better utilization of instruction and data cache memory by using hardware scouting. From Figure 11 we can see that average data cache hit rate of HSSS-IMT is greater with over 2% than that obtained on SS-IMT. This rising is very important for us. By implementing the multithreading technique on a basic scalar processor gave us a worse performance related to this data cache hit rate parameter [4] [5]. These performances of HSSS-IMT show us that there exists the possibility to use this kind of architectures to integrate it in bigger structures in order to create multicore processors. These multicore processors could be used in embedded systems because they contain limited resources (e.g., they have a scalar configuration) and they are able to manage more threads useful to run an RTOS.

Although investigating the power consumption of this architecture wasn't a purpose of this paper, we can anticipate that this type of architecture could be used to create low power energy BCEs due to the simplicity of the processing element [6] [11] [12].

The HPEEC domain should be sustained by developing novel transistor technologies in order to reduce the power consumption but, in the same time, the researchers have to concentrate on how to create new scheduling techniques, basic processing elements or revolutionary memory design.

Acknowledgment

This work was supported by Continental Automotive Systems Sibiu, Romania, grant no. 550/25.02.2010.

References

[1] G.M. Amdahl, "Validity of the single-processor approach for achieving large-scale computing capabilities", Proc. Am. Federation of Information Processing Societies Conf., AFIPS Press, 1967, pp. 483-485.

[2] S. Borkar, "Thousand core chips - a technology perspective", Proc. of the 44th annual Design Automation Conference (DAC 07), 2007, pp. 746-749.

[3] D. Burger and T.M. Austin, "The SimpleScalar tool set, version 2.0", ACM SIGARCH Computer Architecture News, vol. 25, Issue 3, June 1997, pp. 13-25.

[4] H.V. Căpriţă and M. Popa, "Design methods of multithreaded architectures for multicore microcontrollers", Proc. of 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI 2011), Timisoara, Romania, 2011, pp. 427-432.

[5] H.V. Căpriţă and M. Popa, "Multithreaded peripheral processor for a multicore embedded system", Applied Computational Intelligence in Engineering and Information Technology, Springer Berlin Heidelberg, 2012, pp. 201-212.

[6] S. Chaudry, P. Caprioli, S. Yip and M. Tremblay, "High performance throughput computing", IEEE Micro, vol. 25, Issue 3, May 2005, pp. 32-45.

[7] K. Flautner, N. Kim, S. Martin, D. Blaauw and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power", ISCA '02 Proc. of the 29th annual international symposium on Computer architecture, vol. 30, Issue 2, 2002, pp. 148-157.

[8] J.L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium", Journal of Computer, vol. 33, Issue 7, July 2000, pp. 28-35.

[9] M.D. Hill and R.M. Marty, "Amdahl's Law in the Multicore Era", Journal of Computer, vol. 41, Issue 7, July 2008, pp. 33-38.

[10] A. Munir, S. Ranka and A. Gordon-Ross, "High Performance Energy Efficient Multicore Embedded Computing", IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 4, 2012, pp. 684-700.

[11] O. Mutlu, J. Stark, C. Wilkerson and Y.N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors", Proc. of International Symposium on High-Performance Computer Architecture (HPCA 9), 2003, pp. 129-140.

[12] T. Ramirez, A. Pajuelo, O.J. Santana and M. Valero, "Runahead threads to improve SMT performance", Proc. of 14th International Conference on High-Performance Computer Architecture (HPCA 14), 2008, pp. 149-158.

[13] D.M. Tullsen and J.A. Brown, "Handling long-latency loads in a Simultaneous Multithreading processor", Proceedings of the 34th International Symposium on Microarchitecture, December 2001 (MICRO 34), pp. 318-327.

[14] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo and R.L. Stamm, "Exploiting choice: instruction fetch and issue on an implementable Simultaneous Multithreading processor", Proc. of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, 1996, pp. 191-202.

[15] T. Ungerer, B. Robic and J. Silc, "A survey of processors with explicit multithreading", ACM Computing Surveys, vol. 35, no. 1, March 2003, pp. 29-63.