A small footprint interleaved multithreaded processor for embedded systems

Charly Bechara, Aurelien Berhault, Nicolas Ventroux,
Stéphane Chevobbe, Yves Lhuillier, Raphaël David

CEA, LIST,
Embedded Computing Laboratory,
Gif-sur-Yvette, F-91191, FRANCE;
Email: charly.bechara@cea.fr

Daniel Etiemble

Université Paris Sud,
Laboratoire de Recherche en Informatique,
Orsay, F-91405, FRANCE;

***Abstract*—With the increase in the design complexity of MPSoC architectures and the need for more transistor/energy efficient processor architectures, designers are exploiting the parallelism at the thread level (TLP) through the implementation of embedded multithreaded processors. Moreover, future manycore architectures tend to use small footprint RISC cores. In this paper, we present a small footprint, scalar, in-order, 5-stage pipeline, interleaved multithreaded processor with 2 hardware thread contexts for embedded systems and SoC integration. Synthesis results in 40 nm TSMC shows that the multithreaded core area is only 19800 μm^2 and 13.97 kilogates, which is almost equal to a 4KB direct mapped cache memory according to CACTI 6.5 tool [1]. The IMT core has an augmentation of *73.2%* in core area compared to the monothreaded core. The multithreaded core is validated by running a simple bubble-sort application and varying the L1 D$ memory. The average performance gain is 17% compared to the monothreaded core.**

keywords: multithreaded processors, interleaved multithreading, System-on-Chip, embedded systems, RISC

I. Introduction

Traditional high-performance superscalar processors implement several architectural enhancement techniques such as out-of-order execution, branch prediction, and speculation, in order to exploit the instruction-level parallelism (ILP) of a single-thread sequential program. However, due to the limits of ILP [2], a more coarse-grained solution consists of exploiting the parallelism at the thread level (TLP), where multiple threads can be executed in parallel on multicore processors or concurrently on hardware multithreaded processors.

A hardware multithreaded processor [3] provides the hardware resources and mechanisms to execute several hardware threads on one processor core in order to increase its pipeline utilization, hence the application throughput. Unused instruction slots, which arise from pipelined execution of single-threaded programs by a monothreaded core, are filled by instructions of other threads within a multithreaded processor. The hardware threads compete for the shared resources and tolerate pipeline stalls due to long latency events, such as cache misses. These events can stall the pipeline up to 75% of its execution time [4]. Thus, multithreaded processors have two major advantages over other types of processors: 1) they offer the ability to hide latency within a thread (e.g., memory or execution latency) and 2) they achieve high transistor/energy efficiency.

Many general purpose single-chip processors exploit ILP, but as the limits of this approach are being reached, such processors are beginning to support both ILP and TLP, such as Intel(R) Pentium(R) 4 Hyper-Threading (HT) [5]. This allows the processor to exploit a wider range of parallelism, but considerably increases the chips complexity and power consumption, making them unsuitable for embedded applications. For instance, the Intel(R) Pentium(R) 4 processor supports Hyper-Threading with 2 thread contexts (TC), has a die size of 81 mm^2, and consumes more than 86 Watt in 65-nm technology [5]. On the other hand, embedded processors must have a die size in the order of few hundreds μm^2 and most consume in the order of few mW, using the currently available technologies (65 nm and below). Thus, they should support simple technology for exploiting ILP, such as pipelining or VLIW. Non-deterministic ILP boosting mechanisms, such as speculative scheduling or branch prediction, should be avoided. In this context, processing a single thread stream often leaves many functional units of the embedded processor underutilized. To compensate the loss in single-thread performance and to increase the transistor/energy efficiency of the embedded processor, designers are exploiting the parallelism at the thread level (TLP) through the implementation of embedded multithreaded processors [6], [7].

Moreover, future manycore architectures tend to use small footprint RISC cores [8]. Therefore, in this paper study, we will consider the extreme case: a **5-stage pipeline, in-order, single-issue, monothreaded RISC core**. Then, we will support this core with hardware interleaved multithreading (IMT). The following are the main contributions of this paper:

- Design methodology and validation in RTL of a scalar interleaved multithreaded processor for embedded systems from a small footprint RISC monothreaded processor.
- Synthesis results of the monothreaded and the IMT cores that give the designers the accurate occupation rate of each core component.

This paper is organized as follows: Section II discusses related works on different types of multithreading techniques for scalar monothreaded processors. In particular, the interleaved multithreading (IMT) will be retained in this study. Section III introduces briefly the scalar, in-order, 5-stage monothreaded core that we will use as the base processor throughout our study. The monothreaded core will be synthesized in 40 nm TSMC technology. Based on the surface occupation rate, we will deduce the optimal number of thread contexts (TC) that can be integrated. Then, section IV applies the IMT technique to the monothreaded core by adding only the necessary components. In section V, we will validate the IMT core and compare it to the monothreaded core using a simple bubble-sort application. And finally, section VI concludes the paper by discussing the present results along with future works.

II. Related Work

Two types of multithreading techniques exist for the scalar and in-order monothreaded core: interleaved and blocked.

Interleaved multithreading (IMT), also called switch-on-cycle or fine-grain multithreading, is a multithreading technique that issues an instruction from a different thread at every clock cycle using a round-robin scheduler, with zero context-switching overhead. The first well-known architecture which uses IMT is the Denelcor HEP [9]. It supports up to 50 threads in hardware. Tera MTA [10] is a derivative of the HEP with similar properties that supports 128 TCs. These architectures do not use caches, and rely on having a large number of threads to hide the memory latency between successive instructions of a thread. At any point in time, each pipeline stage will contain an instruction from a different thread. Therefore, there is no need for a complex circuitry that handles data dependencies between the instructions, since each thread can have just one instruction in the pipeline. Nevertheless, to support sufficient parallelism and eliminate the need for a 'data forwarding unit' that handles data dependencies, the number of active threads should be equal or greater to the number of pipeline stages. For instance, MIPS 34K [6], a recent IP for MPSoC integration, has a 9-stage pipeline and supports 9 TCs. SUN UltraSPARC T2 [11], a CMT processor used for server architectures, has a 6-stage pipeline for each core and supports 8 TCs. In the IMT, the performance of a single thread is degraded by 1/n, where n is the number of TCs. Thus, IMT architectures are useful for throughput oriented architectures. For example, in embedded systems, Eleven Engineering XInc [12] and Ubicom MASI [13] IMT processors are used in the wireless communication domain. Another researcher has developed an IMT MicroBlaze soft-IP for FPGAs [14].

On the other hand, blocked multithreading (BMT), also called switch-on-event or coarse-grain multithreading, allows a thread to run normally as in sequential mode before being switched out for long latency events such as cache misses, memory loads, failed synchronization [15], or wait for producer data in a streaming execution model. These events normally represent points in execution at which the processor would become idle for a long period of time. In such a case, it is useful to perform a context switch and execute instructions from another thread to fill the otherwise idle cycles. This is only effective when the context switch time is significantly less than the idle period of the event causing the switch [16]. The main advantage of BMT is that it requires a smaller number of TCs for multithreading, which means lower hardware cost than IMT. For instance, Infineon TriCore2 [7] supports 2 TCs for a 6-stage pipeline, PRESTOR-1 [17] supports 4 TCs for a 10-stage pipeline, and MulTEP [18], which is based on Anaconda multithreaded processor [19], supports 2 TCs for a 5-stage pipeline. In addition, each thread can execute at full processor speed as in single-threaded mode. However, careful processor design choices must be taken to not starve other waiting thread contexts. For instance, if the BMT processor is well-dimensioned and the cache misses are almost negligible, this means there will be no context switches, hence other thread contexts will never advance in execution. Thus, for real-time embedded applications, TCs should have priorities to guarantee the response time. For instance, in TriCore 2, TC0 is the main thread and TC1 is a helper thread. Another drawback is the context switch penalty, which is dependent of the number of pipeline stages. In fact, for each thread context switch, the pipeline should be totally flushed and reset.

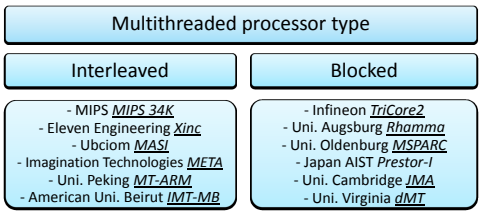

Figure 1. Examples of industrial and research interleaved and blocked multithreading architectures for embedded systems

Examples of recent IMT and BMT processors that exist in embedded systems is shown in Figure 1. To the authors' knowledge, there are lot of misconceptions in the literature of the actual surface overhead of the multithreaded processor compared to its monothreaded counterpart. Some manufacturers claim it is only 5% [5]. It is not clear if they considered the core area alone or with the L1 cache memories. However, we believe that this low number only applies for a big general-purpose processor, and might change for small footprint cores in embedded systems. Therefore, in this next sections, we will investigate in more detail the exact area overhead of an interleaved multithreaded core applied for a small footprint monothreaded core.

III. Monothreaded AntX

The monothreaded core that will be retained throughout our study is called AntX. It has lot of similarities with MIPS-I R3000 described in [20]. AntX is a scalar, in-order, 5-stage pipeline (IF,ID,EX,MEM,WB), monothreaded RISC core (Figure 2), developed by the Embedded Computing Laboratory at CEA LIST. It is a 32-bit architecture designed

specifically to be used as a low-cost control core in a MPSoC environment. Therefore, there are no complex units such as a branch predictor, FPUs, and multipliers. Its register file is composed of 16 32-bit registers.

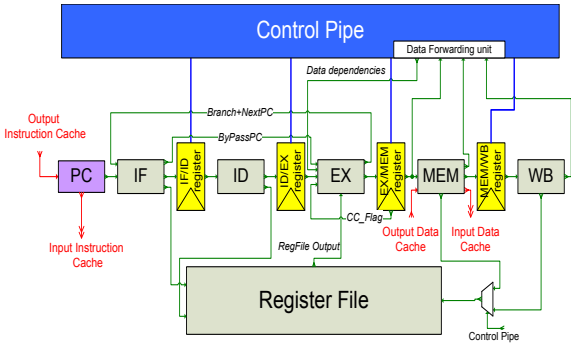

Figure 2. Monothreaded AntX: scalar, in-order, 5-stage, monothreaded processor

AntX comes along with a dedicated GNU toolchain (antx-elf-gcc and all binutils) that is already ported to its ISA. The ISA supports a variable instruction size (16/32 bit) in order to reduce the instruction memory footprint. So, some basic arithmetic/logic/comparison/jump instructions are coded in 16-bit, while other more complex instructions are coded in 32-bit. The Instruction Fetch (IF) unit fetches a 32-bit instruction from the memory and handles the aligned/unaligned instructions in a finite state machine (FSM). Figure 3 shows the different cases of input instruction combinations.

The instruction flow in the pipeline resembles that of the MIPS-I R3000 described in [20]. One exception is that the jump/branch instructions are executed in the EX-stage instead of the ID-stage. They use the ALU in order to calculate the new program counter (PC) address, so the hardware cost of a dedicated adder is compensated. Another difference is that we disabled the delay slot instruction after a jump/branch instruction using the gcc compiler option '-fno-delayed'. The compiler inserts always a 'nop' instruction.

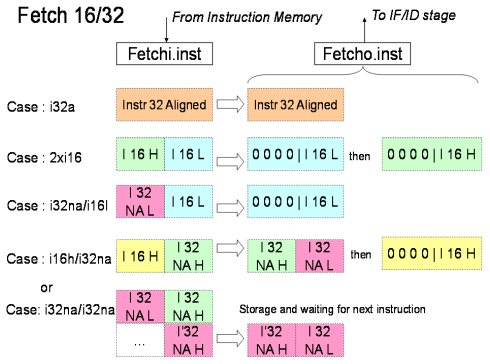

Figure 3. Representation of different cases in AntX Instruction Fetch

The *control pipe* unit is responsible of handling the data dependencies between the instructions in the different pipeline stages. For example, when an instruction in the ID-stage wants to read a data from a specific register, and that data is already calculated but not yet committed by the WB-stage, the *control pipe* will forward the data using the 'data forwarding unit' from the WB-stage to ID-stage without stalling the pipeline. Data forwarding eliminates most of the pipeline hazards (WAR, RAW, WAW). However, 1-cycle pipeline stall latency can still occur due to 2 reasons: branch instructions penalty (if taken-branch) and pipeline interlocks due to load/store instructions in the MEM-stage. The latter is due to memory access latency during a L1 cache hit when load/store instructions are in the MEM-stage. On the other hand, if the data is not present in the L1 cache (cache miss), then the waiting time is more than 1 clock cycle. In fact, those pipeline stalls will degrade the processor performance and stop it from reaching the optimal IPC of 1.

A monothreaded AntX RTL model has been developed in VHDL and has been synthesized in 40 nm TSMC technology using Synopsys dc_shell 2009.06-SP1 (low power, low threshold voltage, worst case) with a frequency of 300 MHz. The surface repartition of each module is shown in Figure 4. The overall core area is 11417 μm^2, which is about 8.05 kilogates. AntX area is smaller than a 2KB direct mapped cache memory according to CACTI 6.5 tool [1]. The average power consumption of this model is 1.67 mW.

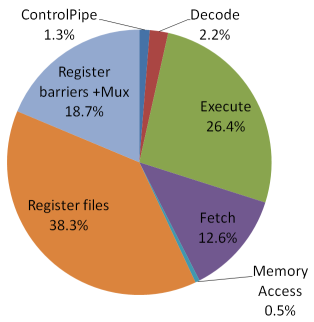

Figure 4. Surface repartition of different components in monothreaded AntX for 40 nm TSMC technology. Total area = 11417 μm^2, Total number of gates = 8.05 kilogates

One clear observation is that the register file occupies 38% of the total core area, which is a significant portion of the tiny monothreaded core. In a multithreaded processor, each TC has its own register file. Therefore, for a multithreaded AntX with 4 TCs, the new core area increase will be more than 100%. This implies there is a diminishing return advantage of implementing an embedded multithreaded processor more than 2 TCs. Another motivation for implementing 2 TCs is that the pipeline stall penalty is 1-cycle for most of the cases. Thus, dependent instructions are only 1 pipeline stage distant from each other and 2 TCs are enough for masking this dependency. Therefore, for the rest of our work, we chose multithreaded processors with 2 TCs. In the next section, we will explore in more details the core die area increase due to IMT.

IV. IMT AntX

IMT is a multithreading technique that issues an instruction from a different thread at every clock cycle using a round-robin scheduler, with zero context-switching overhead. In this section, we will modify the monothreaded AntX RTL model described in section III in order to support interleaved multithreading with 2 hardware TCs. In fact, for a 5-stage pipeline, 2 TCs are sufficient to eliminate the stall conditions due to pipeline interlocks and branch penalties (if taken-branch). AntX IMT with 2 TCs (TC1 and TC2) is shown in Figure 5.

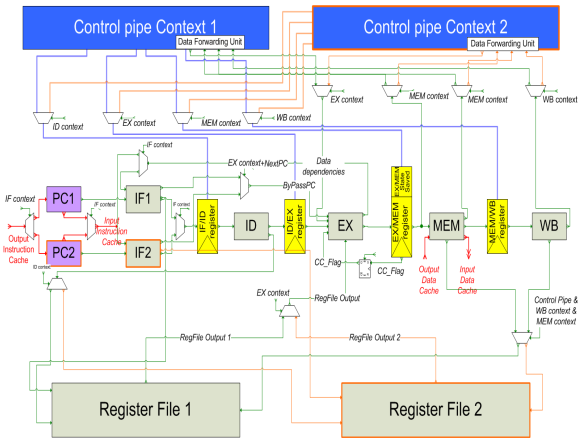

Figure 5. Interleaved multithreaded AntX: scalar, in-order, 5-stage, interleaved multithreaded processor

The following are the main modifications for extending the monothreaded to IMT AntX:

- *Duplicating the register file and PC:* Each TC should have its own register file and PC in order to store and switch the context in zero time overhead.
- *Duplicating control pipe:* The control pipe module is used to manage the instruction flow and dependencies of a TC at each pipeline stage. Therefore, it controls the pipeline and validity of each stage. In IMT, two successive pipeline stages have instructions from different TC. Therefore, to support 2 TCs, either we have to modify the original control pipe (monothreaded version) or duplicate it. According to the synthesis results of the monothreaded AntX (Figure 4), the surface occupation of the control pipe is only 1%. Accordingly, from development and validation time perspectives, we duplicate the control pipe. In addition, a multiplexer is added for each I/O signal belonging to the control pipe. This multiplexer switches between the 2 control pipes depending on the actual TC identifier in the pipeline stage. The multiplexers are not on the critical path of the synthesized architecture.
- *Duplicating IF module:* To manage two different TCs, the IF module can be modified or duplicated. The first one involves modifying the fetch state machine and saving each TC state at each context switch, which incorporates more development and validation time. The second one is easier to implement, since the IF module is already validated. Furthermore, in terms of surface cost, the two solutions would be equivalent, since the modification of the IF module incorporates the addition of a large number of registers to store the TC state. Therefore, the second solution has been preferred. However, a small modification is required for each IF module to handle properly the instruction fetching: the state of the FSM should be delayed. This implies that the FSM is updated every 2 clock cycles, since each TC is processed at half the speed. In our model, the IMT pipeline is not fully blocked if one TC is blocked. As shown in Figure 6, if TC2 is blocked due to I$ miss, then TC1 can continue fetching instructions at half speed.
- *Augmenting the EX/MEM inter-stage register size*: When a data cache miss occurs in the MEM stage for TC1, the pipeline is normally stalled waiting for the data. Meanwhile, TC2 instructions could have proceed their execution and increase the pipeline utilization. Therefore, the EX/MEM register has been increased to save the EX/MEM state that corresponds to TC1 in order to be reloaded when TC1's data arrives as shown in Figure 7. If the state is not saved, the MEM module would have the output from a wrong instruction, and the instruction that caused the data miss would be lost. To guarantee a correct behavior, the PC of the blocked context should not be incremented during a data cache miss.
- *Delaying signals:* Some signals have been delayed so they correspond to the right TC. For instance, the bypass PC signal from IF-stage to EX-stage and the execution flag signal from EX-stage are delayed by 1 cycle. Otherwise, the instruction execution flow would be incorrect.

Cycle	Fetch	Decode	Execute	Mem	WB
N	Instr1 (TC1)	x	x	x	x
N+1	Instr1 (TC2) **I$ Miss**	Instr1 (TC1)	x	x	x
N+2	Instr2 (TC1)	Instr1 (TC2) **BUBBLE**	Instr1 (TC1)	x	x
N+3	Instr1 (TC2) **I$ Miss**	Instr2 (TC1)	Instr1 (TC2) **BUBBLE**	Instr1 (TC1)	x
N+4	Instr3 (TC1)	Instr1 (TC2) **BUBBLE**	Instr2 (TC1)	Instr1 (TC2) **BUBBLE**	Instr1 (TC1)
N+5	Instr1 (TC2) **I$ Hit**	Instr3 (TC1)	Instr1 (TC2) **BUBBLE**	Instr2 (TC1)	Instr1 (TC2) **BUBBLE**

Figure 6. Example of Fetch running in IMT AntX processor during an instruction cache miss

IMT AntX RTL model has been developed in VHDL and has been synthesized in 40 nm TSMC technology (low power, low threshold voltage, worst case) with a frequency of 300MHz. The surface repartition of each module is shown in Figure 8. The overall core area is 19800 μm^2, which is about 13.97 kilogates. IMT AntX area is almost equal to a 4KB direct mapped cache memory according to CACTI 6.5 tool [1].

Cycle	Fetch	Decode	Execute	Mem	WB
N	Instr3 (TC1)	Instr2 (TC2)	Instr2 (TC1)	Instr1 (TC2) **D$ Miss**	Instr1 (TC1)
N+1	Instr2 (TC2) **BUBBLE**	Instr3 (TC1)	Instr2 (TC2)	Instr2 (TC1)	Instr1 (TC2) **BUBBLE**
N+2	Instr4 (TC1)	Instr2 (TC2) **BUBBLE**	Instr3 (TC1)	Instr1 (TC2) **D$ Miss**	Instr2 (TC1)
N+3	Instr2 (TC2) **BUBBLE**	Instr4 (TC1)	Instr2 (TC2) **BUBBLE**	Instr3 (TC1)	Instr1 (TC2) **BUBBLE**
N+4	Instr5 (TC1)	Instr2 (TC2) **BUBBLE**	Instr4 (TC1)	Instr1 (TC2) **D$ Hit**	Instr3 (TC1)
N+5	Instr2 (TC2)	Instr5 (TC1)	Instr2 (TC2) **BUBBLE**	Instr4 (TC1)	Instr1 (TC2)

Figure 7. Example of data cache miss managed by IMT AntX processor using augmented EX/MEM inter-stage register

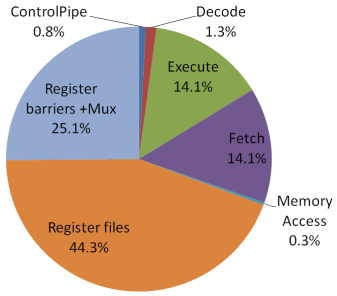

Figure 8. Surface repartition of different components in IMT AntX for 40 nm TSMC technology. Total area = 19800 μm^2, Total number of gates = 13.97 kilogates

The IMT AntX has an augmentation of *73.2%* in core area compared to the monothreaded AntX. This is mainly due to doubling the RF, PC, and IF modules, which is essential for proper IMT functioning. However, if we consider the area of the L1 caches with the core area, then this percentage will drop. For instance, for a L1 I$ and D$ of 2 KB, IMT AntX has an augmentation of only *23.8%* with respect to the monothreaded AntX.

V. Validation

In this part, we provide a case study scenario in order to validate the functionality of the IMT model in an embedded SoC platform and compare its performance to the monothreaded model.

We use a typical processor system environment described in Figure 9. The processor-memory architecture is based on a Harvard architecture with separate L1 instruction cache (I$) and data cache (D$) busses. It implements a 2-level memory hierarchy with L1 I$ and D$ memories, connected with an AHB bus to an on-chip L2 instruction and data memories. The L2 memories contain all the instruction and data codes of the applications.

The processor can be either monothreaded or IMT AntX with 2 TCs. For the IMT AntX, the L1$ memory is segmented per TC in order to limit cache interferences. Therefore, each TC has half the L1$ size compared to the monothreaded AntX.

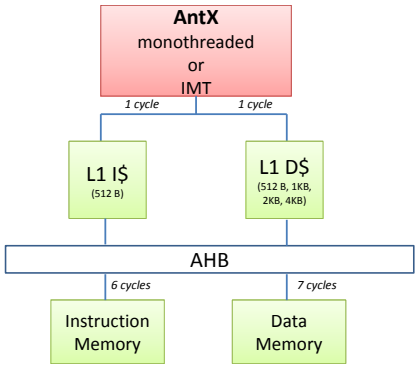

Figure 9. AntX processor system environment with L1 I$ and D$ connected with an AHB bus to on-chip instruction and data memories

For this experiment, we consider a basic bubble-sort application for 100 elements. The application has lot of jump/branch instructions and data dependencies between instructions. We run 2 instances of the application with different elements sequentially on the monothreaded processor, and concurrently on the IMT processor.

In this experiment, we vary 2 platform parameters for a better architecture exploration. First, the processor type can be chosen to be *monothreaded* or *IMT*. Second, the L1 D$ memory size can be set to 512 B, 1 KB, 2 KB, and 4 KB. This will generate different data cache miss percentages as shown in Figure 10. The L1 I$ size is fixed to 512 Byte, which is sufficient for the bubble-sort application and generates only 0.1% of I$ miss. A L1 cache hit takes 1 clock cycle, an access to L2 instruction memory due to L1 I$ miss takes 6 cycles, and an access to L2 data memory due to L1 D$ miss takes 7 cycles on average. L2 memory access time might vary few cycles (1-2 cycles) depending on the AHB arbiter.

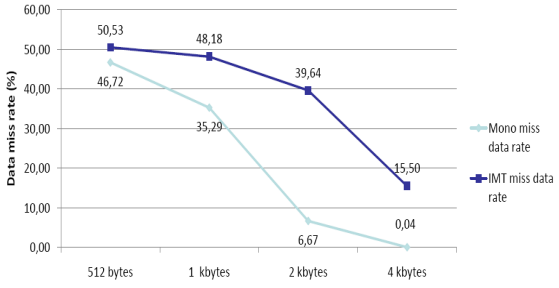

Figure 10. Data cache miss rate for monothreaded and IMT AntX while varying the cache size: 512 Byte, 1 KB, 2 KB, 4 KB

In Figure 11, we show the execution time in cycles for all the configurations. We decompose the total execution time into 4 components: effective execution time, branch instruction penalty time due to 'taken' branches, data dependencies stall time due to pipeline interlocks, and memory stalls time due to cache misses.

As we can notice from Figure 11, the IMT processor

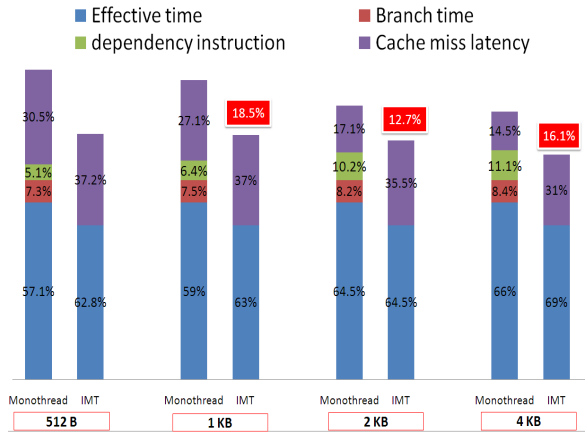

Figure 11. Performance results of monothreaded v/s IMT AntX running two bubble-sort applications on 100 elements. The IMT overcomes the performance of the monothreaded by 17% on average

overcomes the performance of the monothreaded processor for all the cache configurations. The performance gain varies between 12.7% and 20.7%. In fact, the performance gain highly depends on the percentage of data cache misses that each segmented cache generates. Each TC in IMT processor has half the cache size, hence it generates more cache misses and more pipeline stalls due to L2 memory access. Due to its interleaving property, the IMT tolerates the pipeline stalls generated by branch penalties and data dependencies between instructions. Their stall time are hidden completely by executing instructions from another TC, if it is active. It is clear that 2 TCs are sufficient to hide all these latencies for a 5-stage pipeline processor.

VI. Conclusion

This paper presented a small footprint, scalar, in-order, 5-stage pipeline, interleaved multithreaded processor with 2 hardware thread contexts for embedded systems and SoC integration. The RTL model of the IMT processor is based on a RISC monothreaded processor called AntX, which resembles the MIPS-I R3000 [20]. The synthesis results of the monothreaded core showed that there is no area advantage of adding more than 2 TCs for the multithreaded core. In fact, for each TC, the register file, the PC, and the IF module should be doubled to guarantee proper IMT functioning. The overall multithreaded core area is 19800 μm^2 and 13.97 kilogates, which is almost equal to a 4KB direct mapped cache memory according to CACTI 6.5 tool [1]. The IMT AntX has an augmentation of *73.2%* in core area compared to the monothreaded AntX. The IMT RTL model is validated by executing 2 instances of a simple bubble-sort application concurrently, while varying the L1 D$ size. The extracted pipeline statistics showed that the IMT model is able to hide all the pipeline stalls due to data dependencies between instructions and branch penalties. The IMT core gives an average performance gain of 17% compared to the monothreaded core.

For future work, we aim to implement the blocked multithreaded (BMT) processor and compare its surface overhead and performance with the IMT processor. In addition, we want to port more significant benchmarks for embedded systems and run them on the 3 processor models.

Acknowledgements

We thank Alain Chateigner, Erwan Piriou, and Philippe Fauvel for their helpful contributions in this work.

References

[1] Naveen Muralimanohar and Rajeev Balasubramonian. Cacti 6.0: A tool to model large caches.

[2] D.W. Wall. Limits of instruction-level parallelism. In *Int'l Conf. on Architectural Support for Programming Languages and OperatingSystems (ASPLOS)*, Santa Clara, USA, April 1991.

[3] Theo Ungerer, Bortu Robic, and Jurij Silc. Multithreaded Processors. *The Computer Journal*, 45:320–348, 2002.

[4] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, 2005.

[5] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *Micro, IEEE*, 23(2):56 – 65, march-april 2003.

[6] MIPS. Programming the MIPS32® 34K Core Family. Technical report, MIPS Technology, 2005.

[7] Erik Norden. A Multithreaded RISC/DSP Processor with High Speed Interconnect, 2003.

[8] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[9] J. S. Kowalik. Parallel mimd computation: the hep supercomputer and its applications. 1985.

[10] M. Howard and A. Kopser. Design of the Tera MTA integrated circuits. In *Gallium Arsenide Integrated Circuit (GaAs IC) Symposium, 1997. Technical Digest 1997., 19th Annual*, pages 14 –17, oct 1997.

[11] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC. pages 22 –25, nov. 2007.

[12] Y. Le Moullec, C. Leroux, E. Baud, and P. Koch. Power consumption estimation of the multi-threaded xinc processor. pages 210 – 213, nov. 2004.

[13] David Fotland. Ubicom's MASI Wireless Network Processor, 2003.

[14] R. Moussali, N. Ghanem, and M.A.R. Saghir. Microarchitectural Enhancements for Configurable Multi-Threaded Soft Processors. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 782 –785, aug. 2007.

[15] A. Agarwal, R. Bianchini, D. Chaiken, F.T. Chong, K.L. Johnson, D. Kranz, J.D. Kubiatowicz, Beng-Hong Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine. *Proceedings of the IEEE*, 87(3):430 –444, mar 1999.

[16] B. Boothe and A. Ranade. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 214 –223, 1992.

[17] K. Tanaka. PRESTOR-1: a processor extending multithreaded architecture. In *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2005*, page 8 pp., jan. 2005.

[18] Panit Watcharawitch and C Panit Watcharawitch. MulTEP: Multi-Threaded Embedded Processors. In *In An International Symposium on Low-Power and High-Speed Chips (Cool Chips) IV, volume I. IEEE/IEICE/IPSJ/ACM SIGARCH Computer Society*, 2003.

[19] Simon W. Moore. *Multithreaded Processor Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.