Analysis of Execution Efficiency in the Microthreaded Processor UTLEON3

Jaroslav Sykora, Leos Kafka, Martin Danek, and Lukas Kohout.

Institute of Information Theory and Automation of the ASCR,
Department of Signal Processing,
Pod Vodarenskou vezi 4, Prague, Czech Republic.
{sykora, kafkal, danek, kohoutl}@utia.cas.cz
WWW home page: http://sp.utia.cz/

Abstract. We analyse an impact of long-latency instructions, the family blocksize parameter, and the thread switch modifier on execution efficiency of families of threads in a single-core configuration of the UTLEON3 processor that implements the SVP microthreading model. The analysis is supported by code execution in an FPGA implementation of the processor.
By classifying long-latency operations as either *pipelined* (e.g. floating-point operations) or *non-pipelined* (e.g. cache faults) we show that the blocksize parameter that controls resource utilization in the microthreaded processor has profound effects when the latency is *pipelined*, i.e. increasing the blocksize can improve the performance. In the *non-pipelined* long-latency case the efficiency reaches its maximum even with a small value of blocksize beyond which it cannot improve due to occupancy of an exclusive resource (memory bus congestion).
The conclusions drawn in this paper can be used to optimize code compilation for the microthreaded processor. As the compiler specifies the blocksize parameter for each family of threads individually, it can optimize the register file utilization of the processor.

Keywords: Processor architectures, micro-threading, multi-threading, memory latency tolerance, execution efficiency, SVP concurrency model, UTLEON3 processor.

1 Introduction

To combat the increasing complexity of traditional out-of-order multiple-issue processors, modifications of simpler in-order single-issue architectures were proposed that improve their performance without sacrificing their energy efficiency and hardware simplicity. The reduced complexity of in-order single-issue architectures allows for smaller footprint integration, thus shorting internal wires in the IC and achieving higher operating frequencies. At the same time the scaling of the manufacturing process makes placing many such cores on a chip feasible. A homogeneous array of processing cores could also simplify the energy and performance scaling, were it carried out on-line or during the SoC design.

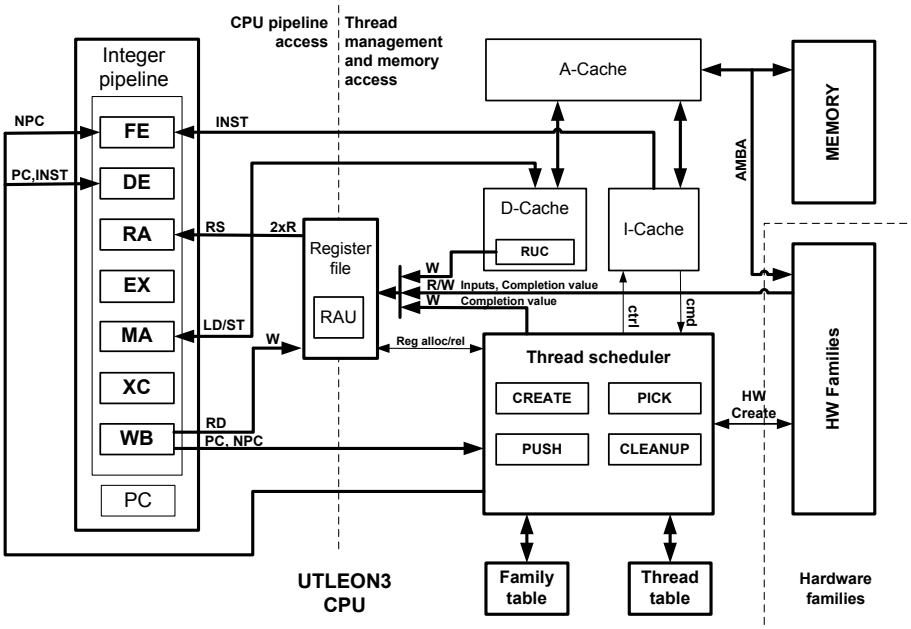


Fig. 1: Block diagram of the UTLEON3 processor. *Pipeline Stages*: *FE*=Instruction Fetch, *DE*=Decode, *RA*=Register Access, *EX*=Execute, *MA*=Memory Access, *XC*=Exception, *WB*=Writeback. *RAU*=Register Allocation Unit, *RUC*=Register Update Controller, *A-Cache*=AMBA–Cache Interface.

Previous architectures Single-threaded architectures focus on extraction of independent instructions (ILP) that can be executed during a stall. One of the in-order execution techniques to accomplish it is the run-ahead execution ([5], [14]) that allows a program to proceed past stalled instruction. The result register of the stalled instruction is marked as poisoned, and so all the dependent instructions can be identified. Once the stall is over, the dependent instructions must be re-executed, and their results correctly integrated into the architectural state. Other techniques try to decouple the producer and consumer instructions ([15], [16]). By executing the producer instructions (such as a memory load) early its effective latency can be partly hidden.

The multicore architectures require many concurrent tasks to be fully utilized. Although a multicore architecture suitable for automatically parallelised sequential code was proposed in [3], it is unclear whether enough independent instructions could be extracted from existing unmodified programs to sustain many cores. Existing architectures usually presume multi-programmed environment (e.g. *OpenSPARC T1/T2* [12]), where processes communicate through shared memory or by message passing.

Others provide ISA extensions to support multiple thread execution [21], but do not automatically spread threads over multiple cores. The *MIPS MT* archi-

tecture [11] introduces only two new unprivileged instructions: `fork` to create a new thread, and `yield` to make a thread wait for an event. The *Sparcle/Alewife* [2] system is based on a slightly modified SPARC architecture. It employs block (coarse) multithreading. The SPARC register windows are used to implement 4 independent thread contexts rather than as a register stack; context switching and thread scheduling is done in software via fast traps; and fine-grain synchronization through *empty/full bits* is implemented in an external Communications and Memory Management Unit and in the cache controller. The *MSparc* [13] architecture is similar to Sparcle, but the context switching mechanism is provided in hardware.

Contributions Previous works [19], [22] have dealt with an analysis of multithreading efficiency with respect to the number of available hardware thread contexts (denoted N, see Table 1), average run length between cache faults (R), cache latency (L), register file size (F), and required context size (number of registers per thread; C). However, as the previous processor architectures did not have the concept of families of threads,[1] they could not analyse the impact of the blocksize parameter (B). In [19] only cache fault latencies (i.e. *non-pipelined* ones) were stochastically analysed, and the number of contexts (N) was considered a machine parameter, while in the SVP/UTLEON3 architecture it is a program-controlled (but implementation bounded) variable. This paper discusses effects of this parameter on the efficiency of the microthreaded execution in the UTLEON3 processor. We show that the blocksize parameter has a similar effect as the number of thread contexts had in the previous architectures. The outcomes of the work can be used to improve the technology of a compiler for the architecture.

2 UTLEON3 Processor Architecture

The *UTLEON3* processor (see [6], and the block diagram in Figure 1) is an implementation of the SVP model (see [9], [10]) for managing concurrency. The SVP model is multicore-aware, meaning concurrent threads are to be automatically spread in a processing *micro-grid* environment (a 2D array of SVP processors). The *UTLEON3* core is written in VHDL, it has a 7-stage in-order single-issue multi-threaded execution pipeline, and it is fully FPGA-synthesizable. It is based on the *LEON3* SPARCv8 embedded processor from *Aeroflex Gaisler* [7]. However, the multicore support hardware has not been implemented in UTLEON3 yet.

Microthreading is a multi-threading variant that decreases the complexity of context management. The goal of microthreading is to tolerate long-latency operations (LD/ST and multi-cycle operations such as floating-point) and to synchronize computation on register access. For an overview of multi-threading see [21].

[1] The concept is explained in paragraph *Families of Threads* in section 2.

A possible speedup generated by micro-threading comes from the assumption that while one thread is waiting for its input data, another thread has its input data ready and can be scheduled in a few clock cycles and executed. Another assumption is that load and store operations themselves need not be blocking since the real problem arises just when an operation accesses a register that does not contain a valid data value.

Register File Traditional multi-threaded architectures often replicate the whole processor state for each thread context supported, including all the architectural (program visible) registers [11], [2], [13]. This simplifies the software programming model, but it requires a large register file to support just few thread contexts. As the processor register file is one of the most expensive units in a CPU [18], its optimal utilisation is very important. Previous experimental work [22], [17] has shown the advantage of reducing the per-thread context size. In [17] the authors reduced the context from 32 to 16 registers (i.e. two mini-threads in one context) without much performance impact.

The UTLEON3 processor allows each thread to individually specify the number of required registers, from 1 up to the architectural limit of 32 registers per thread given by the SPARCv8 instruction encoding. This is done through the `.registers` pseudo-instruction given at the beginning of each thread routine which informs the processor how many registers it must allocate for the thread.

Fine-grained synchronization and communication between threads in one family, and between the processor pipeline and long-latency units (cache, FPU) is accomplished by a self-synchronizing register file (I-structures [4]). Each 32b register in the file is extended with a state information. When a register contains valid data, it is in the *FULL* state. If data are to be delivered into the register asynchronously (from the cache, FPU etc.), but they have not been required by the pipeline yet, the register is in the *PENDING* state. Finally, if the data were required by a thread, but were not available at the time, the register is in the *WAITING* state; only on the $PENDING \Rightarrow WAITING$ transition the thread has to be switched out and suspended.

Thread Scheduling The processor implements blocked (coarse) multithreading, meaning a thread is switched out of the pipeline only when an unsatisfied data dependence (i.e. a dependence on a long-latency operation) has been encountered. This improves the single-thread performance, but it requires the pipeline to have fully bypassed stages. When a thread switch occurs, the affected thread's instructions in the previous pipeline stages must be flushed. For example when a thread in the *Execute Stage* of the pipeline reads a register in the *PENDING* state, it must be switched out as the data in the register is not valid, and thus the two previous pipeline stages (*Decode*, *Register Access*) must be cleared as well – but only if they contain instructions from the same thread. To optimize this case the architecture allows to annotate each instruction with a `swch` modifier that will cause the thread to voluntarily switch itself out of the pipeline early in the *Fetch Stage* (if a compiler or assembler programmer anticipates the instruction depends on a long-latency producer instruction). The voluntary switch causes

Table 1: Parameters. LL=long-latency, FP=floating point.

Parameter	Description	(units)
R	average run length between two LL ops.	(cycles, instructions)
L_N	non-pipelined latency, e.g. a cache fault	(cycles)
L_P	pipelined latency, e.g. FP operation	(cycles)
N	supported number of thread contexts	(number of threads)
B	family blocksize	(number of threads)
C	thread context size	(registers)
S	context switch cost	(cycles)
F	register file size	(registers)

zero overhead in most cases as there is no pipeline bubble inserted. However, the dependent instruction must be cleared in any case and later re-executed, so the synchronization cost in the typical case is 1 cycle, while without the modifier it is 3 cycles. The processor allows for instructions from different threads to be present in distinct pipeline stages at the same time.

Families of Threads Contrary to some previous multi-threaded architectures which often provided only a few (4-8) hardware thread contexts, the UTLEON3 processor supports *tens* to *hundreds* of concurrent hardware thread contexts. To be able to take advantage of the large-scale parallelism offered the threads are created in *families*. Conceptually, threads in one family share data and implement one piece of a computation. In a simple view one family corresponds to one *for*-loop in the classical *C* language; in microthreading each iteration (each thread) of a hypothetical *for*-loop (represented by a family of threads) is executed independently according to data dependences.

As the families of threads can be nested similarly to the classical *for*-loops, the processor can execute many different families, each comprising many threads, at the same time. Families compete for processor resources, mainly for entries in the thread table and registers in the register file. To regulate resource allocation and sharing among families of threads the architecture allows to specify (for each family individually) a so-called *blocksize* parameter. This parameter is optional for it does not affect the *semantics* of the computation, but it affects its pace. *The* blocksize *value limits the maximal number of threads of a given family that are allowed to co-exist at any moment in the thread table.* This enables a compiler or assembly-level programmer to optimize the rate of thread creation in each family.

3 Analysis

We classify long-latency (LL) operations in two groups: (a) *pipelined LL operations* are executed in a fully pipelined unit (e.g. an integer multiplier, a floating-point unit) which means the processor can initiate a new computation in the

unit every clock cycle, and the processing takes L_P cycles. (b) *non-pipelined LL operations* must be served sequentially in a unit as they compete for some exclusive resource, e.g. a system/memory bus in the case of a cache fault, and we assume the operation takes L_N cycles to complete.

Upon creation of a family the total number of threads n in the family (equal to the number of iterations of a hypothetical *for*-loop) and the blocksize parameter B are specified by the program. The scheduler unit then reserves B entries in its thread table and allocates $B \cdot C$ registers in the (large, but shared) register file. Thus, in the family given no more than B threads can execute concurrently at any moment. But if the requested number of resources cannot be allocated (for they were spent on existing k families: $\sum_k (B_k \cdot C_k) \leq F$), the hardware scheduler must reduce the value of B until the allocations succeed so that the computation can proceed.

Clearly each family of threads contributes a different amount of parallelism to the processor execution. By optimizing the value of B for each family a compiler or assembly-level programmer can achieve better utilization of processor resources.

3.1 Impact of the Pipelined Long-Latency Operations

Let us assume a family of n threads with blocksize B. Each thread of the family executes $R-1$ short-latency (one cycle) instructions and one long-latency instruction; thus the run length between two long-latency operations is R cycles. The long-latency operation takes L_P cycles to complete, and it is fully pipelined. The thread switch cost is S cycles.

Figure 2 shows what happens in the processor pipeline with respect to the blocksize B and latency L_P. In Figure 2a the long-latency operation has latency L_P much higher than is the combined computational load of B threads, i.e. $L_P > R(B-1)$. In this case the family operates in the *linear region* because increasing B gains more performance.

In Figure 2b the latency L_P is shorter than $R(B-1)$, and thus the program computation is limited only by the processor pipeline performance (and context switch cost). Increasing the blocksize parameter B cannot gain more performance here because the pipeline is already *saturated*.

The perceived run-time in cycles per LL operation is plotted in Figure 3a. As the total program run-time depends on the number of instructions executed in the processor, in practice we normalize it to obtain execution efficiency to be able to compare execution for different benchmarks. The efficiency of one family in the linear region $\eta_{P,lin}$ can be calculated by considering the R to be the ideal number of *Busy* cycles:

$$\eta_{P,lin} = \frac{Busy}{Busy + Switching + Idle} = \frac{R}{R + S + \frac{L_P}{B}} = \frac{R \cdot B}{(R+S) \cdot B + L_P} \quad (1)$$

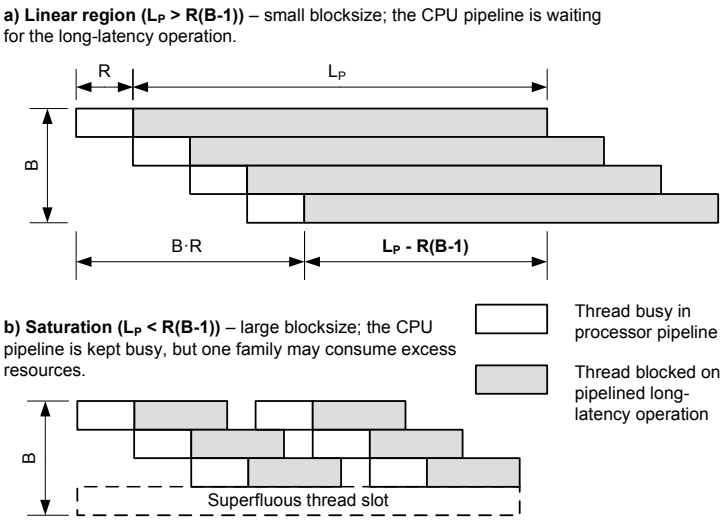


Fig. 2: Thread schedule for pipelined long-latency operations. A new operation can be issued while the previous one is processing, e.g. a pipelined integer multiplication, a floating point operation.

In the saturation the efficiency does not depend on the blocksize, but the context switching cost is more pronounced:

$$\eta_{sat} = \frac{R}{R+S} = \frac{1}{1+\frac{S}{R}} \tag{2}$$

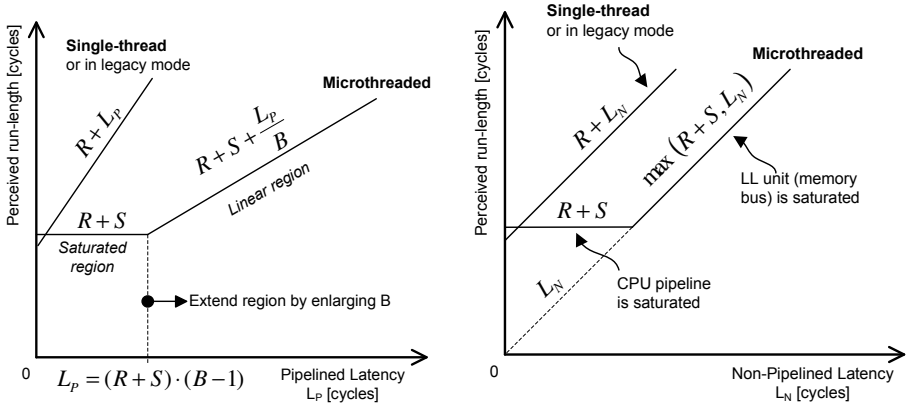
Note that the saturated efficiency η_{sat} is the same for the pipelined and non-pipelined classes of long-latency operations.

3.2 Impact of the Non-Pipelined Long-Latency Operations

A non-pipelined long-latency unit can process at most one request at a time because of an occupancy of some exclusive resource. Each request will take L_N cycles to process; any new request issued in the meantime must be blocked or stored in a FIFO queue to be served later. This is a model of a cache fault where each instance requires an access to the memory bus. An impact of a limited memory bandwidth on a multi-threaded processor was theoretically studied e.g. in [8].

The best case in the microthreaded mode occurs when the processor pipeline or the long-latency unit are saturated (Figure 4). Blocksize $B = 2$ is theoretically sufficient to keep at least one of the two resources fully occupied so that the program perceives minimal latency $max(R + S, L_N)$ (Figure 3b). Thus in the microthreaded mode the efficiency η_N of a family of threads does not depend on the blocksize:

$$\eta_N = \frac{R}{max(R+S, L_N)} \tag{3}$$



(a) Pipelined long-latency operations, e.g. a pipelined FP operation

(b) Non-pipelined long-latency operations, e.g. cache faults.

Fig. 3: Theoretical performance of long-latency operations.

4 Experimental Evaluation on the Microthreaded Processor

Experiments were carried out on an FPGA-synthesizable VHDL model of the UTLEON3 processor. We modified the processor's integer multiplier to simulate a long-latency pipelined unit with an arbitrary delay of L_P cycles. Similarly, the model of the main memory was extended so that it can simulate a non-pipelined delay L_N cycles. A synthetic benchmark program creates a family of threads in which each thread executes $(R-1)$ short-latency (1 cycle) and one long-latency instructions. The family blocksize value is specified by the `setblock` instruction before a family is created. Efficiency of early voluntary context switches is compared by using the `swch` modifier in the first (short-latency) instruction that depends on the long-latency one and comparing it to a run without the `swch` modifier.

The results in Figure 5a show CPU utilization efficiency η_P with respect to the pipelined long-latency operation L_P that was simulated by the modified multiplier unit. The run-length of the test program was $R_P = 15$ cycles. Different blocksizes $B_P = \{6, 8, 16, 24\}$ and the presence of the `swch` modifier were evaluated ($S_{without-swch} = 3$ cycles, $S_{with-swch} = 1$ cycle).

As shown in the analysis the blocksize parameter B determines the point (latency L_P) when the family transitions from the saturated region, where it executes with the maximal efficiency η_{sat} (Eq. 2), to the linear region, where the efficiency $\eta_{P,lin}$ (Eq. 1) decreases with L_P. Presence of the `swch` modifier affects the context switch cost S that influences the saturated efficiency $\eta_{sat} = \frac{R}{R+S}$ (Eq. 2). Without the `swch` modifier the theoretical saturated efficiency with the given $R = 15$ is 0.83, but with the `swch` modifier it is 0.93. This analytical prediction agrees with the measurement in Figure 5a.

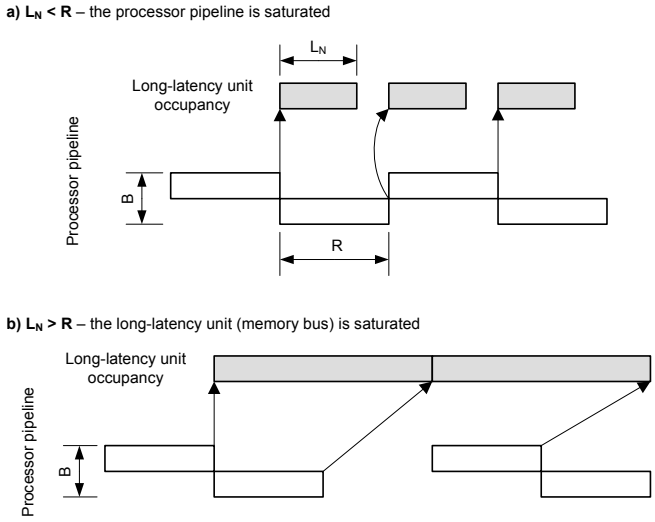


Fig. 4: Thread schedule for non-pipelined long-latency operations (e.g. cache faults). A new operation can be issued on completion of the previous operation.

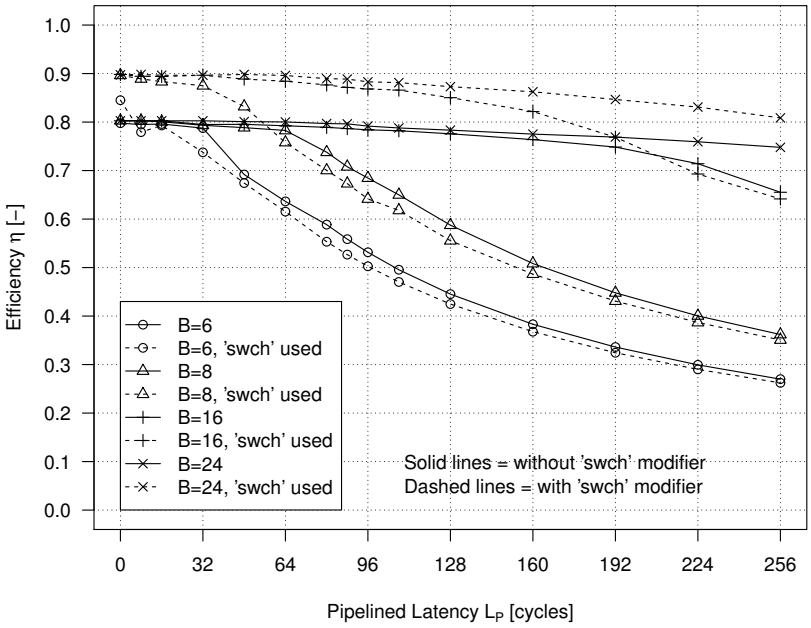
Figure 5b shows the CPU utilization efficiency η_N with respect to a non-pipelined long-latency operation L_N. In this case the run-length of the test program was $R_N = 49$ cycles, and different blocksizes $B_N = \{4, 8, 16\}$ and the presence of the `swch` modifier were evaluated.

As expected the blocksize parameter does not influence the CPU efficiency η_N so much when compared to the previous case. The effect of the `swch` modifier is not very pronounced in the plot, but that is because $R_N \gg S$ in the presented measurement.

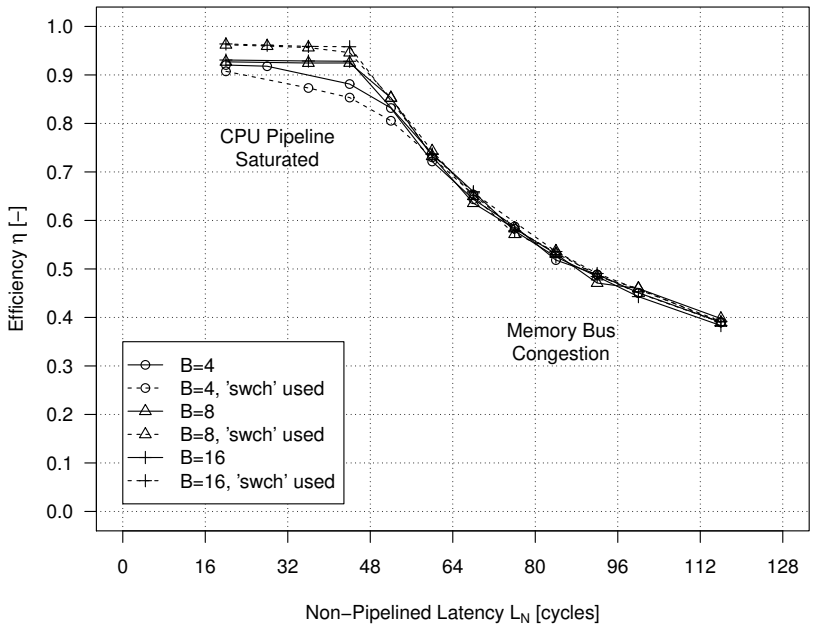
We believe that the chosen values have connection to real-world situations. In the pipelined long-latency operation the $R_P = 15$ cycles corresponds to 6% arithmetic intensity. For example the *Discreet Cosine Transform* (DCT) kernel from the the *Independent JPEG Group* [1] compiled for the SPARC architecture has arithmetic intensity 15%. In the non-pipelined long-latency operation the $R_N = 49$ cycles corresponds to cache miss rate of 2% over all instructions executed; given a typical ratio of memory access instructions of 20% in common programs (every fifth instruction is Load/Store), this implies a 10% D-Cache miss rate. Albeit quite large for contemporary single-threaded processors, this miss rate can be expected in processors executing multithreded workloads that increase presure on the cache subsystem.

5 Conclusions

In the paper we have analysed the impact of long-latency instructions, the family blocksize parameter, and the thread switch modifier on execution efficiency



(a) *Efficiency of pipelined long latency operations.* For short latencies the efficiency is optimal even with small blocksize values (the processor is saturated) and the effect of the 'swch' modifier is most profound. Longer latencies require higher blocksize values to stay in the saturation. Measured for $R_P = 15$ cycles, blocksize $B_P = \{6, 8, 16, 24\}$.



(b) *Efficiency of non-pipelined long latency operations.* Once the memory bus is congested increasing the blocksize value does not improve the execution efficiency. Measured for $R_N = 49$ cycles, blocksize $B_N = \{4, 8, 16\}$

Fig. 5: Experimental results obtained on the FPGA implementation of the processor.

of families of threads in the UTLEON3 processor that implements the SVP microthreading model. Experimental evaluations run on the FPGA implementation of the processor support the analysis.

We classify long-latency operations as either *pipelined* or *non-pipelined*. The analysis shows that the blocksize parameter that controls thread resource allocation in the processor hardware has profound effects when the latency is *pipelined*, i.e. increasing the blocksize value can improve the performance up to the saturation point η_{sat}. In the *non-pipelined* long-latency case the efficiency reaches its maximum even for small blocksize values, beyond which it cannot improve due to the occupancy of an exclusive resource (memory bus congestion).

As the compiler specifies the blocksize parameter for each family of threads individually, the analysis can be used to optimize resource usage. The compiler should specify smaller blocksize values for memory-intensive families of threads to save resources, while compute-intensive families with many pipelined FP operations could benefit from larger blocksize values. Ideally each family should operate just at its saturation point where the efficiency reaches its maximum while the resource utilization is optimal.

Acknowledgment

This work was supported and funded by the European Commission under Project Apple-CORE No. FP7-ICT-215215, and by the Czech Ministry of Education under Project No. 7E08013. The paper reflects only the authors' view; neither the European Commission nor the Czech Ministry of Education are liable for any use that may be made of the information contained herein. For information about the Apple-CORE project see [20].

References

1. Independent JPEG Group. http://www.ijg.org/
2. Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B.H., Yeung, D., D'Souza, G., Parkin, M.: Sparcle: An evolutionary processor design for large-scale multiprocessors. IEEE MICRO 13, 48–61 (1993)
3. Akkary, H., Jothi, K., Retnamma, R., Nekkalapu, S., Hall, D., Shahidzadeh, S.: On the potential of latency tolerant execution in speculative multithreading. In: IFMT '08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies. pp. 1–10. ACM, New York, NY, USA (2008)
4. Arvind, Nikhil, R.S.: Executing a program on the MIT tagged-token dataflow architecture. IEEE Transaction on Computers 39(6), 300–318 (1990)
5. Barnes, R.D., Nystrom, E.M., Sias, J.W., Patel, S.J., Navarro, N., Hwu, W.m.W.: Beating in-order stalls with "flea-flicker" two-pass pipelining. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. p. 387. IEEE Computer Society, Washington, DC, USA (2003)
6. Danek, M., Kafka, L., Kohout, L., Sykora, J.: Instruction set extensions for multi-threading in LEON3. In: Kotásek, Z., et al. (eds.) Proceedings of the 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems DDECS2010. pp. 237–242. IEEE (2010)

7. Gaisler, J., Catovic, E., Habinc, S.: GRLIB IP Library User's Manual. Gaisler Research (2007)
8. Guz, Z., Bolotin, E., Keidar, I., Kolodny, A., Mendelson, A., Weiser, U.C.: Many-core vs. many-thread machines: Stay away from the valley. IEEE Comput. Archit. Lett. 8(1), 25–28 (2009)
9. Jesshope, C.: Scalable instruction-level parallelism. In: Computer Systems: Architectures, Modeling, and Simulation. pp. 383–392. Springer Berlin / Heidelberg (2004)
10. Jesshope, C.R.: μTC - an intermediate language for programming chip multiprocessors. In: Asia-Pacific Computer Systems Architecture Conference. pp. 147–160 (2006)
11. Kissell, K.D.: MIPS MT: A multithreaded RISC architecture for embedded real-time processing. In: Stenström, P., Dubois, M., Katevenis, M., Gupta, R., Ungerer, T. (eds.) HiPEAC. Lecture Notes in Computer Science, vol. 4917, pp. 9–21. Springer (2008)
12. Kongentira, P., Aingaran, K., Olukotum, K.: Niagara: a 32-way multithreaded SPARC processor. IEEE Micro 25(2), 21–29 (2005)
13. Mikschl, A., Damm, W.: MSparc: A Multithreaded Sparc. In: Euro-Par'96 Parallel Processings: Second International Euro-Par Conference, Vol II, LNCS 1124. pp. 461–469. Springer Verlag (1996)
14. Nekkalapu, S., Akkary, H., Jothi, K., Retnamma, R., X., S.: A simple latency tolerant processor. In: IEEE 26th International Conference on Computer Design (2008)
15. Parcerisa, J.M., Gonzalez, A.: Improving latency tolerance of multithreading through decoupling. IEEE Trans. Comput. 50(10), 1084–1094 (2001)
16. Rangan, R., Vachharajani, N., Vachharajani, M., August, D.I.: Decoupled software pipelining with the synchronization array. In: PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. pp. 177–188. IEEE Computer Society, Washington, DC, USA (2004)
17. Redstone, J., Eggers, S., Levy, H.: Mini-threads: Increasing TLP on small-scale SMT processors. In: 9TH INTL SYMP. ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE (HPCA (2003)
18. Rixner, S., Dally, W.J., Khailany, B., Mattson, P., Kapasi, U.J., Owens, J.D.: Register organization for media processing. In: HPCA6. pp. 375–386 (2000)
19. Saavedra-Barrera, R.H., Culler, D.E., von Eicken, T.: Analysis of multithreaded architectures for parallel computing (1990)
20. The Apple-CORE Consortium: Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs. http://www.apple-core.info
21. Ungerer, T., Robič, B., Šilc, J.: A survey of processors with explicit multithreading. ACM Comput. Surv. 35(1), 29–63 (2003)
22. Waldspurger, C.A., Weihl, W.E.: Register relocation: Flexible contexts for multithreading. In: 20th Annual International Symposium on Computer Architecture. pp. 120–130 (1993)