Optimizing Resource Utilization with software–based Temporal Multi–Threading (sTMT)

Vicenç Beltran
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
vbeltran@bsc.es

Eduard Ayguadé
Barcelona Supercomputing Center (BSC)
Technical University of Catalonia (UPC)
Barcelona, Spain
eduard@ac.upc.edu

***Abstract*—Compute and memory access units are two of the most important resources to appropriately manage in current and future multi–/many–core architectures. Memory bandwidth and computational capacity need to be exploited in a combined way to achieve the best system performance. Coarse–grain multi–threading, also known as temporal multi–threading (TMT), is a well known technique that improves overall resource utilization by time–multiplexing the execution of a reduced number of hardware threads that are switched in case of a high–latency event, such as a memory miss. Hence, the processor does not stall on memory misses and the number of in–fly memory operations is increased, improving the overall processor resource utilization.**

In this paper, we propose a software–based implementation of TMT that supports and unbounded number of threads and enables a flexible combination of multiple computational kernels. Our TMT implementation is based on micro–threads that combine fast cooperative and preemptive context switches to overcome some intrinsic limitations of current TMT hardware implementations, such as the reduced and fixed number of hardware threads available. Our proposal is demonstrated with an implementation on the Cell/B.E. which is evaluated using heterogeneous mixes of memory–/CPU–bound kernels. Experimental results show how the proposed technique reduce the execution time of several benchmarks by up to 78%.

I. Introduction

The improvement of absolute processor performance has been one of the main research topics since the development of the first processor, but nowadays, absolute performance is only half of the picture. The goal of current processor designs is to maximize performance/Watt. This trade-off between performance and power consumption is of paramount importance to allow the integration of a large number of cores on the same chip. Thus, features such as out of order execution, simultaneous multi–threading (SMT) or large coherent caches, which were usual on single, dual or quad core designs, have been revisited or even left out on novel many–core designs such as the Cell/B.E. [1] or the Intel SCC [2]. All these techniques improve resource utilization and performance but at the cost of increasing processor complexity and energy consumption. In this paper we will revisit coarse-grain multi–threading, also known as, temporal multi–threading (TMT) [3] in the era of many-core processors.

To achieve a good overall processor efficiency both compute and memory units should be fully utilized. TMT allows to time–multiplex the execution of a small and fixed number of hardware threads (usually 2 or 4) on the same core. Thus, when one thread stalls due to a high–latency event, such as a memory miss, the core can quickly resume the execution of another hardware thread. The net effect is that not only the compute unit utilization increases, but also the memory unit utilization, which has to manage an increasing number of memory operations. However, the actual processor resource utilization is always limited by the intrinsic nature of the computational kernel being executed. If the kernel is memory–bound the compute unit is underutilized, likewise, if the kernel is CPU–bound the memory unit is underutilized.

The availability of multi/many-core designs has stimulated the development of more and more parallel computational kernels that can benefit from many–core processors. It is not unusual to find out systems running workloads composed of multiple parallel kernels with different computational characteristics, some of them memory–bound but others CPU–bound. Task–based programming models, such as [4], are good examples of these environments. This scenario opens up the possibility of mixing up memory–bound kernels and CPU–bound kernels to achieve an optimal processor utilization. To this end, we need to execute an adequate combination of memory–bound and CPU–bound kernel instances that lead to an optimal resource utilization of the CPU and bandwidth resources. Recent hardware TMT implementations, such as the Itanium–2 Montecito [5], only support a fixed and small number of hardware threads. As we will see in the evaluation Section, a fixed and small number of threads do not provide the required flexibility to combine multiple computational kernels in a optimal way and, on the other hand, the number of hardware threads cannot unlimitedly increase due to processor's complexity and power consumption constrains.

In this paper we advocate for a software–based TMT implementation that can dynamically grow or reduce the number of available threads depending on the characteristics of the running workload to optimize the processor's resource utilization. To evaluate the effectiveness of our proposal we have extended our open source micro–threading library [6] for the Cell/B.E. processor to implement a software–based TMT. We chose the Cell/B.E. as the evaluation platform because specific features of this processor, namely the explicit management of the Cell/B.E. processors' local storage, allows us to test our proposal on an actual system, while on other cache–based architectures we can only experiment via simulation.

978-1-4673-2371-0/12/$31.00 ©2012 IEEE

The singular architecture of the Cell/B.E. provides high computational power and memory bandwidth with a simple and efficient hardware design that overcomes the memory wall [7] problem with the use of software–managed local memories. All the load/store operations to/from a local storage are explicitly managed by a memory control flow unit, which is decoupled from the processing unit, thus allowing an individual fine–grained management of both compute and memory bandwidth resources. Hence, to obtain the best performance of the Cell/B.E. processor the programmer needs to explicitly manage data transfers between main memory and each local storage so that computation and data transfer are overlapped. The use of software–managed local memories instead of traditional caches is the most distinctive characteristic of this architecture. In fact, this key feature allows the implementation of a software–based TMT on the Cell/B.E.

The main contribution of this paper is to propose and evaluate the potential and flexibility of software–based TMT, which supports a larger number of threads than any hardware–based TMT implementation. Thus increasing performance and resource utilization for several workloads. Although the experimental evaluation has been done on the Cell/B.E. we believe that this technique and the results obtained can be extrapolated to cache–based processors. However, some hardware modifications discusses on Section VI must be done for this technique to work on cache–based processors.

The rest of the paper is organized as follows: Section II presents the theoretical background about resource utilization. Section III describes the software-only TMT implementation for the Cell/B.E. architecture. Section IV describes the experimental environment, the methodology used, and the obtained results. Section V discusses the related work. Finally, VI concludes the paper and outlines some future work.

II. Theoretical Background

In this section we explain the concepts and techniques used and evaluated in the rest of the paper. First, subsection II-A identifies and describes the main properties of computational kernels, software–based TMT and buffering schemes. Subsection II-B introduces the concept of hardware resource utilization and shows that an interleaved execution of memory–bound and CPU–bound kernels always improves the overall hardware resource utilization.

A. Computational Kernels and Buffering Schemes

In this paper we use the term computational kernel, or kernel for short, to refer to any algorithm that follows the *get–compute–put* pattern. In the *get* step some data located in the main memory is transferred to the local storage/cache. In the *compute* step, an arbitrary computation on this data is performed. Finally, in the *put* step, the result of the *compute* step is transferred from the local storage/cache to the main memory. This *get–compute–put* sequence is iteratively repeated until all the input data has been processed. A wide spectrum of algorithms follow this simple pattern.

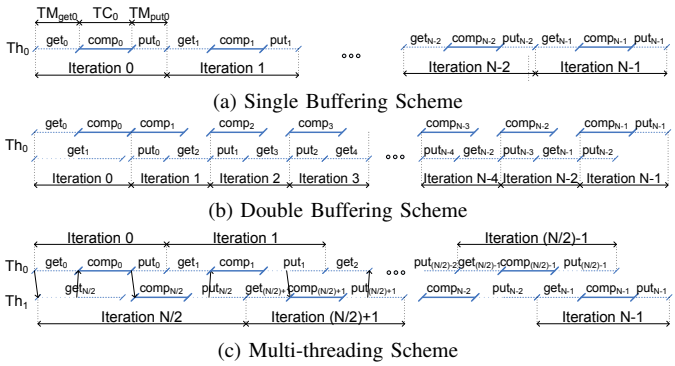


Fig. 1: Thread execution diagrams of the single buffering (a), double buffering (b) and multi–threading (c) schemes

A naive algorithm implementation that follows the *get–compute–put* pattern does not usually achieve the optimal performance because there is a dependency between the *get* and *compute* steps and also between the *compute* and *put* steps that serializes the execution of the three steps and limits the processor performance. Double buffering is a well–known technique used to overlap the *compute* stage of the current iteration with both the *put* step of the previous iterations and the *get* step of the next iteration (on cache–based processors prefetching techniques have the same objective). The main drawback of the double buffering scheme is that it is only feasible if there are no cross–iteration dependencies. Recent works on micro–threading [8][6] have applied this technique to architectures with software–managed local storage to overlap the *compute* steps of one thread with the *get* and *put* steps of other threads and vice-versa.

Figure 1 shows the execution of the same computational kernel with single–buffering 1a, double–buffering 1b and multi–threading 1c. The continuous line represents the active utilization of the compute resources, while the horizontal dashed line represents the use of the memory bandwidth resources. As can be observed in Figure 1a, the single-buffering scheme is the most inefficient, because it is unable to overlap computation and memory transfer times. On the other hand the double–buffering and the multi–threading schemes can simultaneously use both resources, although each one following a different approach. The double–buffering scheme is based on the predictability of the next memory access, while the multi–threading scheme exploits the intrinsic parallelism of the executing kernel. Hence, the double–buffering scheme is only applicable when the access pattern of the kernel is known in advance. Oppositely, the multi–threaded scheme is based on the premise that the kernel work can be split in sub-parts, that is usually the case for applications designed to run on multiple computational cores. The rest of the paper is based on the multi–threading scheme (refer to [6] for a detailed comparison between double–buffering and multi–threading schemes).

Computational kernels can be classified based on their resource utilization as memory–bound, CPU–bound or well–balanced. In the first case, performance is limited by the

memory bandwidth between the processor and the memory. In the second case, the limiting factor is the processor's compute power. In the last case, performance is equally limited by both compute and memory bandwidth resources. Equations 1a and 1b define the data transfer and computation times (T_M and T_C), respectively. A kernel is memory–bound if, and only if, $T_M > T_C$, CPU–bound if, and only if, $T_M < T_C$ and well–balanced if, and only if, $T_M = T_C$. Hence, the execution time of a kernel that overlaps computation an data transfer times is the highest between T_C and T_M, as shown in equation 1c. In summary, T_M measures both the *get* and *put* steps, while T_C measures the *compute* step (which includes the time spent in the computational kernel, as well as, the time required to perform a thread context–switch).

$$T_M = \sum_{i}^{N-1} T_{M_i}, \qquad T_{M_i} = T_{M_{get_i}} + T_{M_{put_i}} \tag{1a}$$

$$T_C = \sum_{i}^{N-1} T_{C_i} \tag{1b}$$

$$T = max(T_C,\ T_M) \tag{1c}$$

B. Hardware Resource Utilization

Memory bandwidth and computing capacity are the two main resources available on a computer system. In this paper we designate memory bandwidth utilization and compute power utilization with the Greek letters α and β, respectively. Both α and β are measured with dimension–less units in the range [0..1]. The memory bandwidth utilization (α) is the time required to transfer all input data to/from main memory (T_M) divided by the total execution time (T). Likewise, the CPU utilization (β) is all the computation time (T_C) divided by the total execution time (T). Finally, we define the overall hardware resource utilization, designated by the Greek letter ρ, as the product of α * β as summarized in equation 2. In this way, the weight of both resources are proportional.

$$\alpha = \frac{T_M}{T} \quad \beta = \frac{T_C}{T} \quad \rho = \alpha * \beta \tag{2}$$

Equation 3 summarizes the relation between resource utilization (ρ), T, T_M and T_C. Notice that if $T_M \neq T_C$ the resource utilization is always sub-optimal ($\rho < 1$).

$$T_M > T_C \Rightarrow \alpha = 1,\ \beta = \frac{T_C}{T_M}, \rho = \alpha * \beta = \frac{T_C}{T_M} < 1 \tag{3a}$$

$$T_M < T_C \Rightarrow \beta = 1,\ \alpha = \frac{T_M}{T_C}, \rho = \alpha * \beta = \frac{T_M}{T_C} < 1 \tag{3b}$$

$$T_M = T_C \Rightarrow \alpha = 1,\ \beta = 1,\ \rho = 1 \tag{3c}$$

Equation 3c shows that for a well–balanced kernel the resource utilization (ρ) is optimal, because both compute capacity and memory bandwidth resources are fully utilized. In contrast, equations 3a and 3b show a sub–optimal resource utilization in both memory–bound and CPU–bound kernels. The goal of this Section is to prove that overall resource utilization can be improved if the execution of a memory–bound kernel (K_m) and a CPU–bound (K_c) kernel is interleaved. The interleaved execution of both kernels is denoted by $K_m \parallel K_c$, while the non interleaved execution of both kernels is denoted by $K_m \sim K_c$. On the interleaved scenario a pool of threads run iterations of both kernels concurrently, on the non–interleaved scenario, a pool of threads first run all the iterations of the first kernel followed by all the iterations of the second kernel. To prove our hypothesis, we first calculate the resource utilization of an non interleaved execution of K_m and K_c, denoted by $\rho(K_m \sim K_c)$. Hence, the resource utilization is $\rho(K_m) * T(K_m) + \rho(K_c) * T(K_c)$ divided by the total execution time $T(K_m \sim K_c) = T(K_m) + T(K_c)$. Equation 4 shows the result of this expression simplified.

TABLE I: A synthetic case study

kernel	T_C	T_M	T	α	β	ρ
K_m	3	6	6	1	$\frac{1}{2}$	$\frac{1}{2}$
K_c	6	4	6	$\frac{4}{6}$	1	$\frac{4}{6}$
$K_m \sim K_c$	9	10	12	$\frac{10}{12}$	$\frac{9}{12}$	$\frac{10}{16}$
$K_m \parallel K_c$	9	10	10	1	$\frac{9}{10}$	$\frac{9}{10}$

$$\rho(K_m \sim K_c) = \frac{\rho(K_m) * T(K_m) + \rho(K_c) * T(K_c)}{T(K_m) + T(K_c)} = \frac{T_C(K_m) + T_M(K_c)}{T_M(K_m) + T_C(K_c)} \tag{4}$$

$$\left(\frac{T_C(K_m)+T_C(K_c)}{T_M(K_m)+T_M(K_c)} > \frac{T_C(K_m)+T_M(K_c)}{T_M(K_m)+T_C(K_c)}\right) \wedge \left(\frac{T_M(K_m)+T_M(K_c)}{T_C(K_m)+T_C(K_c)} > \frac{T_C(K_m)+T_M(K_c)}{T_M(K_m)+T_C(K_c)}\right) \Rightarrow \rho(K_m \parallel K_c) > \rho(K_m \sim K_c) \tag{5}$$

The interleaved execution of K_m and K_c is denoted by $K_m \parallel K_c$, thus, $T_C(K_m \parallel K_c) = T_C(K_m) + T_C(K_c)$ and $T_M(K_m \parallel K_c) = T_M(K_m) + T_M(K_c)$. The resulting execution of $T_C(K_m \parallel K_c)$ is either memory–bound, CPU–bound or well–balanced. If it is well–balanced then the resource utilization is optimal, which implies $\rho(K_m \parallel K_c) > \rho(K_m \sim K_c)$. If the resulting execution is memory–bound then $\alpha = 1$ and $\rho = \beta = \frac{T_C(K_m)+T_C(K_c)}{T_M(K_m)+T_M(K_c)}$. Otherwise, it is CPU–bound and then $\beta = 1$ and $\rho = \alpha = \frac{T_M(K_m)+T_M(K_c)}{T_C(K_m)+T_C(K_c)}$. As shown in Equation 5 in both situations $\rho(K_m \parallel K_c) > \rho(K_m \sim K_c)$. Hence, we can conclude that $\rho(K_m \parallel K_c)$ is always greater than $\rho(K_m \sim K_c)$. Notice that the previous proof does not give us any insight about the expected resource utilization improvement, because it depends on the actual execution time of each kernel.

Table I shows a synthetic case of resource utilization for a memory–bound kernel (K_m) and a CPU–bound kernel (K_c), which use two threads to overlap computation and data transfer times. In the same Table we can observe how the individual execution of both kernels ($K_m \sim K_c$) has a worse resource utilization than its combined execution ($K_m \parallel K_c$). Figures 2a and 2b depict the individual resource utilization of K_m and K_c, while Figure 2c shows the combined execution of

both kernels ($K_m \parallel K_c$), using two micro–threads in each case. It is worth nothing that the combined execution of both kernels is still suboptimal, because the resulting combined execution is memory bound. Thus, to further improve the resource utilization we should combine ($K_m \parallel K_c$) with a CPU–bound kernel. We can repeat this process iteratively until we achieve the optimal resource utilization ($\rho = 1$) or we reach the thread limit. Notice that the number of context switches is only determined by the number of kernel iterations executed, because there is exactly one thread context switch at the end of each kernel iteration. Hence, an increase in the number of threads does not affect the total number of thread's context switches performed, i.e., there is no additional overhead if the number of threads are increased.

In the same way we have proved that the combined execution of a memory–bound kernel with a CPU–bound kernel results in a better resource utilization, the merge of two kernels with same characteristics (both memory–bound, CPU–bound or well–balanced) has the same resource utilization (ρ) as the individual execution of both kernels. Thus, interleaving the execution of several kernels always lead to a resource utilization greater or equal than the individual execution of these kernels.

III. Software–based Temporal Multi-threading (sTMT) on the Cell/B.E.

This section summarizes the main characteristics of the Cell/B.E. architecture and describes the design, implementation, and extensions to our open source CellMT library [6], which is used to implement a flexible sTMT. This library enables the concurrent execution of several computational kernels on the Cell/B.E.

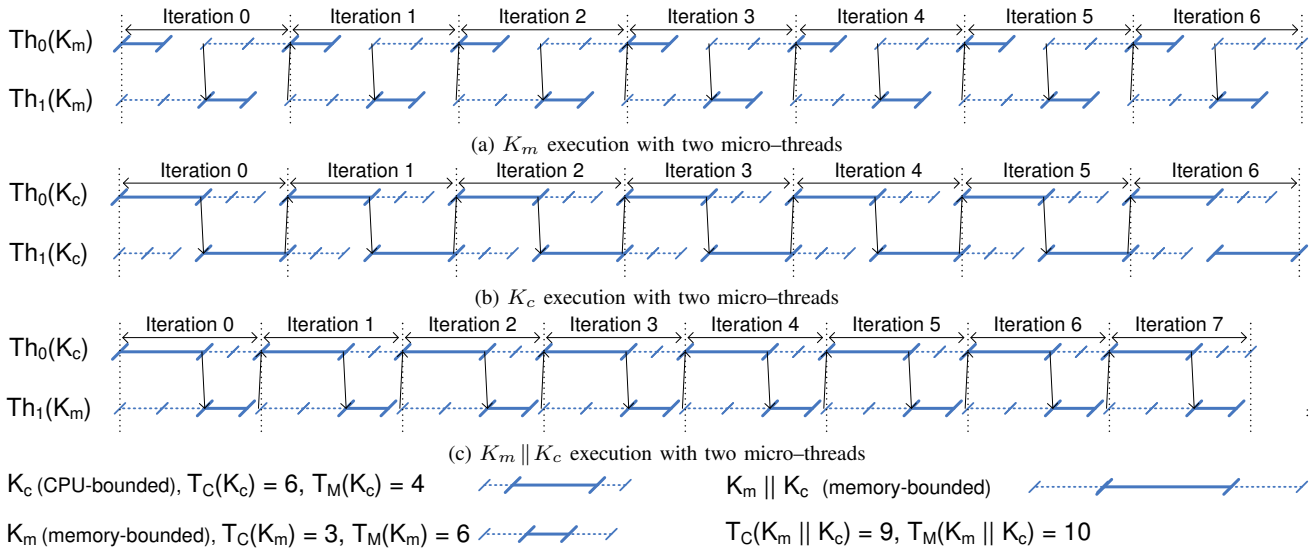
A. Cell/B.E. Architecture

The Cell/B.E. is an heterogeneous processor composed of one main general purpose processor (PPU) and eight specialized cores (SPUs) with software–managed local stores. Each SPU only has direct access to their own local storage, so before any data computation may take place the involved data must be explicitly transferred from the main memory to the local store. The data are transferred with asynchronous DMA operations managed by a specialized memory flow controller (MFC), thus the SPU is able to keep computing while up to 16 DMA operations are in-fly. The ability to issue asynchronous DMA operations between the local stores and the main memory enables an efficient overlapping of computation time and data transfer time, but at the cost of higher software complexity. The most widely used techniques to exploit this processor feature are double–buffering and micro–threading schemes, which have been already discussed in detail in Section II-A.

B. Software–based TMT design and implementation

The CellMT library provides a micro–threaded environment for the SPUs of the Cell/B.E. processor, which enables the concurrent execution of several threads inside each SPU. The use of multiple threads in the same SPU naturally overlaps the computation time of one thread with the data transfer time of other threads, without increasing the code complexity. As shown in [6] and [4] the micro–threaded approach can outperform a hand-coded double buffering scheme, with speedups from 0.96x to 3.2x, while maintaining the complexity of the code comparable to a naive simple–buffering scheme. It is worth noting that none of the previous studies have evaluated the potential of this technique to combine different computational kernels.

The micro–threading library is implemented on a core library that provides all the features and flexibility required to run complex multi-threaded application inside the SPUs. This core library provides a low-level threading API that can be directly called from the SPU application code, although most applications only need to change the blocking DMA



(a) K_m execution with two micro–threads

(b) K_c execution with two micro–threads

(c) $K_m \parallel K_c$ execution with two micro–threads

Fig. 2: Individual execution of K_m (a) and K_c (b) and a combined execution of both kernels (c)

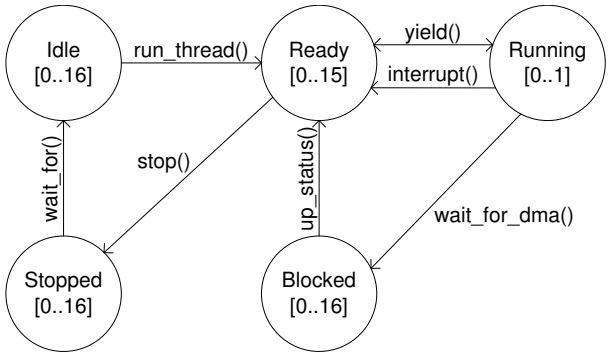


Fig. 3: Thread state diagram of our software–based TMT.

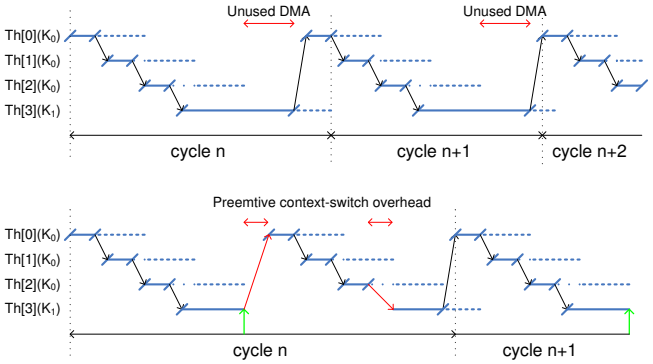


Fig. 4: Diagram of threads execution with interrupts disabled (top) and interrupts enabled (bottom).

wait calls to the waiting function provided by our library. The *wait_for_dma()* and *yield()* functions provided by this library save the current thread state and relinquish the control of the processor to the next ready thread following a round–robin scheduling policy. The cost of the cooperative context switch is around 150 cycles (47 ns), independently of the number of active threads.

Figure 3 shows the state diagram of the up to 16 threads supported by the CellMT library. Initially, there is one thread on the *Running* state and 15 threads on the *Idle* state. When a running thread calls the *run_thread()* function, one of the threads in the *Idle* state changes to the *Ready* state. The thread in the *Running* state can be put on the *Blocked* state with a call to the *wait_for_dma()* function. An interrupt or a call to the *yield()* function move the current running thread to the *Ready* state. When a thread ends, it is put on the *Stopped* state until another thread call the *wait_for()* function, which move the stopped thread to the *Idle* state again. For every context switch, the non–blocking function *update_state()* is called to check the status of the in–fly DMA operations and updates the state of the blocked threads to the *Ready* state if required.

The CellMT library has a constant context–switch time that enables the execution of up to 16 concurrent threads without degrading performance. This property allows to increase the number of threads executing a kernel until the optimal overlapping of computation and data transfer time is achieved. Nevertheless, increasing the number of threads behind this point will not further improve the overall system resource utilization as explained in the next Section.

1) Preemptive context switching: We have extended the CellMT library to also support preemptive context switches. This feature is essential to mimic how hardware TMT works and also to achieve the best performance in some workloads, as we will see in the evaluation Section. The preemptive context switch support allows the preemption of the active thread, but the cost of this operation doubles the cost of a cooperative context switch. We initially implement a timer based approach to preempt the current thread once a specific time has elapsed. The main problem with this approach is that we need to carefully chose the timer value, depending on the kernel characteristics and the current processor and memory load. Hence a inadequate timer or a unexpected change in the workload will cause unnecessary overheads. Fortunately, the Cell/B.E. can fire interrupts based on DMA events. Based on this capability, the implemented policy interrupts the running thread when all the active DMA operations have been completed. This way, the processor is never interrupted while there are in–fly DMA operations. Only when all memory operations have finished, the running thread is preempted to allow other threads to initiate more DMA operations. This method is very robust, because it automatically adapts to the workload executed and the current processor load. Figure 4 shows two simple diagrams to illustrate the difference between the interrupt-disabled policy (top) and the interrupt-enabled policy (bottom). Both diagrams represent the execution of three instances of a memory–bound kernel (K_0) and one instance of a CPU-bound kernel (K_1). On one hand, if interrupts are disabled, the running thread only yields the processor when it need to wait for the completion of a DMA operation. Hence, a high CPU–bound kernel can starve the execution of other memory–bounded kernels, leaving memory bandwidth underutilized. On the other hand, if interrupts are enabled, the running thread can be interrupted when all the in–fly DMA operations are completed and the execution of one blocked thread will be resumed. Notice on the bottom half of Figure 4 that the overhead of a preemptive context switch is bigger than the overhead of a cooperative context switch, as we need to save and restore all the processor's registers when the thread is preempted and resumed respectively. Additionally, the absolute number of thread context–switches performed also increase proportionally to the number of interruptions done. Thus, this interrupt policy is only useful if the overhead of the additional preemptive context switches makes up for the increase of DMA operations. Section IV-C focus on the performance evaluation of this interrupt policy.

IV. Experimental Evaluation

In this Section we describe the methodology and benchmarks used to validate the techniques and the concepts presented in previous Sections. All the experiments have been conducted on a QS20 blade powered with two Cell/B.E proces-

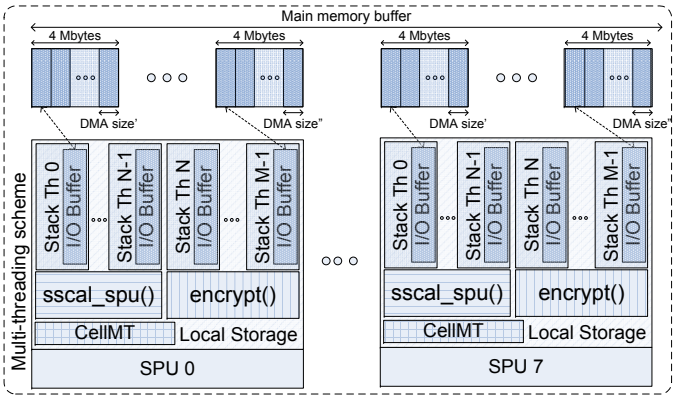


Fig. 5: Our benchmarking framework configured with the *sscal* and the *encrypt* kernels.

sors. All the software has been developed with the IBM SDK 3.1. Subsection IV-A presents the synthetic benchmark and the computational kernels used in the experiments. Subsection IV-B describes the methodology used to select the experiments and configuration parameters tested. Finally subsection IV-C presents and discusses the obtained results.

A. Synthetic Benchmark

To evaluate the feasibility and performance of the concepts and the techniques explained in previous Sections we have developed a framework to systematically evaluate the performance of various computational kernels. This framework allows the concurrent execution of up to 16 computational kernels on each SPU. However, for the sake of clarity, all the configurations evaluated have been done with only two different computational kernels, but each one with several threads: the memory–bound *sscal* kernel and the CPU–bound *encrypt* kernel. *Sscal* is a kernel provided by the standard IBM BLAS Level 1 library that scales a vector by a constant. On the other hand, the *encrypt* kernel, provided by the standard IBM cryptographic library, encrypts a data–block with the AES encryption algorithm in ECB mode. Both kernels can work with a input block of arbitrary size, which allows the evaluation of each kernel with different DMA transfer sizes.

The framework can be parametrized to execute each kernel with different parameters, such as number of threads, DMA buffer size or number of iterations to perform. Figure 5 shows the synthetic benchmark configured with the *sscal* and *encrypt* kernels to run with N and M-N threads, respectively. Each micro-thread has an associated circular input/output buffer of 4 MBytes in main memory. The benchmark can be configured to keep running until a number of iterations are completed or a configurable period of time is elapsed. Each thread reads the input buffer in chunks of *DMA size* bytes and yields the processor to another thread. When the input data is ready on the SPU local storage, the thread processes the input data with the suitable kernel and puts back the result to the main memory. The implementation of our software–based TMT library allows the framework to gather execution statistics,

TABLE II: *Sscal* and *encrypt* measured T_M (a), T_C (b), optimal resource utilization (c) and performance (d). The results marked in grey are used on Section IV-B

Kernel \ DMA size	256	512	1024	2048	4096	8192	16384
sscal	196	392	784	1570	3144	6297	12592
encrypt	196	392	784	1570	3144	6297	12592

(a) T_M (ns). Include *get* and *put* steps

sscal	105	123	147	194	289	479	859
encrypt	1178	2211	4279	8414	16692	33233	66317

(b) T_C (ns). Include *compute* step and context switch time (46 ns)

sscal	0.537	0.314	0.187	0.124	0.092	0.076	0.068
encrypt	0.167	0.177	0.183	0.187	0.188	0.189	0.190

(c) Theoretical maximum resource utilization (ρ)

sscal	40733	20423	10203	5096	2545	1270	635
encrypt	6794	3618	1870	951	479	241	121

(d) Theoretical maximum *get-compute-put* (iterations/ms)

such as number of iterations completed, total execution time, total wait time and number of bytes transferred for each thread. In order to keep the overhead of the statistics minimal, we only record the total number of iterations performed by each kernel and the total execution time.

The first experiment is to measure the exact T_M and T_C of both kernels, running on eight SPUs. In order to measure T_M, which includes both *get()* and *put()* steps, we have coded an octuple–buffering scheme, which only reads and immediately writes back data from/to main memory without doing any data processing. Table II(a) shows the measured times for the seven DMA transfer sizes evaluated. T_M only depends on the DMA transfer length so, as expected, the results for both kernels are identical. On the other hand, to measure T_C without any DMA influence, we have implemented a simple loop that process each block of data with the corresponding kernel, but issuing empty DMA operations. The measurements in Table II(b) include the measured *compute* step time and the cost of one thread context switch (46 ns).

The data in Table II(c) shows the optimal resource utilization that can be achieved with the individual execution of each kernel (i.e. when the compute and DMA transfer times are fully overlapped). This data is derived from the measured T_M and T_C, using Equations 2 and 3. Likewise, Table II(d) shows the optimal (maximum) throughput of an individual kernel execution, which is calculated taking into account the eight SPUs available on the Cell/B.E. As expected, $T_M > T_C$ is true for all the configurations of the memory-bound *sscal* kernel, likewise $T_M < T_C$ is true, for all the configurations of the CPU–bound *encrypt* kernel. With the number of iterations done by each kernel and the data already measured of Table II it is straightforward to calculate the α, β and ρ of a combined kernel execution.

B. Methodology

In the previous Subsection we have measured the optimal performance and resource utilization of the *sscal* and *encrypt*

kernels when executed individually. These measurements will be used as a baseline to compare the performance of the combined execution of both kernels. On a combined kernel execution, the first step is to decide how many threads should execute each kernel in order to maximize performance and resource utilization. Hence, the goal is maximize Equation 2, which can be rewritten as Equation 6.

$$\rho = \alpha * \beta = \frac{T_M * T_C}{T^2} \tag{6}$$

We need to extend the previous Equation to an arbitrary number of kernels. If we consider *N* different kernels, lets say from K_0 to K_{N-1} and each kernel is executed by X_i threads, the values of T_M and T_C are $\sum_i^{N-1} X_i * T_M(K_i)$ and $\sum_i^{N-1} X_i * T_C(K_i)$ respectively. If we recall Equation 1c, T is defined as the max between T_M and T_C. This statement is true when all the threads execute the same kernel, but when threads execute different kernels, we need to consider the case in which the execution of one kernel iteration can not be overlapped by the execution time of the rest of threads, resulting in the augmented expression that is the denominator of equation 7. This equation can be solved as a linear–fractional optimization problem that satisfies the restrictions of Equation 8. The result of this optimization problem is the number of threads (X_i) for each kernel *i* that result in an optimal hardware resource utilization.

$$\rho = \frac{\left(\sum_i^{N-1} X_i * T_M(K_i))\right) * \left(\sum_i^{N-1} X_i * T_C(K_i))\right)}{max\left(\begin{array}{l}\sum_i^{N-1} X_i * T_M(K_i),\ \sum_i^{N-1} X_i * T_C(K_i), \\ max\left(X_i * (T_M(K_i) + T_C(K_i))\right)\end{array}\right)^2} \tag{7}$$

$$Constraints:\ \forall_i X_i \geq 1, \sum_i^{N-1} X_i \leq 16 \tag{8}$$

We have solved this optimization problem for a combined execution of *sscal* and *encrypt* kernels when configured with a DMA transfer of 4096 and 512 bytes respectively. Table II shows in grey the values of T_M and T_C of each kernel. The result of solving this optimization problem, which must satisfy the restrictions of Equation 8, leads to a configuration of five and eight threads to achieve an optimal ρ of 0.985.

The previous formula is also used to devise the number of threads required to optimally execute individual kernels. For instance, the individual optimal resource utilization of the *sscal* and *encrypt* kernels configured to work with a DMA transfer of 4096 and 512 bytes respectively, which is marked in grey on Table II(c), can be achieved in both cases with only two threads.

We have run a combined execution of both kernels with the optimal thread configuration calculated on the previous

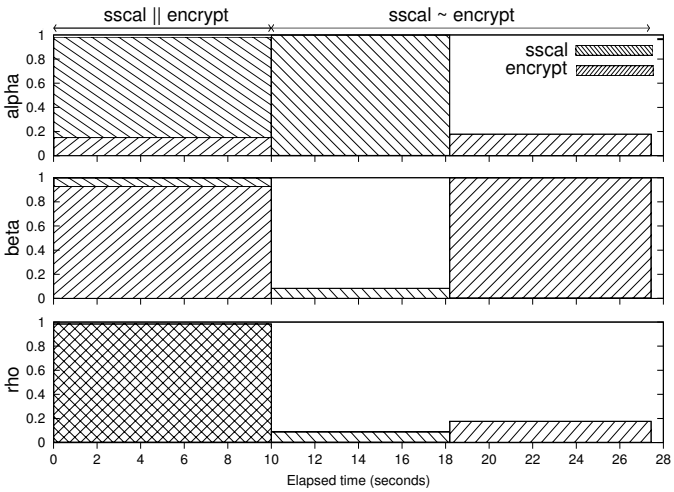


Fig. 6: Detailed resource utilization (α, β and ρ) of a combined execution of *sscal* and *encrypt* (left), versus an individual execution of *sscal* (middle) and *encrypt* (right).

paragraph for 10 seconds, recording the aggregate number of iterations done by each kernel to calculate α, β and ρ. Then, we execute *sscal* and *encrypt* kernels individually (each one with two threads) until the respective number of iterations done by the combined kernel execution are completed. Figure 6 shows α, β and ρ for the combined execution of both kernels, the individual execution of the *sscal* kernel, and finally the individual execution of the *encrypt* kernel. The combined execution of *sscal* and *encrypt* kernels have a performance of 2201 and 3345 iter/ms respectively, which results in a ρ of 0.983. On the other hand, the individual execution of the *sscal* and *encrypt* kernels results in a performance of 2545 and 3618 iter/ms respectively, which corresponds to the optimal individual performance shown in grey on Table II(d). The individual execution of *sscal* and *encrypt* are 8.19 and 9.25 seconds respectively, for a total of 17.44 seconds, while the combined execution performs the same amount of work in only 10 seconds, which result in a speedup of 1.44x.

Equation 7 is also useful to calculate the exact proportion of iterations that should execute each kernel to achieve the optimal performance. We only need to set one of the X_i to 1 and remove the restrictions that the rest of X_i must be integers. Table III shows the result of solving the Equation 7 for 49 combinations of *sscal* and *encrypt* kernels. For instance, the optimal resource utilization for the previous combined execution of *sscal* and *encrypt* can only be achieved with 1.57 iterations of the *encrypt* kernel for each *sscal* iteration. In the previous experiment we have used 5 and 8 threads to approximate the 1.57 optimal proportion (8/5 = 1.6). Likewise, if we configure both kernels with the same number of thread, we can also achieve the optimal ρ scheduling 1.57 times more frequently the threads of kernel *encrypt* than the threads of kernel *sscal*. As we will see in the next Section, these techniques are not always effective. The grey cell of Table III shows all the combinations that can not be optimally executed

TABLE III: Thread proportion to achieve an optimal ρ.

encrypt \ sscal	256	512	1024	2048	4096	8192	16384
256	10.78 1	3.65 1	1.54 1	1 1.40	1 2.91	1 5.93	1 11.96
512	19.99 1	6.77 1	2.85 1	1.32 1	1 1.57	1 3.20	1 6.45
1024	38.39 1	13.00 1	5.48 1	2.54 1	1.22 1	1 1.66	1 3.36
2048	75.19 1	25.46 1	10.74 1	4.98 1	2.40 1	1.18 1	1 1.71
4096	148.84 1	50.40 1	21.25 1	9.85 1	4.75 1	2.33 1	1.15 1.00
8192	295.91 1	100.19 1	42.25 1	19.58 1	9.44 1	4.63 1	2.30 1.00
16384	590.21 1	199.84 1	84.28 1	39.06 1	18.82 1	9.23 1	4.58 1.00

by the lack of available threads and other issues that we address in the following Section. It is worth noting that only 9 of the 49 kernel combinations of Table III can achieve the optimal resource utilization and performance with only two threads, which is the usual number of hardware threads present on current processors. If we double the number of threads from two to four, then 18 of the 49 kernel combinations could achieve the optimal performance, but 31 kernel combinations will still need many more threads to perform optimally. This data makes clear that the number of hardware threads of current multi–processors is not enough to achieve the optimal performance in most situations.

C. Experimental Results

On the previous Subsection we have studied how a combined execution of a memory–bound kernel and a CPU–bound kernel can dramatically improve performance and overall resource utilization. Although, a combined execution always result in better performance than a individual execution, to obtain the best performance we need to select the appropriate number of threads to execute each kernel. Thus, a complete software–based TMT implementation should include an heuristic to dynamically decide the number of threads required to execute each kernel based on lightweight profiling information, however the implementation of this heuristic is out of the scope of this paper. For the purpose of this paper, and to avoid the variability that an heuristic will introduce, we have pre–calculated the optimal number of threads used in each experiment. In this way, we can clearly identify the best performance that can be achieved with this technique.

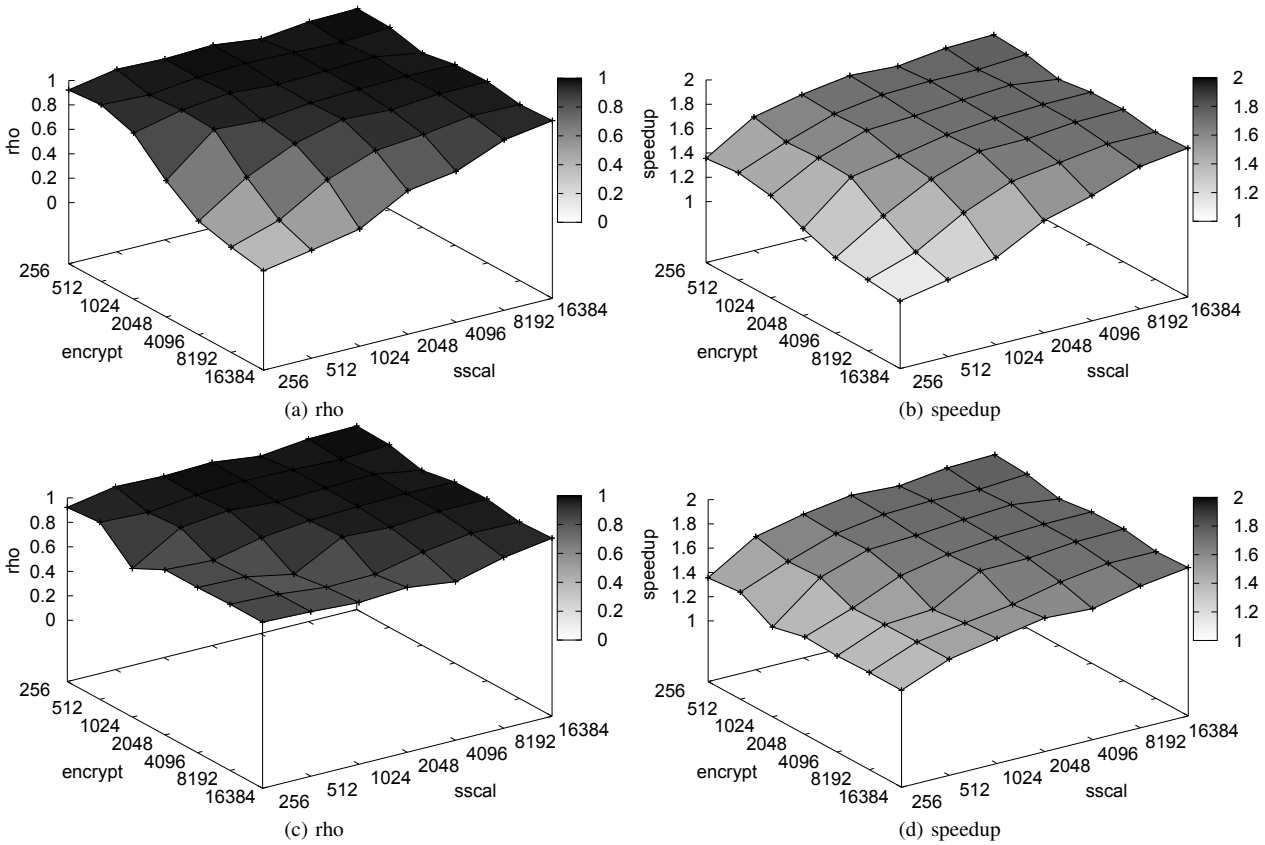
With the data on Table III and Equation 7 we have calculated, for each configuration evaluated, the optimal number of threads used to execute the *sscal* and *encrypt* kernels. Each configuration have been executed for 30 seconds, recording the number of iterations completed by all the threads of each kernel. With the result of these executions and the data of Table II as a baseline, we can easily calculate the resource utilization and the performance of each configuration. Figure 7a shows the resource utilization for all the configurations evaluated, while Figure 7b shows the speedup versus the individual kernel executions of Table II(d). Both Figures have the *x* and *y* axis in log-scale and have a similar shape, as resource utilization and speedup are two metrics strongly correlated. The Figures can be clearly divided in two zones, which correspond with the white and grey cell of Table III. On one hand, all the *white* configurations have a high ρ and a speedup that range from 0.92 and 1.35 to 0.99 and 1.78, respectively. On the other hand, the *grey* configurations have a ρ and a speedup that progressively decrease from around 0.8 and 1.3 to 0.31 and 1.06, respectively. With Table III as a reference, ρ and performance decrease progressively from top to bottom and right to left. It is worth to remark that even the combined kernel executions marked in grey have better performance and resource utilization than the individual execution of the two kernels shown in Table II.

The limited performance of the *grey* configurations is caused by the huge difference between $T_C(encrypt) \gg T_M(sscal)$. For example, in the case of the fist configuration in light grey, to fully overlap the *compute* step of the encrypt kernel we will need at least 39 threads executing an instance of the *sscal* kernel, but our current implementation is limited to 16 threads. Although, we could easily extend our current implementation to support more than 16 threads or schedule 39 *sscal* iterations (threads) for each *encrypt* iteration (thread), another limitation will still limit the actual performance. The current Cell/BE processor only support up to 16 concurrent DMA operations, and hence, if we try to issue a DMA operation when there are already 16 DMA operations on–the–fly, the processor stalls until one of the pending DMA operations completes. This is a hardware limitation that we cannot easily overcome.

The only feasible solution to solve the above–mentioned problem is to split the large *compute* step into smaller parts using the preemptive mode implemented in our software–based TMT library, as detailed in Subsection III-B. Our preemption policy only interrupts the running thread if there are other threads waiting for the completion of a DMA operation. In this scenario, only when all the in–fly DMA operations are completed the running thread is preempted and one of the other threads is resumed.

Figures 7c and 7d show resource utilization and speedup with our interrupt policy enabled, respectively. The shape of both Figures are similar to the shape of Figures 7a and 7b, the only difference is that the ρ and speedup of the *grey* configurations have significantly improved. The lowest utilization of 0.31 have increased to 0.85, which is still lower than the average ρ of 0.95 of the *white* configurations due to the additional overhead of the preemptive context switches. Although the cost of a preemptive context switch doubles the cost of a cooperative context switch, the absolute cost is still very low. Only using a cooperative scheduling policy keeps the number of context–switches proportional to the number of kernel iterations executed, however the introduction of a preemptive scheduling policy can dramatically increase the number of context–switches performed per kernel iteration. For instance, a encrypt *compute* step of the configuration 256/16384 is preempted on average 61.4 times before it can complete.

The configurations in light grey are the first to be affected by the thread shortage, but at the same time, the increase on



(a) rho (b) speedup

(c) rho (d) speedup

Fig. 7: Resource utilization and speedup of the combined kernel executions with interrupts disabled (top) and enabled (bottom).

performance is not enough to compensate the overhead of the additional preemptive context–switches. The point is that for these configurations, the cost of increasing the number of context switch in terms of β do not compensate the gains in terms of α. These configurations will be the most benefited from hardware support to accelerate preemptive context–switches.

Although each of the evaluated configurations only combine two different kernels with a fixed computational and memory bandwidth demand, the results can be extrapolated to any complex applications that can be split in several phases, each one with a fixed computational and memory bandwidth demand. We have evaluated the resource utilization of other memory–bound and CPU–bound kernel combinations that include more than two kernels. The results obtained have the same shape as the results presented for the *sscal* and *encrypt* kernels. The only noticeable difference is the size of the grey zone on Table III, which depends on the difference between T_M and T_C of memory–bound and CPU–bound kernels, respectively.

V. Related Work

Simultaneous Multi-threading (SMT) [3] is a well–known and widely–used technique to interleave the execution of various kernels. This technique is implemented with hardware threads that share the compute and memory resource available on the processor. Hardware threads enable the concurrent execution of different kernels, thus improving hardware resource utilization. This technique allows a very fine–grained interleave of kernel execution at the instruction level. Everything is done in hardware and totally transparent to the kernels executed. The main limitation of this technique is that each hardware thread requires its own register file, which increase the processor complexity and power requirement. Hence, current implementations of SMT have a small (2 or 4) and fixed number of hardware threads. Coarse–grain multi–threading, also know as temporal multi–threading (TMT) [3], is also a well–known technique implemented in cache–based processors such as the Itanium–2 Montecito [5]. This technique increases the overall processor utilization by time multiplexing the execution of a reduced number of hardware threads that are switched in case of a high-latency event, such as a memory miss. Hence, the processor does not stall on memory misses and the number of in–fly memory operations increases, improving the overall resource utilization in a transparent way. TMT implementations also have a fixed and reduced number of hardware threads due to complexity and power constrains.

As far as we know, there are no hardware implementation of TMT for processors with software–managed local storage's such as the Cell/B.E. On these kind of architectures, the high–latency DMA memory operations are explicitly encoded in the program code and can thus be easily identified and managed. This property enables the implementation of micro–threading

techniques without any explicit hardware support. Previous works, such as [6], [8], have successfully used micro–threads on the Cell/B.E. to hide high latency memory access and overlap computation and data transfer times. These techniques based on micro–threads usually outperform double-buffering [9], [10], software cache [11], and pre–fetching techniques [12]. It is worth noting that none of the previous work has studied the implications and potential of combining different workloads (memory–bound/CPU–bound). Hence the previous works achieve the best possible resource utilization for a given kernel, but not the optimal hardware resource utilization, as is the case of this work.

Micro–threading techniques execute each kernel as a burst of instructions between DMA operations, because the context switches between threads are cooperative. Consequently, if there are burst of computation too larger (the case of *encrypt* kernel with a large DMA transfer size), some threads could starve the execution of another threads, leaving the memory bandwidth resource underutilized. Our sTMT implementation solve this problem with preemptive context switches driven by a novel interrupt policy based on the actual processor resource utilization.

VI. Conclusions and Future Work

In this paper we have proved the feasibility and usefulness of software–based temporal multi–threading (sTMT) to optimize system resource utilization on a Cell/B.E. processor. Software–based TMT enables the combined execution of different kernels in a efficient way. The experimental results, that combine memory– and CPU–bound kernels, show speedups that range from 1.30x to 1.78x. Moreover, the hardware resource utilization is also improved, with most of the results obtained close to the optimal hardware resource utilization. Even the most unbalanced kernel combinations achieve a high resource utilization thanks to our novel thread context switch policy based on the dynamic hardware resource utilization of the processor. To back up our claims, we have formalized the hardware resource utilization of an arbitrary workload set, and verified it with two actual kernels. For these two computational kernels we have evaluated an exhaustive number of configurations varying the computational intensity and DMA length. The evaluation of other pairs of memory-/CPU-bound kernels led to similar results that support our claims.

It is worth noting that to achieve the best performance and resources utilization for a wide range of kernel combinations, a large number of threads is required to fully overlap computation and data transfer times. Current hardware implementations of TMT on cache-based processors have a reduced and fixed number of hardware threads, which can perform a preemptive thread context switch in a small number of cycles (15 cycles on a Itanium–2 Montecito). Despite the impressive performance of hardware context switch, our evaluation show that in most situations to achieve the best resource utilization it is necessary a larger number of threads. On the other hand, our software–based implementation of TMT is very competitive, but hardware support to accelerate cooperative and, specially, preemptive context switches will further increases the applicability and performance of this technique in even more situations. As a future work, we plan to investigate hybrid implementations of TMT to combine the flexibility of the software approach with the context-switch performance of the hardware approach.

Another topic that we will further investigate is the feasibility and applicability of the techniques presented in this work on cache–based processors. Although, all the experimental evaluation has been done on a processor with a software-managed local storage, we believe that the ideas and techniques presented in this paper are also relevant to cache-based processors. However, some major modifications should be considered to port software–based TMT to a cache–based processor. To name a few, the ability to raise interrupts to notify software about high-latency events (ex: L3 miss), or the modification of the OS thread scheduling to explicitly take advantage of software–based TMT.

Acknowledgements

This work was supported by the Ministry of Science and Innovation of Spain (CICYT) [TIN-2007-60625].

References

[1] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its first implementation," *IBM DeveloperWorks*, November 2005.

[2] J. H. et al., "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS," in *Proceedings of the 2010 International Solid-State Circuits Conference. CA, USA*, 2010, pp. 108–109.

[3] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, 2003.

[4] R. Ferrer, V. Beltran, M. González, X. Martorell, and E. Ayguadé, "Achieving high memory performance from heterogeneous architectures with the sarc programming model," in *MEDEA '09: Proceedings of the 10th MEDEA workshop on MEmory performance*. New York, NY, USA: ACM, 2009, pp. 15–21.

[5] C. McNairy and R. Bhatia, "Montecito: A dual-core, dual-thread itanium processor," *IEEE Micro*, vol. 25, pp. 10–20, 2005.

[6] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé, "CellMT: A Cooperative Multithreading Library for the Cell/B.E." in *HiPC '09: Proceedings of the 16th annual IEEE International Conference on High Performance Computing, Cochin, India*, 2009, pp. 245–253.

[7] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, March 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[8] M. F. Ahmed, R. Ammar, and S. Rajasekaran, "Novel micro-threading framework on the cell broadband engine," in *ISCC '09: Proceedings of the IEEE Symposium on Computers and Communications*, 2009, pp. 570–575.

[9] T. Chen, Z. Sura, K. O'Brien, and J. K. O'Brien, "Optimizing the Use of Static Buffers for DMA on a CELL Chip," in *LCPC*. Springer Berlin / Heidelberg, 2006, pp. 314–329.

[10] Jonathan Bartlett, "Programming high-performance applications on the Cell/B.E. processor," April 2007, http://www.ibm.com/developerworks/library/pa-linuxps3-6/.

[11] T. Chen, T. Zhang, Z. Sura, and M. Gonzalez, "Prefetching irregular references for software cache on cell," in *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008, pp. 155–164.

[12] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos, "Dma-based prefetching for i/o-intensive workloads on the cell architecture," in *CF '08: Proceedings of the 2008 conference on Computing frontiers*. New York, NY, USA: ACM, 2008, pp. 23–32.