Introducing hardware TLP support in the Cell processor

Roberto Giorgi, Zdravko Popovic, Nikola Puzovic
Department of Information Engineering, University of Siena, Italy
http://www.dii.unisi.it/~ {giorgi | popovic | puzovic}

***Abstract*— The focus of our study is the support for fine/medium grained Thread Level Parallelism (TLP) by using a hardware scheduling unit and relying on existing simple cores. Simple cores are grouped into clusters in order to provide a scalable solution. As a proof of concept, we use an implementation based on the Cell Broadband Engine (CBE). Cell is a multiprocessor on a chip developed by Sony, Toshiba and IBM that contains one general purpose core and eight coprocessor elements that accelerate the multimedia and vector processing. The aim of this paper is to present a possible implementation of DTA (Decoupled Threaded Architecture) that is based on the Cell processor, while keeping the scalability of the original DTA.**

***Index Terms*—thread level parallelism, multicore processors**

I. INTRODUCTION

CURRENT trends in computer architecture have brought to the market multicore architectures to utilize the ever increasing number of transistors, which are available on a single chip. Future general purpose processors are expected to have a number of cores at least an order of magnitude bigger than it is now [1]. A good representative of this trend is the Cell processor [2, 3] that was created by the cooperation of Sony, Toshiba and IBM. This is a heterogeneous multicore architecture with a single PowerPC core and eight SIMD cores.

However, programming for multicore architectures is still an open issue and research topic in both academia and industry. TLP is one opportunity that can be exploited by the multicore architectures. Each core can run a separate thread, and inside them it can exploit even other levels of parallelism (ILP, DLP). Even though multicore peak computational power can be very high, programming them is not a trivial problem. In the case of Cell, for instance, the programmer defines tasks statically in order to let them execute on SIMD cores as well as the data transfers for them (to and from main memory). Previously, we experimented successfully with the DTA architecture [4], which is able to efficiently exploit fine/medium grained TLP that is available in the programs. The hardware proposed in DTA could provide mechanisms for efficient and scalable thread scheduling and synchronization also in existing multicore architecture like the Cell.

In this paper, we present an implementation of DTA support for the Cell architecture. The aim of this hardware solution is to exploit available TLP. The idea is to map DTA cores to Cell SPE cores (see Section I.A) and introduce simple additional hardware for thread scheduling and synchronization as in DTA [4], thus making it dynamic.

The rest of the paper is organized as follows. In the rest of this section we present a brief review of the Cell architecture and then a brief introduction to the DTA architecture. Section II describes the implementation of DTA hardware support in Cell architecture. In Section III, we show initial results obtained on this platform. Section IV gives the conclusions.

A. The Cell Processor

The Cell processor is a multiprocessor on a chip that combines one general purpose PowerPC Processing Element (PPE) with eight SIMD accelerators (called SPE - Synergistic Processor Element) [2, 3].

The PPE consists of one PPU (PowerPC Processor Unit) and caches (L1 + L2). The PPU is a dual-issue in-order processor based on the IBM PowerPC architecture. In typical usage of the Cell processor, the PPU is responsible for running the operating system and for managing the SPEs.

The SPE contains one Synergistic Processor Unit (SPU), Memory Flow Controller (MFC) and a 256KB of Local Store (LS) memory. The SPU is a dual-issue in-order processor that works only with data that are present in its Local Store and accesses main memory and other Local Stores through MFC controller. Each MFC contains a Memory Management Unit (MMU) for address translation and a DMA controller.

The Element Interconnect Bus (EIB) in the Cell processor connects the PPE, SPEs, off-chip memory and I/O devices. It has one address bus and four 16-byte data rings for data transfers.

B. DTA Architecture

DTA [4] is an architecture that derives from previously proposed execution models like SDF (Scheduled Data-Flow) [5] and related ones like TAM (Threaded Abstract Machine) [6]. Threads communicate with each other in a producer-consumer fashion, and a thread will start its execution only when all its data are ready in a local memory (which we also call *Frame Memory*). Each thread has assigned a number of input data it needs (synchronization count - SC). SC is decremented every time a data is stored in a local memory of a thread. When it reaches zero, the thread is ready for the execution. In this way we have a dataflow-like communication between threads - dataflow at thread level. An example of thread synchronization in DTA is shown in Fig. 1. This example shows one thread (fib(n)) that forks another three threads (fib(n-1), fib(n-2) and sum). It also shows the data dependencies among these threads, and the synchronization counts that are associated with them.

DTA is a proposal for a many-core architecture. Processing elements (PE) in DTA are grouped into nodes, where

dimension of each node is determined with a constraint that each PE must be reachable in one cycle. This is done in order to be able to deal with the wire delay problem. On the other hand, communication between nodes is slower, and interconnection network is more complex, but this is necessary to achieve scalability as the available number of transistors increases.

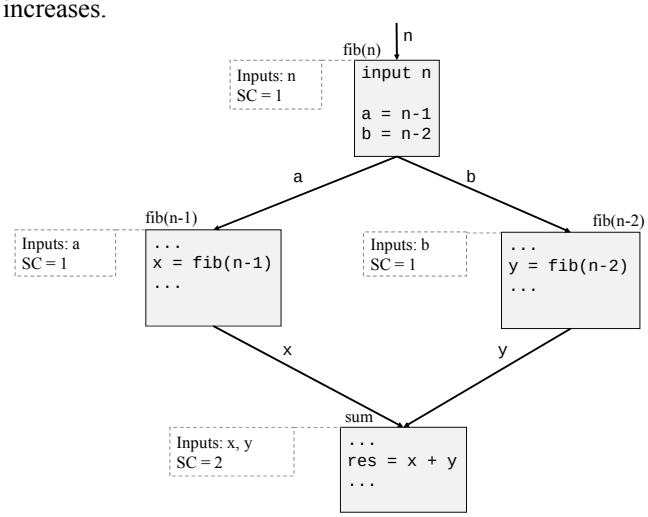

Fig. 1. An example of thread synchronization in DTA with recursive Fibonacci

There are three specific hardware structures that DTA uses plus a minimum support in the instruction set. First one is a frame memory. It is a fast memory associated with each processor used for storing the thread's input data. Processor can access this memory in one cycle and we assume no stalls in the pipeline in the case of frame memory access. Second one is a local scheduler element (LSE). Each PE contains one LSE that manages local frames and forwards request for resources to the DSE (Distributed Scheduler Element). DSE is a third DTA specific hardware structure (one per node). It is responsible for distributing the workload between processors in the node, and for forwarding it to other nodes. DSE, together with all the LSEs, makes the hardware scheduling unit of DTA architecture, which provides functionality of dynamic distribution of the workload between processors. Schedulers communicate among themselves by sending messages. These messages can signal the allocation of the new frame (FALLOC request and response messages), releasing a frame (FFREE message) and storing the data in remote frames (further details in paper [4]).

II. DTA SUPPORT IN CELL

We assume that the compiler or programmer identifies portions of the programs that we name TLP activities. These threads execute only on SPUs, and PPU will be responsible for sending the TLP activities to the SPUs by forking the first thread. After that, the DTA-specific hardware will take care of the execution of these activities. When the last DTA thread completes its execution, it will notify the PPU that the execution is completed. In order to support this, a minimal DTA support must include the following:

- Distributed Scheduling Element (DSE) is needed in order to distribute the threads and to perform the load balancing among SPUs.
- Local Scheduling Element (LSE) together with the Frame Memory is needed in order to manage the threads that are running on a single SPU.
- Support for DTA-specific instructions in the SPUs.

The Frame Memory can be mapped straightforwardly into the SPE's Local Store. The rest of this section briefly explains the CellSim simulator, and then presents the DTA-specific modifications that are needed in the simulator. The modules that are implemented correspond to the actual hardware that would be needed in the Cell processor with DTA support. For now, we have implemented only the version with one node (one PPU and eight SPUs).

A. Cell

We use the CellSim in order to model the Cell processor [7, 8]. The high level structure that it models is shown in Fig. 2. It is a modular simulator implemented using UNISIM framework [9]. Although the Cell processor has one PPU and eight SPUs, the CellSim simulator has been built with the support for even more PPUs and SPUs (all in one node) in order to create diverse configurations.

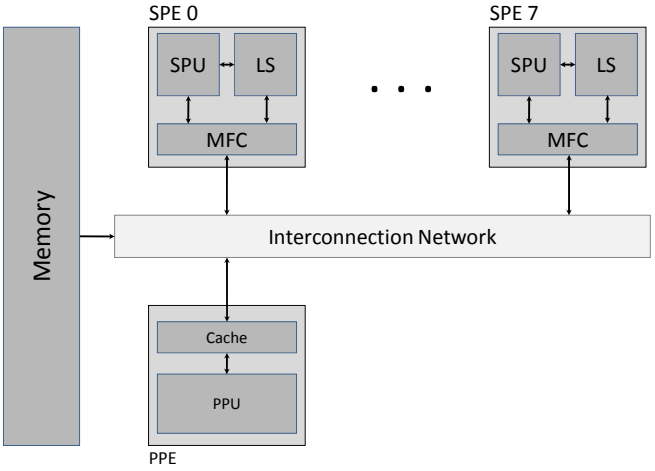

Fig. 2. The modular structure of the CellSim simulator: the configuration with 1 PPE and 8 SPEs. This configuration models the Cell processor.

For each hardware component of the Cell processor (PPU, SPU, LS, EIB) there is a module that simulates it. The modules that simulate the PPU and SPU are instruction set simulators (ISS) and their parameters, such as the number of instructions that they can execute per cycle can be changed. All memories in the simulator (main memory and LS) are simulated by the same type of module whose latency and the number of ports can be configured. Instead of modeling the EIB of the Cell processor, CellSim utilizes a K-Bus topology for the interconnection network (IN) module. In this way, the network is scalable, and more than one PPU and eight SPUs can be attached to it. Correspondence with the real Cell Processor has been validated previously [7, 8].

B. DTA specific modifications

The additional modules that are needed to simulate the DTA architecture in CellSim are shown in Fig. 3 and Fig. 4. Since

DSE is responsible for distributing the threads and for load balancing between all nodes, it is placed near to the PPU so that it can be reached easily from each SPU. The arbiter that was added between DSE and the bus is a simple combinatorial module that distributes the transactions from the bus either to the cache or to the DSE (Fig. 3). The DSE has a table with N entries (N=8 in this case) that contains the number of frames assigned to each SPE. It contains also combinatorial logic to select the SPE that will accept the new frame based on the same algorithm as in [4]. On the other hand, LSE is responsible for handling the scheduling of threads that are executing on one SPU [4], and it is placed inside the SPE.

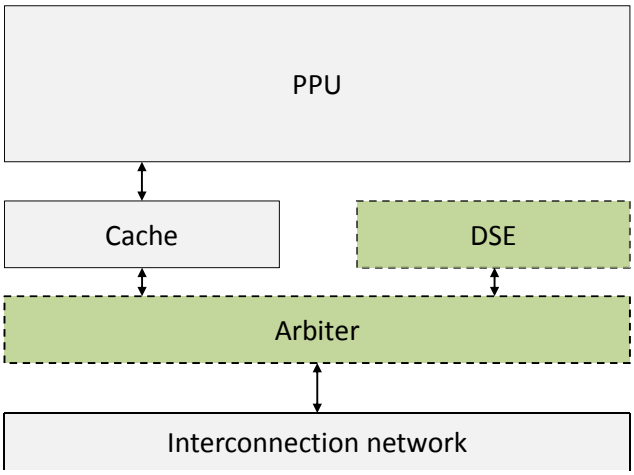

Fig. 3. Integration of the DSE into the CellSim

The LSE is implemented as a part of the MFC (Fig. 4). In this way, it can use the capabilities of the MFC to manage the local store and communicate with the rest of the system.

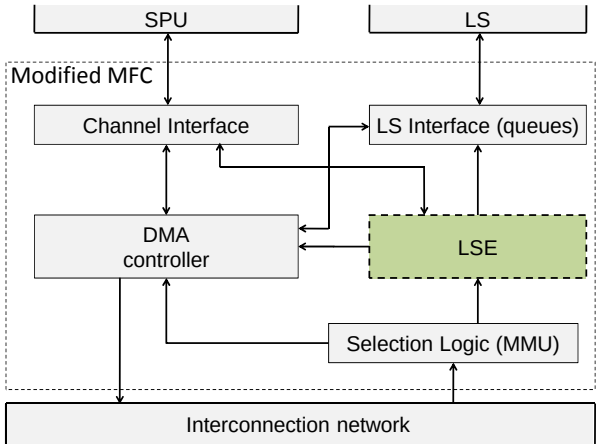

Fig. 4. Integration of the LSE into the CellSim's SPE module

The LSE contains queues for managing threads and frames in the SPE: the *Pre-Load Queue* and the *Waiting Table* (detailed in previous work [4, 5]) and briefly recalled below.

The *Pre-Load Queue* holds the continuations of the threads (their instruction pointers and the address of their frames) that are ready for execution (SC == 0). It is implemented as a circular queue with a configurable capacity.

The *Waiting Table* contains the continuations that have been scheduled but are not yet ready for execution (SC ≠ 0). It is implemented as an associative array with one entry per frame. The array is indexed by frame number, and each entry contains the IP and SC. Frame buffer is a circular queue that contains the addresses of each free frame in the SPE. The size of these tables and queues can be configured by passing different parameters to the simulator. The physical structure of the local store has remained unchanged as in the CellSim and part of it is used to implement the Frame Memory. LSE is intercepting accesses that are coming to LS from the interconnection network, and handles them in case when they are relevant to the execution of the DTA program (writes to the frame memory and control space of the LSE).

In order for the SPUs to execute DTA code, a tool has been developed that translates programs from the native DTA assembly to the Cell SPU assembly language. For the majority of the native DTA instructions the appropriate SPE instructions have been determined. However, for the instructions that were dealing with DTA threads this was not possible since they required DTA support. Hence, some new instructions have been added for the SPU ISA essentially for handling thread lifetime (such as *falloc* for allocating a new frame, *ffree* for deallocating frames, *tstop* that notifies the LSE that a thread has finished its execution) and frame memory accesses (such as *store* for storing data to other threads' frames and *load* for loading data from the current frame).

The Cell processor memory map consists of several regions that are dedicated to mapping the local stores, SPU control registers, PPU control registers, etc. The CellSim simulator implements this mapping as well. Main memory is mapped at the beginning of the address space. Then, the local stores of each SPU in the system are mapped. Registers of each PPU in the system are mapped after, and, at the end the MFC registers of each SPU. We have used this fact in order to be able to access and control the schedulers.

The control registers of the DSE are mapped in the space reserved for the registers of the corresponding PPU. The arbiter in front of DSE and PPU decides which transactions from the interconnection network are going to DSE and which are for PPU based on the destination address. These registers are used for communication with LSEs (for creating, destroying and dispatching threads) and with other DSEs (for load balancing among nodes in the future versions of the simulator). The control registers of each LSE are mapped inside the space reserved for the local store of the corresponding SPE. Since LSE is a part of the MFC, it will intercept the messages that are destined to it.

Each Local Store is partitioned as shown in the Fig. 5. The beginning of the LS is reserved for the code of the threads (space from address 0 to A_{code_end}). The local store also contains frames (n_{frames} frames, each of them with a fixed *number of entry-per-frame* called n_{epf}) that are starting from the address A_{frame_start}. After the frames, a control space is starting from the address $A_{control}$. These addresses are used by LSE to receive communication from the corresponding DSE. All these parameters can be configured in the simulator.

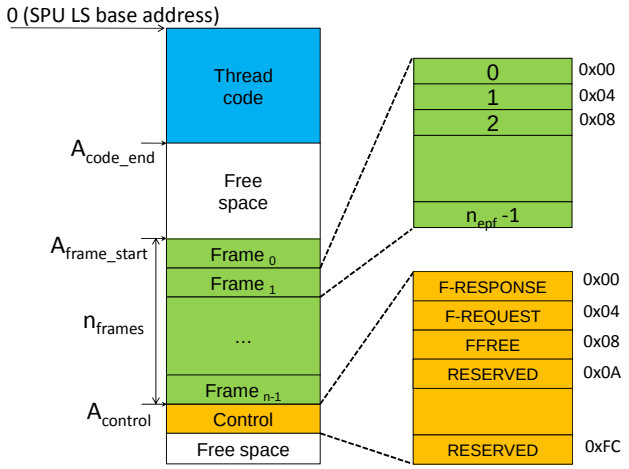

Fig. 5. DTA extension memory mapping inside one Local Store

III. EXPERIMENTAL RESULTS

A. Methodology

In order to verify the model and to obtain performance results we have used both native DTA simulator [4] and Cell-DTA simulator (in various configurations), which is a UNISIM extension [9] of the CellSim simulator [7, 8], and finally we used a Playstation-3 with a real Cell processor and Linux Fedora-8. The native DTA simulator is written to model an ideal DTA processor, and it models an ideal memory with no latencies. The pipelines that are used are simple, in-order pipelines. The interconnection among processors in the DTA is modeled as a standard bus in all the experiments that were performed.

For simulating the Cell with DTA extension, we extended the CellSim with the modules that are described in the previous section. In order to achieve a reasonable comparison with the native DTA simulator, we have considered two different configurations characterized by different values for memory latency (*ml*), and local store latency (*lsl*):

- Cell-DTA [ml=150, lsl=6] is the basic configuration where the latency of local store is 6 cycles and memory latency is 150 cycles. This configuration is used for modeling the realistic Cell processor.
- Cell-DTA [ml=1, lsl=1] is the ideal configuration where both local store and memory latencies are 1 cycle. This configuration was tested so we could compare more directly the results with the native DTA simulator.

For the initial tests we have used several simple programs (the first three also used in [5]):

-- Matrix multiply is a program that multiplies two square matrices of order *n* (where n is 32). In order to parallelize it, the matrices have been divided in 2, 4 or 8 parts and sent to different threads for processing. Hence, all tests with this program use 2, 4 or 8 processors.

-- Fibonacci is the recursive program for calculating Fibonacci numbers of *k* (where k is 10). It is a simple program that forks a lot of threads to execute recursive function calls. The main purpose of this benchmark is to test how the system handles a vast amount of small threads

-- Zoom is a program that zooms into one part of the input picture. It is parallelized by sending different parts of the picture to different threads, hence creating independent threads. Input is an *n* by *n* picture (where n is 64).

-- Bitcount (bitcnt) from the MiBench [10] suite is a program that counts bits in several different ways for a certain number of iterations (input parameter that is 10000 in our case). Parallelization is performed by unrolling both the main loop that calls different functions and loops inside each function that perform the iterations.

All these programs were hand-coded previously for the native DTA architecture, and then (automatically) translated with a tool (except for matrix multiply) in order to use the Cell SPU ISA with DTA extensions. In the original DTA version, all programs that were using memory were communicating with it directly, and in Cell-DTA versions we have implemented simple pre-fetching mechanism. All data that are needed by the thread were preloaded into the LS, and then read from the LS. Also, in order to give a fair comparison, we have optimized some of these benchmarks for Cell-DTA in order to eliminate the hazards (by moving instructions and by renaming registers).

B. Experimental Results

Even when compared with an "ideal" Cell-DTA configuration (ml=1, lsl=1) (Fig. 6), the native DTA has a substantial advantage memory is served within a pipeline cycle and without bus contention. We made these experiments in order to validate the DTA-Cell simulator against the original DTA. Since CellSim models a realistic Cell processor, it was impossible to completely bypass the queues and the bus. In cases of bitcnt, zoom and matrix multiply this comes from the fact that these benchmarks have memory accesses which are slower in Cell-DTA then in native DTA. The difference is more visible in the case of zoom and matrix multiply since they are memory-intensive benchmarks. When using Cell-DTA with memory latencies set to 1 the differences become smaller, but native DTA is still faster. This is due to the fact that SPU pipelines are modeled with dependencies, and even though the memory latency is small (1 in case of Cell-DTA [ml=1, lsl=1]), the memory access from the SPU needs to go through the bus and to wait in the queues, which wasn't the case in native DTA simulator. This is also the reason why Fibonacci is slower on Cell-DTA. This behavior was quite expected as the native DTA simulator was mainly built for testing the correctness of the execution model, TLP support and scalability, and not to assess the absolute validity of the execution time. Nevertheless, we need to work on optimization of our DTA mapping which is still in its infancy, and there is space for a more efficient implementation.

In fact, we can verify this also observing the average pipeline utilization for all benchmarks (Fig. 7). It can be seen that utilization is higher in case of native DTA than in case of Cell-DTA for all the benchmarks except for matrix multiply. This is also due to the fact that native DTA implements non-blocking frame allocation [4] (that is not available here), and that it uses perfect memory, so there are no pipeline stalls due

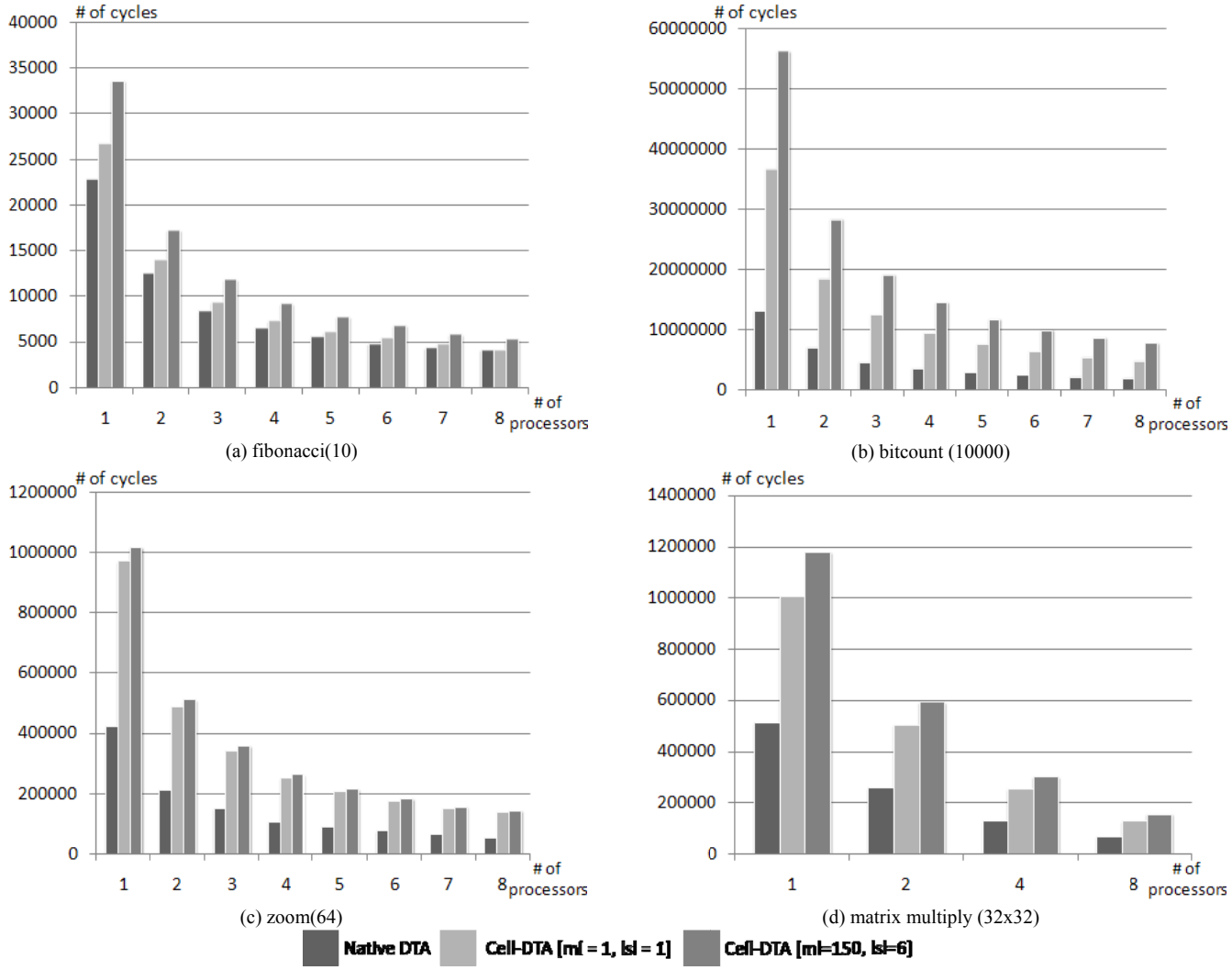

Fig. 6. Execution time for the four benchmarks. The Y-axis represents the execution time in cycles, and the X-axis represents the number of SPUs (PEs or processors in native DTA). In brackets, for each benchmark we report the input parameters that we used.

to memory or frame accesses. In case of matrix multiply, the benchmark is heavily optimized in the Cell version, hence the better utilization.

A quite interesting result is the scalability for the DTA extension to Cell. The results are comparable to the native DTA results (Fig. 8). The speedup has been calculated with respect to the execution time when running on only one SPU (or on one processor in case of native DTA). The obtained speedup is good, almost ideal in some cases. The worst speedup is obtained in the case of Fibonacci. In fact, this latter program generates a vast number of small threads, and the thread management becomes a significant factor. This is also the case for bitcnt, but the impact is lower.

C. *Comparison with the Cell*

All four programs have been implemented for the Cell architecture in order to compare it with the Cell-DTA. We have implemented the matrix multiply and zoom so that each of the eight SPUs calculates one part of the destination matrix. The bitcnt program is implemented so that each of the seven functions is statically allocated to one of the SPUs (one remains free). In case of execution on DTA-Cell, programs are always using all 8 SPUs that are available in the system.

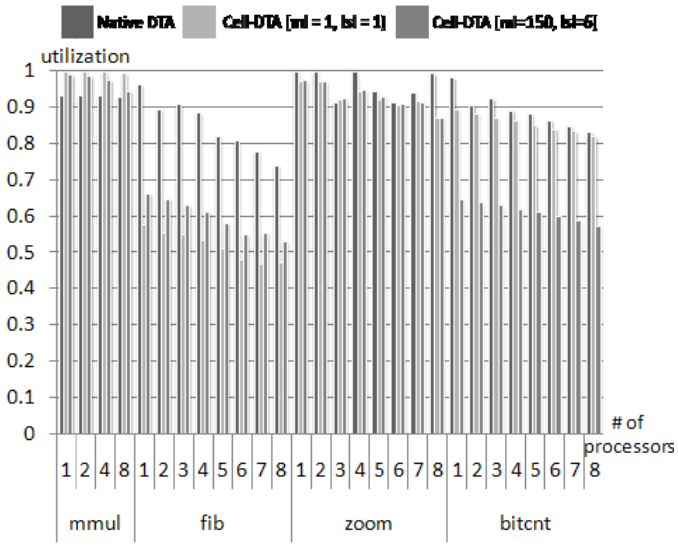

Fig. 7. Average pipeline usage of the four benchmarks. The Y-axis represents the average pipeline usage and the X-axis represents the number of SPUs (processors in native DTA).

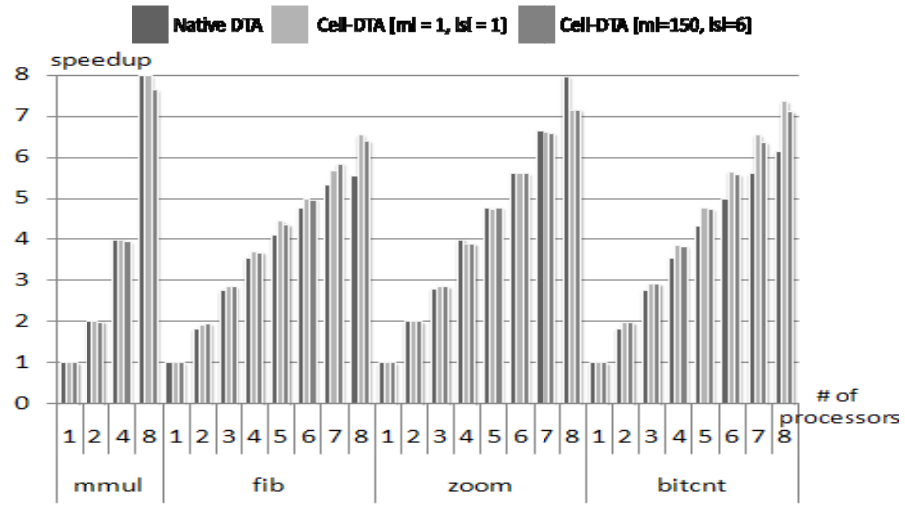

Fig. 8. Scalability of the four benchmarks. The Y-axis represents the speedup (baseline is the configuration with one SPU (processor)), and the X-axis represents the number of SPUs (processors in native DTA).

Fibonacci uses an algorithm that is recursive. Since the Cell programming model doesn't support non-trivial dynamic synchronization among SPUs, it would be difficult to implement it. Such implementation could lead to DTA-like software synchronization mechanisms which we will consider in our future work. Table I presents the comparison. We can see that in all cases the results are Cell-DTA outperforms the original Cell. In case of Fibonacci, the reason is obvious (one vs. eight SPUs) and in case of other programs the dynamic scheduling and synchronization of DTA are giving the advantage to Cell-DTA model.

TABLE I
CELL-DTA VS. NATIVE CELL EXECUTION*

Benchmark	Cell-DTA cycles	Cell cycles	Improvement
Fibonacci	5263	11561	54%
Bitcount	7945637	8196021	3%
MatrixMult.	154294	196908	21%
Zoom	142037	341381	58%

*The execution time in the table is given in the number of cycles obtained using CellSim, and improvement is calculated with respect to the execution time on a real Cell.

IV. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a hardware implementation of DTA TLP support for the Cell architecture. Initial results show that the solution is scalable, and that it improves the performance of the Cell processor. We need to use more benchmarks in order to further explore the feasibility of the DTA support. For mapping portion of source code that could benefit from TLP acceleration we plan to mark TLP activities using OpenMP directives [11].

V. ACKNOWLEDGEMENTS

The authors wish to thank Prof. Alex Ramirez and Prof. Xavier Martorell for giving us access to CellSim simulator and tools. We thank Sylvain Girbal and Olivier Temam for their assistance on using the UNISIM framework. This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6) and HiPEAC Network of Excellence (contract IST-004408 – FP6).

REFERENCES

1. Vangal, S., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Iyer, P., Singh, A., Jacob, T., Jain, S., Venkataraman, S., Hoskote, Y., Borkar, N.: An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. IEEE International Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers (2007) pp. 98-589.

2. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev **49** (2005) pp. 589-604.

3. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation: a performance view. IBM J. Res. Dev. **51** (2007) pp. 559-572.

4. Giorgi, R., Popovic, Z., Puzovic, N.: DTA-C : A Decoupled multi-Threaded Architecture for CMP Systems. 19th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2007, Gramado, Brasil (2007) pp. 263-270.

5. Kavi, K.M., Giorgi, R., Arul, J.: Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation. IEEE Transaction on Computers **50** (2001) pp. 834-846.

6. Culler, D.E., Goldstein, S.C., Schauser, K.E., Eicken, T.V.: TAM - a compiler controlled threaded abstract machine. J. Parallel Distrib. Comput. **18** (1993) pp. 347-370.

7. Cabarcas, F., Rico, A., Rodenas, D., Martorell, X., Ramirez, A., Ayguade, E.: CellSim: A Cell Processor Simulation Infrastructure. HiPEAC ACACES-2006, L'Aquila, Italy (2006) pp. 237-240.

8. Cabarcas, F., Rico, A., Rodenas, D., Martorell, X., Ramirez, A., Ayguade, E.: A Module-based Cell Processor Simulator. HiPEAC ACACES-2007, L'Aquila, Italy (2007) pp. 279-282.

9. August, D., Chang, J., Girbal, S., Gracia-Perez, D., Mouchard, G., Penry, D.A., Temam, O., Vachharajani, N.: UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. IEEE Comput. Archit. Lett. **6** (2007) 45-48.

10. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on - Volume 00. IEEE Computer Society, pp. 3-14.

11. The OpenMP API specification for parallel programming. (http://openmp.org)