Exploiting DMA to enable non-blocking execution in Decoupled Threaded Architecture

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

Abstract

DTA (Decoupled Threaded Architecture) is designed to exploit fine/medium grained Thread Level Parallelism (TLP) by using a distributed hardware scheduling unit and relying on existing simple cores (in-order pipelines, no branch predictors, no ROBs).

In DTA, the local variables and synchronization data are communicated via a fast frame memory. If the compiler can not remove global data accesses, the threads are excessively fragmented. Therefore, in this paper, we present an implementation of a pre-fetching mechanism (for global data) that complements the original DTA pre-load mechanism (for consumer-producer data patterns) with the aim of improving non-blocking execution of the threads.

Our implementation is based on an enhanced DMA mechanism to prefetch global data. We estimated the benefit and identified the required support of this proposed approach, in an initial implementation. In case of longer latency to access memory, our idea can reduce execution time greatly (i.e., 11x for the zoom benchmark on 8 processors) compared to the case of no-prefetching.

1. Introduction

Many-core architectures are currently an attractive solution for efficient and scalable utilization of the increasing number of transistors available on a single chip. These kinds of architectures are studied in both academia and industry. Recent examples of such architectures include IBM Cyclops [1], UltraSPARC T2 [2], TRIPS [3], Plurality [4] and Intel Polaris [5]. These architectures indicate that future general purpose processors are expected to have a number of cores at least an order of magnitude bigger than now.

DTA is a many-core multithreaded architecture for exploiting fine/medium grained TLP that is available in the programs [6][7], by providing mechanisms for scalable thread scheduling, synchronization and decoupling of their memory accesses. Local variables and synchronization data are communicated via a fast frame memory. However, accessing global data from any point in a program is possible, and might not be completely replaced by accesses to frame memory through the compiler analysis. Here, we present an implementation of a pre-fetching mechanism (for global data) that complements the DTA original pre-load mechanism (for consumer-producer data patterns) for fully non-blocking execution of the threads, although it can be generalized to other existing multithreaded architectures with few modifications. For example, it can be implemented on the Cell processor [8]. We used DTA implementation for the Cell as a framework for our experiments.

The rest of the paper is organized as follows. Section 2 recalls the DTA concept briefly. In Section 3, we present how DTA can benefit from pre-fetching combined with the pre-loading mechanism. In Section 4, we show current initial implementation and some results obtained on the reference platform. Finally we discuss the related work and give conclusions.

2. DTA Memory Model (DTA-MM) and required support

DTA [6][8] is an architecture that derives from previously proposed execution models like SDF [9] and related ones like TAM (Threaded Abstract Machine) [10] and EARTH (Efficient Architecture for Running THreads) [11]. DTA executes TLP activities, which are, by definition, portions of a program that exhibit thread level parallelism. The TLP activities are usually offloaded to DTA hardware where they are executed in parallel (for example, in the case of the Cell processor, TLP activities are offloaded by general purpose processor to SPEs, which execute them in parallel). In comparison with SDF, DTA adds the concept of clustering the resources in order to address wire-delay[6], a fully distributed scheduler and a communication protocol to implement exchange of synchronization messages [6], a major benefit being a scalable design. A TLP activity is further divided into threads, which can be executed in parallel. A threads are logically divided into smaller parts which are called code blocks (explained below). 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 frame memory. Each thread has a number of input data it needs (tracked by a per-thread Synchronization Counter

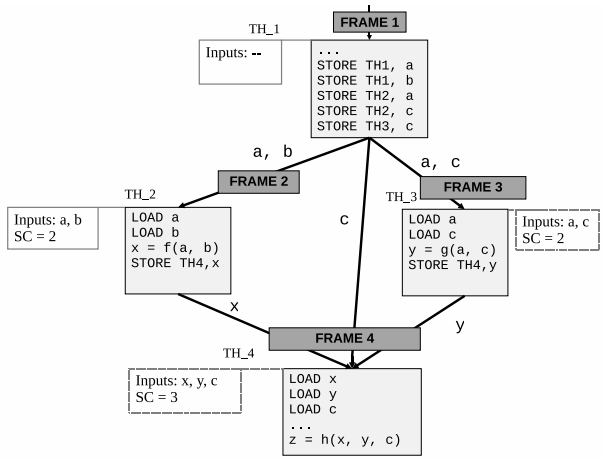


Figure 1. Simple example of thread communication in DTA. Data is sent to other threads using STORE instructions, and is read using LOAD instructions.

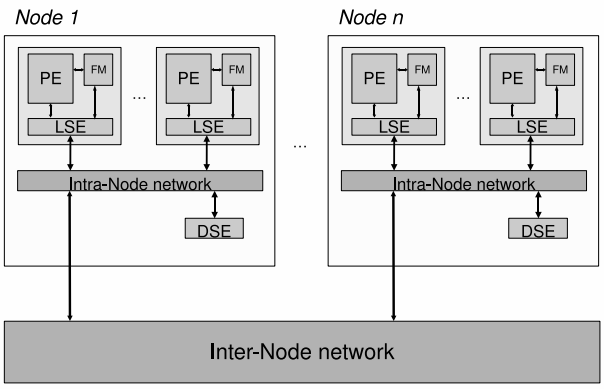


Figure 2. High level conceptual view of the DTA.

- SC). SC is decremented every time a datum is stored in a thread frame. When SC reaches zero, the corresponding thread is ready for the execution. In this way, we have a dataflow-like communication between threads - dataflow at thread level (Figure 1).

Processing Elements (PE) in DTA are grouped into nodes [6], where the node size is small enough to avoid the wire delay problem (e.g., all elements are synchronized with the same clock) [12]. On the other hand, the communication between nodes is slower as we rely on a more complex interconnection network. High level organization of DTA is shown in Figure 2.

There are three specific hardware structures that DTA uses plus a minimum support in the instruction set that was described before [6] and we recall it briefly here. The first one is a Frame Memory (FM). It is a local memory, associated with each processing element, that it is used

Table 1. Thread management instructions needed for execution of DTA programs

Name	Description
FALLOC	Creates a new frame by sending request to the scheduler.
FFREE	Releases a frame.
STOP	Notifies the LSE that thread has completed its execution.
LOAD	Loads the data from the frame of the current thread.
STORE	Stores the data to the frame of another thread.

for storing the thread's input data. Because of this specific memory, we have four instruction types for accessing memory: reading and writing to frame and main memory [6]. The second specific hardware structure is a Local Scheduler Element (LSE). Each PE contains one LSE that manages local frames and forwards requests for resources to a DSE (Distributed Scheduler Element). The 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 when internal resources are finished. DSEs from all nodes, together with all LSEs, constitute the (hardware) Distributed Scheduler (or DS) of DTA. Scheduler elements communicate among themselves by sending messages. These messages can signal the allocation of a new frame (FALLOC-Request and FALLOC-Response messages), releasing a frame (FFREE message) and storing the data in remote frames (further details are in the paper [6]). In order to manage the lifetime of each thread, we need a few additional instructions in the ISA (Table 1). In cases when the processing element does not have instructions for accessing main memory, they also need to be added (for example in Cell SPU).

Each DTA thread consists of three code blocks: pre-load (PL), execution (EX) and post-store (PS) [6][9]. Each code block is executed in a phase with a corresponding name (left part of Figure 3). In the pre-load phase, a thread reads all the necessary input data from the assigned frame and writes them into registers. In the execution phase, a thread manipulates the data placed in registers, and in post-store phase such thread writes its results into the frames of other threads.

Each DTA thread must pass through these states:

1) "Wait for a Frame" - A frame must be assigned to a thread before it can receive any data from other threads.
2) "Wait for stores" - Each thread needs to wait for all data to become ready in the frame. When the SC reaches zero the thread will pass into the next state.
3) "Ready" - All thread's data are ready in the frame memory and the thread can start as soon as the pipeline becomes available.
4) "Execution" - When the pipeline becomes available the thread will start its execution. The execution is divided into three parts:

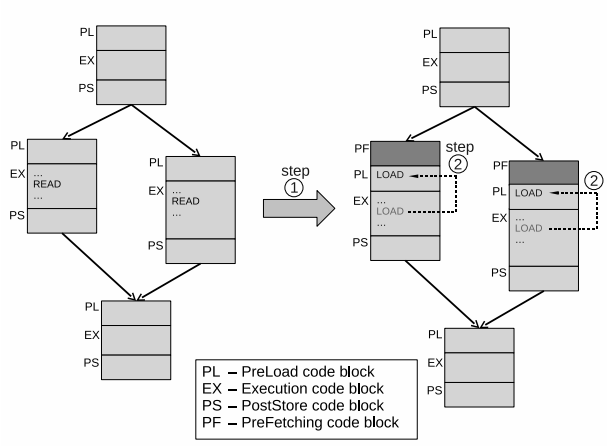


Figure 3. Lifetime of a thread and DMA related modifications (on the left) in order to support prefetching. Prefetching code blocks (PF) are added (step 1) and memory accesses are changed to access local memory (step 2).

a) Pre-load (PL) - In this phase data is read from the frame memory.
b) Execution (EX) - The code of the thread is executed.
c) Post-store (PS) - Sending data to other threads.

Although there must be no access to the frame memory in the execution phase, accesses to the main memory can still occur in the original DTA model. These accesses cause stalls in the pipeline (we name these accesses READ and WRITE for the sake of differentiating them from the accesses to frame memory - LOADs and STOREs). The prefetching mechanism presented in this paper is dealing with these stalls.

3. Making DTA-MM more efficient through DMA-based pre-fetching

In this section, we describe the general prefetching mechanism that we propose. Since it is possible to access global data structures from any point in a program, we focus here on those accesses. Some modifications are needed to DTA threads to prefetch global data (Figure 3). In the case when there are no main memory accesses, threads will remain unchanged as in the original DTA. However, when there are READ instructions, the compiler will modify the threads in order to add prefetching code. For each thread that contains a generic memory accesses, one new code block (PreFetch or PF code block, Figure 3) will be created that will initiate the transfer from main memory to local memory. In order to decouple the accesses to the main memory, all READ instructions that the thread contained are replaced by the compiler with LOAD instructions that now accesses the prefetched data in the local memory and are moved into the PL code block.

We could deal with accesses to global data by further partitioning threads, and this would have been possible also in the original DTA design [6], but in such case a possible adverse effect is the creation of too small threads (fine grained), typically when the program uses complex data structures such as arrays, linked-lists, data block pointers. For this reason, the prefetching mechanism has been designed to address these issues acting in two directions:

1) The prefetching can be tuned in order to prefetch not a single datum but more data depending on the situation; this operation is actually scheduled with a priority given by the Control-Data Flow Graph (CDFG) of the program;
2) The hardware is designed so that prefetch on such complex structures are facilitated.

This prefetching mechanism needs to be implemented both on the side of the compiler and in the architecture. The compiler has to recognize when a thread uses different types of global data, and be able to insert the prefetch instructions in the PreFetch (PF) code block. On the architectural side, we use the DMA unit in order to transfer the data from the main memory to the local memory. The changes that are needed in order to implement the prefetching (with respect to the original DTA) are as follows, and the rest of this section gives some details on how these changes should be implemented:

- The lifetime of a thread needs to be changed in order to support the code block that will prefetch the data.
- Local scheduler needs to be modified in order to handle different types of transfers for global data.
- Compiler needs to be adapted in order to modify the DTA threads to add the prefetching code for basic data types.

In particular, two additional states need to be inserted (Figure 4): 2a) "Program DMA" and 2b) "Wait for DMA". If there is data to prefetch, a PF code block is responsible for programming the DMA transfer.

Depending on the block of data that is accessed from within the thread (e.g., array access, linked list access, pointer access), the compiler will insert instructions to program the DMA unit to prefetch the entire data structure or only parts of it. Once the transfer is completed, a standard DTA synchronization mechanism (Synchronization Counter) can be used to notify the scheduler that the thread can continue its execution. This could be implemented also using split-transaction network, but in case where thread accesses array with a certain stride between elements it could generate too many transactions (and DMA performs it in one transaction).

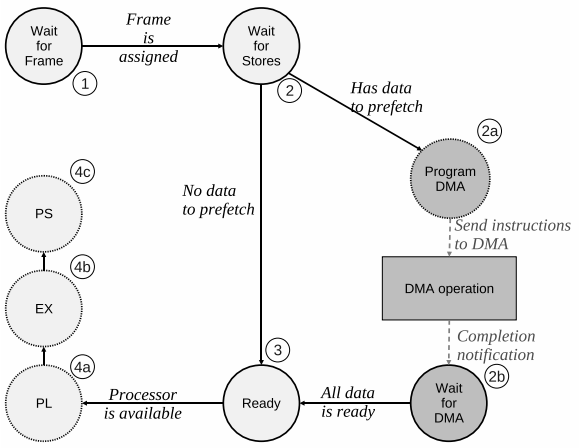


Figure 4. The lifetime of a thread in DTA with prefetching enabled. The states that are introduced with respect to the original DTA are shown with darker background.

4. Initial implementation and preliminary results

In order to verify the mechanism that is proposed above, we have implemented its initial version that supports the prefetching of generic object from memory. The rest of this section explains the experimental methodology that was used and gives the initial results.

4.1. Experimental methodology

We use a modified version of CellSim [13] with DTA support [8]: we refer to this modified version as "CellDTA" in the following text. CellSim is a modular simulator that is based on the UNISIM framework [14] to simulate the Cell processor, and then extended with DTA-specific modules.

The PPE (Power Processing Element) of the Cell processor is used to initiate the DTA TLP activities, and threads are executed on the SPEs (Synergistic Processing Element). Each SPE contains a SPU (Synergistic Processing Element) which executes code, Local Store and a MFC (Memory Flow Controller). SPU is an in-order SIMD processor which can issue two instructions in each cycle (one memory and one calculation). It does not contain any branch prediction, but relies on the compiler to give hints on branches. It also does not have any caches, but uses the local store to store data and instructions.

For the purpose of implementing DTA support, we have added one DSE to the Cell processor, and one LSE to each SPE. In order to store the code of DTA threads that execute on the SPU and to hold the frames that are needed locally, we use the Local Store. For this experiment, we use a part of the LS in order to store the data that was prefetched from the main memory. The parameters of the memory subsystem used in simulations are given in Table 4. The SPU has been changed to add DTA-specific instructions (Table 1) and instructions for accessing main memory.

Table 2. Parameters of the memory subsystem used in simulations.

Memory	Parameter	Value
Main memory	Size	512 MB
	Latency	150 cycles
	Number of ports	1
Local Store	Size	156 kB
	Latency	6 cycles
	Number of ports	3

Table 3. Instructions for accessing both frame and main memory.

Name	Description
LS address	The address in the local store where data will be stored.
MEM address	The address in the main memory where data is located.
Data size	The size of the data that will be transfered.
Tag ID	The ID of the DMA operation that will later be used by the LSE to check if the transfer is completed or not.

Table 4. Parameters of the communication subsystem used in simulations.

	Parameter	Value
Bus	Number of buses	4
	BW of each bus	8 bytes/cycle
	Total BW	8.1 GB/s at 2.4 GHz
MFC (DMA controller)	Command Queue size	16
	Command latency	30 cycles

Each of the SPEs contains a DMA unit, which is used by DTA threads to transfer the prefetched data. The programming of this unit is performed via MFC and commands are sent using existing Cell SPE instructions. The parameters that are used to program the DMA unit in Cell are in Table 3. DMA needs address and size for the data block access. Additionally, the address where this data will be stored in the LS has to be sent, together with the TAG ID, which is used to read the status of the initiated transfer in the DMA unit. The parameters of the communication subsystem that are used in the simulations are in Table 2.

4.2. Benchmarks

All the benchmarks are hand-coded for the original DTA and then translated for Cell-DTA version. Prefetching code blocks are added by hand following the principles described in the previous sections. The benchmarks are:

- The bitcount from the MiBench [15] suite is a program that counts bits for a certain number of iterations (input parameter). Its parallelization has been performed by

unrolling both the main loop and the loops inside each function. This benchmark is used in order to test the scalability of the architecture. Global data that is used by some of the functions in the program is prefetched in the threads where it was needed. Experiments are performed with 10000 iterations - bitcnt(10000) in the figures.

- Matrix multiply (mmul) is a program that multiplies two matrices. Threads that run in parallel are calculating parts of the output matrix. The number of threads is always a power of two, and the program is always executed on a number of cores that is power of two. Inputs are two n by n matrices. Prefetching of the parts of the input matrices is performed in the threads that are calculating the output matrix. Experiments are performed with matrices of size 32 by 32 - mmul(32) in the figures.
- 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 PEs. Input is an n by n picture. Parts of the input image are prefetched in the threads that are calculating the zoom. Experiments are performed with a picture of size 32 by 32 - zoom(32) in the figures.

4.3. Preliminary results

Before applying the prefetching mechanism, we have analyzed the behavior of benchmarks to determine the amount of execution time in which the processor is stalled when waiting for memory. The programs were executed with eight SPUs in conditions described in previous section. The results are shown in Figure 5.

The y-axis of the Figures 5 shows the benchmark name and x-axis shows the percentage of time spent in certain phases of the execution: Working - when the SPU works without stalls; Idle - when the SPU has no ready threads to execute; Memory Stalls - when SPU waits for a response from main memory (including the time that a request to memory spends on the network); LS Stalls - when SPU is waiting for a response from the Local Store; LSE Stalls - when the SPU waits for a response from the LSE and Prefetching - prefetching overhead, which is due to the fact that SPU must spend some time in order to program the DMA unit. In an implementation where LSE has two available pipelines (SP and XP) [6], it can overlap this with the execution of other threads, but in the CellDTA this is not yet available. In case of bitcnt, there are LSE stalls, which are due to the fact that this benchmark is forking vast amount of threads in small amount of time and the LSE can't keep up (a possible solution is to use virtual frame pointers [6], but we did not include this feature in the current version of the CellDTA simulator). In all three benchmarks, a significant amount of time is spent while waiting for memory. Accesses to the Local Store (for reading frame memory) are mostly hidden (overlapped with the execution) and therefore LS stalls constitute only a 2% of execution time of bitcnt and they are negligible in the case of mmul and zoom. In order to fully understand the behavior of benchmarks, we have extracted the dynamic instruction count (Table 5). We report the total number of executed instructions, as well as number of instructions that are accessing frame and main memory.

Table 5. Number of executed instructions in all benchmarks (total, instructions for accessing frame and main memory).

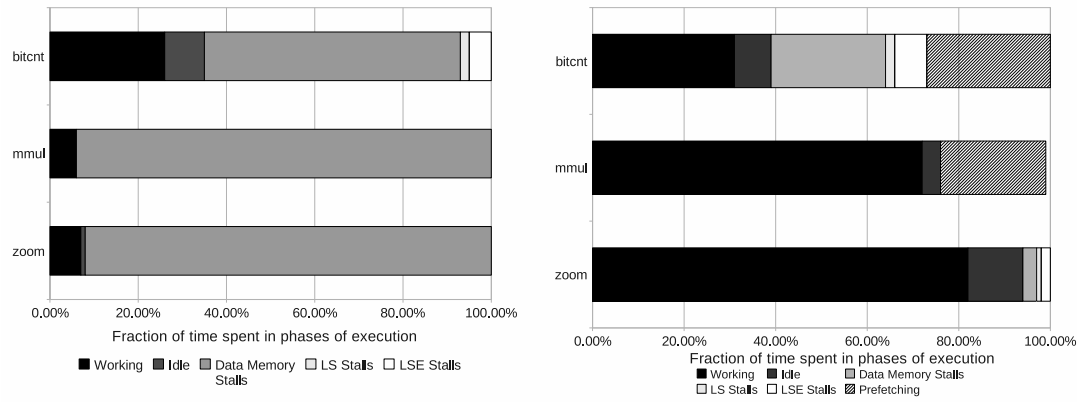
Benchmark	Total	LOAD	STORE	READ	WRITE
bitcnt	9415559	806593	806593	192366	2814
mmul	341422	73	73	65536	1024
zoom	353425	4672	4672	32768	16384

In bitcnt, 58% of time is spent waiting for main memory (Figure 5a), while READ instructions represent 2% of total executed instructions and LOAD instructions represent 8.5% of all executed instructions (Table 5). As we can see (Table 5), data is mostly exchanged using frame memory and accesses to global data within threads (READ instructions) are due to the reading of global arrays that are used by some of the functions that are counting bits. The prefetching decouples 62% of READ instructions. Other READ instructions are left in the program because it is not worthwhile to decouple them. In certain threads of bitcnt, a thread is reading one element of the 256-element array, and the element to be read is not known before the execution starts, so the entire array needs to be prefetched. In this case, it is faster to leave one memory access inside the thread rather than prefetch all elements of the array when only one will be used.

In matrix multiply, 94% of time is spent waiting for memory (Figure 5a), while (Table 5) READ instructions represent 19% of all instructions and the number of accesses to frame memory is negligible. The accesses to global memory are due to the fact that the input matrices are stored in main memory, and read from there by the threads that are calculating the result. Prefetching decouples all global memory accesses, in this case.

Zoom spends 92% of time waiting for memory (Figure 5a), while (Table 5) READ instructions are 9.2% of all instructions. Similarly to matrix multiply, input data is stored in main memory, and read from there by threads that are calculating the output image. Prefetching decouples all global memory accesses, also in this case. As we can see, the percentage of time spent while waiting for memory and the number of accesses to main memory are high and this gives a lot of space for the prefetching technique to work.

Finally, we show the final effect on the execution time when prefetching is enabled (figures 6a, 7a, 8a) in comparison with the execution time without prefetching and



(a) No prefetching

(b) With prefetching

Figure 5. The breakdown of average SPU execution time on original CellDTA with eight SPUs and memory latency set to 150.

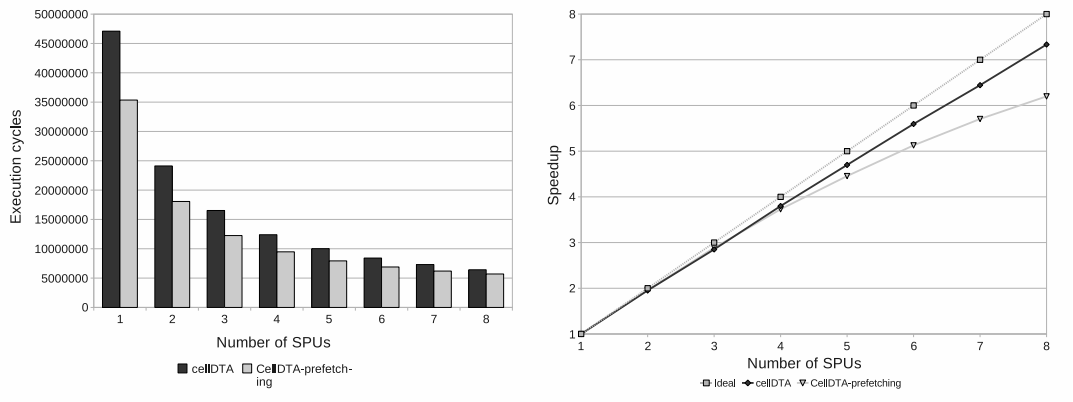
the scalability (figures 6b, 7b, 8b) for our benchmarks. When prefetching is enabled, in cases of mmul and zoom all needed data is transferred to the local store before executing the EX code block (see figure 4), while in the original DTA design [6] a transfer from the main memory is created each time a READ operation is performed. This means that in case of no prefetching the CellDTA is not using all available bandwidth, since each READ instruction fetches only 4 bytes of data (and the network can support transfers of 32 bytes in one cycle). On the other hand, when prefetching is used, DMA unit can fully utilize the bandwidth. In fact, as expected, performance is much better when prefetching is enabled: speedup with respect to the original CellDTA is 1.13 times for bitcnt (Figure 6a), 11.18 times for matrix multiply (Figure 7a) and 11.48 times for zoom (Figure 8a). The reduced speedup in the case of the bitcnt benchmark is due to the fact that we do not decouple all the global access, but only a portion of them (this shall be considered in the next releases of our simulator). The scalability (in all cases) is a little worse with respect to the original architecture. More importantly, the execution times are reduced in comparison with original architecture, and it is especially visible in cases of matrix multiply and zoom. Prefetching overhead (Figure 1.b) is 19% in case of bitcnt, 28% in case of matrix multiply and it is negligible in case of zoom. In case of bitcnt, memory stalls still account for 26% of execution time, while in case of the other two benchmarks memory stalls are completely eliminated.

Figure 9 shows the pipeline usage for CellDTA with and without prefetching. Naturally, the usage is much higher when prefetching is performed because operations with local store are much faster than operations with main memory, and latencies are much smaller. Obtained results are in line with the results presented in Figure 5, meaning that the improvement in pipeline usage is mostly due to the amount of memory stalls that were present in the architecture without prefetching.

As our interest is mostly on the execution model of the threads and the decoupling of their accesses, our simulator does not yet include the cache module (still under development), we performed another set of experiments by setting all memory latencies in the system to one cycle. In this way, we investigate the best situation when cache accesses would always hit and we compare the results with the previous experiments (which represent the opposite extreme situation, when cache would always miss). The speedup is similar to the case of long memory latency (1.01 times in case of mmul and 1.34 times in case of zoom), and the pipeline usage is improved. In case of bitcnt, prefetching has slowed down the execution because only 5% of the time was spent waiting for memory, while prefetching overhead is 34%. The execution time is almost equal to the case with long latencies and prefetching. Considering that prefetching introduces a little overhead, this indicates that this perfetching scheme can almost eliminate the need for caches.

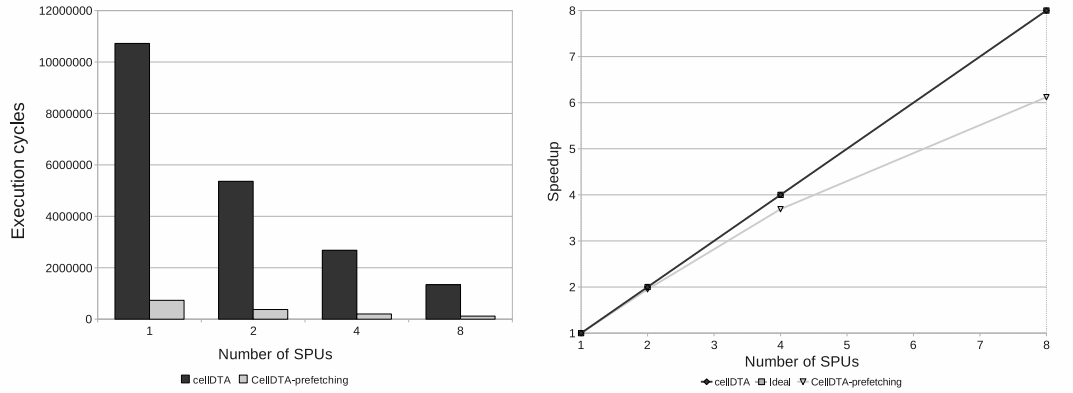
5. Related Work

Multi-core architectures have gained a lot of attention in the industry recently. IBM Cyclops-64 (C64) [1] is a multi-core multithreaded chip that is currently under development. It contains 80 processors and each processor has two SRAM memory banks that can be configured either as scratchpad or global memory. Plurality [4] is a multi-core system that uses a pool of RISC processors with uniform memory, hardware scheduler, synchronizer and load balancer. SUN Microsystems' UltraSPARC T2 [2] is a multithreaded multicore chip capable of running 64 threads at the same time. The main difference between these architectures and the DTA is the scheduled dataflow programming model that DTA uses [9].



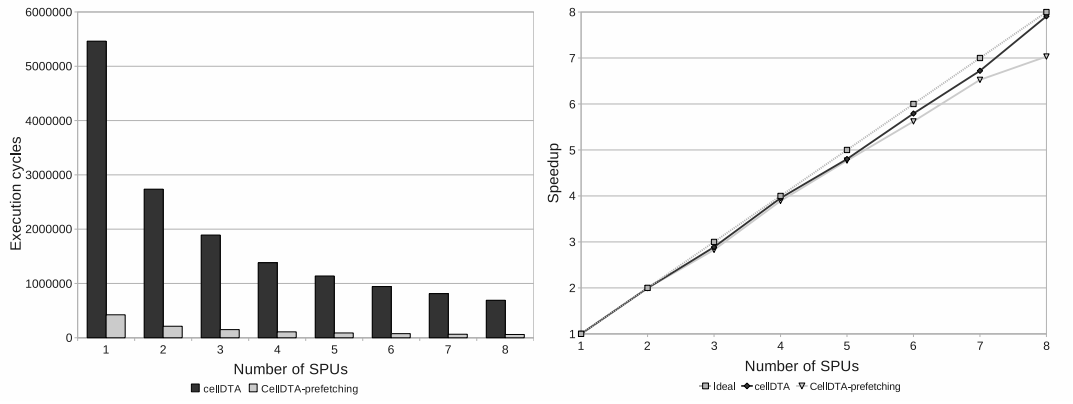
(a) Execution time (b) Scalability

Figure 6. Results for bitcnt(10000) when memory latency is set to 150 cycles.



(a) Execution time (b) Scalability

Figure 7. Results for mmul(32) when memory latency is set to 150 cycles.



(a) Execution time (b) Scalability

Figure 8. Results for zoom(32) when memory latency is set to 150 cycles.

There are several examples of research in multi-core architectures in academia. Speculative Data-Driven Multi-threading (DDMT) [16] is an architecture that is based on dataflow at thread level like DTA. Main difference is that

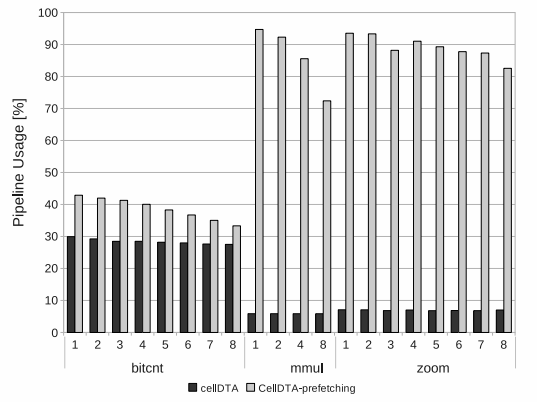


Figure 9. Pipeline usage for programs with and without prefetching.

this concept has static scheduling while in DTA scheduling is done dynamically at run-time in hardware. TRIPS [3] uses "medium size" tiling by allowing several different types of tiles. Some of them can be reconfigured in order to exploit different types of parallelism. While DTA employs dataflow execution at the thread level, and control flow-like execution inside the thread, TRIPS does the opposite. The EARTH architecture [11] contains two levels of threads - threaded procedures and fibers. Threaded procedures are invoked asynchronously in parallel, and they are divided into fibres - fine-grain threads that synchronize in dataflow-like manner. Fibers in EARTH are similar DTA threads, with the difference that DTA threads have decoupled memory accesses. TAM [10] defines a self-scheduled machine language with parallel threads, which communicate in dataflow manner. The difference between TAM and DTA is that TAM only provides a machine language that can be compiled to run on any multiprocessor system without hardware support.

6. Conclusion

In this paper, we have presented a mechanism to address accesses to generic data in the DTA in order to be able to execute threads in fully non-blocking fashion. We have explained one possible prefetching mechanism and presented its initial implementation. The simulation environment was DTA model implemented on Cell processor in UNISIM framework (using CellSim simulator). We have seen from the initial results that in all test cases execution is faster. For the memory intensive benchmarks (mmul and zoom) this speed up is very significant. We can also notice that pipeline utilization is almost perfect when prefetching is used, which proves that concept is correct. This was just an initial implementation. As a part of the future work we are planning to fully automate the entire process (both compiler and architecture parts) and to experiment with some other advanced mechanism and with more complex benchmarks.

Acknowledgments

This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6) and by the HiPEAC2 Network of Excellence (FP7) contract IST-217068.

References

[1] G. Almási, C. Caşcaval, J. G. Castaños, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren, Jr., "Dissecting cyclops: a detailed analysis of a multithreaded architecture," *SIGARCH Comput. Archit. News*, vol. 31, no. 1, pp. 26–38, 2003.

[2] M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson *et al.*, "UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC," in *Solid-State Circuits Conference, 2007. ASSCC'07. IEEE Asian*, Jeju, Republic of Korea, 2007, pp. 22–25.

[3] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 422–433, 2003.

[4] "Plurality architecture." [Online]. Available: http://www.plurality.com/architecture.html

[5] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob *et al.*, "An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, 2007, pp. 98–589.

[6] R. Giorgi, Z. Popovic, and N. Puzovic, "DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems," in *Proceedings of IEEE SBAC-PAD*, Gramado, Brasil, Oct. 2007, pp. 263–270.

[7] R. Giorgi, Z. Popovic, N. Puzovic, A. Azavedo, and B. Juurlink, "Analyzing scalability of deblocking filter of h.264 via tlp exploitation in a new many-core architecture," in *Proceedings of the 11th EUROMICRO-DSD*, Parma, Italy, sept 2008, pp. 189–194.

[8] R. Giorgi, Z. Popovic, and N. Puzovic, "Introducing hardware tlp support for the cell processor," in *Proceedings of IEEE International Workshop on Multi-Core Computing Systems*. Fukuoka, Japan: IEEE, March 16-19, 2009 2009, pp. 1–6, accepted for publication.

[9] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation," *IEEE Transaction on Computers*, vol. 50, no. 8, pp. 834–846, Aug. 2001.

[10] D. Culler, S. Goldstein, K. Schauser, and T. Von Eicken, "TAM-A Compiler Controlled Threaded Abstract Machine," *Journal of Parallel and Distributed Computing*, vol. 18, no. 3, pp. 347–370, 1993.

[11] H. Hum, O. Maquelin, K. Theobald, X. Tian, X. Tang, G. Gao, P. Cupryk, N. Elmasri, L. Hendren, A. Jimenez *et al.*, "A design study of the EARTH multiprocessor," in *Proceedings of the IFIP WG10. 3 working conference on Parallel architectures and compilation techniques table of contents*. IFIP Working Group on Algol Manchester, UK, UK, 1995, pp. 59–68.

[12] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," in *Proceedings of the IEEE*, 2001, pp. 490–504.

[13] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "A module-based cell processor simulator," in *HiPEAC ACACES-2007*, L'Aquila, Italy, Jul. 2007, pp. 279–282.

[14] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani, "UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development," *Computer Architecture Letters*, vol. 6, no. 2, pp. 45–48, 2007.

[15] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, 2001, pp. 3–14.

[16] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-Driven Multithreading Using Conventional Microprocessors," *IEEE Transactions On Parallel and Distributed Systems*, pp. 1176–1188, 2006.