Implementing Fine/Medium Grained TLP Support in a Many-Core Architecture

Roberto Giorgi, Zdravko Popovic, Nikola Puzovic

Department of Information Engineering, University of Siena, Italy
http://www.dii.unisi.it/~ {giorgi | popovic | puzovic}

Abstract. We believe that future many-core architectures should support a simple and scalable way to execute many threads that are generated by parallel programs. A good candidate to implement an efficient and scalable execution of threads is the DTA (Decoupled Threaded Architecture), which is designed to exploit fine/medium grained Thread Level Parallelism (TLP) by using a hardware scheduling unit and relying on existing simple cores. In this paper, we present an initial implementation of DTA concept in a many-core architecture where it interacts with other architectural components designed from scratch in order to address the problem of scalability. We present initial results that show the scalability of the solution that were obtained using a many-core simulator written in SARCSim (a variant of UNISIM) with DTA support.

Keywords: many-core architectures, DTA

1 Introduction

Many-core architectures offer an interesting possibility for efficiently utilizing the increasing number of transistors that are available on a single chip. Several many-core architectures have been developed in industry [1-3] and have been proposed in academia research projects [4, 5].

Although many-core architectures offer hundreds of computational cores, they have to be properly programmed in order to utilize their computing power potential [6]. Decoupled Threaded Architecture (DTA) is a proposal for exploiting fine/medium grained TLP that is available in programs [7]. Even though other types of parallelism are typically present in programs, like Instruction Level Parallelism (ILP) and Data Level Parallelism (DLP), they are not the focus of this paper: we assume that the overall architecture will be offloading parts of the computation with TLP potential on small "TLP-accelerators", e.g., simple in-order cores, and that other types of accelerators could take care of ILP and DLP. DTA also provides distributed hardware mechanisms for efficient and scalable thread scheduling, synchronization and decoupling of their memory accesses. Previous research experimented with DTA using a simplified framework in order to prove the concept [7]. In this paper, we

present an initial implementation of DTA support in a heterogeneous many-core architecture that is compatible with the SARC project [8] architecture, and we describe the hardware extensions that are needed for DTA support.

2 DTA Concept

The key features of DTA concept are: i) communication and ii) non-blocking synchronization among threads, iii) decoupling of memory accesses that is based on the Scheduled Data-Flow (SDF) concept [9], iv) clusterization of resources in nodes (differently from SDF) and v) the use of a distributed hardware scheduler (which was centralized in SDF). Data is communicated via frames, which are portions of local memory assigned to each thread. A per-thread synchronization counter (SC) is used to represent a of input data that the thread needs. This counter is decremented each time a datum is stored in a thread's frame, and when it reaches zero (when all input data have arrived) that thread is ready to execute. In this way, DTA provides a dataflow-like communication between threads - dataflow at thread level, and a non-blocking synchronization (threads can be synchronized using SC and while one thread is waiting for data, processors are available to execute other ready threads).

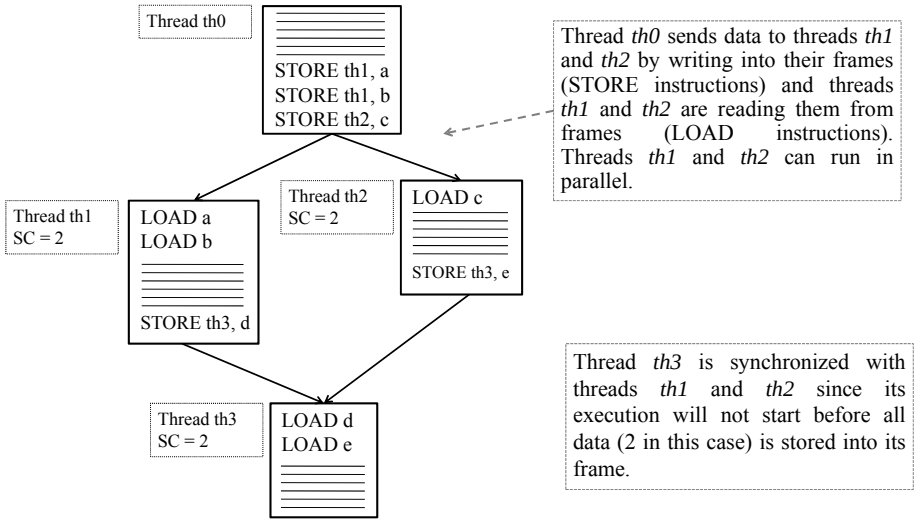


Fig. 1. An example of communication and synchronization among threads in DTA.

Threads in DTA are logically divided into smaller phases, which are called code blocks. At the beginning of a thread, Pre-Load (PL) code block reads the data from the frame and stores them into registers. Once the PL phase completes, the Execution (EX) code block starts, and it reads data from the registers and performs calculations. At the end of a thread, Post-Store (PS) code block writes data to the frames of other

threads. Another possibility, like in SDF architecture [9], is to use more than two types of pipelines, one to handle PL and PS code blocks – named Synchronization Pipeline (SP) - and the other type to execute EX code blocks – named Execution Pipeline (XP); in this work, we don't want to lose the flexibility of using existing and smaller cores. Communication of data via frames is preferable, but it is not always possible to replace accesses to global data in main memory with accesses to frame memory so the threads can access main memory at any point during execution, and in this case DMA-assisted prefetching mechanism can be used to completely decouple memory accesses [10]. In order to overcome the effects of wire delay, Processing Elements (PEs) in DTA are grouped into nodes. The nodes are dimensioned so that all PEs in one node can be synchronized using the same clock [7], and that fast communication can be achieved among them using a simple interconnection network inside a node. 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.

The first specific hardware structures that DTA uses is a Frame Memory (FM). This is a local memory that is located near each PE and it is used for storing thread's data. Access to a frame memory is usually fast and shouldn't cause any stalls during execution. Another DTA-specific hardware structure is the Distributed Scheduler (DS) that consists of Local Scheduler Elements (LSEs) and Distributed Scheduler Elements (DSEs). Each PE contains one LSE that manages local frames and forwards request for resources to the DSE. Each node contains one DSE that is responsible for distributing the workload between processors in the node, and for forwarding it to other nodes in order to balance the workload among them. DSE together with all LSEs provides functionality of dynamic distribution of the workload between processors. Schedulers communicate between 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 [7].

Besides these structures, DTA requires a minimal support in the ISA of the processing element for the creation and management of DTA threads. This support includes new instructions for assigning (FALLOC) and releasing (FFREE) frames, instructions for storing data to other thread's frames (STORE) and for loading data from frame (LOAD). In the case when PEs cannot access the main memory directly, instructions for reading and writing data to and from main memory are also needed.

Further details on DTA, as well as one possible implementation are given in Section 4.

3 Heterogeneous Many-Core Architectures

Future chip multiprocessor architectures aim to address scalability, power efficiency and programmability issues. In order to achieve the goal of creating a scalable chip, the architecture is typically subdivided into nodes (**Fig. 2**) that are connected via a Network on Chip (NoC), where each node contains several processors (or domain-

specific accelerators). The communication inside a node is done by using a faster network that connects all the elements that belong to one node (crossbar for example). Since the network inside a node is very efficient, and adding more elements to the node would cause the degradation of its performance, the architecture scales up by adding more nodes to the configuration, and not by increasing the node size. Efficient many-core architectures should also be designed to target diverse application domains, from single threaded programs to scientific workloads. In order to address these domains, the architecture can contain heterogeneous nodes (**Fig. 2**).

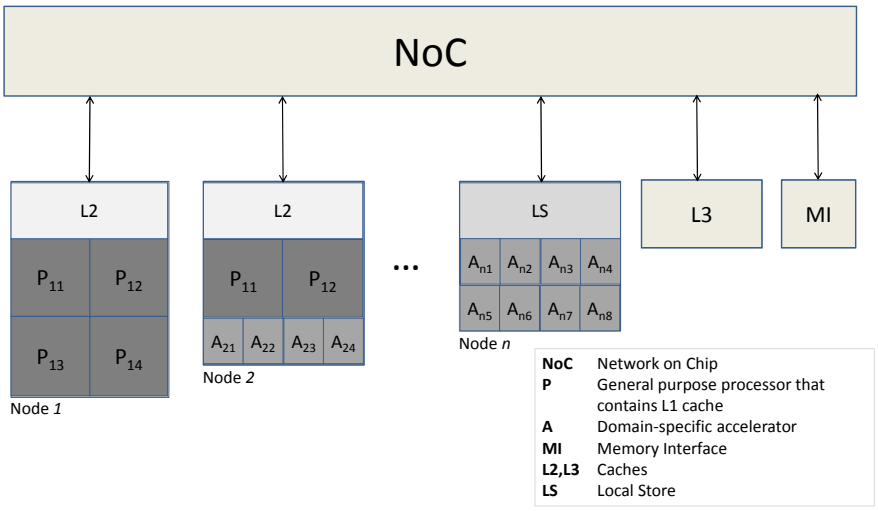


Fig. 2. An instance of many-core architecture with heterogeneous nodes

Each node contains a mix of general purpose processors and/or accelerators together with local memory (such as shared L2 cache or a Local Store). Typically a general purpose processor performs the control of the nodes and provides operating system services, and may address aggressive Instruction Level Parallelism (ILP) needs. On the other hand, domain specific accelerators will speed-up applications that have specific processing needs (such as vector or multimedia). We use a shared memory model as it simplifies the programmability of the machine. Several programming models for shared memory multiprocessors have been considered recently, such as OpenMP [11] and Capsule [12].

4 Implementing DTA in Many-Core Processor

A possible instance of a many-core processor with DTA support should contain a memory controller, and multiple DTA nodes that contain DTA accelerators (**Fig. 3**). The system needs to contain a general purpose processor (P) that is responsible for sending the code to the DTA nodes, and for initiating the DTA activity. A crossbar is

used for providing a fast communication for elements inside a node, which can be a part of the more complex Network on Chip (NoC) that is used to connect the nodes. The Distributed Scheduler is located in each node and since it will communicate mostly with the LSEs inside the same node it is attached directly to the crossbar.

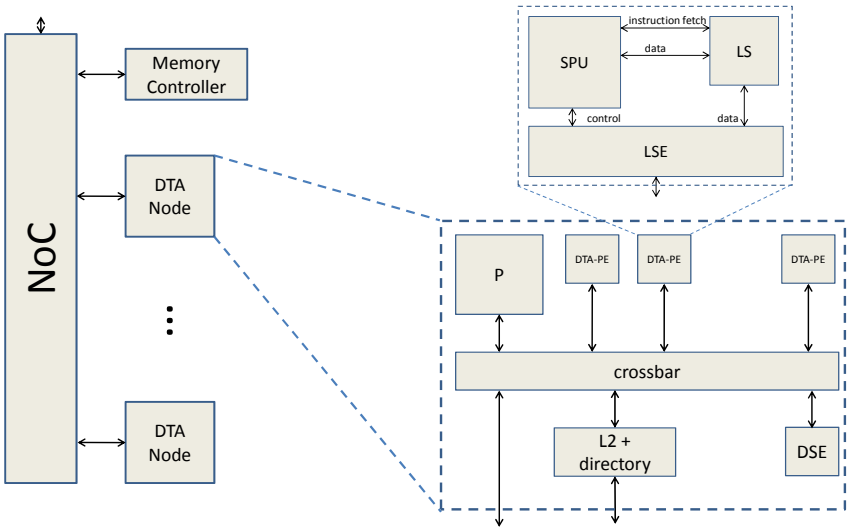


Fig. 3. An instance of a many core architecture with DTA support.

In this study, the DTA accelerators are based on the Synergistic Processing Unit (SPU) [13] from the Cell processor. SPU is an in-order SIMD processor, which can issue two instructions in each cycle (one memory and one calculation). In order to keep the processor simple, the designers didn't implement any branch prediction and SPU relies on the compiler to give hints on branches. It also doesn't have any caches, but uses the local store to store data and instructions. For the purpose of running DTA programs, SPU is extended with the Local Scheduling Element, and frames for threads that execute on one SPU are stored in the Local Store (LS). The SPU's ISA is extended with DTA-specific instructions, and communication with the rest of the system is handled by the LSE. A SPU with DTA-specific extension is called DTA-PE (**Fig. 3**). Since the SPU contains only one pipeline, all code blocks (PL, EX and PS) will execute on it in sequence. However, SPU's pipeline is able to issue one memory and one execution instruction at the same time and for instance it can overlap a load from LS with subsequent instructions.

The LSE manages threads that execute on one DTA-PE, and it contains structures with information about the current state of the DTA-PE and frame memory: the Pre-Load (PLQ) queue and Waiting Table (WT).

The Pre-Load Queue contains information about threads that are ready to run (SC = = 0). It is implemented as a circular queue and each entry contains the Instruction Pointer (IP) of the first instruction of the thread and address of the thread's frame (Frame Pointer – FP).

The Waiting Table contains information about threads that are still waiting for data (SC != 0). Number of entries in the WT is equal to the maximal number of frames that are available in the DTA-PE, and it is indexed by a frame number. Each entry in the WT contains the IP and FP of the thread and synchronization count, which is decremented on each write to the thread's frame. Once the SC reaches zero, IP and FP are transferred to the PLQ.

In order to be able to distribute the workload optimally, the DSE must know the number of free frames in each DTA-PE. This information is contained in the Free Frame Table (FFT) that contains one entry with the number of free frames for each DTA-PE. When a FALLOC request is forwarded to a DTA-PE, the corresponding number of free frames is decremented, and when a FFREE message arrives the number of entries is incremented. Since it may happen that a FALLOC request cannot be served immediately, a Pending FALLOC Queue (PFQ), which stores pending frame requests. Each entry in this queue contains the parameters of the FALLOC request, and the ID of the DTA-PE that sent the request. When a free frame is found, the corresponding entry is removed from this queue.

Most of the additional hardware cost that DTA support introduces comes from the structures needed for storing information about threads. These costs are expressed in **Error! Reference source not found.** for implementation with one DTA node, using **rbe** (register bit equivalent) [14] as a unit of measure. The register bit equivalent equals the area of a bit storage cell – a six transistor static cell with high bandwidth that is isolated from its input/output circuits [14]. In the remainder of this section we will give the estimate of the hardware cost that DTA introduces for the case of one node.

Table 1. Storage cost of DTA components expressed in register bit equivalent units: $n_{DTA\text{-}PEs}$ – number of DTA-PEs in the node, $size_{FP}$ – size of FP in bits, n_{PFQ} – number of PFQ entries, $size_{IP}$ – size of IP in bits, $size_{SC}$ – size of SC in bits

Component	Structure	Size [rbe]
DSE	FFT	$size_{FFT\text{-}entry} * n_{DTA\text{-}PEs}$
	PFQ	$n_{PFQ} * (size_{IP} + size_{SC} + size_{ID})$
LSE	PLQ	$n_F * (size_{IP} + size_{FP})$
	WT	$n_F * (size_{IP} + size_{FP} + size_{SC})$

The parameters that influence the hardware cost of DTA support are the number of DTA-PEs in one node ($n_{DTA\text{-}PEs}$), number of frames in each DTA-PE (n_F), number of bits needed to store the Synchronization Counter ($size_{SC}$ – in bits), Instruction Pointer size ($size_{IP}$ – in bits) and the number of entries in the PFQ (n_{PFQ}).

The required storage size for keeping an FP entry ($size_{FP}$) in the LSE is $log_2 n_F$ bits, since instead of keeping the entire address it is enough to keep the frame number from which the address can be reconstructed using simple translation. The Pre-Load Queue contains n_F entries (FP and IP), and the Waiting Table contains n entries (FP, IP and SC). The frames are kept in Local Store and no additional storage is needed for them.

The Free Frame Table in the DSE contains $n_{DTA\text{-}PEs}$ entries, where each entry can have the value from zero to n_F since n_F is the maximum number of free frames in one DTA-PE (hence, the size of an entry is $size_{FFT\text{-}entry} = log_2 (n_F+1)$ bits). The Pending

FALLOC Queue contains n_{PFQ} entries where each entry contains the IP ($size_{IP}$ bits), SC ($size_{SC}$ bits) and the ID of the sender ($size_{ID} = log_2 n_{DTA\text{-}PEs}$). The total size of the structures needed for hardware support is the sum of the costs for LSEs and the cost of the DSE, and it is the function of $n_{DTA\text{-}SPEs}$ and n_F.

Take for example a DTA node with $n_{DTA\text{-}PEs}$ = 8 DTA-PEs, where each DTA-PE has 256kB Local Store, Instruction Pointer of $size_{IP}$=32 bits, maximal value for SC of 256 (hence, $size_{SC}$=8 bits), n_F = 128 frames per DTA-PE (each frame with 64 4-byte entries). And the DSE that has the possibility to store 8 pending FALLOC requests (one from each DTA-PE). The frames occupy 32kB in each Local Store (256kB total) and the rest of the LS can be used for storing the code and other data that cannot be communicated using frame memory and needs to be fetched from the main memory (using DMA unit for example). In this case, the PreLoad Queue has 4992 bits and the Waiting Table has 6016 bits of storage, which gives total of 1.3 kB in the LSE. The Pending FALLOC Queue takes 392 bits in the DSE, and the Free Frame Table has 64 bits of storage which yields total of 49 B in the DSE. Hence, all needed structures in one node have 10.8 kB of storage space, which is 0.5% when compared to the total LS size. If we double the number of frames in each DTA-PE to n_F = 256 (taking ½ of the LS), the required storage space increases to 22 kB, which is 1.07% of the LS size. Increasing the number of frames even more, to 384 (taking ¾ of the LS), the total size is 33.05 kB which represents 1.6% of the LS size. Based on these values, and neglecting small contributions to the total size, we arrive to the formula:

$$size\ (n_{DTA\text{-}PEs}, n_F) = K_1 * n_{DTA\text{-}PEs} * n_F * (2 * \log_2 (n_F) + K_2)$$

where K_1 = 11/85, K2 = 70 and the size is expressed in bits.

5 Experimental Results

5.1 Methodology

In order to validate the DTA support, we extended the SARCSim simulator and tested it with several simple kernel benchmarks. SARCSim is a many-core simulator that is based on the UNISIM framework [15], and developed to simulate the SARC architecture [8].

SARCSim/UNISIM allowed us to use already existing modules (such as processors, network and memory) and to implement only the DTA-specific extensions. The configuration used for performing simulations is the one described in a previous section, and we have varied memory latencies throughout the experiments in order to determine if the memory is the limiting factor for the scalability. The size of the LS used in experiments is 256kB per DTA-PE. The tested configuration didn't have caches implemented, and all requests went directly to the memory. However, we have performed the tests with memory latency set to one cycle in order to simulate the situation in which the caches are present, and requests always hit.

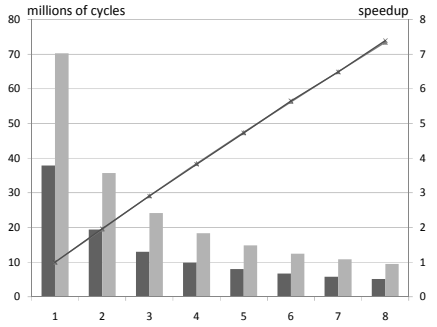
The benchmarks that are used for performing tests are:

- The bitcount (bitcnt) from the MiBench [16] suite is a program that counts bits in various ways for a certain number of iterations (an input parameter). Its parallelization has been performed by unrolling both the main loop and loops inside each function.
- Fibonacci (fib) is a program that recursively calculates Fibonacci numbers. Each function call is a new DTA thread. The main purpose of this benchmark is to create a vast number of small threads in order to stress the DTA scheduler.
- Matrix multiply (mmul) is a program that just does what the name implies. Calculations are performed in threads that work in parallel. Number of working threads is always power of two. Inputs are two n by n matrices.
- Zoom is a image processing kernel for zooming. It is parallelized by sending different parts of the picture to different processors. Input is an n by n picture.

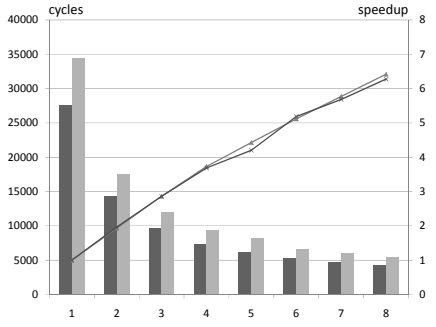
All these benchmarks were first hand-coded for the original DTA architecture, and then translated in order to use the SPU ISA with DTA extensions.

5.2 Preliminary Results

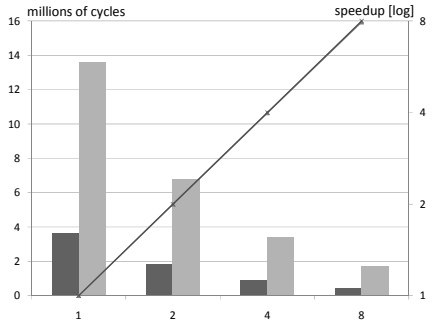
The first set of experiments shows the scalability of the DTA TLP support when number of DTA-PEs is increased from 1 to 8 in one node (**Fig. 4**).



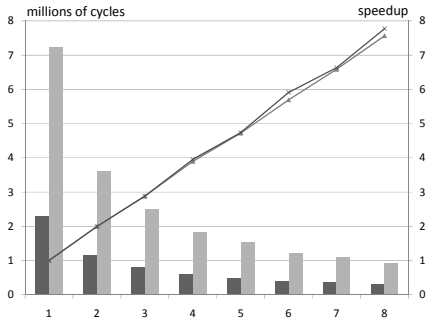
a) Bitcnt (10000)



b) fibonacci (10)



c) mmul(32)



d) zoom (32)

Fig. 4. Execution times and speedup when varying memory latency. Execution time is shown using bars, and speedup using lines. The X axis shows number of DTA-PEs.

All benchmarks scale well except for Fibonacci, as the number of requests for new threads exceeds the DSE's capabilities. We have encountered this situation in a previous study [7] and we overcame this problem by using a virtual frame pointers, as described in the same study. In this work, we didn't consider yet the use of the virtual frame pointer. As expected, the configuration with memory latency set to 1 cycle has lower execution time than the configuration with memory latency set to 150 cycles. However, the scalability is the same in both cases, and speedup is near to the ideal.

6 Related Work

Most of the leading hardware producers have introduced their many-core architectures recently. Examples are Cyclops-64 [1], which is a multi-core multithreaded chip currently under development by IBM, UltraSPARC T2 [2] from SUN Microsystems , and Plurality [3], which uses a pool of RISC processors with uniform memory, hardware scheduler, synchronizer and load balancer. DTA mainly differs from these architectures in the execution model, which is based on the Scheduled DataFlow in the case of DTA.

Academic research projects are also focusing on many-core architectures. Speculative Data- Driven Multithreading (DDMT) [5] exploits the concept of dataflow at thread level like DTA. The main difference is that DDMT uses static scheduling while in DTA scheduling is done dynamically at run-time in hardware. TRIPS [4] uses tiling paradigm with different types of tiles. These tiles are reconfigurable in order to exploit different types of parallelism. TRIPS uses dataflow concept inside a thread, and control flow at the thread level, which is the opposite of what DTA does. TAM [17] defines a self-scheduled machine language with parallel threads, which communicate in dataflow manner among them and a machine language that can be compiled to run on any multiprocessor system without any hardware support (unlike DTA which has HW support).

7 Conclusions

In this paper, we have presented one possible implementation of TLP support for a many-core architecture, which targets fine/medium grained threads via hardware scheduling mechanism of the DTA. The initial test show that scalability of the architecture is promising in all the cases up to 8 processors per node.

The overall conclusion is that since this implementation for TLP support scales good, it suites well in the many-core environment. As a future work, we want to test different configurations with more nodes and to implement some techniques present

in native DTA (e.g. virtual frame pointers). We will focus also on a tool that would allow us to automatically extract DTA code from high level programming languages by using methods like OpenMP, which would allow us to perform tests with more benchmarks.

Acknowledgements. This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6) and by the HiPEAC Network of Excellence (FP6 contract IST-004408, FP7 contract ICT-217068).

8 References

1. Almási, G., et al., *Dissecting Cyclops: a detailed analysis of a multithreaded architecture.* SIGARCH Comput. Archit. News, 2003. **31**(1): p. 26-38.
2. Shah, M., et al. *UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC*. in *IEEE Asian Solid-State Circuits Conference, 2007. ASSCC '07*. 2007. Jeju.
3. *Plurality architecture*. Available from: http://www.plurality.com/architecture.html.
4. Sankaralingam, K., et al., *Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture*, in *Proceedings of the 30th annual international symposium on Computer architecture*. 2003, ACM Press, pp. 422-433: San Diego, California.
5. Kyriacou, C., P. Evripidou, and P. Trancoso, *Data-Driven Multithreading Using Conventional Microprocessors.* IEEE Trans. Parallel Distrib. Syst., 2006. **17**(10): p. 1176-1188.
6. Harris, T., et al., *Transactional Memory: An Overview.* IEEE Micro, 2007. **27**(3): p. 8-29.
7. Giorgi, R., Z. Popovic, and N. Puzovic, *DTA-C : A Decoupled multi-Threaded Architecture for CMP Systems*, in *19th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2007*. 2007: Gramado, Brasil. p. 263-270.
8. *SARC Integrated Project*. Available from: www.sarc-ip.org.
9. Kavi, K.M., R. Giorgi, and J. Arul, *Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation.* IEEE Transaction on Computers, 2001. **50**(8): p. 834-846.
10. Giorgi, R., Z. Popovic, and N. Puzovic. *Exploiting DMA mechanisms to enable non-blocking execution in Decoupled Threaded Architecture*. in *Accepted for Workshop on Multithreaded Architectures and Applications*. 2009. Rome, Italy.
11. *The OpenMP API specification for parallel programming*. Available from: http://openmp.org.
12. Pierre, P., L. Yves, and T. Olivier, *CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs*, in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 2006, IEEE Computer Society.
13. Kahle, J.A., et al., *Introduction to the cell multiprocessor.* IBM J. Res. Dev, 2005. **49**: p. 589-604.
14. Flynn, M.J., *Computer Architecture*. 1995, Sudbury, Masachusetts: Jones and Bartlett Publishers.
15. August, D., et al., *UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development.* IEEE Comput. Archit. Lett., 2007. **6**(2): p. 45-48.
16. Guthaus, M.R., et al., *MiBench: A free, commercially representative embedded benchmark suite*, in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on - Volume 00*. 2001, IEEE Computer Society, pp. 3-14.
17. Culler, D.E., et al., *TAM - a compiler controlled threaded abstract machine.* J. Parallel Distrib. Comput., 1993. **18**(3): p. 347-370.