A Case Study on the Design Trade-off of a Thread Level Data Flow based Many-core Architecture

Zhibin Yu, Andrea Righi, Roberto Giorgi
Department of Information Engineering
University of Siena (UNISI)
Via Roma, 53100, Siena, Italy

Abstract— **With the potential of overcoming the memory and power wall, the many-core/multi-thread has become a trend in processor design area. However, this architecture is far from ripeness because it also companies with many challenges such as scalability and larger architecture design space compared with mono-core architectures. In many-core design space, Data-Flow based architectures are alternatives that deal with concurrency, long memory latencies, and synchronization stalls efficiently. Nevertheless, even in this sub-area, there are still a lot of factors affecting the scalability and performance of the architecture. In this paper, we explore the design trade-offs for Decoupled Threaded Architecture (DTA) which is a data-flow many-core architecture. By using a well known bio-informatics benchmark, ClustalW, we evaluate various DTA configurations with different number of synchronization and execution pipelines. We find that the configuration which consists of two synchronization pipelines (SP) and one execution pipeline (EP) for each processing element(PE) achieves almost the same performance as the configuration consisting of two SPs and two EPs for each processing element. By employing the former configuration, we can save 32.5% of the area required for each DTA processing element.**

Keywords-Multi-threaded;Many-core;Scalability;Programability; Dataflow architecture.

I. INTRODUCTION

Due to the limited performance gains from mono-core architectures, the industry has already shifted gears to deploy architectures with multiple cores and threads [22]. Although multi/many-core architectures promise a significant performance potential, it is not trivial to obtain performance improvement from these architectures. Besides traditional difficulties, new challenges such as scalability and programmability are arising. Further, the design space of multi/many-core architectures is much larger than that of mono-core architectures, leading more difficulties to design them. The Data-Flow architecture [2] is an alternative that deals with concurrency, long memory latencies, and synchronization stalls efficiently. We have designed a multi-threaded architecture named Decoupled Threaded Architecture (DTA) based on Data-flow architecture [1].

Decoupled Threaded Architecture (DTA) is a multi-threaded architecture based on the Scheduled Data Flow (SDF) execution paradigm. The way in which data is communicated among threads and the decoupling of memory accesses from execution are the main differences between DTA/SDF and other multithreaded programming models. Data is exchanged between threads via frames which are portions of local memory assigned to each thread. Each thread is associated with a Synchronization Counter (SC) that represents the number of input data needed by the thread. This counter decreases each time when a datum arrives in the thread's frame. Once it becomes zero, which means all needed input data are ready, the thread is ready to execute. In this way, DTA provides dataflow at thread level and a non-blocking synchronization. This is one of the key differences between the DTA and the original Data-flow architecture which provides dataflow at instruction level.

In this paper, we provide a case study for the design trade-off our previously proposed DTA architecture. We employ the well-known bio-informatics application Clustal-W to evaluate various DTA configurations. Especially, we look for the optimal combinations of the number of synchronization and execution pipelines. We show that the configuration of 2 SPs and 1 EP achieves the same performance as that of 2 SPs and 2 EPs. Therefore, we can save 32.5% of the area required for each DTA processing element by using the former configuration.

The rest of this paper is organized as follows. Section II briefly describes an overview of the DTA architecture. Section III provides an analysis of the benchmark Clustal-W. The experimental methodology is given in Section IV and Section V provides the results and analysis. Section VI surveys the related work and we conclude the paper in Section VII.

II. OVERVIEW OF THE DTA ARCHITECTURE

A. *The Execution Model*

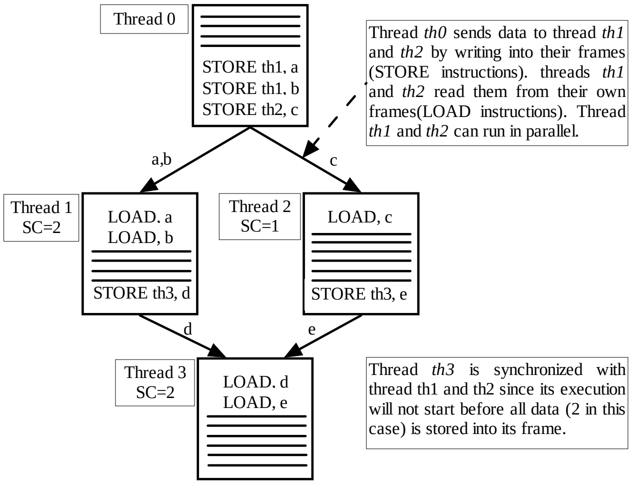
DTA executes TLP (Thread Level Parallelism) activities of a program — portions of a program that exhibits Thread Level Parallelism (TLP). A Compiler (or a programmer) identifies parallel parts of the program and marks them as TLP activities. When these activities are encountered during execution, they are launched to the DTA hardware where they are executed in parallel. For example, in the DTA implementation on the Cell processor [3], TLP activities are launched by the general purpose processor (Power PC) to the DTA-enabled Synergistic Processor Elements that execute DTA threads.

The DTA architecture decouples the memory accesses from execution. This helps the threads exchange data in the data flow manner. To achieve this, a new memory concept, frames which are portions of local memory assigned to each thread, is introduced. One thread has one frame which is used

ISBN: 978-1-61208-154-0

to store the input data for the thread. In order to indicate whether all the input data needed by a thread is in the corresponding frame, the DTA architecture employs a Synchronization Counter (SC). This counter represents the number of input data needed by a thread. When one datum of a thread arrives at its frame, the SC of it decreases by one. Once a SC becomes zero, the corresponding thread is ready to execute. The execution of each DTA thread can be split into three phases: load, execution, and store. In the load phase, input data are loaded from the frame memory; in execution phase, the thread is executed; in the store phase, the outputs of the thread are written to the frames of other threads. Preemptive execution is not allowed and a thread voluntarily releases the processor each time when it switches between phases. Therefore, the memory access of a thread is decoupled from the execution of it.

An example of thread synchronization in DTA is shown in Figure 1. Thread *th0* executes first and creates threads *th1*, *th2* and *th3*. Since the thread *th1* needs two input variables *a* and *b*, its SC is set to 2 when the thread is created. Similarly, the SCs of threads *th2* and *th3* are set to 1 and 2 respectively at the beginning. When *th0* executes the first STORE instruction (used to send data to another thread), it will store *a* into the frame of the thread *th1*. Meanwhile, the SC of *th1* is decreased by 1. After the second STORE, the SC of *th1* becomes 0 and *th1* is ready for its execution. When the third STORE of *th0* completes, *th2* is also ready and both threads (*th1* and *th2*) can run in parallel if there are two cores available. When their execution completes, the output data are stored into the frame of thread *th3*.



Thread *th0* sends data to thread *th1* and *th2* by writing into their frames (STORE instructions). threads *th1* and *th2* read them from their own frames(LOAD instructions). Thread *th1* and *th2* can run in parallel.

Thread *th3* is synchronized with thread th1 and th2 since its execution will not start before all data (2 in this case) is stored into its frame.

Figure 1. An example of thread synchronization

To overcome the long wire delay issue [4], the DTA architecture clusters resources into nodes, a different approach from the SDF architecture. Figure 2 shows the overview of our DTA architecture. As shown in Figure 2, each node contains several processing elements (PE) that are interconnected via a fast and simple intra-node network. Each node is dimensioned so that all PEs in a single node can be synchronized by using the same clock. The Nodes are connected by a slower inter-cluster network.

B. *The DTA Architecture*

The DTA architecture employs two kinds of schedulers to schedule workloads among the computing elements: Distributed Scheduler Elements (DSEs) and Local Scheduler Elements (LSEs), which are illustrated in Figure 2. Each PE contains one LSE that manages local frames and forwards request for resources to the DSE. For example, when a remote store arrives at a local frame of a thread, the LSE decreases the SC of the thread and stores the datum to the frame. When a PE requests a new frame, the request is forwarded to the DSE. Each node contains one DSE that balances the workloads among processors in the node. The elements of the schedulers communicate with each other by using messages [1].

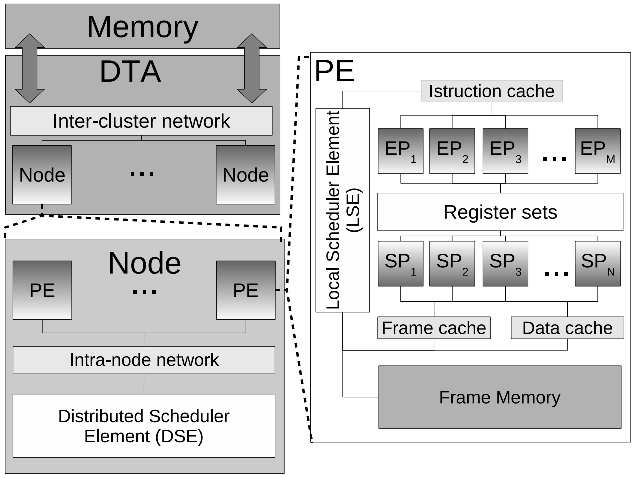


Figure 2. DTA architecture design. A processing element (PE) is composed of many execution pipelines (EP) and synchronization pipelines (SP) with common register sets, frame, data, and instruction cache. Processing elements are grouped by nodes to break scalability limits.

There are two types of memory in the DTA architecture:

- **Frame Memory (FM)**. It is private for each thread and contains input data for the thread;
- **Global Memory (GM)**. It is shared among threads and is accessible via the inter-cluster network;

Both frame memory and global memory use a direct- mapped cache of 32KB (no additional overhead is considered in case of cache hits) to reduce the memory latency. Reading from frame memory is always faster than reading from global memory. On the other hand, writing data to both types of memory is always non-blocking since the LSEs take the write request and process it. The PE is free to continue executing as soon as possible after the request is passed to a LSE. In order to reduce the memory latency, DTA programs tend to use the frame memory as much as possible.

C. *The Processing Element*

The processing elements in DTA can be either off-the-shelf processors or specialized DTA processors with separate pipelines for different phases of each thread. When off-the-shelf processors are used, they need to be modified in order to include LSE, frame memory (usually cache or local store is used [3]) and several DTA-specific instructions which are used to manage the lifetime of threads and exchange data among threads. In order to extract precise statistic information for each phase of a thread's execution, we use DTA-specific processing elements in this study.

Each PE contains several pipelines, a register set, a frame memory, and a Local Scheduler Element. There are two types of pipelines in each PE: Synchronization Pipeline (SP) and execution pipeline (EP). SP is responsible for executing load and store phases. EP is responsible for the execution phase of a thread. All pipelines in the PE share the same register set. When a thread starts to execute, the LSE assigns one of the available register sets to it. Then the thread passes trough different pipelines to execute its load, execution and store phases. When the thread finishes its execution, its frame and register set become available for other threads.

All requests that originate from a PE (either for memory accesses or for new resources) are handled by its LSE. If the request is local (mapped to internal frame memory), then it is served immediately. On the other hand, in case of a request for remote resource (new frame or a remote memory location), the LSE forwards the request to the DSE (in case of a request for a new frame or a store to a remote frame) or to the main memory (in case of a request for a remote memory location). In the both cases, the request must pass through intra-cluster network, causing a longer latency.

D. Terminologies

In this paper we use the following terminologies to differentiate the types of threads:

- **Pthread**: a thread created by an application [5]. These threads are specified by the programmer and can have arbitrary length.
- **DTA-thread**: a sequence of load, execution and store phases. Each pthread contains many small DTA threads that are generated by the compiler.

III. The Analysis of Clustal-W

In molecular biology, Clustal-W [6] is an important program for the simultaneous alignment of nucleotide or amino acid sequences. It implements the most widely used approach of multiple sequence alignments that use a heuristic search known as progressive alignment. However, the progressive alignment algorithm suffers a high computational complexity and consequently it may take a lot of time to complete. A traditional technique to speedup this task is to parallelize the application as much as possible. As a result, the progressive alignment algorithm is a perfect candidate to fully utilize the machine resources and to stressfully test the overall performance as well as the scalability of massive parallel systems.

In this work, we use the parallel implementation of Clustal-W that comes from the BioPerf [7] benchmark suite. This version of Clustal-W is divided in three phases:

1) The first phase, pair-wise alignment, takes between 60% and 80% of the execution time in a traditional uni-processor machine.
2) The second stage forms a phylogenetic tree using the Neighbor-Joining algorithm with the aligned sequences generated at the previous step.
3) The third step progressively aligns the sequences according to the tree branching order obtained at the second step.

In order to better understand the performance of Clustal-W, we characterize it by using Oprofile [8]:

```
Counted CPU_CLK_UNHALTED events
samples    %          image name        symbol name
526230  50.0853       clustalw-smp        parallel
271232  25.8152       clustalw-smp        pdiff_reverse
215633  20.5234       clustalw-smp        pdiff_forward
15849    1.5085       clustalw-smp        pdiff
14632    1.3926       clustalw-smp        calc_prf1
2535    0.2413        clustalw-smp        diff
1731    0.1648        clustalw-smp        prfalign
1534    0.1460        clustalw-smp        aln_score
```

The above profiling results show that the pairwise alignment — function *parallel()* — is the most CPU-consuming part of the execution. Over 50% of execution time spent in it. Hence, we focused our analysis on this function.

IV. Experimental Methodology

A. The Methodology

As a starting point, we use the Clustal-W implementation that is parallelized at the application level by using Pthreads primitives. In order to better utilize the underlying architecture, the pthread library was implemented by using DTA assembly primitives. Whenever a new pthread is requested in the code, the compiler creates a DTA thread to exploit the dataflow execution model. Instead of relying on locking and semaphores, pthread primitives for synchronization are written to rely on the synchronization counters and the dataflow communications that are supported by the DTA hardware.

The standard ANSI-C version of Clustal-W is translated into DTA assembly code by using a modified Scale [18] compiler. This compiler is extended with a DTA backend and custom implementation of libraries for the generic I/O operations (*open()*, *read()*, *write()*, . . .). The tests are executed on a DTA simulator which is based on the code of sdfsim-3.0.0 [19].

For the input dataset, we use the standard input dataset from the BioPerf suite, 1290.seq (66 sequences of length almost 1100). The input is replicated 32 times to create at least 512 workers via *pthread_create()* and to avoid "empty" worker threads. All the created threads have a non-empty input sequence to align. The total number of sequences to be aligned during each run is 66 x 32 = 2112.

ISBN: 978-1-61208-154-0

B. The Decomposition of the code

Since the main goal of this work is to better understand the TLP exploitation from the point of view of the architecture, we focused on the statistics for the highlighted section of the code (see Section III). This section starts after all threads are created. Namely, it starts after the call to the function *pthread_cond_broadcast()*.

As mentioned before, the main idea is to allow the DTA to coexist with other types of processors in the system (e.g. general purpose processors) and to launch only the parallel part of the code to a dedicated DTA hardware. In this way, the General Purpose Processor (GPP) executes only for the sequential part of the application. The GPP can be optimized to exploit the available Instruction Level Parallelism (ILP) of the program that is not suitable for DTA. On the other hand, the TLP-optimized DTA hardware runs the threaded portion of the code. By analyzing the code and the arguments of *pthread_create()* function, it is fairly easy to identify the portion of the code that has the high degree of parallelism. Therefore, even at compile time, we can decide how to split the code into sequential and parallel parts.

V. RESULTS AND ANALYSIS

Firstly, we estimate the instruction mix that is created at runtime:

Total number of instructions:	2,323,534,075
Total frame-memory references:	1,097,224,437
Total data-memory reference:	91,925,906
Total READs from frame:	548,612,219
Total WRITEs to frame:	548,612,218
Total READs from memory:	78,648,743
Total WRITEs to memory:	13,277,163
Total number of DTA-threads created:	50,259,676
Instructions per DTA-thread (average):	47.2305

As we can see from the dynamic statistics, 51.12% of the instructions are memory accesses but only 3.96% of them are the data memory access. The 96.04% of memory accesses goes to the frame memory and these accesses are used for the communication among DTA-threads. This is evidence that the compiler is able to generate many smaller DTA threads inside each thread created by pthread library. Since the frame memory locates near the processor, the advantage of DTA is to leverage this memory for communication among threads instead of using other ways of communication (e.g. function calls with a stack frame).

The second set of experiments test the scalability of several different configurations of DTA while varying the miss latency. The results are shown in Figure 4 and Figure 5. In the DTA architecture, the stalls in the pipelines are only a consequence of LOAD misses. These stalls delay the fetch and completion of future instructions. Therefore, only the Synchronization Pipelines are affected by pipeline stalls. Increasing the memory latency reduces the contribution of the EPs to the total execution time. On the other hand, the activity of SPs strongly influences the performance in terms of total execution time and speedup. This can be seen from the graphs for execution time since the configuration with 2 SPs and 1 EP performs almost as well as the configuration with 2 SPs and 2 EPs. Also, increasing the memory latency gives better speedup (but the execution time increases also) since there is a higher probability to use more SPs.

When the latency is increased, the amount of stalls in SPs, as shown in Figure 6, is also increased so do the number of threads in the system as illustrated in Figure 7. The number of threads is increased because there are many new threads are waiting. If we have a longer stall in a pipeline and a new DTA-thread arrives, there is a less probability to find an available SP that can start to execute the load phase of the thread. This means that the benefit of increasing the number of SPs is greater than increasing the number of EPs when the memory latency increases.

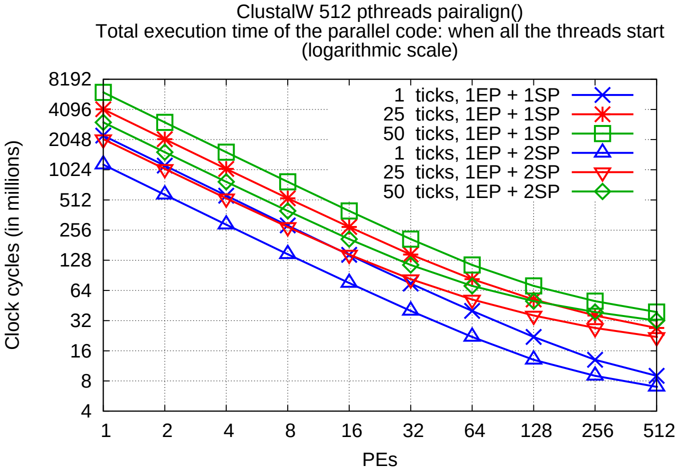


Figure 4 Total execution cycles with variant PE configurations and Load miss latency. Memory latencies are 1, 25, and 50 clock cycles (ticks)

This fact is due to the memory latency issue and execution model. Statistically, at any time of the execution of a program, 2/3 of the available DTA-threads need to be served by a SP (due to load and store phases) and only 1/3 of the threads need an EP to continue their execution. This is also in line with the obtained results on execution time and speedup that are discussed above. Hence, in a dataflow base multithreaded architecture, it is possible to get the same performance by using a cheaper configuration, namely, by decreasing the number of architectural elements that are dedicated to calculations (EPs) and by increasing the number of architectural elements dedicated to communication (SPs).

To quantify the area saved by decreasing the number of EPs, we evaluated the number of transistors and the required area for all configurations that we used in our experiments. Our method is based on the analytical method provided by the "SimpleScalar directed estimation tool" [20]. This tool takes the number of functional units (such as ALUs, Load/Store units,...) as input and produces the estimate of required transistor count and the area. The results are displayed in Table 1. As shown in Table 1, the "1 EP + 2 SPs" configuration saves about 32.5% of the eara with respect to the "2 EPs + 2 SPs" configuration while keeps the same performance, leading to a space and energy efficient architecture.

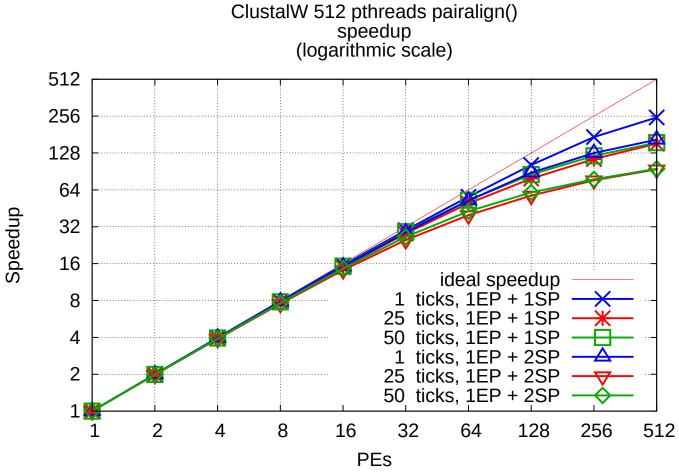


Figure 5. Speedup with different PE configurations and LOAD miss latency. The memory latencies are 1, 25, and 50 clock cycles (tickets).

VI. RELATED WORK

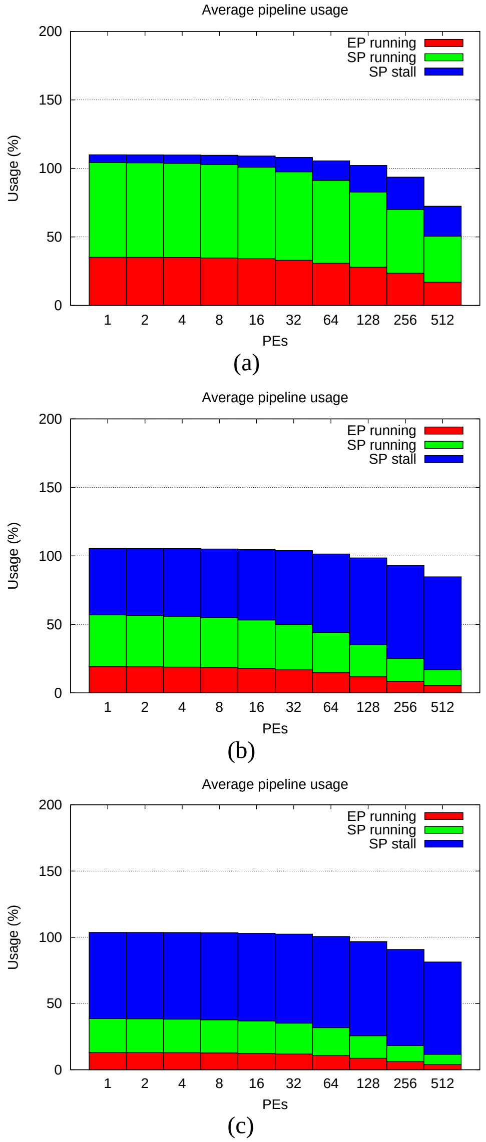
Several researchers in the past have studied the performance of multiple versions of parallel Clustal-W in a wide range of different multi-core architectures. The solution presented by [9] uses message-passing libraries on a PC cluster (ClustalW-MPI). Besides the software approach, new approaches that are using reconfigurable hardware (such as FPGA) have been presented [10]. Vandierendonck et al. [11] explored the performance of a Clustal-W implementation optimized for the Cell BE architecture (ClustaIW-Cell). Liu et al. also explored the optimizations of Clustal-W using the GPU acceleration of nvidia GeForce 7800 GTX (ClustalW-GPU) [12].

From the hardware perspective many decoupled architectures have appeared in the past few years. Speculative Data- Driven Multithreading [13] is an architecture that is based on decoupling principle. This architecture identifies miss streams, i.e. streams of instructions that are likely to cause cache misses and executes them in a multithreaded fashion in order to perform pre-fetching. HiDisc (Hierarchical Decoupled Instruction Stream Computer) [14] is an architecture that reduces memory latency by pre-fetching at both hardware and software level. Pre-fetching is accomplished by separating the instruction stream into one for regular execution and one for memory accesses.

Moreover, multi-core/many-core architectures have gained the most attention in the industry recently. IBM Cyclops-64 (C64) [15] is a multi-core-on-a-chip processor that consists of 80 processors (or cores). Each processor has two SRAM memory banks that can be configured either as scratchpad or global memory. Plurality [16] is a multi-core system that uses a pool of RISC processors with uniform memory, hardware scheduler, synchronizer and load balancer. SUN Microsystems UItraSPARC T2 [17] is a multithreading multi-core chip capable of running 64 threads at the same time. The main difference between the existing architectures and DTA is that DTA is a multithreaded architecture that uses the scheduled dataflow programming model and decouples

TABLE I
AREA AND TRANSISTOR USAGE ESTIMATION:
PIPELINES AND DIFFERENT PROCESSING ELEMENTS

Element	Number of Transistors	Area (in M λ^2)
1 EP	419,597	685.47
1 SP	225,554	368.49
1 PE = 2 EP + 2 SP	1,290,302	2,107.92
1 PE = 1 EP + 2 SP	870,705	1,422.45

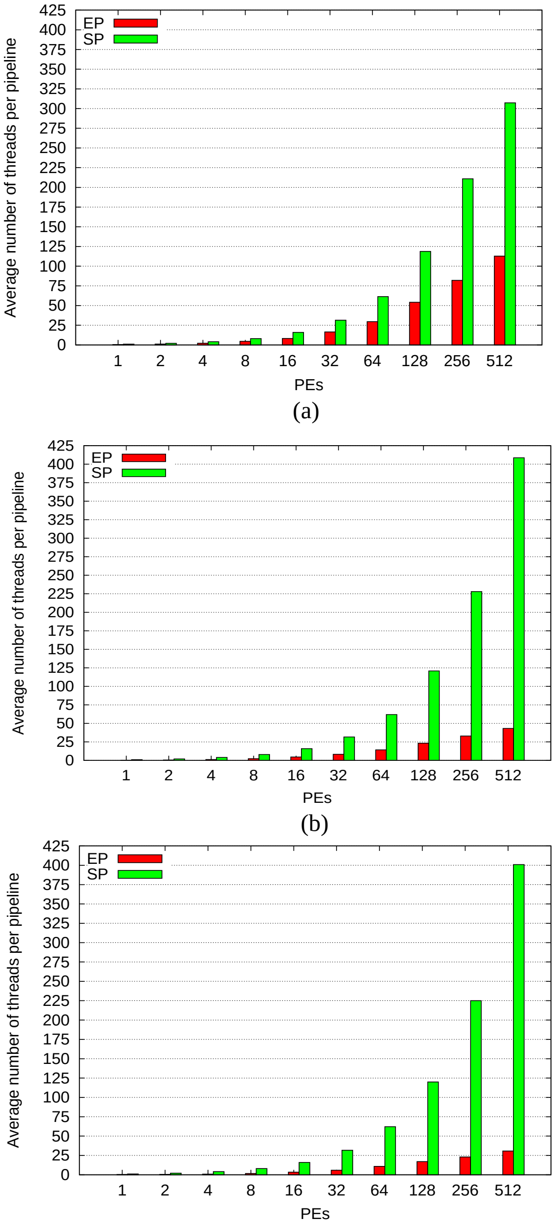


(a)

(b)

(c)

Figure 6. Average pipeline usage (1PE = 1 EP + 1 SP) with a Load miss latency of 1, 25, and 50 clock cycles, denoted by (a), (b), and (c). Total usage can be greater than 100% when EP and SP code are executed in parallel



(a)

(b)

(c)

Figure 7. Average number of DTA-threads running in EPs and SPs using different LOAD miss latencies: 1, 25, 50 clock cycles(ticks)

the memory accesses from the threads' execution.

VII. Conclusion

This paper analyzes the tradeoffs in design of a multi-threaded architecture, and their effect on the exploitation of the Thread Level Parallelism. By using the bio-informatics application Clustal-W as a benchmark, we evaluate our Decoupled Threaded Architecture (DTA) with different number of architectural elements that are dedicated for computation and for communication among threads. We have experimentally verified that the contribution of the hardware dedicated to communication (Synchronization Pipelines — SPs) is much greater than the contribution of the hardware dedicated to computation (Execution Pipelines — EPs). This shows that our architecture is based on the coarse-grained dataflow execution model that emphasizes the communication in a producer-consumer fashion. We found that the configuration in which each Processing Element (PE) is composed of 1 EP and 2 SPs is able to achieve the performance which is very close to that of the configuration with "2EP + 2SP". This can save the area of each PE and obtain benefits in terms of power significantly.

Acknowledgements

We thank Prof. Krishna Kavi for providing the modified Scale compiler and for his useful comments. This work was partly funded by the European FP7 project TERAFLUX id. 249013[21], HiPEAC IST-217068, and IT PRIN 2008 (200855LRP2).

References

[1] R. Giorgi, Z. Popovic, and N. Puzovic, "Dta-c: A decoupled multi-threaded architecture fhr cmp systems," in *Proceedings of IEEE SBAC-PAD*, Gramado, Brasil, Oct. 2007, pp. 263-270. [Online]. Available: http://www.dii.unisi.it/ popovic/docs/Giorgi-DTA.pdf

[2] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation," *IEEE TRANSACTIONS ON COMPUTERS*, pp. 834-846, 2001.

[3] R. Giorgi, Z. Popovic, and N. Puzovic, "Introducing hardware TLP support or the Cell processor," in *Proceedings of IEEE International Workshop on Multi-Core Computing Systems. Fukuoka, Japan*. Los Alamitos, CA, USA: IEEE Computer Society, Mar 2009, pp. 657-662.

[4] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490-504, Apr 2001.

[5] B. Nichols, D. Buttlar, and J. P. Farrell, "Pthreads programming: A POSIX standard for better multiprocessing," *Reilly, California*, 1996.

[6] J. D. Thompson, D. G. higgins, and T. J. Gibson, "CLUSTAL w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *NUCLEIC ACIDS RESEARCH*, vol. 22, pp. 4673-4673, 1994.

[7] D. A. Bader, Y. Li, T. Li, and V. Sachdeva, "BioPerf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications," in *Proceedings of the IEEE International Workload Characterization Symposium*. 2005, pp. 163-173.

[8] J. Levon and P. Elie, "Oprofile: A system profiler for linux," Web site: *http://oprofile. sourceforge. net* , 2005.

[9] K. B. Li, "ClustalW-MPI: ClustalW analysis using distributed and parallel computing," *Oxford Univ Press*, 2003, vol. 19.

[10] T. Oliver, B. Schmidt, D. Nathan, R. Clemens, and D. Maskell, "Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW," *Oxford Univ Press*, 2005, vol. 21.

[11] H. Vandierendonck, S. Rul, M. Questier, and K. D. Bosschere, "Experiences with parallelizing a bio-informatics program on the cell BE," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4917, p. 161, 2008.

[12] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, "GPU-ClustalW: using graphics hardware to accelerate multiple sequence alignment," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4297, p. 363, 2006.

[13] A. Roth and G. S. Sohi, "Speculative Data-Driven multithreading," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, vol. 37, 2001.

[14] W. W. Ro, S. P. Crago, A. M. Despain, and J. L. Gaudiot, "Design and evaluation of a hierarchical decoupled architecture," The *Journal of Supercomputing*, vol. 38, no. 3, pp. 237-259, 2006.

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

[16] "Plurality architecture." [Online]. Available: http://www.plurality.com/architecture.html. June 29, 2011.

[17] M. Shah, J. Barreh, T. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, *et al*., "UItraSPARC 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.

[18] K. S. McKinley, J. Burrill, M. D. Bond, D. Burger, B. Cahoon, J. Gibson, J. E. B. Moss, A. Smith, Z. Wang, and C. Weem, "The Scale compiler," *Technical report*, University of Massachusetts, 2001. http://ali-www. cs. umass.-edu/scale, 2005.

[19] SDFsim 3.0.0, http://csrl.unt.edu/sdf/sdfhowto.php June 29, 2011.

[20] M. Steinhaus, R. Kolla, J. L. Larriba-Pey, T. Ungernr, and M. Valero, "Transistor count and Chip-Space estimation of simulated microprocessors," *Research report UPC-DAC-2001-16*, UPC Barcelona Spain, 2001.

[21] http://www.Teraflux.eu June 29, 2011.

[22] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, Dec 2009, Notre Dame, IN, USA, pp. 469-480.