DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems

Roberto Giorgi, Zdravko Popovic, Nikola Puzovic
Faculty of Information Engineering, University of Siena, Italy
{giorgi, popovic, puzovic} @ dii.unisi.it

Abstract

One way to exploit Thread Level Parallelism (TLP) is to use architectures that implement novel multithreaded execution models, like Scheduled Data-Flow (SDF). This latter model promises an elegant decoupled and non-blocking execution of threads. Here we extend that model in order to be used in future scalable CMP systems where wire delay imposes to partition the design.

In this paper we describe our approach and experiment with different distributed schedulers, different number of clusters and processors per cluster to show good scalability of our architecture. We describe our approach and present initial results on system scalability and performance. We suggest design choices to improve the scalability of the basic design.

1. Introduction

As the technology improvements have made possible to place more than one processor on a chip, single-Chip Multi-Processors (CMP) with many processing elements have become reality [1].

The architecture we are introducing is based on the Scheduled Data-Flow (SDF) execution paradigm [2] [3]. SDF exploits simple yet powerful paradigm that is based on coarse grained dataflow and multithreading. Program is partitioned in non-blocking threads where all memory accesses are decoupled from the execution. Each thread is divided into three phases: pre-load phase (where thread reads all data), execution phase (where thread is executed without performing any memory accesses) and post-store phase (where results are written back).

Scheduled Dataflow Architecture (SDA) is the original implementation of the SDF execution paradigm. Basic SDA processing element contains pipelines for thread execution, frame memory and the logic to support the lifetime of the threads. Despite its simplicity, original SDA faces several problems that cause limitations to the usability of the architecture. In SDA, each pipeline must be able to communicate with local memory, register file and control logic in one cycle, and this is a reasonable assumption as long as we have a few pipelines in a processor. However, as the number of pipelines grows, and with limitation imposed by the wire delay problem, it becomes harder for all components in the processor to communicate with each other in one cycle, and this becomes the limiting factor of the SDA scalability

Our main objective is to exploit the TLP but in terms of decoupled access/execute, non-blocking threads. We have expanded the SDF architecture in order to make it scalable, and tried to improve the scalability by clusterizing resources and balancing workload among the processing cores.

The rest of the paper is organized as follows: First we introduce the SDF execution paradigm in Section 2. In Section 3, we describe our idea, Decoupled Threaded Architecture (DTA-C, where "c" stands for clustered). Section 4 contains experimental results, and in sections 5 and 6 we discuss future research and present the conclusions.

2. Execution paradigm

In order to explain the execution paradigm, we are going to use the example given in Figure 1. In this example a program forks four threads. Thread 1 (TH_1) reads variables a, b and c from the input and sends them to other threads. Threads TH_2 and TH_3 can execute in parallel because calculation of *f(a, b)* and *g(a, c)* is independent while TH_4 needs the results calculated by these two. In order to ensure that any thread won't start executing before all of its data is ready, a synchronization count (SC) has been associated to each of them. This synchronization count contains the number of input data that thread needs in order to run. In our example, TH_2 needs to wait for *a* and *b*, so its synchronization count is 2. When data is stored for a thread, synchronization count is decremented and once it reaches zero that thread is ready to execute.

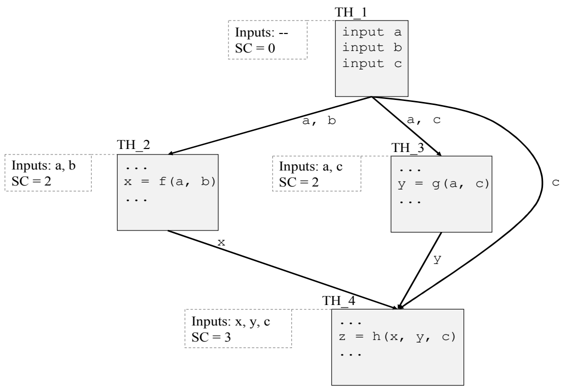

Figure 1. Example program in SDF.

For communicating data among threads, SDF execution model uses frames. When a new thread is created, the system assigns it with a fixed size frame that can be written by other threads and read only by the thread that it belongs to. In implementations of SDF paradigm (SDA, DTA-C), a frame is a part of a frame memory, which is a small on-chip memory.

As mentioned in the introduction, each thread in SDF is separated into three phases: pre-load phase, execution phase and post-store phase. In the pre-load phase, data is read from the frame memory, shared data is read from the main memory and stored into local registers. In the execution phase, thread executes without any memory accesses, and uses only the data that are in the register file. When the computation is finished, thread performs its post-store phase where it writes data to other threads' frames and writes data to main memory if needed.

3. DTA-C description

DTA-C (Decoupled Threaded Architecture - Clustered) is based on the execution paradigm of SDF. DTA-C adds the concept clusterizing resources. Each cluster in the architecture has the same structure and can be considered as a high-level tile. Figure 2 shows the overall DTA-C.

A cluster consists of one or more Processing Elements (PEs) and a Distributed Scheduler Element (DSE). The set of all DSEs constitutes the Distributed Scheduler (DS). The amount of resources (in particular the number of PEs) that can be placed in one cluster is defined by the fact that we want all resources within a cluster (e.g., PEs) to be accessible in one clock cycle. This property of the cluster logically leads to the use of a fast interconnection network inside the cluster (*intra-cluster* network), and the use of a slower but more complex network for connecting all clusters (*inter-cluster* network). The actual amount of processing elements that can fit into one cluster will depend on the technology that is used.

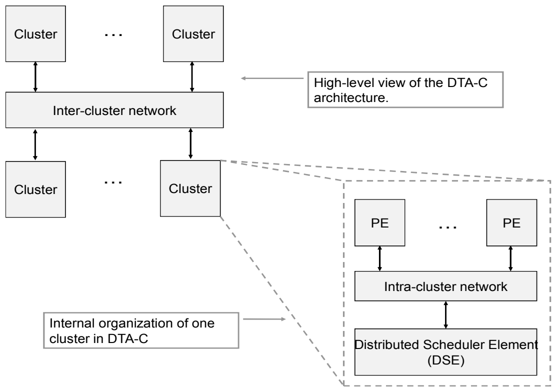
Cluster
...
Cluster
High-level view of the DTA-C architecture.
Inter-cluster network
Cluster
...
Cluster
PE
...
PE
Intra-cluster network
Internal organization of one cluster in DTA-C
Distributed Scheduler Element (DSE)

Figure 2 Architecture organization.

Each processing element in the architecture has its Unique Processing-element Identifier (UPI), which consists of pair: cluster id – CID, processing element id – PID. Also, each frame in the architecture has its Unique Frame Identifier (UFI), which is defined by the triplet: CID, PID, and frame number – FN. Here, CID and PID are designating the cluster and processing element where the frame resides, and FN is the number of the frame in the local frame memory. The amount of threads assigned to one PE is also limited by the fact that the frames of all assigned threads are stored in the local frame memory, which has a limited capacity. When data is sent to a frame, stores are sent to the inter-cluster network only if CID is different from the CID of the sender. This simplifies the routing and reduces bandwidth requirements (described in Section 3.2).

3.1 PE Architecture

Figure 3 shows the structure of a processing element in DTA-C tile. Each processing element contains pipelines, frame memory, register file and local scheduler. There are two pipelines: synchronization pipeline (SP) that is responsible for executing pre-load and post-store phases and execution pipeline (XP) that is responsible for the execution phase of the thread. Both pipelines share the same register set, so that thread can change pipelines without context switching.

The Local Scheduler (LS) is responsible for communicating with other processors and clusters. When we refer that a PE is communicating with someone, it actually means that the communication is done by the local scheduler of that PE. This is just to avoid the confusion between local and distributed scheduler. Local scheduler serves the requests for new resources and for data communication (Section 3.2). Frame memory is a fast word–addressable memory that holds frames for the threads assigned to this processor.

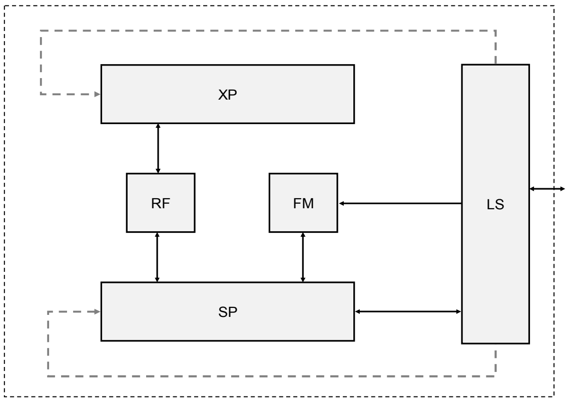

Figure 3. The structure of the processing element.

3.2 Scheduling

We have a two level scheduling: i) handled by the Local Scheduler (LS) inside each PE and ii) handled by the Distributed Scheduler Element (DSE) inside each cluster.

The Distributed Scheduler Element (DSE) consists of a list, that we call FFT (Free Frame Table), which contains information about the number of available frames in its cluster (ordered by the availability) and a FALLOC Request Queue (FRQ), which accounts for Frame Allocation (FALLOC) requests. The Local Scheduler and the Distributed Scheduler are responsible for assigning resources (i.e., continuations and frames) to the threads and for keeping track of processor usage in order to balance the load in the system. The main control data is packed into (what we call) a Continuation. The Continuation is a placeholder for the information needed to handle the lifetime of the thread, similar to [3], and a frame is the storage for the data needed for thread execution. The requests for obtaining new frames for the threads are sent from the PEs to the DSE and data and control information can be exchanged even between PEs, if they are in the same cluster, otherwise it must be done through the DSE that can do the inter-cluster routing. The DSE serves requests by sending the Continuation containing the information about the frame pointer of the assigned frame. The communication among PEs in the same cluster, PEs and their DSE and among DSEs of different clusters is all done via messages.

There are six types of messages: FallocRequest (request for the new frame allocation by sending the continuation), FallocResponse (response to frame allocation sending the continuation with the frame identifier assigned), FFree (request to release the frame), Broadcast (availability request to other clusters), BroadcastResponse (availability response) and DataStore (sending data to other threads). All messages except for DataStore are dealing with resource assignment – creation and release of continuations.

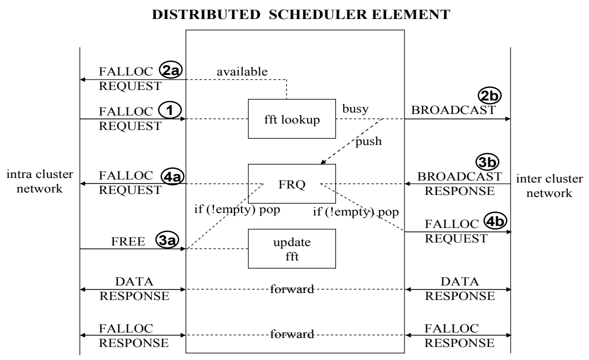

Figure 4. Algorithm of Distributed Scheduler Element – phases in the operation of DSE are numbered from 1 to 4.

Sending, receiving and processing of messages are done in schedulers, both local and distributed ones. The Distributed Scheduler Element is forwarding messages from one processing element to another and makes decision about where the new thread is going to be executed (Figure 4). When the DSE receives a new FallocRequest message, it checks the availability list for available frames in the local cluster. If there is any that is available, it forwards the message to the processing element that has the lowest number of threads assigned to it (inside that cluster). If all of them are busy and without any more available frames, it puts the message in FRQ and sends a Broadcast message to all the other clusters. Messages can be removed from the queue in two cases: i) when Ffree message arrives; this means that some thread has finished its execution on one of the local PEs and therefore has execution resources available or ii) when BroadcastResponse message arrives from some other cluster that can execute new thread. In the first case, DSE just forwards the message from the FRQ to the local PE that sent Ffree message. In the second case the message is forwarded to (remote) DSE of another cluster, which then forwards the message to a PE in its cluster. Other messages (DataStore message and FallocResponse message) are simply forwarded to their destination.

The Local Scheduler operates as described in Figure 5. When it receives a FallocRequest message, it dedicates one free frame for the execution of the new thread and then sends a FallocResponse message to the processing element that issued the request. When a FallocResponse message is received, it means that a threads' continuation has a frame identifier assigned to it, and a PE inside that cluster can continue the execution. A DataStore message is sent for each STORE instruction and, upon the reception of this message, the related data is stored in the destination frame. If multiple STOREs need to be sent to one destination, we can group them into one DataStore message. If the destination frame is in the same processing element, no message is sent and the store is done locally. When all the data for a thread is stored, that thread can start execution (Synchronization Count equal to zero).

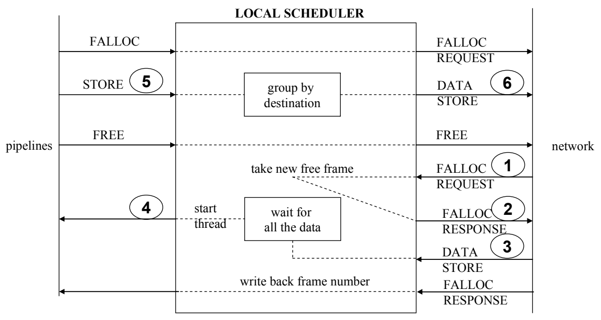

Figure 5. Algorithm of Local Scheduler – phases in the operation of LS are numbered from 1 to 6.

With this algorithm for allocating new frames, we have introduced additional delay for frame allocation (FALLOC) instructions that was not present in original SDF architecture. This comes from the need to distribute threads across many processors and to achieve parallel execution. However, we try to eliminate this delay by a technique which is described in the next section.

3.3 Non-blocking frame assignment

Figure 6 shows the situation that occurs when a request for a new thread is issued. On the left, we can see that the pipeline executing current thread blocks and waits for the response from the Distributed Scheduler. In order to overcome this problem, we are introducing two new tables that will reside in the local scheduler - *Map Table* and *Store Buffer* (Figure 6 on the right). When the request is sent to the DSE, the processor continues the execution of the current thread without stopping. Instead of a real frame pointer, it receives a virtual frame pointer (VFP), which is unique for each thread, and is generated locally by the processor. At the same time, a new entry with generated VFP is added in the map table. When the response from scheduler arrives the map table entry with corresponding VFP is updated with a real frame pointer (a Unique Frame Identifier or UFI), and the actual location of the frame is associated with the VFP.

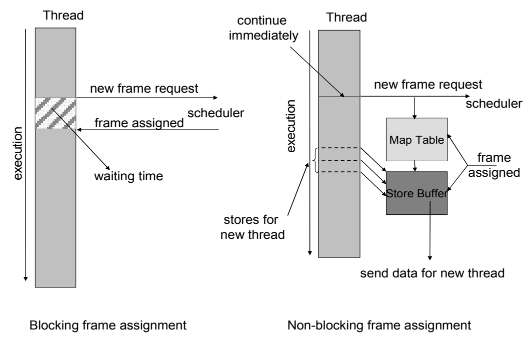

Figure 6. Non blocking resource assignment.

The only instructions that actually use a Unique Frame Identifier are STORE instructions, which store data to the frame of the new thread. Since execution continues immediately after the request is issued, it may happen that a STORE instruction executes before the response has arrived. In that case, we put the VFP of the new thread together with the frame offset and data to be stored into the Store Buffer. When the response arrives, we update the Map Table and associate the newly received Unique Frame Identifier to the appropriate VFP. In case that there are pending STOREs in the Store Buffer, we map their VFP to the real Unique Frame Identifier (only for the one that just arrived) and send the data to the appropriate location. In the other case, when a STORE occurs after the response, we already have the real Unique Frame Identifier for the new thread, so we can perform mapping immediately and send the data to its location.

With this approach, processor is free to continue the execution, while waiting for the frame response is done in parallel, so no blocking due to frame assignment occurs during the execution of the thread. Even in the case when response doesn't arrive before the thread finishes its execution, we don't need to wait for it and processor can take some other thread and start to execute it. When the response arrives, all pending stores in the Store Buffer will be resolved and data will be sent to the appropriate frames. Of course, if the Map Table becomes full, we will still suffer from blocking, but the chances of that to happen are small unless we have a huge network contention and a big number of newly created threads at the same time. This approach is evaluated in Section 4.3.

4. DTA-C Experiments

For the evaluation of DTA-C, we have developed a cycle-accurate simulator in C++. This simulator implements the DTA-C as described in the previous section and uses the perfect memory model. Our initial goal was to estimate the scalability and different scheduling algorithms, so the assumption that the main memory can be accessed in one cycle is acceptable for this purpose.

Since compiler is still an open research issue, we had a limited choice of benchmarks to work with. Our analysis were performed with *radix-sort* (SPLASH2, [4]) and *bitcnt* (MiBench, [5]), both of which were hand-coded.

4.1 Scalabilty

Figure 7 shows the speedups (where the baseline is the number of execution cycles on one processing element) for radix-sort benchmark. Number of clusters is represented on the x-axis, and speedup is on the y-axis. In all the Figures, the number of Processing Elements Per Cluster (PEPC in Figures 7, 8, 9, 10, 12) is constant along one line, and number of clusters is varied in order to demonstrate the scalability of the architecture.

In this configuration, radix sort benchmark forks 512 threads that sort the data. Since the time to fork all threads is significantly smaller then the execution time of each of them, we can assume that all threads can run at the same time (if there are enough processing elements to execute them all). We can see from this Figure that scaling is almost perfect (e.g., speedup of 57 for 64 processing elements, (i.e. 16 PEPC and 4 clusters).

Bitcnt benchmark exhibits slightly different behavior. Running time of each forked thread (that serves for counting) is comparable with the time needed to fork all of the unrolled threads. In comparison with radix sort, this benchmark has also more pre-load/post-store communication, so network bandwidth is a limiting factor here. Speedup for this benchmark is shown in the Figure 8.

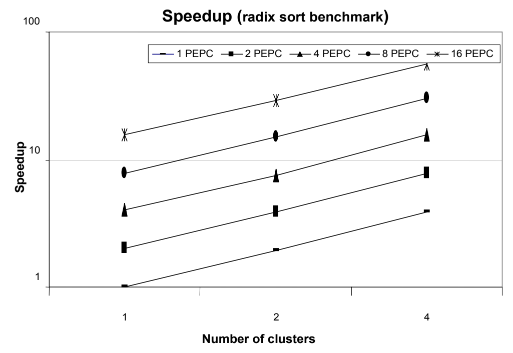

Figure 7. Scalability of radix-sort benchmark. PEPC is the number of Processing Elements Per Cluster.

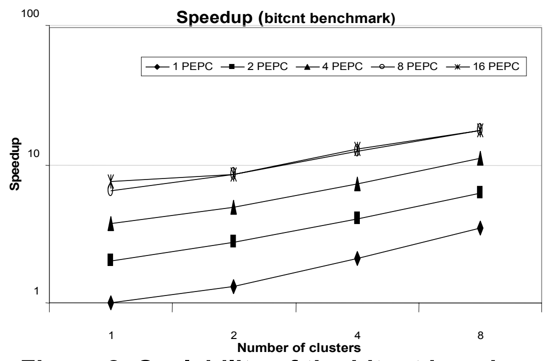

Figure 8: Scalability of the bitcnt benchmark.

It can be seen that speedups for the configuration with 8 processing elements per cluster and the configuration with 16 processing elements per cluster are almost the same. This saturation occurs because intra-cluster bus is not very large (128 bits wide) and messages are transferred only at a rate of one at each bus cycle. When we enable the intra-cluster network to send two messages in one cycle (by using a double issue bus) scalability improves.

Figure 9 compares the configurations with a double issue bus (Double Speed - DS) and the configurations with a baseline bus (Single Speed - SS). We can see that when using faster interconnection, the architecture scales well as we increase the number of Processing Elements Per Cluster (PEPC) from 8 to 16. It can also be seen that the configurations with faster network perform much better then the ones with regular network. The configuration with 8 processing elements and a double issue bus is faster than the corresponding configuration with regular bus, and also faster than the configuration with 16 Processing Elements Per Cluster and a baseline bus.

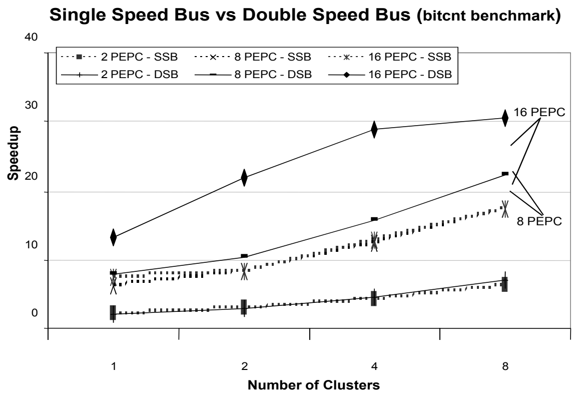

Figure 9. Scalability of bitcnt benchmark with faster intra-cluster network.

4.2 Improving the scheduling

In order to determine the best scheduling algorithm for the distributed scheduler, we have experimented with several different scheduling algorithms.

The simple scheduling algorithm, originally implemented in the architecture always tries to allocate a new frame in the current cluster, and forwards the request to the remote cluster only if there is a lack of resources. However, this may not be an optimal approach, because it doesn't consider the dependencies among the threads, thus we have considered (what we call) *ISA-helped scheduling* as an alternative. This approach is similar to a "coarse" level scheduling for WaveScalar architecture [6], presented in [7].

The idea is that the program can give scheduling hints to the hardware by using modified instructions for new frame creation. One new instruction was added to forces the Distributed Scheduler Element to send a request for new frame to a remote cluster, while in the case of a regular FALLOC instruction the behavior remains the same (trying to serve the request within the cluster where it originated). Hardware cost of this solution is negligible – e.g., a single counter to keep track of the next cluster that will serve the request.

Figure 10 shows the comparison between simple and ISA-helped scheduling. We have modified the bitcnt benchmark to force the forking of all threads that execute one function to the same cluster (a different cluster for each function if possible). In all configurations, ISA-helped scheduling outperforms the simple scheduling algorithm. With bigger number of clusters, ISA-helped "affinity" scheduling achieves better distribution in workload balance among clusters, and outperforms the simple scheduling almost by a factor of 2.

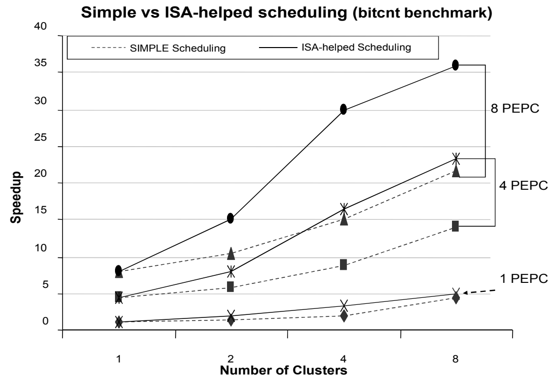

Figure 10. Simple vs. ISA-helped scheduling.

4.3 Non-blocking resource assignment

Figure 11 shows the comparison between the blocking and non-blocking resource assignment.

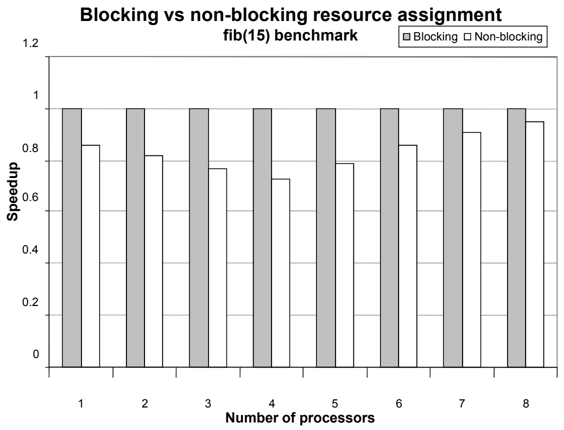

Figure 11. Blocking vs non-blocking resource assignment.

All results were normalized to the number of execution cycles for the configuration with blocking resource assignment (vertical value of 1), and the number of processing element was varied. In order to obtain these results, we have used double issue bus as interconnection (to eliminate the saturation caused by the network). Simple recursive version of the Fibonacci(N) program was used (N=15). As the graph shows, non-blocking approach is always faster, with performance improvements ranging from 5.2% to 23.6%.

4.4 Comparison with TMS320C6711

In order to compare the DTA-C with a real world system, we have compared it with TMS320-C6711 (TMS) [8]. This processor is a VLIW with 6 ALUs (2 integer and 4 floating-point) and 2 multipliers. Figure 12 shows this comparison.

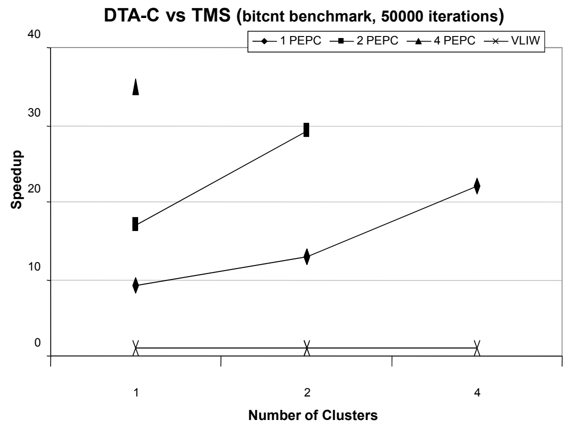

Figure 12. Comparison of DTA-C and TMS320-C6711.

Since TMS has eight execution units, we have compared it with DTA-C configurations that are the most similar to it. Each configuration had up to 4 processing elements (total of 8 pipelines) distributed in 1, 2 or 4 clusters, and each of them used simple scheduling and prefect memory. All configurations outperform TMS by the factor ranging from 9 up to 35. This comes from the fact that bitcnt benchmark for DTA-C is hand-parallelized and a lot of TLP is exposed in it while for TMS we used a TMS320C6000 C/C++ compiler. Another reason is that bitcnt is an integer benchmark, and doesn't use any of the available floating-point units.

The configuration with only one cluster shows the best performance because all communication is going trough intra-cluster network. On the other hand, configurations with more clusters are less efficient because part of the communication needs to go trough the slower inter-cluster network, but still, they are better than TMS.

5. Related Research

Decoupled architectures have been studied for a long time. Among the first decoupled processors was Astronautics ZS-1 [9] processor that had separated instructions into two streams, one for fixed-point/memory address computation and other one for floating point calculations, and two streams were executed in parallel. More recently, several new decoupled architectures have appeared. Speculative Data-Driven Multithreading (DDMT) [10] 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 prefetching. HiDisc (Hierarchical Decoupled Instruction Stream Computer) [11] is an architecture that reduces memory latency by prefetching at both hardware and software level. Prefetching is accomplished by separating the instruction stream into one for regular execution and one for memory accesses.

The main difference between DTA-C and above mentioned decoupled architectures is that DTA-C fully exposes its programming model to the programmer who can exploit the available TLP. Also, all threads in DTA-C are explicitly separated into phases, and decoupling of memory accesses is complete.

MLCA [12] is a paradigm for parallel architectures that exploits parallelism at the task level using similar techniques that are used in a modern superscalar processors for exploiting ILP – register renaming and out-of-order execution. Similar to DTA-C is that implicitly this architecture uses dataflow at the task level but the programming model is still sequential.

One of the possible solutions for CMP systems are tiled architectures. Tiled architectures have some number of tiles (e.g. resources like processing elements, memory modules, register files, etc.) that are replicated on the chip and connected via on-chip network. Several recent proposals for CMPs are in fact tiled architectures: RAW [13], TRIPS [14], WaveScalar [6]. All of these architectures show good scalability but they are different in terms of how do they exploit the available parallelism in programs.

RAW architecture [12] is a multiprocessor architecture that tries to overcome the wire delay problem by using very simple in-order MIPS processors as tiles. Since processing elements are small and connected only to their neighbors, it is ensured that all wires are shorter than the length or width of a tile. This architecture is similar to DTA-C in sense that its resource division is performed based on what can be reached in one cycle, but it employs a conventional programming model that may not be suitable for future parallel systems and applications.

TRIPS [13] uses “medium size” tiling by allowing several different types of tiles for processing elements, memory, cache (instruction and data) and registers. Some of the above mentioned tiles can be reconfigured in order to exploit different types of parallelism. While

DTA-C employs dataflow execution at the thread level, and control flow-like execution inside the thread, TRIPS does the opposite by employing control flow execution at the thread level and dataflow execution inside the thread.

WaveScalar [6] has a huge number of simple processing elements, which communicate operands in such a way to employ dataflow execution model. This architecture employs real dataflow execution at the instruction level, and unlike DTA-C doesn't target TLP.

6. Conclusions

DTA-C uses a paradigm that is based on a non-blocking execution of threads and decoupling of memory accesses. Since threads communicate in a dataflow-like manner, this communication is used also for the synchronization among the threads. In this way, we are reducing the need for the use of regular synchronization primitives inside the threads and enable the non-blocking execution. Uniform tiles with simple processing elements enable us to clusterize resources in order to achieve better usage of the available transistors on the chip and obtain good scalability of the architecture.

With selected benchmarks from MiBench and SPLASH-2 benchmark suites we have shown that the architecture is scalable and identified the bottlenecks in our architecture. We have shown that by increasing the speed of the interconnection network, and by using advanced scheduling algorithm (both of which were identified as bottlenecks) we can increase the scalability even further.

7. Acknowledgements

This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6).

References

[1] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. Cambridge, Massachusetts, United States: ACM Press, pp. 2-11, 1996.

[2] K. Kavi, J. Arul, and R. Giorgi, "Execution and Cache Performance of the Scheduled Dataflow Architecture," *SPRINGER Journal of Universal Computer Science*, vol. 6, pp. 948-967, 2000.

[3] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation," *IEEE Transaction on Computers*, vol. 50, pp. 834-846, 2001.

[4] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd annual international symposium on Computer architecture*. S. Margherita Ligure, Italy: ACM Press, pp. 24-36, 1995.

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

[6] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. San Diego, CA, USA: IEEE Computer Society, pp. 291, 2003.

[7] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers, "Instruction scheduling for a tiled dataflow architecture," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. San Jose, California, USA: ACM Press, pp. 141-150, 2006

[8] "Texas Instruments TMS320C6711 Datasheet."

[9] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon, "The Astronautics ZS-1 processor," in *1988. ICCD '88., Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. Rye Brook, NY pp. 307-310, 1988

[10] A. Roth and G. S. Sohi, "Speculative Data-Driven Multithreading," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. Monterrey, MEXICO: IEEE Computer Society, pp. 37, 2001.

[11] 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, pp. 237-259, 2006.

[12] T. Abdelrahman, A. Abdelkhalek, U. Aydonat, D. Capalija, D. Han, I. Matosevic, K. Stewart, F. Karim, and A. Mellan, "The MLCA: A Solution Paradigm for Parallel Programmable SoCs," in *2006 IEEE North-East Workshop on Circuits and Systems*: pp. 253, 2006

[13] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, pp. 25-35, 2002.

[14] 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," in *Proceedings of the 30th annual international symposium on Computer architecture*. San Diego, California: ACM Press, pp. 422-433, 2003.