Decoupled Threaded Architecture

Roberto Giorgi, Zdravko Popovic, Nikola Puzovic[1]

Dept of Information Engineering, University of Siena, Via Roma 56, 53100 Siena, Italy

ABSTRACT

Decoupled Threded Architecture (DTA)[2] is designed to exploit Thread Level Parallelism (TLP) by using a sea of simple cores grouped into cluster for providing a scalable solution that copes with wire delay.
Our goals are i) to provide an aggressive mechanisms for decoupling memory accesses deriving from simple and complex data structures; ii) to implement a non-blocking execution of the threads.
Here we illustrate some of the concepts related to our research in implementing DTA.

KEYWORDS: multicore chip, non-blocking execution, decoupling

1 Introduction

Our architecture, DTA (Decoupled Threaded Architecture) is based on SDF execution paradigm [1, 2] and extends its main properties, like non-blocking execution of threads and decoupling of memory accesses from execution. The architecture is organized into clusters, where each cluster contain processing elements and portions of a distributed scheduler that takes care of allocating tasks on available processors in the same or some other cluster.The distributed scheduler details has been introduced in previous work [3]
The fact that in original SDF execution paradigm each thread needs to be strictly divided into pre-load, execution and post-store phases imposes limitations in thread creation for the compiler. When compiler encounters a memory access, it must find a way to move it to pre-load phase, or when that is not possible (for example when address needs to be calculated) it needs to crate new (sequential) thread. This leads to the fact that the size of the threads that can be extracted is very limited wand this fact directly affects the amount of available TLP.
In order to overcome this limit, we reviewed completely the way generic memory access happen
Since READ instructions can occur anywhere inside the thread, decoupling as proposed in SDF is violated. One part of our current research is to try and identify all memory accesses that will violate decoupling. Once this is done, data can be pre-loaded before thread starts to execute, and decoupled execution will be preserved. Details are given in Section 3.
Morover, one of our design goals is also to map our architecture as closely as possible on existing microarchitectures. To this end, we rely on a single pipeline for processing all

[1] {giorgi, popovic, puzovic}@dii.unisi.it
[2] This work is part of IST-FET project SARC funded under the EU's 6th Framework Programme.

(non-blocking) instructions instead of using two dedicated pipeline (execution+memory) as in the Scheduled Dataflow Architecture (SDA).
A new scheduling mechanism has introduced additional delay which affects the performance. We are investigating the possibility of non-blocking resource assignment by allowing the thread to continue even if response didn't arrive. Details are given in Section 2.

2 Non blocking resource assignment

Each thread in our architecture needs a frame in order to start getting the data from other threads and eventually to start the execution. Frame assignment is done in the scheduler. In a naïve implementation, when the new thread is forked processor sends the request for the new thread to the scheduler and continues the execution only when the response arrives. During the time spent in waiting for the response from scheduler the pipeline is stalled. This response delay doesn't exist in original SDF. It comes from the need to distribute threads across many processors and achieve parallel execution. We have measured this time with simple *Fibonacci* benchmark, and results for standard DTA architecture, as well as for DTA with double speed interconnection network (DTA-DS) are presented in Figure 1.

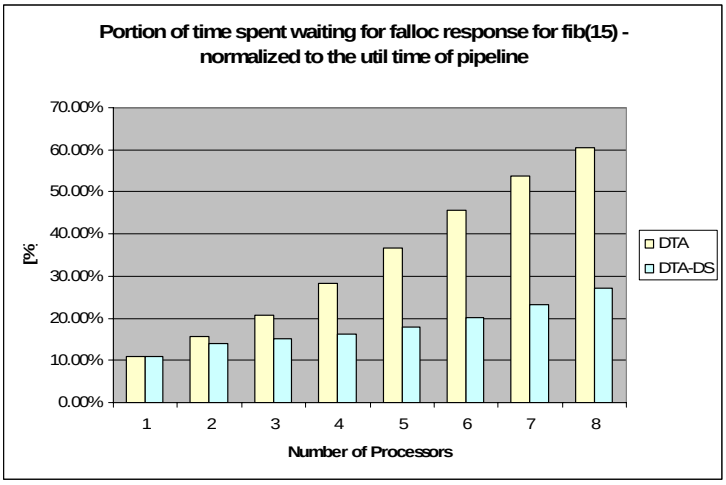

Figure 1. Portion of execution time spent in waiting for FALLOC response

In a smarter implementation we considered the following idea. Two tables are used - *map table* and *store buffer* (Figure 2). When the fork of the new thread occurs, processor continues without stopping. Instead of a frame pointer, it receives a virtual frame pointer, which is local for each processor. At the same time, request for new frame is sent to the scheduler and a new entry in the map table is added. Processor is free to continue the execution and waiting for the frame response is done in parallel. When the response from scheduler arrives the map table entry with corresponding virtual frame pointer is updated.
The only instructions that actually use the frame pointer are store instructions which store data for the new thread in its frame. If a store instruction arrives before the response from the scheduler, we must add a new entry in the store buffer. Later on, when the the frame

arrives, the store buffer will be checked for pending stores and they will be processed in that moment.

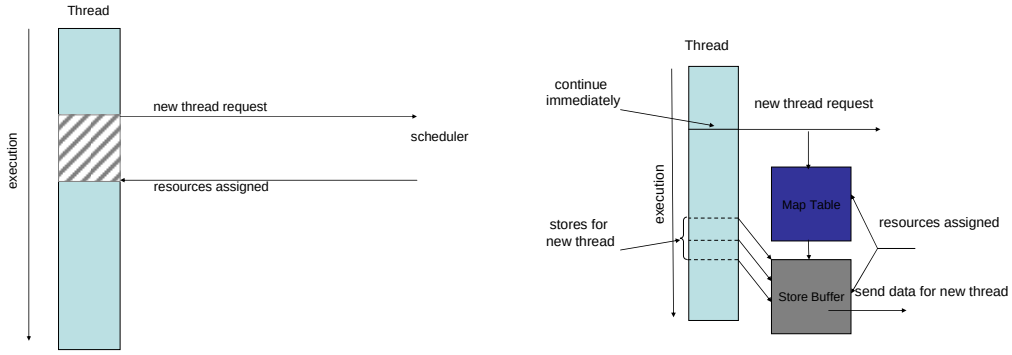

Figure 2. Blocking (on the left) and non blocking resource assignment (on the right)

Even in the case when responses from scheduler arrive later and we don't achieve to speed-up the starting time of new threads, execution time of the current thread is shorter and processor becomes free earlier.

3 Decoupling memory accesses

Since READ instructions (that read data from the main memory) can cause cache misses, this may cause pipeline stalls and can violate decoupling of memory accesses.
Figure 3 on the left shows current situations, where READ instructions are mixed with the rest of the code. On the right, a new pre-load phase has been added in order to solve the problem of memory accesses. Our approach is to identify memory accesses that can cause blocking at compile time, and to give information about them to the hardware. This identification can be done either automatically by the compiler, or programmer could insert it manually.
While the instructions that access memory are known at compile time, actual location for the access in most cases is known only at run time. Programs need to be modified in order to calculate these addresses before the thread that is performing memory accesses is forked. We can use the knowledge on complex data structures (like arrays and linked lists) in order to extract this information on memory accesses, to calculate the addresses and write them to the appropriate places in the frame. Before thread starts executing, a Load Unit will read the information on memory accesses from the frame, and it will program a RDMA [ref] mechanism in order to initiate loading of data. Only when all data (both data from the frame and data from the main memory) are available locally, thread can start its execution. Once data is fetched, it must remain in cache during the time when it is needed by the thread.

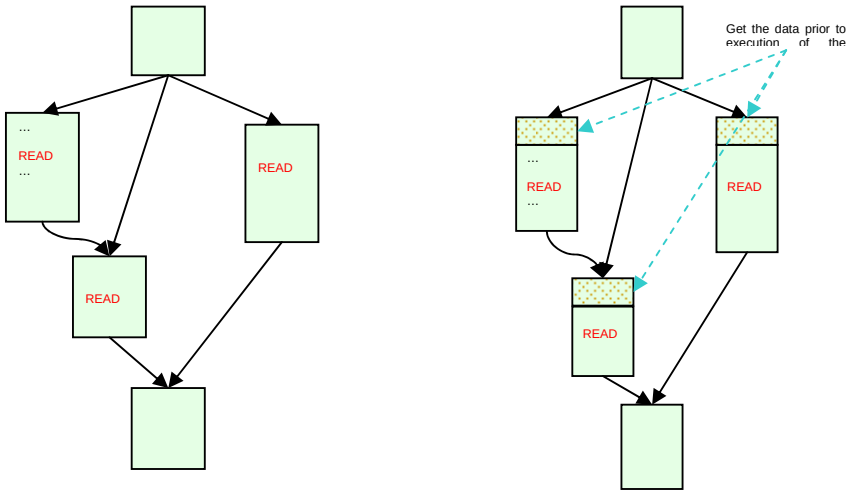

Figure 3. Mechanism for pre-loading data. On the left, current generic situation is shown (arrows represent the data being communicated). On the right, preload phase has been added to each thread that will include all memory accesses within the thread.

Since capacity of the cache is limited, it may occur that two (or more) fetched blocks are conflicting, and one (or more) of them will be evicted from the cache. Also, if invalidation from other processor occurs after loading data, depending on the coherence protocol, data may be invalidated. Solution for this problem has been investigated in the area of the predictable real-time systems with cache memories. Some proposals in the literature [4] try to address this unpredictability, namely "cache partitioning" and "cache locking". Also, lot research has been performed in area of reducing miss rate for direct-mapped and set-associative caches. Since these misses are coming form conflicts, we can use some of the techniques in order to reduce the number of conflicts and try to keep the data in the cache. The approaches for reducing the number of conflict misses, are based on using different placement functions in case of a conflict, like hash-rehash caches [5] and xor-based placement [6]. One of the objectives of our research is to investigate the usage of these techniques and to try and improve them so that they can be used to keep the data in the cache as long as it's needed and to eliminate as much potential conflicts as possible.

4 References

[1] 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.

[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] R. Giorgi, Z. Popovic, "Core Design and Scalability of Tiled SDF Architecture", *HiPEAC ACACES-2006*, ISBN:90-382-0981-9, L'Aquila, Italy2, July 2006, pp. 145-148.

[4] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. San Diego, CA, USA: ACM Press, 2003.

[5] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Trans. Comput. Syst.*, vol. 6, pp. 393-431, 1988.

[6] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *Proceedings of the 11th international conference on Supercomputing*. Vienna, Austria: ACM Press, 1997.