MT-SDF: Scheduled Dataflow Architecture with mini-threads

Domenico Pace
University of Pisa
Pisa, Italy
col.pace@hotmail.it

Krishna Kavi
University of North Texas
Denton, Texas, USA
kavi@cse.unt.edu

Charles Shelor
University of North Texas
Denton, Texas
cfshelor@sbcglobal.net

Abstract—In this paper we show a new execution paradigm based on Decoupled Software Pipelining in the context of Scheduled Dataflow (SDF) architecture. We call the new architecture MT-SDF. We introduce mini-threads to execute loops as a software pipeline.

We permit the mini-threads to share registers. We present a qualitative and quantitative comparison of the mini-threads with the original SDF architecture, and out-of-order superscalar architecture. We use several benchmark

Key words: Dataflow Architecture, Decoupled Software Pipelines, Multithreaded Architecture, and Shared Registers.

I. INTRODUCTION

Achieving high performance is possible when multiple activities can be executed concurrently. The concurrency must not incur large overheads to be effective. A second issue that must be addressed is the synchronization and/or coordination of concurrent activities. These actions often lead to sequentialization of parallel activities, thus defeating the potential gains of concurrent execution. Thus effective use of synchronization and coordination are essential to achieving high performance. One way to achieve this goal is through speculative execution whereby it is speculated that concurrent activities do not need synchronization or predict the nature of the synchronization. Successful speculation will reduce sequential portions but mis-speculation leads to overheads for undoing the speculative execution.

Implementation of these ideas in traditional control-flow (or speculative superscalar) architectures requires extensive software and hardware analyses to expose inherent concurrencies in applications, and complex recovery mechanisms when speculation fails.

More recently, a software technique known as Decoupled Software Pipelining (DSWP) [8, 9, 10] tries to eliminate or reduce dependencies among iterations of loops by spreading the dependent operations across multiple iteration of the loop in a pipelined fashion. However, implementation of DWSP on multicore processors requires efficient and fine-grained communication among cores.

GPUs and GP-GPUs are receiving considerable interest from high performance community. GPU processors include a large number of small threads, which can be used to execute applications with large-scale data parallelism. However these processors are difficult to program and the performance is limited by the transfer of data between the primary processing cores and GPUs.

We believe that the data flow computational model presents a better choice to processor architecture, both to implement scientific applications and applications with limited data parallelism [2]. In our previous research, we developed the Scheduled Dataflow [4, 5, 6, 7] that can be viewed as hybrid dataflow/control flow architecture. SDF threads use data flow execution model, while instructions within a thread are executed in order so that conventional pipelines and memory hierarchies can be used. In this paper we describe an extension to SDF where SDF threads contain mini-threads. The mini-threads contain both private and shared registers; shared registers can be used to communication among mini-threads. The MT-SDF mini threads can be used to execute the pipeline stages created using the Decoupled Software Pipelining (DSWP) approach.

The rest of the paper is organized as follows. Section II describes DSWP; Section III describes the original SDF architecture; Section IV shows how SDF is extended with mini-threads and Section V includes our experimental results.

II. DECOUPLED SOFTWARE PIPELINING

Software pipelining has been used to extract higher levels of parallelism, primarily in VLIW architectures. DSWP uses a similar mechanism to effectively tolerate variable latency stalls imposed by memory loads. DWSP is used to parallelize recursive data structure (RDS) loops to execute as two concurrent Threads: a critical part (CP) thread comprising the traversal slice and an off-critical part (off-CP) thread comprising the computation slice. For example, consider the following loop:

```
while (prt = prt → next ) {
ptr → val = ptr →v a l + 1 ;
```

978-1-4799-5247-2/14 $31.00 © 2014 IEEE
DOI 10.1109/DFM.2013.18

}

The traversal slice consist of the critical part code, prt = prt->next , and the computation slice is ptr->val=ptr->val+1. A DSWP parallelization of this loop consists of:

```
while(prt = prt →next{        while(prt = consume()){
  produce (ptr)                 ptr →val =ptr →val+1;
}                             }
```

TRASVERSAL LOOP COMPUTATION LOOP

The produce () function enqueues the pointer onto a queue and the consume() function dequeues the pointer. If the queue is full, the produce function will block waiting for a slot in the queue. The consume function will block waiting for data, if the queue is empty. In this way, the traversal and computation threads behave as a traditional decoupled produce-consumer pair. To reduce overhead, these threads communicate using a Synchronization Array (SA), a dedicated hardware structure for pipelined inter-thread communication. The abstraction of the SA is sets of blocking queues accessed via produce and consume instructions. The produce instruction takes an immediate dependence number and a register as operand. The value in the register is enqueued in the virtual queue identified by the dependence number. The consume instruction dequeues data in a similar fashion.

To apply similar techniques for DOACROSS loops with loop carried dependencies [3], this technique is extended, leading to PS-DWSP [9,10]. To better understand how PS-DSWP (and in general DSWP) works, consider the example shown in Figure 1. Figure 1(b) illustrates the Program Dependence Graph for the C code in Figure 1(a). In order to partition the instruction of the loop, DSWP first groups the instructions into Strongly Connected Components and then DSWP creates the Directed Acyclic Graph (DAG). DSWP can extract a maximum of 7 threads in this example. In practice, the performance of this loop is limited by the execution time of the SCC formed by statements F and G (assuming the loop is repeated several iterations). A key observation is that FG cannot be partitioned by DSWP but it can be replicated, so that multiple threads concurrently execute this SCC for different iterations of the outer loop (different element of the "p" list). There are dependencies carried by the outer loop in the SCCs AJ, CD, I. The first two are difficult to eliminate, the third SCC can be subjected to reduction, allowing it to be replicated. PS-DSWP can partition the DAG into two stages: a first sequential stage containing A J, B, and CD, and a second, parallel stage containing E, FG, H, I. This parallel stage can be replicated to concurrently execute in as many threads as desired, with the performance limited only by the number of iterations of the outer loop and the slowest stage in the pipeline.

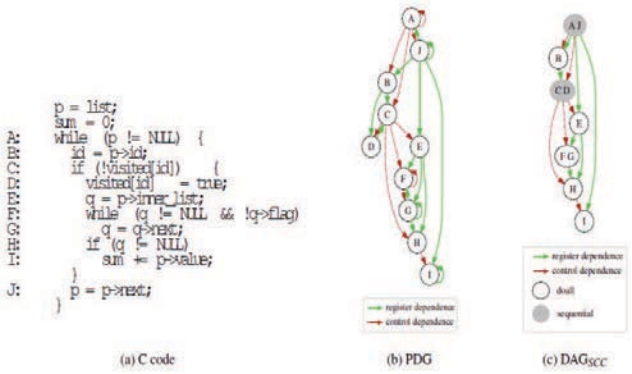

Figure 1: PS-DWSP Example

Figure 2 sketches the code that PSDSWP generates for the previous example. While not shown in this figure, the actual transformation generates code to communicate the control and data dependencies appropriately, and to add up the sum reduction after loop exit.

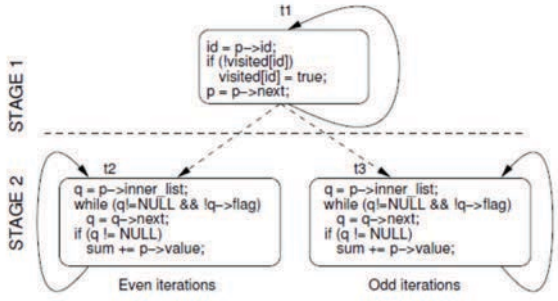

Figure 2. PS-DSWP applied to code in Figure 1

III. OVERVIEW OF SDF ARCHITECTURE

Scheduled Dataflow (SDF) [4,5,6,7] uses non-blocking threads where threads are enabled only when they receive all necessary inputs (*data driven*); and the architecture decouples all memory access from the execution pipeline. The architecture uses two different processing pipelines: Execution Pipeline (EP) for computations and Synchronization Pipeline (SP) for accessing memory. SP prepares an enabled thread by preloading all the data for the thread in its private register set; SP also stores results of completed threads into memory, thus enabling other threads. This decoupling leads to 3 phases of execution: *preload, execute, post-store.* Each thread's context is fully described by its continuation <*FP, IP, RS, SC*>. FP is the frame pointer representing storage allocated for the thread where it receives its inputs; IP is its instruction pointer, RS is the identification of the private register set assigned to the thread, and SC is the

synchronization count indicating the number of inputs needed to enable the thread. A scheduling unit (SU) manages continuations and schedules them either on SP or EP, depending on the state of the continuation. Figure 3 shows a simple SDF program.

```
CODE main
     PUTR1 main.1
     FORKEP R1 ;switch to execution pipeline
main.1:
     PUTR1 thread  ;instruction pointer to the code
     PUT 1 , R2  ;synchronization count
     FALLOC R1, R2, R11  ;allocating the frame memory for the thread
     PUTR1 main.2
     FORKSP R1  ;switch to synchronization pipeline
main.2:
     PUT 5, R5
     STORE R5, R11|1  ;now the thread is enabled and starts its execution
     FFREE  ;deallocate the frame memory and the register set
     STOP

CODE thread
     LOAD RFP|1, R5  ;the threads takes the data from its frame memory
       :
       : ;computation
       :
     FFREE  ;deallocate the frame memory and the register set
     STOP
```

Figure 3. An SDF Program Example

In main.1, a new thread is created using FALLOC, which allocates a new frame for the thread and stores IP and SC in the frame; this instruction is executed by EP. Data for the new thread is provided using STORE instructions, which are executed by SP. When the thread is ready to execute, it starts at CODE, by first moving data from frame memory to registers using LOAD instructions, executed by SP. A thread moves between SP and EP using FORKSP or FORKEP instructions. The frame memory and register sets are returned when the thread completes post-storing results, using an FFREE instruction.

IV. DWSP APPLIED TO SDF

To optimize the support for DSWP concepts we implemented a new level of threads within SDF: we call them mini-threads. We refer to the new architecture as MT-SDF. The mini-threads are completely contained within SDF threads. Unlike SDF threads, no frame memories are allocated to mini-threads; instead a register set is allocated when a mini-thread is created (using RSALLOC). This way, the mini-thread can receive its inputs directly in its registers, eliminating the preload phase of SDF threads. Min-threads do not use dataflow like enabling. A mini-thread becomes ready to execute under the control of the parent SDF thread, and we use SPAWNSP instruction. Figure 4 gives an example of MT-SDF code.

The mini-threads are placed in the mini-threads queue in order to distinguish them from SDF (macro) threads that may access Frame memories.

```
          :
          :
     PUT planck, R4
     PUTR1 main.create
     FORKEP R1
main.create:
     RSALLOC R11
     RSSTORE R5, R11|5
     SPAWNSP R4, R11

     ADD R3, R5, R5
     LT R5, R6, R8
     PUTR1 main.create
     FORKEP R8, R1
     PUTR1 main.end
     FORKSP R1
main.end:
     FFREE
     STOP

CODE planck
          :
          :
```

Figure 4. MT-SDF Program Example

Speculative Execution.

In addition, speculation can be used with mini-threads. The concept is an extension of speculative SDF threads reported previously in [4,7]. Speculative mini-threads are created by the SPECSPAWNSP instruction that creates a speculative continuation that consists of a 5-tuple: *<IP, RS, EPN, RIP, ABI>*. EPN is the epoch number: this value is used for the committing order of the mini-threads. RIP is the re-try instruction pointer used in case of mis-speculation. ABI is the address buffer ID that is used to store the addresses of speculatively read data; MESI like coherency will detect violations on speculatively read data items. Speculative threads commit strictly in the order of epoch numbers. When a thread is considered for commit, and no data access violations are found in the ABI buffer associated with the thread, the commit controller will schedule the thread for commit. If there is a violation, the commit controller sets the IP of that continuation to RIP and places the thread into the non-speculative queue for re-execution.

Register Organization.

In many cases, several mini-threads need the same inputs (e.g., base address for arrays, constant values). To facilitate this, we view the register sets used by mini-threads as partially shared (or global) and partially private registers. In the current implementation, each mini-thread has 32 private integer and 32 floating point registers (R0 to R31 and F0 to F31). All threads share 32-integer and 32-floating point registers (R32 to R63 and F32 to F63). This approach is similar to register windows used in SPARC architecture [1]. The parent thread can now store common data in shared or global register for use by all threads. Figure 5 below shows the structure of

registers in MT-SDF

Since reduction operations are very common in scientific applications, we have included reduction as a basic operation on shared registers. Thus mini-threads can use reduction when storing their results into shared registers.

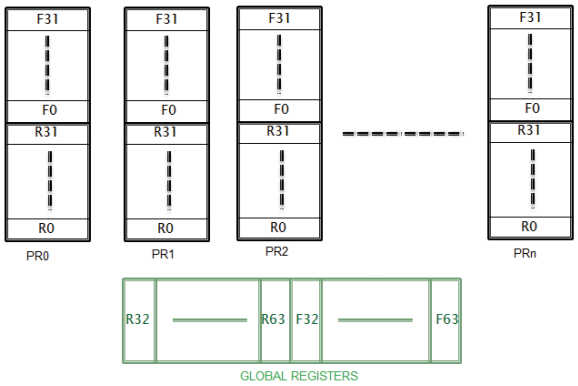

Figure 5. Shared Registers in MT-SDF

Impact of shared registers. To quantitatively evaluate the shared register set feature of MT-SDF, we used the dot product program. Figure 6 shows the execution time of 4 different implementations of the dot product program. In the figure, we show results using 10,000 element arrays, but use either 50 or 100 threads. In each case, we compare the number of cycles needed when using shared registers with reduction operation and using a single thread that performs reduction operation (which minimizes the complexity of hardware, but the reduction thread can only be activated after all threads have completed their computation - the original SDF model of execution).

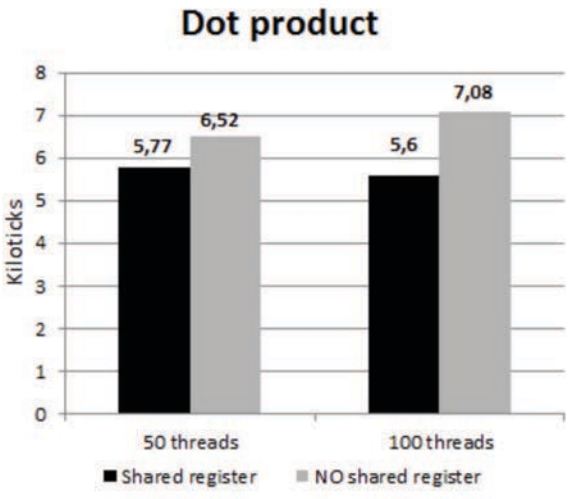

Figure 6. Evaluation of Shared Registers

V. EVALUATION OF MT-SDF

In this section we include comparison of MT-SDF with SDF using several benchmark kernels. These programs were hand-coded and executed using an extended version of the SDF simulator. We relied on hand-coded examples, since no optimizing compiler is available for MT-SDF at this time.

a) Matrix Multiplication benchmark: First we analyze the results of the matrix multiplication (MM) benchmark. In this benchmark we used two 20x20 matrices. Figure 7 shows the thread structure implemented in this program. Note that we are using DSWP for coding the application - we use the same structure for both SDF and MT-SDF implementation. In this version we used two concurrent mini-threads to optimize the execution of the inner loop of the matrix multiplication program. We used a shared register where each mini-thread can store its partial result. The MM benchmark exhibits both thread level parallelism and instruction level parallelism.

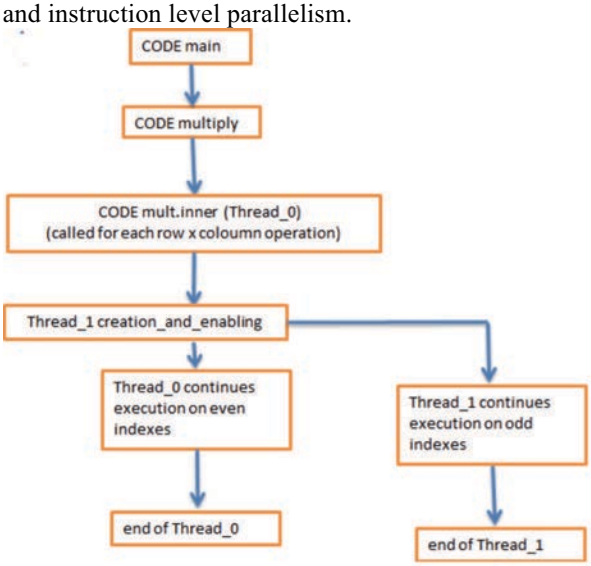

Figure 7. Coding Matrix Multiplication

The following figure shows the comparison of execution times of MT-SDF and SDF

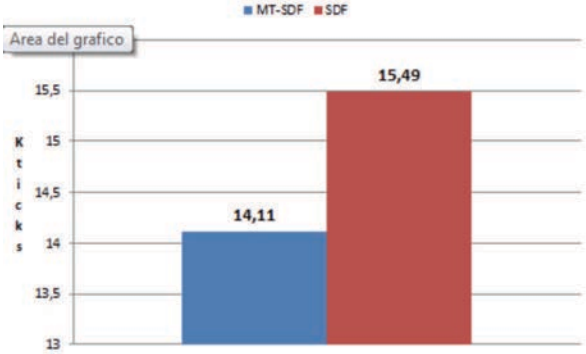

Figure 8. Matrix Multiplication Comparison

As can be seen from Figure 8, MT-SDF outperforms SDF. MT-SDF needs 9% fewer execution

cycles than SDF. Mini-threads lead to better utilization of both SP and EP pipelines. Figure 9 shows the utilization rates of the pipelines for the MT-SDF version compared to the SDF version of the program. These results indicate that MT-SDF utilizes the hardware resource more effectively.

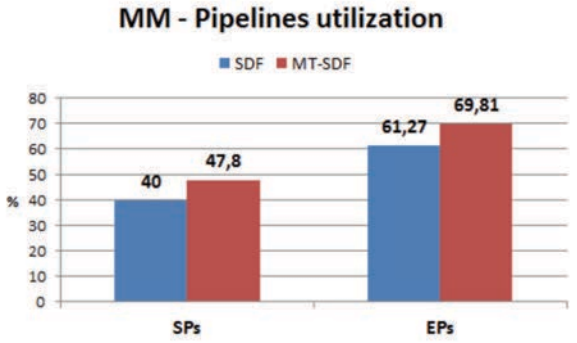

Figure 9. Utilization Rates for Matrix Multiplication

We also experimented with increasing thread level parallelism, using 4 and 5 threads for inner loop. The results are shown in Figure 10. The version with 5 MT is only 1% faster than the version with 2 MT. This is due to the overhead of the creation of a mini-thread is comparable to the computational load (4 MUL instructions and 4 ADD instructions) of the mini-thread itself.

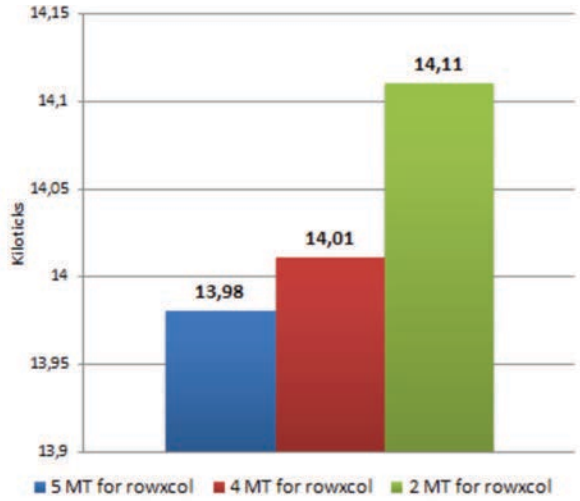

Figure 10. Increasing Thread Level Parallelism

b) Fast Fourier Transform: FFT exhibits higher degrees of thread level parallelism and higher computational load than matrix multiply. We used Cooley-Tukey[1] algorithm and used speculative mini-threads. Figure 11 shows the improvement in execution cycles. In this case the improvements due to the utilization of mini-threads is more evident. MT-SDF needs 45% fewer execution cycles to complete its execution when compared SDF version.

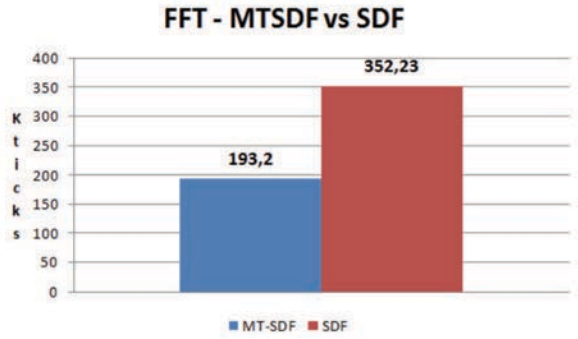

Figure 11. FFT Comparisons

Analyzing the execution pipeline utilization rates (not shown as a figure) and assuming 200 execution pipelines installed:

- SDF version uses only 162 EPs;
- MT-SDF version uses all the 200 EPs.

If more EPs are available, the program will use them also, leading to even better performance. This implies MT-SDF can generate and use higher levels of parallelism.

c). Monte Carlo method to estimate the PI and Planckian Distribution: These two benchmarks present a common behavior, but the Planckian Distribution has a lower degree of synchronization constraints. The following figure shows the results for Planckian distribution. (note: has Pi as well?)

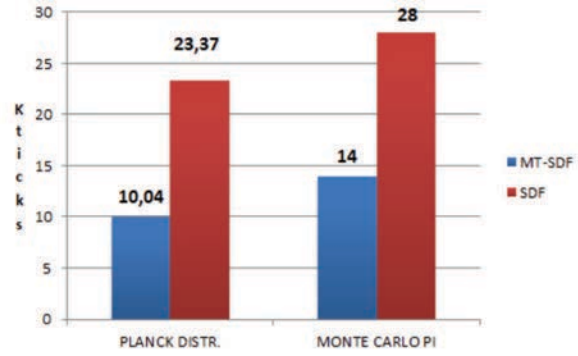

Figure 12. Planckian Comparisons

The Planckian Distribution benchmark contains a DOALL loop: in this case MT-SDF outperform SDF, saving 57% in execution cycles. Figure 13 shows the pipelines' utilization rates for the Planckian Distribution benchmark. As evident from Figure 13, the utilization rate in MT-SDF is consistently higher than in SDF, especially for the SPs.

1 http://it.wikipedia.org/wiki/Trasformata_di_Fourier_veloce#Algoritmo_di_Cooley-Tukey

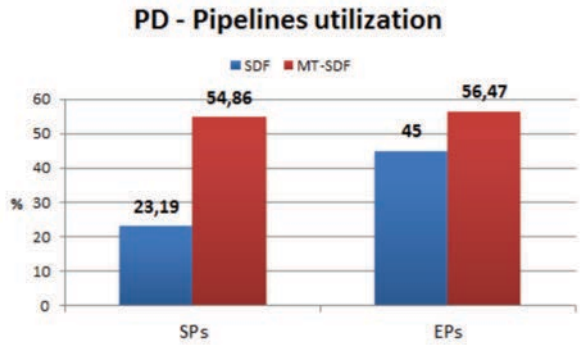

Figure 13. Utilization Rates for Planckian Distribution

c) Tri Diagonal Elimination: This benchmark, like the Planckian Distribution benchmark, is one of the kernels in the Livermore Loops suite. The C code is shown below

```
for ( i=1 ; i<n ; i++ ) {
    x[i] = z[i]*( y[i] - x[i-1] );
}
```

As can be seen, there is a loop-carried dependence. Iteration k needs the result from iteration k-1. To optimize the execution of this benchmark, we coded it using the DSWP technique. The following figure shows the execution cycles needed by MT-SDF compared to those needed by SDF.

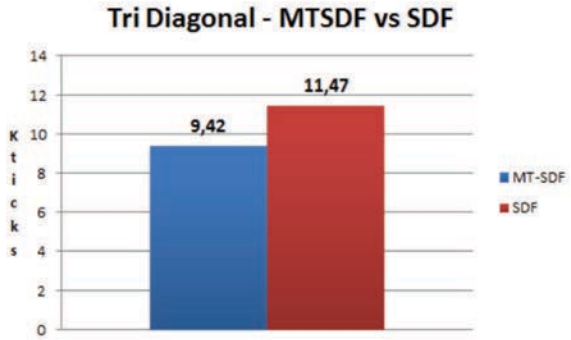

Figure 14. Tri-diagonal Comparisons

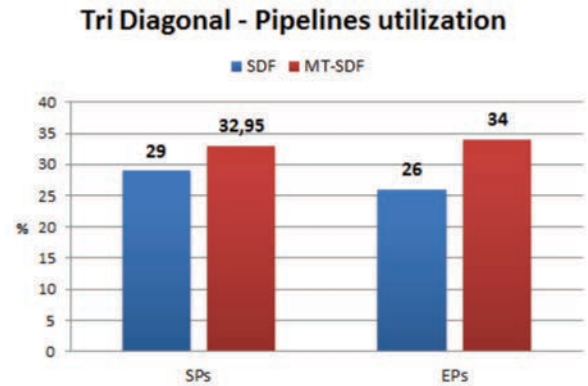

Figure 15. Utilization for Tri-diagonal Elimination

MT-SDF outperforms SDF, saving 18% execution cycles. As in other cases, mini-thread fast activation paradigm allows for better pipelines utilization (Figure 15).

MT-SDF vs Out of Order Superscalar.

This section shows the comparison of MT-SDF architecture with a superscalar out-of-order (OOO) processor simulated through the SimpleScalar simulator [2] . For MT-SDF we used the best configuration in terms of number of SPs and EPs, to achieve the best possible performance. For the simulated superscalar processor we used the most common configuration supported by the SimpleScalar simulator (see Table 1 below). We used the same memory access latencies for both architectures

Table 1: Parameters used for Simplescalar

Superscalar Parameter	Value
Number of integer ALU	8
Number of integer multiplier/dividers	8
Number of memory system ports	8
Number of floating point ALU	8
Number of floating point multipliers/dividers	8
Instruction fetch queue size	32
Instruction Decode width (insts/cycle)	16
Instruction issue (insts/cycles)	16
Register Update unit size	128
Load/Store queue size	64
Branch Prediction	Bimodal with 2048 entries

Figure 16 shows the chart that summarizes the results. The benchmarks considered have different levels of parallelism.

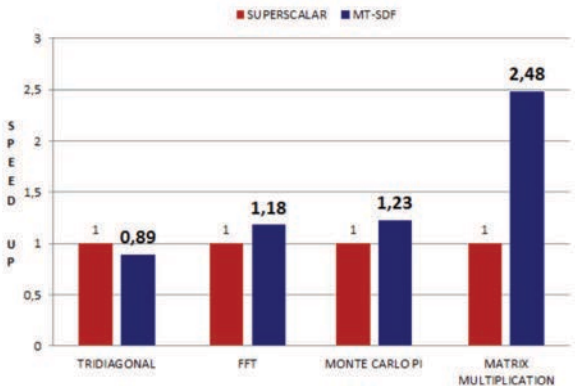

Figure 16: MT-SDF vs Superscalar

For the Tri-Diagonal Elimination benchmark the superscalar processor out-performs MT-SDF because it can exploit the inherent instruction level parallelism. On the other hand, to reach that performance the superscalar processor uses a lot of resources. MT-SDF uses only 2 SPs (equivalent to two memory port to access the memory) and 1 EP. For other benchmarks, MT-SDF out-performs the superscalar processor. In

[2] http://www.simplescalar.com

particular for the matrix multiplication benchmark, MT-SDF can exploit the inherent thread-level and data-level parallelism.

VI CONCLUSIONS

We extended the Scheduled Dataflow architecture with new features: this new architecture is called MT-SDF. The main characteristics of MT-SDF are another level of threads (mini-threads), shared registers and reduction operations with shared registers. These new features lead to substantial performance improvements for both DOALL and DOACROSS loops, when compared to the original SDF and out of order superscalar architecture. Using shared registers to store data that are common to several threads we can achieve at least 10% speed-up over SDF. The reduction capability with shared registers permits a better exploitation of thread-level parallelism when reduction operation is needed.

Mini-threads are the most important extension to SDF. Mini-threads are introduced to support Decoupled Software Pipelining (DWSP). Several benchmarks have shown that is possible to achieve between 9% and 57% speedup over SDF. Compared to a superscalar out-of-order processor, MT-SDF performs better when there is a high degree of thread level parallelism, but only slightly worse for applications with low degree of thread level parallelism, particularly if the threads need to utilize mutual exclusion on shared resources.

VII REFERENCES

[1]. Tom Germond, David L.Weaver. SPARC Architecture Manual version 9, 1994.

[2]. A.R. Hurson and K.M. Kavi. Dataflow Revival – a renewed interest in dataflow architecture", Wiley Encyclopedia of Computer Science, pp 890-901, Volume 2, ISBBN 978-0-471-38393-2, Jan 2009.

[3]. A.R. Hurson, J.T. Lim, K.M. Kavi and B. Lee "Parallelization of DOALL and DOACROSS loops - a survey", Advances in Computers, Vol. 45, pp 54-105, (Edited by M. Zerkowitz), Academic Press 1997.

[4]. Krishna Kavi, Wentong Li and Ali Hurson. "A non-blocking multithreaded architecture with support for speculative threads", Proceedings of the 8th International Conference on Algorithms, Architectures and Applications of Parallel Processing (ICA3PP-208), Cyprus, June 9-11, 2008, Proceedings published by Springer-Verlag, LNCS 5022, pp 173-184

[5]. Krishna Kavi, Joseph Arul, and Roberto Giorgi. Execution and cache performance of the scheduled dataflow architecture. Journal of Universal Computer Science, Special Issue on Multithreaded Processors and Chip Multiprocessors , 6:948– 967, 2000.

[6]. Krishna M. Kavi, Roberto Giorgi, and Joseph Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. IEEE Trans. Comput. , 50(8):834–846, August 2001.

[7]. Wentong Li, Krishna Kavi, Afrin Naz, and Phil Sweany. Speculative thread execution in a multithreaded dataflow architecture. In Proceedings of the 19th ISCA Parallel and Distributed Computing Systems , 2006.

[8]. Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel stage decoupled software pipelining. In Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization , CGO '08, pages 114–123, New York, NY, USA, 2008.

[9]. Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In Proceedings of the 13th InternationalConference on Parallel Architectures and Compilation Techniques , PACT '04, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[10]. Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. Performance scalability of decoupled software pipelining. ACM Trans. Archit. Code Optim. , 5(2):8:1–8:25, September 2008.

Acknowledgements. This research is supported in part by the NSF Net-Centric Industry/University Cooperative Research Center, its industrial memberships and by NSF Grant #1237417. Domineco Pace spent Spring 2013 at UNT, working on his MS thesis.