Performance Evaluation of a Non-Blocking Multithreaded Architecture for Embedded, Real-Time and DSP Applications

Krishna M. Kavi and Joseph Arul
The University of Alabama in Huntsville
Roberto Giorgi
The University of Siena, Italy

Abstract
This paper presents the evaluation of a non-blocking, decoupled memory/execution, multithreaded architecture known as the Scheduled Dataflow (SDF). The major recent trend in digital signal processor (DSP) architecture is to use complex organizations to exploit instruction level parallelism (ILP). The two most common approaches for exploiting the ILP are Superscalars and Very Long Instruction Word (VLIW) architectures. On the other hand, our research explores a simple, yet powerful execution paradigm that is based on non-blocking threads, and decoupling of memory accesses from execution pipeline. This paper compares the execution cycles required for programs on SDF with the execution cycles required by programs on Superscalar and VLIW architectures.
Key Words: Multithreaded architectures, Superscalars, VLIW, Decoupled Architectures.

1. Introduction

The major recent trend in digital signal processor (DSP) architecture is to use complex organizations to exploit instruction level parallelism (ILP). The two most common approaches for exploiting the ILP are Superscalars and Very Long Instruction Word (VLIW) architectures. Superscalars rely on hardware techniques to find independent instructions and issue them to independent functional units. VLIW, on the other hand relies on a compiler to schedule independent instructions statically. A different approach for improving processing performance, particularly to bridge the performance gap between processors and memory, is multithreading. There is a consensus that multithreading achieves higher instruction issue rates on processors that contain multiple functional units [8,9,16], We believe that the use of non-blocking threads is appropriate for improving the performance of Superscalar and VLIW architectures.

Our architecture differs from other multithreaded models in two ways: i) our programming paradigm is based on non-blocking functional threads, which eliminates the need for runtime instruction scheduling, and ii) complete decoupling of all memory accesses from execution pipeline. The underlying functional non-blocking model permits for clean separation of memory accesses from execution (which is very difficult to coordinate in other programming models). Since our achitecture performs no runtime instruction scheduling, our architecture requires less complex hardware and potentialy achieve energy savings -- it was stated that a significant power is expended by instruction issue logic of modern Superscalar architectures, and the power consumption increases quadratically with the size of the instruction issue width [12,18]. In this paper we present a comparison our architecture with conventional Superscalar architecture containing multiple functional units and aggressive Out-of-Order instruction issue logic using SimpleScalar Tool Set [3]. We have also compared the performance of our architecture with VLIW architectures using Texas Instruments TMS320C6000 VLIW processor simulator tool-set[15], and the Trimaran infrastructure[1]. Since we target our processor architecture for real-time, embedded and DSP applications, we present our evaluations using benchmarks that reflect these applications (viz., Matrix Multiplication, FFT, a picture zooming applications).

In Section 2 we present research that is most closely related to ours. In Section 3 we present our SDF architecture in detail. Section 4 discusses the methodology that we used in our evaluation and shows our numerical results for real programs.

2. Related Research and Background

Decoupling memory accesses from the execution pipeline to overcome an ever-increasing processor-memory communication cost was first introduced in [13]. Decoupled ideas were recently used in a multithreaded architecture known as Rhamma [5]. Rhamma uses blocking threads requiring many more thread context switches than our non-blocking threads. Moreover, Rhamma does not group all Load instructions together into "pre-load" and all Store instructions together into "post-store" as done by SDF. Because of these differences, SDF outperforms Rhamma [6].

Dataflow architectures are the most recognized implementations of functional computational model [10,11]. Our architecture extends ETS [10,11] and Cilk models [4].

---
[1] http://www.trimaran.org

3. The Scheduled Dataflow Processor (SDF)

The basic processing element in our architecture consists of two units: Synchronization Pipeline (SP) and Execution Pipeline (EP). SP is responsible for scheduling enabled threads on EP, pre-loading thread context (i.e., registers) with data from the thread's (Frame) memory, and post-storing results from a completed thread's registers into the (Frame) memories of destination threads. More detailed description of our architecture can be found [2,6,7].

3.1 Execution Pipeline

Figure 1 shows the block diagram of the Execution Pipeline (EP). EP executes computations of a thread using only registers. Instruction fetch unit behaves like a traditional fetch unit, relying on a program counter to fetch the next instruction. EP executes instructions sequentially with no dynamic instruction issue, nor out-of-order instruction execution. As with any multithreaded system, SDF uses multiple register sets to support active threads; and the achievable thread-level parallelism depends on the number of hardware contexts.

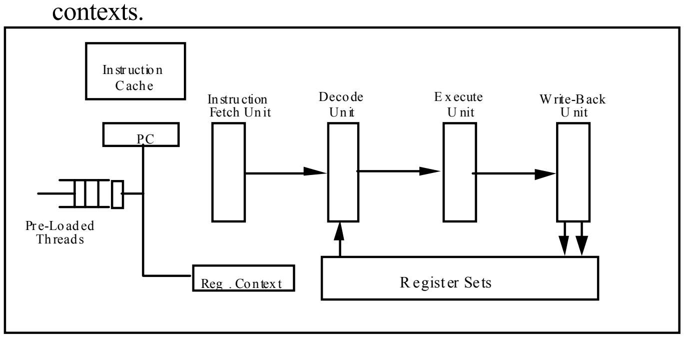

Figure 1. Execution Pipeline (EP).

3.2 Synchronization Pipeline

Figure 2 shows the organization of the memory pipeline of the Synchronization Processor (SP). Here we deal with pre-load and post-store instructions, as well as I-Fetch[2] and I-Store for accessing array and structured data items. In addition to accessing memory, Synchronization Pipeline (SP) holds thread continuations awaiting inputs and allocates register contexts for enabled threads. In our architecture a thread is created using a FALLOC instruction which takes two arguments: an instruction pointer (IP), and a synchronization count (Synch Count) indicating the number of enabling inputs needed. FALLOC returns a frame pointer in a register after allocating a frame and storing IP and Synch Count in the first two locations of the allocated frame. The frame pointer returned by FALLOC will be utilized to store data in the spawned thread's frame memory.

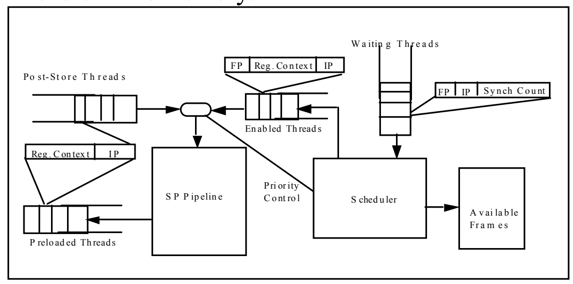

Figure 2. Overall Organization of the SP

An enabled thread (when the Synch Count becomes zero) is scheduled by allocating a register context to it. Threads are created using FALLOC, and thread is moved betweenEP and SP using FORKEP and FORKSP.

3.3. Instruction Set Architecture of SDF.

We first show how the instructions executed by EP would look-like using a simple example (Figure 3).

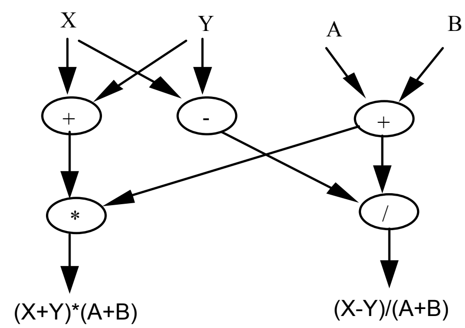

Figure 3. A simple dataflow graph.

Each node of the graph will be translated into a SDF instruction. The two source operands destined for a dyadic SDF instruction are stored in a pair of registers associated with that instruction.

ADD	RR2, R11, R13	/ A+B, Result in R11 and R13
ADD	RR4, R10	/ compute X+Y, Result in R10
SUB	RR4, R12	/ compute X – Y, Result in R12
MULT	RR10, R14	/ (X+Y)*(A+B), Result in R14
DIV	RR12, R15	/(X-Y)/(A+B), Result in R15

The use of separate pairs of registers with each instruction is akin to the reservation stations (using Tomosulo's approach) or dynamic renaming of registers as utilized by most modern Superscalar architectures . The register assignment for instructions is done statically by the compiler and requires no hardware support to. Assuming that the inputs A, B, X and Y to the graph of Figure 3 are available in R2, R3, R4 and R5, respectively (this is achieved during pre-load), the five instructions shown

[2] We use I-structure memory for arrays and structures. Information on I-structures can be found in most dataflow literature. Index computation is performed by EP while the actual access to I-structures is achieved by SP. Simple index calculations can be done by SP directly.

above will be executed *sequentially* and perform the necessary computations. Note that the source operands are specified as a pair of registers using "RR", for example, ADD RR2, R11, R13 adds R2 and R3, and stores the result in R11 and R13. Our instructions still retain the functional nature – there are no write-after-read and write-after-write dependencies with our instructions.

In our architecture, SP pre-loads data in a thread's register set before scheduling the thread on EP (and EP never accesses memory). Assume that the code block of Figure 3 (viewed as a thread) receives the four inputs (A, B, X, Y) from other threads; these inputs will be saved in the frame until the thread is enabled for execution. When enabled, a register context is allocated to the thread and the input data for the thread from its frame memory is "pre-loaded" into its registers. Assuming that the inputs for the thread are stored in its frame (RFP) at offsets 2, 3, 4 and 5, the first four LOAD instructions shown below pre-load the thread's data into registers R2, R3, R4, R5 of the register set allocated for the thread.

```
LOAD  RFP| 2,   R2    / load A into R2
LOAD  RFP| 3,   R3    / load B into R3
LOAD  RFP| 4,   R4    / load X into R4
LOAD  RFP| 5,   R5    / load Y into R5
LOAD RFP| 6,    R6    /FP for returning 1st result
LOAD RFP| 7,    R7    /frame offset for 1st result
LOAD RFP| 8,    R8    /FP for returning 2nd result
LOAD RFP| 9,    R9    / frame offset for 2nd result
```

After the pre-load, the thread is scheduled for execution on EP. The EP then uses only its registers during the execution of the thread body (code shown previously). Let us assume that the results generated by MULT and DIV in our code example (i.e., R14 and R15) are needed by two other threads. The frame pointers and frame-offsets for the destination threads are made available to the current thread in registers R6, R7, R8 and R9 as shown in the pre-load code above (the last 4 LOAD instructions). Note that the frame pointers are returned by FALLOC instructions as described previously, and these pointers can be passed to other threads.

```
STORE   R14,  R6|R7        / store first result
STORE   R15,  R8|R9        / store second result
```

These STORE instructions transfer (or post-store) the results of the current thread (i.e., from MULT in R14 and DIV in R15) to frames pointed to by R6 and R8 at frame-offsets contained in R7 and R9. SP executes STORE instructions after a thread completes its execution at EP.

4. Evaluation of Scheduled Dataflow (SDF)

In this paper, we characterize our architecture based on execution cycles for actual programs using our instruction level simulator. At present the simulator assumes a perfect cache (viz., all memory accesses take one cycle). However, we examined the expected cache behavior using traces from program examination [2]. Our results indicate that SDF produces cache miss behaviors similar to those for Superscalar systems. Previously we reported a comparison of our architecture with a single threaded RISC architecture using DLX simulator [7]. In this paper we will compare our SDF with Superscalar architectures with multiple functional units and Out-of-Order instruction issue logic as facilitated by the SimpleScalar Tool Set [3]). We will also present comparisons of SDF with VLIW architectures as facilitated by Texas Instruments TMS320C6000 VLIW processor simulator tool-set [15], and the Trimaran[3] infrastructure. Since we target our architecture for embedded and DSP applications, we chose Matrix Multiply, FFT and a picture zooming program [14]. We chose these applications since they exhibit different characteristics. Matrix multiply can be written to exploit both thread level and instruction level parallelism; FFT exhibits higher degrees of thread level parallelism with increasing data sizes; and Zoom [14] consists of 3 nested loops and substantial amount of instruction level parallelism in the middle loop (but only small degrees of thread level parallelism).

4.1. SDF vs. Superscalar

In the first experiment, we compared the execution performance of SDF with a Superscalar processor by varying the number of functional units (we varied the number of Integer and Floating point units in Superscalar, and varied the number of SPs and EPs in SDF). For comparisons purposes we set the number of functional units in Superscalar (#Integer ALUs + #Floating Point ALUs)[4] equal the number of SPs and EPs (#SPs + #EPs). Table 1 shows the parameters we used for Superscalar. We have used the compiler provided with SimpleScalar toolset to generate highly optimized code for the benchmarks.

It is our contention that conventional Superscalar systems do not scale well with increasing number of functional units and the scalability is limited by the

[3] http://www.trimaran.org

[4] It is not our intention to state that integer units equate to SP's or floating-point units are the same as EPs. For our initial comparisons, we are hoping that this first order approximation will be fair in terms of functional units.

instruction fetch/decode window size and the RUU size. As stated previously, the power consumed by the instruction issue logic increases quadratically with the window width [12,18]. SDF relies on thread level parallelism, and the decoupling of memory accesses from execution. SDF performance can scale better with a proper balance of workload among SPs and EPs. For the Superscalar, we show execution cycles for both In-Order (shown as I-O in Tables 2-4) and Out-of-Order (shown as O-O in tables 2-4) instruction issue. In all systems, we set all instruction cycles to 1, and assume perfect cache.

Table 1: Superscalar Parameters For Tables 2-4

Superscalar Parameter	Value
Number of Functional Units	Varied
Instruction Issue Width	64
Instruction Decode Width	64
RUU	64
LSQ	64
Branch Prediction	Bimodal with 2048 entries

In Table 2 we show the data for Matrix Multiply. As can be noted, when we add more SPs and EPs (correspondingly more Integer and Floating Point functional units in Superscalar), SDF outperforms Superscalar architecture, even when compared to complex out-of-order scheduling used by Superscalars (shown in bold in Table 2). For both systems, we unrolled the innermost loop 5 times; for SDF, we spawned 10 threads to execute in parallel. SDF's performance overtakes the Superscalar architecture with 3SPs and 3EPs. This is because, SDF can exploit the functional units with available thread level parallelism and decoupled memory accesses. The effect of decoupling memory accesses can be observed from table -- adding more SPs improves the performance more significantly than when EPs are added. SDF performance can be further improved by using more than 10 active threads (or register contexts). The scalability of SDF can more easily be seen from Figure 5. The X-axis shows the number of functional units (#SP+#EP for SDF; #Integer ALUs + #FP ALUs for Superscalar). The figure shows the execution times for 150*150 matrix multiplication.

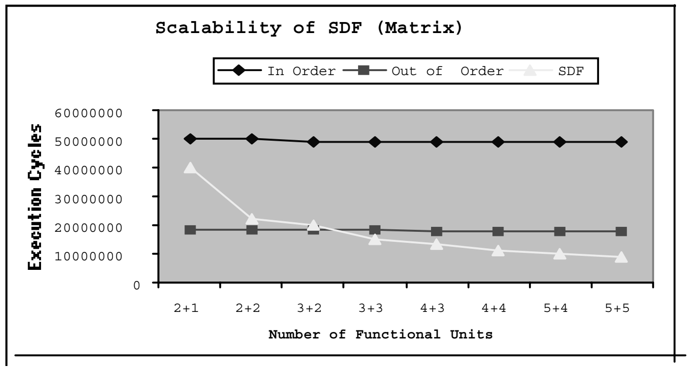

Figure 5. Scalability of SDF (Matrix Multiply

The next table (Table 3) shows the results for FFT. For small data sizes, SDF performs worse than the Out-of-Order Superscalar execution, due to a lack of significant thread-level parallelism. As the data size increases, SDF exploits available thread-level parallelism and outperforms Out-of-Order Superscalar for FFT (for data sizes are greater than 256)

Table 2. Comparing SDF with Superscalars (Matrix Multiply)

Data Size		Superscalar 2INT ALU 1FP ALU	SDF 2SP 1EP	Superscalar 2INT ALU 2FP ALU	SDF 2SP 2EP	Superscalar 3INT ALU 2FP ALU	SDF 3SP 2EP	Superscalar 3INT ALU 3FP ALU	SDF 3SP 3EP
50*50	I-O	1890104		1890104		1867200		1867200	
	O-O	712396	1504297	712396	860782	706877	756707	706877	**574242**
100*100	I-O	14824104		14824104		14633700		14633700	
	O-O	5532202	11843442	5532202	6660012	5511587	594160[illegible]	5511587	**4440772**
150*150	I-O	49763150		49763150		49110246		49110246	
	O-O	18514510	39762487	18514510	222277[illegible]	18468811	1992491	18468409	**14819482**

Data Size		Superscalar 4Int ALU 3FP ALU	SDF 4SP 3EP	Superscalar 4Int ALU 4FP ALU	SDF 4SP 4EP	Superscalar 5Int ALU 4FP ALU	SDF 5SP 4EP	Superscalar 5Int ALU 5FP ALU	SDF 5SP 5EP
50*50	I-O	1867200		1867200		1867200		1867200	
	O-O	680321	**507197**	680321	**430957**	680321	**381247**	680321	**345027**
100*100	I-O	14633700		14633700		14633700		14633700	
	O-O	5306381	**3970682**	5306381	**3330992**	5306380	**298270[illegible]**	5306380	**2665472**
150*150	I-O	49110246		49110246		49110246		49110246	
	O-O	17782453	**13308457**	17782453	**11115592**	17782453	**999060[illegible]**	17782453	**8894002**

Table 3. Comparing SDF with Superscalars (FFT)

Data Size		Superscalar 2INT Alu 1FP Alu	SDF 2SP 1EP	Superscalar 2INT Alu 2FP Alu	SDF 2SP 2EP	Superscalar 3INT Alu 2FP Alu	SDF 3SP 2EP	Superscalar 3INT Alu 3FP Alu	SDF 3SP 3EP
8	I-O	20418		20418		19933		19933	
	O-O	9377	10045	9377	9526	8645	8556	8202	8528
16	I-O	36038		36038		35394		35394	
	O-O	15737	24303	15737	22927	14550	20385	13997	20337
32	I-O	78794		78794		77695		77695	
	O-O	32902	57444	32902	54012	30515	47740	29621	47608
64	I-O	201547		201547		199069		199069	
	O-O	81519	133139	81519	124915	75952	110003	73937	109731
128	I-O	577851		577851		570906		5770906	
	O-O	228095	303518	228095	284382	214191	249774	208285	249070
256	I-O	1816758		1816758		1794386		1794386	
	O-O	703548	**682417**	703548	**638705**	664954	**559665**	644899	**558353**
512	I-O	6165028		6165028		6086525		6086525	
	O-O	2350656	**1516660**	2350656	**1418356**	2235095	**12409448**	2161717	**1238580**

Once again, SDF performance scales better with added functional units than that of a Superscalar. Thus for larger data sizes, SDF can more effectively utilize functional units than Superscalar systems that rely only on ILP from a single threaded programming model. SDF employs two levels of parallelism- thread level parallelism, and the overlapped execution of memory accesses with the execution of arithmetic instructions. The fact that the performance improves when more SPs are added indicates that the decoupling of memory accesses can benefit from more memory pipelines (contained in SP's). Thus, the data shows the benefits of both multithreading (as demonstrated by the ability to exploit greater thread-level parallelism with larger data sizes) and decoupled memory accesses (as shown by improved performance with added SPs).

Table 4 shows the data for Zoom. Once again, the performance of SDF scales better than Superscalar. With 3 SPs and 2 EPs, SDF outperforms even the Out-of-Order Superscalar system, shown in bold in Table 4.

Table 4. Comparing SDF with Superscalars (Zoom)

Data Size		SS 1INT 1FP	SDF 1SP 1EP	SS 2 INT 1 FP	SDF 2SP 1EP	SS 2 INT 2 FP	SDF 2SP 2EP	SS 3 INT 2 FP	SDF 3SP 2EP	SS 3 INT 3 FP	SDF 3SP 3EP
50	I-O	528100		499976		499976		499573		499573	
	O-O	416625	464765	221253	314032	221253	230072	170235	**163907**	170235	**153542**
100	I-O	2094969		1994236		1994236		1993829		1993829	
	O-O	1660478	1855370	877150	1254357	877150	915057	696002	**655907**	696002	**611707**
150	I-O	4989542		7486216		7486216		4785812		4785812	
	O-O	3994462	4171875	2108470	2821032	2108470	2061057	1693042	**1476057**	1693042	**1374402**
200	I-O	8387641		7986709		7986709		7986302		7986302	
	O-O	6613286	7414280	3503558	5014057	3503558	3661977	2779131	**2624357**	2779131	**2441917**

Data Size		SS 4 INT 3 FP	SDF 4SP 3EP	SS 4 INT 4 FP	SDF 4SP 4EP	SS 5 INT 4 FP	SDF 5SP 4EP	SS 5 INT 5 FP	SDF 5SP 5EP	SS 6 INT 5 FP	SDF 6SP 5EP
50	I-O	499573		499573		499573		499573		499573	
	O-O	165210	**115887**	165210	**115452**	160151	**92892**	160151	**92837**	160151	**77912**
100	I-O	1993827		1993827		1993827		1993827		1993827	
	O-O	656328	**460317**	656328	**459667**	646252	**368747**	646252	**368417**	646252	**308777**
150	I-O	4785811		4785811		4785811		4785811		4785811	
	O-O	1638111	**1033277**	1638111	**1032567**	1615054	**827232**	1615054	**826827**	1615054	**693567**
200	I-O	7986300		7986300		7986300		7986300		7986300	
	O-O	2624712	**1834797**	2624712	**1833337**	2587702	**1468757**	2587702	**1468057**	2587702	**1229557**

4.2. SDF vs. VLIW

The Texas Instrument's TMS320C6000 family of DSP processors uses very long instruction word (VLIW) architecture. The newest member of the TMS320C6000 family, the 'C647X, brings the highest level of performance for processing data by utilizing 8 functional units, two register files, divided into two data paths. Each data path consists of a Multiplier, an Adder, a Load/Store units and one unit for managing control-flow (branch and compare instructions). We used a simulator and accompanying tools (including optimizing compiler and profiling tool). We have set instruction execution and memory access cycles to match in SDF and TMS320C64X. For SDF we utilize 8 functional units (4SPs and 4EPs)[5]. We have started working with Trimaran[6] tools. In this paper we will compare SDF with Trimaran using default configurations and optimizations (using a total of 9 functional units, a maximum unrolling of 32 iterations, and several other complex optimizations).

Table 5 shows the data for Matrix Multiplication. TMS 'C6000 performs rather poorly because the optimized version relies on unrolling of only 5 iterations (unlike Trimaran, which uses 32 iterations). SDF achieves better performance than TMS 'C6000 because we rely on thread level parallelism -- the data in Table 5 uses 10 active threads. Trimaran outperforms SDF because of the Herculean optimization efforts made by the compiler. SDF's performance can be improved by performing some similar optimizations and/or increasing the number of active threads. Trimaran exploits greater ILP since it examines 32 loop iterations (and this can be noticed with larger data sizes where Trimaran can sustain higher issue rates).

Table 5. SDF vs VLIW (Matrix Multiplication)

Matrix Multi					
Data Size	SDF	Trimaran	TMS 'C6000 optimized	SDF/Trimaran	DF/TMS C'6000
50*50	430957	331910	1033698	1.29841523	0.416908033
100*100	3330992	2323760	16199926	1.43344924	0.205617729
150*150	11115592	4959204	86942144	2.24140648	0.127850447

The next table (Table 6) shows the results of comparing SDF with TMS 'C6000 and Trimaran for FFT benchmark. Similar to the data in Table 3, SDF outperforms Trimaran VLIW system for large data sizes (greater than 256). As shown previously in Table 3, SDF scales better with more functional units. Thus for larger data sizes, SDF can more effectively utilize functional units than either Superscalar or VLIW systems that rely only on ILP from a single threaded programming model.

Table 7 shows the comparisons of SDF with the two VLIW systems under investigation (TMS C'6000 and Trimaran) for Zoom. SDF consistently outperforms both systems . SDF performance gains improve slightly for larger data sizes.

5. Conclusions

Our goal is the search for a viable architecture that can efficiently support fine-grained threads and decouple memory accesses from execution pipeline. To this end, we presented a non-blocking multithreaded architecture, called SDF. In this paper we presented a performance comparison of SDF with Superscalar and VLIW architectures. The results are very encouraging. Our data shows that SDF scales better than conventional Superscalar systems when more functional units are added. The data presented shows the performance gains due to the decoupling of memory accesses - SDF shows more dramatic performance improvements when more SPs are added, compared to the improvements when more EPs are added.

While decoupled access/execute implementations are possible within the scope of conventional architectures, multithreading (particulalry non-blocking) model presents greater opportunities for exploiting the separation of memory accesses from execution pipeline. In our model, threads exchange data only through the frame memories of threads (array data is provided through I-structure memory). The use of frame memories for thread data permits for a clean decoupling of memory accesses into pre-loads and post-stores. This can lead to greater data localities.

6. References

[1] A. Agarwal, et. Al. "Sparcle: An evolutionary processor design for multiprocessors", *IEEE Micro,* pp 48-61, June 1993.

[2] J. Arul, K.M. Kavi and S. Hanief. "Cache Performance of Scheduled Dataflow Architecture", *Proc. of the 4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP2000),*pp 110-123

[3] D. Burger and T. M. Austin. "The SimpleScalar Tool Set Version 2.0", *Tech Rept. #1342*, Department of Computer Science, University of Wisconsin, Madison.

[4] R. D. Blumofe, *et. al.*, "Cilk: An efficient multithreaded runtime system", *Proc of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPoP)*, July 1995.

[5] W. Grunewald, T. Ungerer, "A Multithreaded Processor Design for Distributed Shared Memory System," *Proc. Advances in Parallel and Distributed Computing*, 1997.

[6] K.M. Kavi, H.-S.Kim, J. Arul and A.R. Hurson "A decoupled scheduled dataflow multithreaded

[5] We concede that this may not be fair, since the processing units in SDF (SP and EP) are homogeneous, while the functional units in VLIW are not.

[6]See http://www.trimaran.org

architecture", *Proceedings of I-SPAN-99, June 23-25, 1999*, pp 138-143.

[7] K.M. Kavi, R. Giorgi and J. Arul. "Comparing execution performance of Scheduled Dataflow Architecture with RISC processors", *Proc. of PDCS-0,* Aug. 8-10, 2000, pp 41-47

[8] V. Krishnan and J. Torrellas. "A chip-multiprocessor architecture with speculative multithreading", *IEEE Trans. on Computers,* Sept. 1999, pp.866-880.

[9] M. Lam and R.P. Wilson. "Limits of control flow on parallelism", *Proc. of the 19th Intl. Symposium on Computer Architecture*, pp 46-57, May 1992.

[10] G.M. Papadopoulos and K.R. Traub. (1991). "Multithreading: A Revisionist View of Dataflow Architectures," *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 342-351.

[11] G.M. Papadopoulos and D.E. Culler. "Monsoon: An explicit token-store architecture", *Proc. of 17th Intl. Symposium on Computer Architecture*, pp 82-91.

[12] S. Onder and R. Gupta. "Superscalar execution with direct data forwarding", *Proc of the International Conference on Parallel Architectures and Compiler Technologies*, Paris, Oct. 1998, pp 130-135.

[13] Smith, J.E. "Decoupled Access/Execute Computer Architectures", *Proc of the 9th Annual Symp on Computer Architecture*, pp 112-119.

[14] H. Terada, et. al. "DDMP's: Self-timed super-pipelined data-driven multimedia processor", *Proceedings of the IEEE*, Feb. 1999, pp 282-296

[15] "TMS320C6000 CPU and Instruction Set Reference Guide", January 2000.

[16] J. Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P. C. Yew. "The Superthreaded processor architecture", *IEEE Trans. on Computers*, Sept. 1999, pp. 881-902.

[17] D.M. Tullsen, *et al*., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp 392-403.

[18] K. Wilcox and S. Manne. "Alpha processors: A history of power issue and a look at the future", *Cool Chips Tutorial in conjunction with MICRO-32*, Haifa, Israel, Dec. 1999.

Acknowledgement: This Paper was supported by NSF grants: CCR-9796310, EIA-9805216 and EIA-9820147.

Table 6 Comparing SDF with VLIW (FFT)

Data Size	SDF	Trimaran	TMS 'C6000	SDF/Trimaran	SDF/TMS 'C6000
8	*8148*	4622	26717	1.762873215	0.304974361
16	*19323*	12391	73456	1.559438302	0.263055435
32	*45028*	31665	213933	1.422011685	0.210477112
64	*103491*	81375	619241	1.271778802	0.167125562
128	*234766*	214685	2040729	1.093537043	0.115040263
256	**525457**	595211	6943638	0.882807945	0.075674596
512	**1163956**	1768441		0.658181981	

Table 7. Comparing SDF with VLIW (Zoom)

Data Size	SDF	Trimaran Optimized	TMS C'6000 Optimized	SDF/Trimaran	SDF/TMS 'C6000
50*50*4	**115452**	157770	144201	0.7317741	0.800632451
100*100*4	**459667**	630520	641625	0.72902842	0.716410676
150*150*4	**1032567**	1418270	1480525	0.72804685	0.697433005
200*200*4	**1833337**	2521020	2959430	0.72722033	0.619489902
250*250*4	**2862857**	3938770	4729593	0.72684036	0.605307264

.