Scalability Of Scheduled Dataflow Architecture (SDF) With Register Contexts

Joseph M. Arul
Fu Jen Catholic University, Taiwan
and
Krishna M. Kavi
University of North Texas, **USA**

Abstract

Our new architecture, known as Scheduled DataFlow (SDF) system deviates from current trend of building complex hardware to exploit Instruction Level Parallelism (ILP) by exploring a simpler, yet powerful execution paradigm that is based on dataflow, multithreading and decoupling of memory accesses from execution. A program is partitioned into non-blocking threads. In addition, all memory accesses are decoupled from the thread's execution. Data is pre-loaded into the thread's context (registers), and all results are post-stored after the completion of the thread's execution. Even though multithreading and decoupling are possible with control-flow architecture, the non-blocking and functional nature of the SDF system make it easier to coordinate the memory accesses and execution of a thread. In this paper we show some recent improvements on SDF implementation, whereby threads exchange data directly in register contexts, thus eliminating the need for creating thread frames. Thus it is now possible to explore the scalability of our architecture's performance when more register contexts are included on the chip.

Keywords: Scheduled Dataflow Architecture, Superscalar, superspeculative, Multithreaded architectures.

1. Introduction

In today's computer industry, von Neumann or control flow architecture, which dates back to 1946, still dominates the programming model. In order to overcome performance limitations of this model, modern architectures rely on instruction-level parallelism [1], by executing multiple instructions every cycle, often executing instruction in an order other than that specified by the program. Alternatively, multiple independent instructions can be packed into a wide instruction word (VLIW) so that the component instructions can be executed simultaneously. In order to increase the number of instructions that can be issued (either statically or dynamically), speculative execution is often employed [2]. The single threaded programming model limits the instruction-level parallelism that can be exploited in modern processors. Smith, in [3] promotes building general-purpose micro architectures composed of small, simple, interconnected processors (or execution engines) running at very high clock frequencies. Smith advocates a shift from instruction-level parallelism to instruction-level distributed processing with more emphasis on inter-instruction communication along with dynamic optimization and a tight interaction between hardware and low-level software [3].

Dataflow architecture [4, 5, 6] is an alternative to the von Neumann model. However previous attempts to develop practical systems based on dataflow model have failed for numerous reasons [15]. Hybrid models that combine the two alternatives have also been explored. Our SDF architecture can be viewed as a new hybrid approach. SDF also presents an alternative to the instruction level parallelism, albeit different from Smith's instruction level distributed processing.

The memory hierarchies present yet another challenge in the design of high-performance processors, since multiple levels of the hierarchy may need to be traversed to obtain a data value. Decoupling of memory accesses from execution can alleviate the memory-CPU performance gap [7]. The main feature of this decoupling is to separate the operand accesses from execution. We believe that to fully benefit from decoupling we must employ multithreading with multiple register contexts such that several threads are ready to execute. Our SDF combines decoupling with non-blocking multithreading. This architecture differs from other

multithreaded architectures by using non-blocking threads and the instructions of a thread obey dataflow (or functional) properties. The deviation of the decoupled Scheduled DataFlow (SDF) system from "pure" dataflow is a deviation from data driven execution (or token driven execution) that is traditionally used for its implementation. Section 2 will present the background and related research. In Section 3 we describe our new model of architecture and briefly summarize our threaded code generation. Section 4 describes the comparison of SDF with that of superscalar architecture and VLIW. Concluding remarks and future work will be presented in the last section.

2. Background and Related Research

Even though dataflow architecture provided the natural elegance of eliminating anti- and output-dependencies, it performed poorly on sequential code. In an eight-stage pipeline machine such as Monsoon, an instruction of the same thread can only be issued to the dataflow pipeline after the completion of its predecessor instruction. Besides, the token matching, waiting-matching store, introduced more bubbles or stalls in the execution stage(s) of the dataflow machines. In order to overcome these drawbacks, researchers explored hybrid of dataflow/control-flow models along with multithreaded execution. In such models several tokens within a dataflow graph are grouped together as a thread to be executed sequentially under its own private program counter control, while activation and synchronization of threads are data-driven. Such hybrid [8, 9, 11, 12] architectures deviate from the original model, where the instructions fetch data from memory or registers instead of having instructions deposit operands (tokens) in "operand receivers" of successor instructions. The architecture presented in this paper, SDF, is one such hybrid architecture, designed to overcome the limitations of pure dataflow model as well as those described in section 1 (that pertain to control flow models). Its features include decoupling of memory accesses from execution pipeline, non-blocking multithreading, dataflow program paradigm and scheduling of instructions in a control-flow like manner.

Simultaneous Multithreading (SMT) [10] allows multiple independent threads to issue instructions each cycle to a superscalar processor. SMT attempts to combine thread level parallelism and instruction level parallelism when running multiple applications on the same processor. Thus, all the available threads compete for and share all of the superscalar's resources every cycle. Each application may exhibit different amounts of parallelism. The choice of implementing *Instruction Level Parallelism* (ILP) or *Thread Level Parallelism* (TLP) depends on the application and data. When per thread ILP is limited, TLP can achieve more parallelism. By combining ILP and TLP, SMT claims to achieve greater throughput and significant program speedups. SMT research also studies how TLP stresses other hardware structures such as memory system, branch prediction and cache misses. SMT has the advantage of flexible usage of TLP and ILP, fast synchronization, and a shared L1 cache over functional units. Although SDF does not rely on ILP, we contend that the decoupling and non-blocking models lead to fine-grained threads, yielding effectively similar performance gains as when ILP within coarse-grained threads are exploited.

3. Scheduled Dataflow (SDF) Execution model and Code Generation

In this section we briefly describe the Scheduled Dataflow architecture model and its code generation. For the conventional multiprocessor system, the program is explicitly partitioned into processes based on programming constructs such as loops, procedures, etc. In order to run the program on SDF, the program must explicitly be partitioned into threads of a sufficient granularity to balance TLP with the overheads of creating threads. A compiler must generate a number of code blocks, where each code block consists of several instructions from one or more threads that can be executed to completion either on *Synchronization Processor* (SP) or *Execution Processor* (EP) without interruption. *Synchronization Count* (SC) is the number of inputs needed for a thread before it can be scheduled for execution. A *Frame* is allocated when a thread is created. All data needed for the thread, including SC, is stored in the frame memory. When a thread receives the needed inputs, the Pre-load code of the thread moves the data from the frame memory into the register context allocated to the thread. Post-store code of a thread stores data from a thread's registers into the frame memories of awaiting threads. The SP is responsible for the pre-load and post-store portions of the code. Once a thread's registers are loaded with input data, it executes on EP without blocking or accessing memory.

In this paper we present a new optimization to the above sequence of activities. When a thread is created, we will first check to see if the thread can be allocated a register context directly (instead of allocating a frame memory). It will then be possible for the thread to receive inputs directly in its registers (instead of first receiving in its frame memory and then moving the data into registers during the pre-load portion of the code). However if no register context is available on a thread creation, we will allocate a frame as in the original SDF

implementation. In this event, a thread receives inputs in its frame memory. Our goal is to explore how the performance of SDF changes with the number of register contexts. This provides multiple scalability options within the SDF architecture: the architecture can be scaled using multiple SPs and EPs, as well as with multiple register sets.

To understand the difference between the use of frame memory (using FALLOC on thread creation) and registers (using RALLOC) on thread creation, consider the following code examples. In Figure 1, each of the threads consists of a pre-load, execute and post-store sections. Thread one (main) stores the data for thread two (loop) in the frame allocated when FALLOC RR4, R10 is executed. The pointer to the frame memory is returned in R10; R4 contains the address of the first instruction for loop thread, while R5 contains the synchronization count for loop (note the notation RRn refers to an even-odd pair of registers n and n+1). The STORE instructions of main thread reflect the passing of data to loop thread.

Since SDF is a non-blocking multithreaded architecture, it is necessary for the main thread to create an additional thread (see FALLOC RR4, R11) that waits for the results form the loop thread.

```
//Thread One
code main
LOAD RFP|2, R2
PUTR1 midmain
FORKEP R1
STOP
midmain: PUTR1 loop
        MOVE R1, R4
        PUTR1 5
        FALLOC  RR4, R10  ; frame allocated
            PUTR1 result          ; allocated with the
            MOVE R1, R4           ; frame memory
            PUTR1 1
            MOVE R1, R5
        FALLOC   RR4, R11  ; thread to create the
            PUTR1 finmain         ; result
            FORKSP
            STOP
finmain:   STORE R11, R10|2  ;store the return pointer
            PUTR1  1
            STORE R1, R10|3  ;store the SC for the return
            STORE R0, R10|4  ; store i value
            STORE R2, R10|5  ; store N value
            STORE R0, R10|6  ; store the result as 0

       // Thread two (loop)
       code loop
       LOAD RFP|2, R2        ; Load return ptr
       LOAD RFP|3, R3        ; Load return SC
       LOAD RFP|4, R4        ; Load i value
       LOAD RFP|5, R5   ; Load N value
       LOAD RFP|6, R6        ; Load result=0
       PUTR1  midloop
       FORKEP  R1
            STOP
midloop:  [If  i< N; then call another loop thread and
increase the i value]
            PUTR1 finloop
            FORKEP R1
            STOP

finloop:   [depending on the if condition return the result to
the thread called]
            //Thread three
            code result
            [The result is read and displayed]
```

Figure 1. Usage of FALLOC and STORE instructions.

In the second code example shown in Figure 2, we try to allocate register context when a thread is created using RALLOC. Thus the instruction RALLOC RR4, R10 in Figure 2 allocates a register set for the loop thread (the register set number is returned in R10). Now the main thread stores data for the loop thread directly in the allocated registers as shown by the RSTORE instructions in Figure 2. Here we eliminated the pre-load code for loop thread. In this example since we are creating only 2 threads (and at any given time there are at most 3 register sets needed), we will assume RALLOC will be successful in allocating a register set. However, in realistic programs, we need to fall back to creating threads using FALLOC when RALLOC fails to allocate register sets. This will increase the code size; however, it permits the exploration of the scalability of SDF as more register contexts are allocated.

```
//Thread One
        code main
        LOAD RFP|2, R2
        PUTR1 midmain
        FORKEP R1
        STOP
midmain: PUTR1 midloop
        MOVE R1, R4
        PUTR1 5
        MOVE R1, R5
        RALLOC RR4, R10  ; thread allocated
        PUTR1 midresult          ; allocated with the frame
        MOVE R1, R4
        PUTR1 1
        MOVE R1, R5
        RALLOC RR4, R11  ; thread to receive the
        PUTR1 finmain            ; result
        FORKSP R1
        STOP
finmain:  RSTORE R11, R10|2  ; store the return ptr
        PUTR1 1
        RSTORE R1, R10|3   ; store the SC for the ret
        RSTORE R0, R10|4   ; store i value
        RSTORE R2, R10|5   ; store N value
```

```
        RSTORE R0, R10|6   ; store the result as 0
        FFREE
        STOP
                //Thread two adds one to N
[This portion of the code is eliminated]
midloop: [If i<n then call another loop
        thread and increase the i value]
        PUTR1 finloop
        FORKSP R1
        STOP
finloop:  [depending on the if condition
        return the result to the thread called]
        // Thread three
        code result
        [The result is read and displayed]
```

Figure 2. Usage of FALLOC and STORE instructions

3.1 Execution Pipeline

As can be seen from Figure 3, the execution pipeline consists of four pipeline stages; *instruction fetch, decode, execute and write back. Instruction fetch unit* behaves like a traditional fetch unit, relying on a program counter to fetch the next instruction. *Decode and register fetch unit* obtains a pair of registers that contains the two source operands for the instruction. *Execute unit* executes the instruction and sends the results to write-back unit along with the destination register numbers. *Write back unit* writes (up to) two values to the register file. In SDF architecture, a pair of registers is viewed as source operands for an instruction. Data is stored in either the left or right half of a register pair by a previous instruction. As can be seen, the execution pipeline behaves more like a conventional pipeline (e.g., MIPS) while retaining the primary dataflow properties (single assignment, and flow of data from instruction to instruction). This eliminates the need for complex hardware for detecting write-after-read (WAR) and write-after-write (WAW) dependencies and register renaming, as well as unnecessary thread context switches on cache misses.

3.2 Synchronization Pipeline

As can be seen from Figure 4 below, the synchronization pipeline consists of six stages: *instruction fetch, decode, effective address, memory access, execute and write-back.* As mentioned earlier, the synchronization pipeline handles pre-load and post-store instructions. The *instruction fetch unit* retrieves an instruction belonging to the current thread using Program Counter (PC). The *decode unit* decodes the instruction and fetches register operands (using a register set). The *effective address unit* computes address for LOAD and STORE instructions. LOAD and STORE instructions only reference the frame memories of threads, using a Frame Pointer (FP) and an offset into the frames; both of which are contained in registers. The *memory access unit* completes LOAD and STORE instructions. Pursuant to a post-store, the synchronization count of a thread is decremented. The *write-back unit* completes LOAD (pre-load) and I-FETCH instructions, by storing the values in appropriate registers.

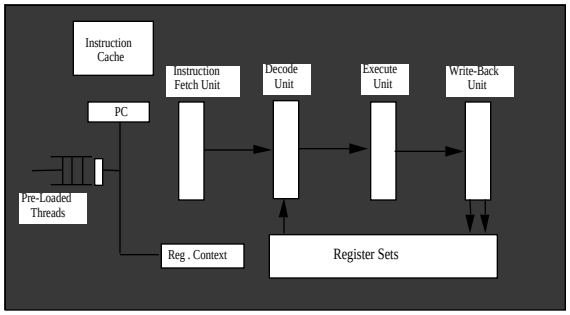

Figure 3. General Organization of the Execution Pipeline.

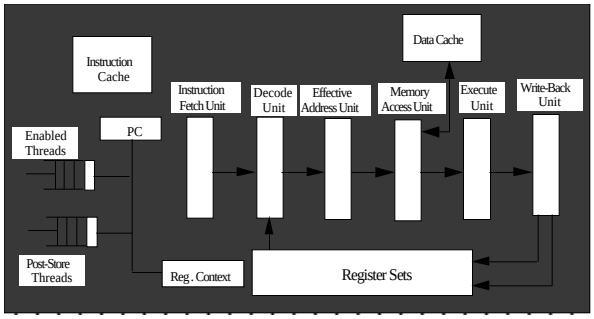

Figure 4. General Organization of the Synchronization Pipeline.

4. Comparisons of SDF with Superscalar architecture

Our previous papers (e.g., [14,15]) presented some comparisons of SDF with a MIPS like DLX, superscalar and VLIW architectures. In this paper, we will present some additional data along these lines, but also emphasize the scalability of SDF as register sets are added. Our contention is that SDF offers a viable alternative to superscalars and VLIW models. SDF architecture requires less complex hardware than superscalars (since SDF performs no dynamic instruction scheduling), and require no extra-ordinary techniques to utilize all instruction slots as needed in VLIW systems. And we will show that SDF scales better than superscalars and VLIW as more functional units and more register sets are added. In our experiments we set the number of functional units the same in SDF, superscalar and VLIW systems. For the superscalar, both in-order and out-of-order instruction issue are utilized. For the superscalar system, both

Table 1. SDF Versus Superscalar for the Program Matrix Multiply.

Data Size	SDF (Cycles)	SS-IO (Cycles)	SS-OO (Cycles)	SDF-SS-IO Speedup	SDF-SS-OO Speedup
50*50	1,720,885	1,968,235	1,174,250	1.1438	0.6824
100*100	13,318,705	15,434,835	9,170,850	1.1589	0.6886
150*150	44,453,530	51,811,474	30,747,479	1.1656	0.6917

instruction fetch and decode width and instruction issue width are set to eight. Register Update Unit (RUU) and Load/Store Queue (LSQ) are set to 32. The SDF system does not perform any dynamic instruction scheduling, eliminating complex hardware (e.g. scoreboards or reservation stations). At present, the SDF system uses no branch prediction. It is also important to note that the Simplescalar tool set[1] performs extensive optimizations and dynamic instruction scheduling. Several benchmarks were used to collect data and compare the performance of SDF and superscalar architecture. Matrix multiplication program will be used to explain in detail the performance of this architecture and its characteristics. The data shown in Table 1 is obtained when 10 concurrent threads are spawned in SDF.

As seen in the table, the SDF system outperforms in-order superscalar system (SS-IO) for all data sizes. Since in this experiment we used only 2 functional units (1SP and 1EP), SDF cannot cope with the thread level parallelism available in the program. However, the compiler for superscalar architecture was able to exploit the ILP available in the application to achieve higher out-of-order instruction issue rates (columns labeled SS-OO in Tables 1-3). In the next experiment we will show that as more functional units are added, and if the application exhibits thread level parallelism, SDF outperforms superscalar systems. Tables 2 and 3 show the comparison of SDF with superscalar system as more functional units are added. Superscalar does not scale well with increased number of functional units and the scalability is limited by instruction fetch/decode window size and the RUU size. It should be remembered that the instruction issue hardware is the most complex (and power hungry) component of modern architectures [18]. The SDF system relies primarily on thread level parallelism, and the decoupling of memory accesses from execution.

As more SPs and EPs are added (correspondingly more integer and floating point functional units in superscalar), the SDF system outperforms superscalar architecture, even when compared to complex out-of-order scheduling used by the superscalar system (SS-OO). The SDF system performance overtakes that of the out-of-order superscalar architecture with 3 SPs and 3 EPs (corresponding to 3 INT and 3 FP ALUs in the superscalar system). In fact, adding more functional units in the superscalar system requires more complex issue hardware for finding dynamic instructions to schedule and requires larger instruction issue/decode widths as well as renaming registers. In the SDF system adding more SP and more EP does not complicate the system since there is no dynamic instruction scheduling involved. Table 3 clearly shows the scalability of SDF as compared to

Table 2.SDF Versus Superscalar for Matrix Multiplication of Different Data sizes

Data size		Superscalar (cycles) 2INT ALU 1FP ALU	SDF (cycles) 2SP 1EP	Superscalar (cycles) 2INT ALU 2FP ALU	SDF (cycles) 2SP 2EP	Supserscalar (cycles) 3INT ALU 2FP ALU	SDF (cycles) 3SP 2EP	Superscalar (cycles) 3INT ALU 3FP ALU	SDF (cycles) 3SP 3EP
50*50	IO	1,890,104	1,504,297	1,890,104	860,782	1,867,200	756,707	1,867,200	**574242**
	OO	712,396		712,396		706,877		706,877	
100*100	IO	14,824,104	11,843,442	14,824,104	6,660,012	14,633,700	5,941,602	14,633,700	**4,402,772**
	OO	5,532,202		5,532,202		5,511,587		5,511,587	
150*150	IO	49,763,150	39,762,487	49,763,150	22, 227,742	49,110,246	19,924,912	49,110,246	**14,819,482**
	OO	18,514.510		18,514,510		18,468,811		18,468,409	

[1] http://www.cs.wisc.edu/~mscalar/simplescalar.html

Table 3. SDF Versus Superscalar for Matrix Multiplication with more than 6 Functional Units

Data size		Superscalar (cycles) 4INT ALU 3FP ALU	SDF (cycles) 4SP 3EP	Superscalar (cycles) 4INT ALU 4FP ALU	SDF (cycles) 4SP 4EP	Supserscalar (cycles) 5INT ALU 4FP ALU	SDF (cycles) 5SP 4EP	Superscalar (cycles) 5INT ALU 5FP ALU	SDF (cycles) 5SP 5EP
50*50	IO	1,867,200	**507,197**	1,867,200	**430,957**	1,867,200	**381,247**	1,867,200	**345,027**
	OO	680,321		680,321		680,321		680,321	
100*100	IO	14,633,700	**3,970,682**	14,633,700	**3,330,992**	14,633,700	**2,982,702**	14,633,700	**2,665,472**
	OO	5,306,381		5,306,381		5,306,380		5,306,380	
150*150	IO	49,110,246	**13,308,457**	49,110,246	**11,115,592**	49,110,246	**9,990,607**	49,110,246	**8,894,002**
	OO	17,782.453		17,782,453		17,782,453		17,782,453	

superscalar system (where no performance gains can be seen as more functional units are added).

4.1. Comparison of SDF with VLIW

This section presents the performance of the SDF system as compared with VLIW architectures as facilitated by the Texas Instruments TMX3206000[2] VLIW processor simulator tool-set (which includes an optimizing compiler and profiling tool), and the Trimaran infrastructure. For these two systems (VLIW and SDF), instruction execution and memory access cycles are set to match. The SDF system utilizes 8 functional units (4 SPs and 4 EPs) as compared to the 8-wide VLIW architecture. The SDF system also is compared with the Trimaran simulator using default configurations and optimizations (using a total of 9 functional units, a maximum loop unrolling of 32, and several other complex optimizations). Table 4 presents the data for matrix multiplication. TMS ‘C6000 does not perform well because the optimized version relies on the unrolling of only 5 iterations (unlike Trimaran, which uses 32 iterations). The SDF system achieves better performance than TMS ‘C6000 because it relies on thread level parallelism. The SDF system used 10 active threads for data shown in Table 4.

4.2. Scalability of SDF with Register Contexts

As indicated in Section 3. 1 (Figures 1 and 2), the original SDF (the data shown thus far in Tables 1-4) used frame memories for storing data for threads. This requires “pre-loading” of data from the frame memory into the register context of threads (after a thread’s synchronization requirements are satisfied, and a register context is allocated to the enabled thread). Instead, we could allocate register contexts for thread on creation so that data can be directly stored in the registers of the thread, thus eliminating memory accesses for storing and loading data. This is possible if sufficient register contexts are available to support the thread level parallelism exhibited by the application. When sufficient register sets are not available, we will fall back to the use of frame memories. This causes some overhead in first trying to allocate register sets and then allocating frames. The code size will also increase.

The data in Table 5 shows the scalability of SDF with more register sets. The data needs very careful examination. We varied the data size, number of functional units, TLP (by spawning different number of threads) and register sets. When only one

Table 4. Comparing SDF with VLIW (Matrix Multiplication).

Matrix Multiplication				
Data Size	**SDF**	**Trimaran**	**TMS ‘C6000**	**SDF/Trimaran**	**SDF/TMX ‘C6000**
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

[2] TMS320C6000 DSP Code Composer Studio by Texas Instruments

http://www.ti.com/sc/docs/tools/dsp/6ccsfreetool.html

Table 5. Limited Register sets usage and the thread allocation of Matrix Multiplication.

N=50	1 Thread	2 Threads	5 Threads
1SP 1EP	4985034 (3 Reg.Set)	2381565 (12 Reg.Set)	2330723 (70Reg.Set)
2SP 2EP	4985911 (")	1246765 (")	1166734 (")
3SP 3EP	4985011 (")	916006 (")	778910 (")
4SP 4EP	4985011 (")	810961 (")	585357 (")
N=100	**1 Thread**	**2 Threads**	**5 Threads**
1SP 1EP	38929984 (3 Reg.Set)	18581732 (12 Reg.Set)	18369157 (70Reg.Set)
2SP 2EP	38929961 (")	9582703 (")	9186942 (")
3SP 3EP	38929961 (")	7069808 (")	6126821 (")
4SP 4EP	38929961 (")	6164461 (")	4597262 (")
N=150	**1 Thread**	**2 Threads**	**5 Threads**
1SP 1EP	130334934 (3 Reg.Set)	62194407 (12 Reg.Set)	61689117 (64Reg.Set)
2SP 2EP		31862481 (")	30847669 (")
3SP 3EP		23473169 (")	20568612 (")
4SP 4EP		20445811 (")	15430183 (")

thread is spawned (second column), no performance improvements can be observed by increasing the number of functional units (i.e., SPs and EPs). Also no improvements in performance can be gained by adding additional register sets, since the application required only 3 register sets. When we increased the TLP by spawning 2 threads for each of the 3 loops of matrix multiplication (third column in Table 5), the performance improvements can be observed with additional functional units. Additional register sets (beyond 12) will not increase the performance since at this TLP only 12 register sets are needed. This column of data also shows the scalability of SDF with increased TLP and increased number of functional units. The last column further demonstrates the scalability of SDF. Here we spawned 5 threads at each loop. This level of TLP requires a maximum of 127 register sets. The data in the fourth column of Table 5 is for 64 register sets. It should be noted that the increased number of register sets (from 3 in second column, to 12 in third column to 64 in fourth column) shows performance gains only with increased TLP and increased number of functional units. We plan to further investigate the performance impacts of increasing TLP, functional units and register sets.

5. Conclusion and Future work

The data comparing the SDF system with multiple units of superscalar and VLIW shows that it is possible to build a simple (no out-of-order scheduling), cost effective machine when compared to complex hardware technology needed in modern superscalar and VLIW processors. Simple hardware and exploiting multithreading, decoupling and dataflow paradigm techniques can achieve this reduction. The data shows that it is possible to build an architecture with non-blocking threads and the decoupling of execution from memory accesses. Since the execution unit uses only registers, it proceeds with no bubbles or stalls in the pipeline. The hardware used in SDF can be much simpler. The SDF system uses no dynamic instruction scheduling. The hardware for dynamic instruction scheduling, such as scoreboard and reservation stations are not required. These hardware savings can be used to allocate more register sets in order to take advantage of the scalability of SDF as demonstrated in this paper. The research also demonstrates the scalability of SDF as more functional units are added to meet the TLP available in the application. There are only 4 pipeline stages in the EP and 6 in the SP. The shorter pipelines can also benefit in case of branch mispredictions. These pipelines also can be subdivided to increase clock speed as is done in modern Pentium processors. There are several projects for implementing multiple processors on the same chip (CMP) as an effective use of extra chip area. This is to take advantage of inter and intra instruction level parallelism. In the SDF system, the register sets are allocated to threads and the threads are scheduled on different units at different times. Such scheduling will be complex in an on-chip microprocessor. Further experiments can be performed with multiple processing elements with multiple units of SPs and EPs. Since each thread carries its own state (i.e., continuation) threads from multiple applications (or user processes) can be mixed freely on SDF. This will be very complex to achieve in conventional multiprocessor system, CMPs and SMTs. Currently a compiler for this architecture is under development. In the future, several benchmarks such as SPEC-2000 can be used

to show the improvements in the cycle count of the overall architecture and its performance.

Reference

[1]. Wall D. W., "Limits on instruction-level parallelism," *Proc. of 4th Intl. Conf. on Architecture Support for programming Languages and Operating Systems (ASPLOS-4)*, April 1991, pp.176-188.

[2]. Sato, T., "Quantative evaluation of pipelining and decoupling a dynamic instruction scheduling mechanism," *Journal of Systems Architecture*, Vol. 46, No.13, Nov. 2000, pp. 1231-1252.

[3]. Smith, J. E., "Instruction-level distributed processing," *IEEE Computer*, Vol.34, No.4, April 2001, pp. 59-65.

[4]. Kavi, K. M., and Shirazi, B., "Dataflow architecture: Are dataflow computers commercially viable?," *IEEE Potentials*, Oct. 1992, pp. 27-30.

[5]. Papadopoulos, G. M., and Culler, D. E., "Monsoon: An explicit token-store architecture," *Proc. of the 17th Intl. Symposium on Computer Architecture (ISCA-17)*, May 1990, pp. 82-91.

[6]. Papadopoulos, G. M., "Implementation of a general purpose dataflow multiprocessor," Tech. Report TR-432, *Laboratory for computer Science*, MIT, Cambridge, MA, Aug. 1988.

[7]. Smith, J. E., "Decoupled access/execute computer architectures," *Proc. of the 9th Annual Symposium on Computer Architecture*, May 1982, pp. 112-119.

[8]. Arvind and Nikhil R. S., "Executing program on the MIT Tagged-token dataflow architecture," *IEEE Transactions on Computers*, Vol. 39 No. 3, 1990, pp. 300-318.

[9]. Gao, G. R., "An efficient hybrid dataflow architecture model," *Journal of Parallel and Distributed Computing*, Vol. 19, 1993, pp. 279-307.

[10]. Tullsen, D. M., Eggers, S. J., Levy, H. M., and Lo, J. L., "Simultaneous multithreading: Maximizing on-chip parallelism," *In Annual Intl. Symposium on Computer Architecture* (ISCA-22), June 1995, pp. 392-403.

[11]. Culler, D. E., Goldstein, S. C., Schauser, K. E., and Eicken, T. V., "TAM - A compiler controlled Threaded Abstract Machine," *Journal of Parallel and Distributed Computing 18*, 1993, PP. 347-370.

[12]. Ang, B. S., Arvind and Chiou, D., "StarT- the next generation: Integrating global caches and dataflow architecture," *Technical Report 354, Laboratory for Computer Science*, MIT, Cambridge, MA, 1995.

[13]. Grünewald, W. and Ungerer, T., "Towards extremely fast context switching in a blockmultithreaded processor," *Proc. of the 22nd Euromicro Conf.*, Sept. 1996, pp. 592-599.

[14]. Kavi, K. M., Arul, J., and Giorgi, R., "Execution and cache performance of the scheduled dataflow architecture," *Journal of Universal Computer Science, Special Issue on Multithreaded Processors and Chip Multiprocessors*, Vol. 6, No. 10, Oct. 2000, pp. 948-968.

[15]. Kavi, K. M., Kim, H. S., Arul, J., and Hurson, A. R., "A decoupled scheduled dataflow multithreaded architecture," *Proc. of the International Symposium on Parallel Architectures Algorithms and Networks* (I-SPAN99), June 1999, pp. 138-143.

[16]. Lo. J. L., Eggers, S. J., Emer, J. S., Levy, H. M., Stamm, R. L., and Tullsen, D. M., "Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading," *ACM Trans. on Computer Systems*, Aug. 1997, pp. 332-354.

[17]. Mitchell, N., Carter, L., Ferrante, J. and Tullsen, D., "ILP vs TLP on SMT," *Proc. of Supercomputing*, Nov. 1999.

[18]. Watson., I. and Gurd, J. R., "A prototype data flow computer with token labeling," *Proc. of the National Computer Conference*, AFIPS Proceedings 48, 1979, pp. 623-628.

Acknowledgement. Some of the original SDF work was supported in part by a grant from the US National Science Foundation, # CCR-9796310.
The authors wish to acknowledge the contributions of Roberto Giorgi and Hyong-Shik Kim.