Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation

Krishna M. Kavi, *Senior Member*, *IEEE*, Roberto Giorgi, *Member*, *IEEE*, and Joseph Arul, *Student Member*, *IEEE*

Abstract—In this paper, the Scheduled Dataflow (SDF) architecture—a decoupled memory/execution, multithreaded architecture using nonblocking threads—is presented in detail and evaluated against Superscalar architecture. Recent focus in the field of new processor architectures is mainly on VLIW (e.g., IA-64), superscalar, and superspeculative designs. This trend allows for better performance, but at the expense of increased hardware complexity and, possibly, higher power expenditures resulting from dynamic instruction scheduling. Our research deviates from this trend by exploring a simpler, yet powerful execution paradigm that is based on dataflow and multithreading. A program is partitioned into nonblocking execution threads. In addition, all memory accesses are decoupled from the thread's execution. Data is preloaded into the thread's context (registers) and all results are poststored after the completion of the thread's execution. While multithreading and decoupling are possible with control-flow architectures, SDF makes it easier to coordinate the memory accesses and execution of a thread, as well as eliminate unnecessary dependencies among instructions. We have compared the execution cycles required for programs on SDF with the execution cycles required by programs on SimpleScalar (a superscalar simulator) by considering the essential aspects of these architectures in order to have a fair comparison. The results show that SDF architecture can outperform the superscalar. SDF performance scales better with the number of functional units and allows for a good exploitation of Thread Level Parallelism (TLP) and available chip area.

Index Terms—Multithreaded architectures, dataflow architectures, superscalar, decoupled architectures, Thread Level Parallelism.

1 Introduction

The performance gap between processors and memory has widened in recent years and the trend appears to continue in the foreseeable future. In this paper, we present an architecture that can overcome this problem, with better scalability than superscalar processors with increased number of pipelines. Our architecture is based on multithreading and dataflow concepts.

Multithreading has been touted as a solution to minimize the loss of CPU cycles by executing several instruction streams simultaneously. While there are several different approaches to multithreading, there is a consensus that multithreading, in general, achieves higher instruction issue rates on processors that contain multiple functional units (e.g., Superscalar and VLIW) or multiple processing elements (i.e., Chip Multiprocessors) [11], [23], [24], [40], [42], [44]. Nevertheless, research is open to find an appropriate multithreaded model and implementation to achieve the best possible performance. Recent efforts like the MP98 [15] show that attention to data-dependencies and hardware support for forking multiple threads help increase the performance.

We have found that the use of nonblocking dataflow-based threads are appropriate for improving the performance of superscalar architectures. Dataflow ideas are often utilized in modern processor architectures. However, these architectures rely on conventional programming paradigms and perform runtime transformations of the control-flow programs into dataflow programs, requiring complex hardware to detect data and control hazards, and reorder and issue multiple instructions.

Our architecture differs from other multithreaded architectures in two ways: 1) Our programming paradigm is based on dataflow, which eliminates the need for runtime instruction scheduling, thus reducing the hardware complexity significantly and 2) complete decoupling of all memory accesses from execution pipeline. The underlying dataflow and nonblocking models permit for clean separation of memory accesses from execution (which is very difficult to coordinate in other programming models). Data is preloaded into an enabled thread's register context prior to its scheduling on the execution pipeline. After a thread completes execution, the results are poststored from its registers into memory. The instruction set implements dataflow computational model, while the execution engine relies on control-flow-like sequencing of instructions (hence the name Scheduled Dataflow). Unlike Superscalar, our architecture performs no (dynamic) Out-of-Order execution and thus eliminates the need for complex instruction issue and retiring hardware. These hardware savings could be utilized to include either more processing units on a chip or more register sets to increase the degree of multithreading (i.e., Thread Level Parallelism). Moreover, it was stated that a significant power is expended by instruction issue logic and the power consumption increases quadratically with the size of the instruction issue width [27], [43]. Some researchers are exploring

- *K.M. Kavi and J. Arul are with the Department of Electrical and Computer Engineering, University of Alabama at Huntsville, Huntsville, AL 35899. E-mail: kavi@ece.uah.edu, arul@crash1.eb.uah.edu.*
- *R. Giorgi is with the Departimento di Ing. Informazione, Università di Siena, 56 Via Roma, 53100 Siena, Italy. E-mail: giorgi@acm.org.*

Manuscript received 1 Sept. 2000; accepted 8 Feb. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 113587.

mechanisms to construct dependence graphs at runtime to guide which instructions should be examined for readiness (rather than all instruction in the issue window) [27]. In our architecture, we perform no dynamic instruction scheduling (or select one or more instructions for issue from a large set of instructions). Thus, our approach could naturally obviate the need for runtime construction of dependence graphs.

We have translated several programs into our SDF instruction set. Using a cycle-level simulator developed at the University of Alabama in Huntsville (UAH), we have compared the execution performance of our architecture with that of conventional superscalar architecture with multiple functional units and aggressive Out-of-Order instruction issue logic as facilitated by the SimpleScalar tool set [10].

In Section 2, we will present research that is most closely related to ours. In Section 3, we will present our Scheduled Dataflow architecture in detail. Section 4 will discuss the methodology that we used in our evaluation and Section 5 will show our numerical results for real programs.

2 Related Research and Background

2.1 Decoupling Memory Accesses From Execution Pipeline

Decoupling memory accesses from the execution pipeline to overcome the ever-increasing processor-memory communication cost was first introduced in [34]. Since then larger cache memories have been used to alleviate the memory latency problem. But, the gap between processor speed and average memory access time is still a major limitation in achieving high performance. Increasing cache capacities, while consuming an increasingly large silicon area on processor chips, often results in diminishing returns. Decoupled architectures may again present a solution to leaping over the "memory wall." Decoupled ideas were recently used in a multithreaded architecture known as Rhamma [17]. Rhamma uses conventional control-flow programming paradigm and blocking threads, hence requiring many more thread context switches than our nonblocking dataflow threads. Moreover, SDF groups all Load instructions together into "preload" and all Store instructions together into "poststore." An analytical comparison with the Rhamma architecture was presented in [21] and, on that basis, we found that SDF outperforms Rhamma.

2.2 Dataflow Model and Architectures

The dataflow model and architecture have been studied for more than two decades and held the promise of an elegant execution paradigm with the ability to exploit inherent parallelism available in applications [4], [5], [12], [14], [28], [29], [30]. However, actual implementations of the model have failed to deliver the promised performance. Nevertheless, several features of the dataflow computational model have found their place in modern processor architectures and compiler technology (e.g., Static Single Assignment (SSA) [13], register renaming, dynamic scheduling and Out-of-Order instruction execution [18], I-structure-like synchronization [1], [6], nonblocking threads [8]). Most modern processors utilize complex hardware techniques to detect data and control hazards, and dynamic parallelism—to bring the execution engine closer to an idealized dataflow engine. It is our contention that such complexities can be eliminated if a more suitable and direct implementation of the dataflow model can be discovered. Some of the limitations of the pure dataflow model that prevented its practical implementations include the following: 1) too fine-grained (instruction level) multithreading, 2) difficulty in exploiting memory hierarchies and registers, and 3) asynchronous triggering of instructions.

Many researchers have addressed the first two limitations of dataflow architectures [12], [20], [30], [35], [37], [38], [39]. Our current architecture specifically addresses the third limitation. Some researchers have proposed hybrid designs in which the dataflow scheduling is applied only at thread level (i.e., macro-dataflow), while each thread is comprised of conventional control-flow instructions [16], [19], [31]. In such systems, the instructions within a thread do not retain functional properties and hence, introduce Write-After-Write (WAW) and Write-After-Read (WAR) dependencies. This in turn requires complex hardware to perform dynamic instruction scheduling. In our system, the instructions within a thread still retain functional properties of dataflow model and thus eliminate the need for complex hardware. The results (or data) flow from instruction to instruction; the data destined to an instruction is stored in registers exclusively assigned for the instruction. Our deviation, in the Scheduled Dataflow (SDF) system, from pure dataflow is a deviation from data driven asynchronous[1] execution (or token driven execution) that is traditionally used for the implementation of "pure" dataflow processors.

3 The Scheduled Dataflow Processor

We will first show how it is possible to "schedule" dataflow instructions. Let us consider a simple dataflow graph, shown in Fig. 1, and the corresponding SDF code. Each node of the graph will be translated into an SDF instruction. The two source operands (i.e., input data) destined for a dyadic SDF instruction are stored in a pair of registers specifically assigned to that instruction; a pair consists of even-odd registers; for example, RR2 refers to registers R2 and R3 within a specified thread context. Predecessor instructions store the data in either the left or right half of a register pair, as dictated by the data dependencies of the program. Unlike in conventional dataflow architectures—e.g., Monsoon [29], Tagged-Token Dataflow Architecture (TTDA) [7]—in our architecture, an instruction is not scheduled for execution immediately when the operands are matched (i.e., available). Instead, operands are saved in the register-pair associated with the instruction and the enabled instruction is scheduled for execution based on

1. It is often believed that dataflow means parallel execution. The dataflow model of computation only exposes the inherent parallelism and the parallelism can only be exploited if multiple functional units or processing elements are available. In the presence of a single processing element (or functional unit), dataflow instructions still execute sequentially, albeit asynchronously.

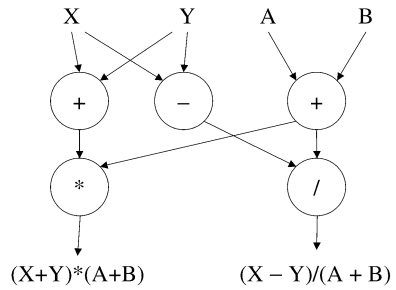


Fig. 1. A simple dataflow graph.

compile time ordering of the dataflow graph, eliminating the asynchronous execution implied by dataflow.

Assuming that the inputs A, B, X, and Y to the dataflow graph of Fig. 1 are available in R2, R3, R4, and R5, respectively (this is achieved during preload, as explained below), the five instructions shown in Fig. 2 will be *scheduled for execution sequentially* and perform the necessary computations, as indicated by the graph in Fig. 1. Note that a pair of registers is specified as source operands with each instruction. For example, ADD RR2, R11, R13 adds the contents of registers R2 and R3 and stores the result in R11 and R13. Our instructions still retain the functional nature of dataflow—each instruction stores its results in the registers that are specifically associated with destination instructions. There are no Write-After-Read (WAR or conceptually equivalent anti) and Write-After-Write (WAW or equivalent output) dependencies with our instructions.

The code shown in Fig. 2 is for the Execution Pipeline (EP). Since our architecture is a decoupled multithreaded system, we use two separate units: Synchronization Pipeline (SP) and Execution Pipeline (EP). SP prepares enabled threads for execution on EP by preloading threads' context (i.e., registers) with data from the threads' *Frame Memories* (Frame Memory is a portion of memory allocated to a thread) and poststoring results from completed threads' registers in frame memories of destination threads.

To illustrate the preload concept, consider Fig. 1 and the SDF code shown in Fig. 2. Assume that the code block of Fig. 2 (viewed as a thread) receives the four inputs (A, B, X, Y) from other threads. Inputs to a thread are saved in the frame memory allocated for the thread when the thread is created and a thread is enabled for execution only when it receives all inputs (as specified by its synchronization count). When enabled, a register context is allocated to the thread and the input data for the thread is "preloaded" from its frame memory into its registers.

Assuming that the inputs for the thread (A, B, X, and Y) are stored in its frame (pointed by RFP) at offsets 2, 3, 4, and 5, the first four LOAD instructions of Fig. 3 (executed by SP) preload the thread's data into registers R2, R3, R4, R5 of the register context allocated for the thread. After the preload, the thread is scheduled for execution on EP. The EP then uses only its registers during the execution of the thread body (Fig. 2). Consider 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 register couples R6 | R7 and R8 | R9, as shown in the preload code above (the last four LOAD instructions of Fig. 3).

The instructions shown in Fig. 4 transfer (or poststore) 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. As can be observed from this example, when a thread is created, it is necessary to provide the thread with the destination thread's frame pointers and offsets.

3.1 Continuations

To better understand the implementation of the SDF architecture, we need to focus first on the dynamic scenario that can be generated at run time. We need to introduce the concept of continuation: a continuation in our architecture is simply a four-value tuple, designated as <FP, IP, RS, SC>, where FP is the Frame Pointer (where thread input values are stored), IP is the Instruction Pointer (which points to the thread code), RS is a Register Set (a dynamically allocated register set), and SC is a Synchronization Count (the number of values needed to enable that thread). Each thread has an associated continuation. At a given time a thread continuation can be one of the following, where "**- -**" means that the value is either undefined or unnecessary:

- Waiting Continuation (WTC) or **<FP, IP, - -, SC>**
- Preload Continuation (PLC) or **<FP, IP, RS, - ->**
- Enabled Continuation (EXC) or **<- -, IP, RS, - ->**
- Poststore Continuation (PSC) or **<- -, IP, RS, - ->**

Thus, at a given time, a thread can be in one of four possible states: WTC, PLC, EXC, or PSC (Fig. 5) based on its continuation. After being created (in status WTC), a thread's continuation moves from "preload" (or PLC) status at SP to "execute" (or EXC) status at EP and finishes in "poststore" (PSC) status again at SP.

A Scheduler Unit (SU) handles the management of continuations and processing resources. In our design, the SU is very simple and can be implemented in hardware using a PLA. We now describe the details of the main functional units in our architecture: the EP, the SP, and the SU.

```
ADD   RR2,  R11, R13 ; compute 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      ; compute (X+Y)*(A+B), Result in R14
DIV   RR12, R15      ; compute (X-Y)/(A+B), Result in R15
```

Fig. 2. Code corresponding to the previous dataflow graph.

```
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  ; frame pointer for returning first result
LOAD RFP|7, R7  ; frame offset for returning first result
LOAD RFP|8, R8  ; frame pointer for returning second result
LOAD RFP|9, R9  ; frame offset for returning second result
```

Fig. 3. A sample preload code. This one corresponds to the previous example.

3.2 Execution Pipeline (EP)

Fig. 6 shows the block diagram of the Execution Pipeline (EP). Remember that EP executes computations of a thread using only registers. The instruction fetch unit behaves like a traditional fetch unit, relying on a program counter to fetch the next instruction.[2] We rely on compile time analysis to produce the code for EP so that instructions can be executed in sequence and assure that the instruction data already available in its pair of source registers.

The *instruction fetch* unit fetches an instruction belonging to the current thread using PC. The *decode* (and register fetch) unit decodes the instruction and obtains a pair of registers that contains (up to two) source operands for the instruction. The *execute* unit executes the instruction and sends the results to the write-back unit along with the destination register numbers. The *write-back* unit writes (up to) two values to the register file. As can be seen, the Execution Pipeline (EP) behaves more like a conventional pipeline while retaining the primary dataflow properties: Data flows from instruction to instruction. Moreover, the EP does not access data cache memory and, hence, requires no pipeline stalls (or context switches) due to cache misses.

3.3 Synchronization Pipeline

Fig. 7 shows the organization of the Synchronization Pipeline (SP), which mainly deals with memory accesses. Here, we deal with preload and poststore instructions. The pipeline consists of the following stages: the *instruction fetch* unit fetches an instruction belonging to the current thread using PC. The *decode* unit decodes the instruction and fetches register operands (using a Register Set). The *effective address* unit computes the effective address for LOAD and STORE instructions. LOAD and STORE instructions only reference the Frame memories[3] of threads, using a frame-pointer (FP) and an offset into the frame, both of which are contained in registers. The *memory access* unit completes LOAD and STORE instructions. Pursuant to a poststore, the synchronization count of a thread is decremented. The *write-back* unit completes LOAD (preload).

3.4 Scheduling Unit (SU)

In our architecture, a thread is created using a FALLOC instruction. FALLOC allocates a frame (accessible by a Frame Pointer FP) and initializes the frame by storing an Instruction pointer (IP) for the thread and a Synchronization Count (SC), which indicates the number of inputs needed to enable the thread. The FALLOC thus creates a WTC (**<FP, IP, - -, SC>**).

In order to speed up frame allocation, fixed sized frames for threads are preallocated and a stack of indices pointing to the available frames is maintained. The Scheduling Unit actually carries out this operation by popping an index from that stack. The SP pushes deallocated frames when executing FFREE instruction subsequent to poststores of completed threads. This policy permits fast context switching and creation of threads.

When some thread completes its execution and "poststores" results (performed by SP), the synchronization counts of each awaiting (WTC) thread are decremented. The SU takes care of checking when the synchronization count becomes zero. Then, it allocates a Register Set (RS) to that thread. The register sets are viewed as circular buffers for assigning (and deallocating) register contexts to enabled threads.

The thread's continuation becomes a PLC (**<FP, IP, RS, - ->**) and it is scheduled for execution on SP for preload. Then, SP loads the thread's data from its frame memory into the register context allocated. Upon the completion of preload, the thread continuation (in state EXC) is handed off to the Execution Processor (EP), using a FORKEP instruction. After the execution stage, we use FORKSP to move this thread back to SP.

FALLOC and FFREE take two cycles in our architecture. FORKEP and FORKSP take four cycles to complete. These numbers are based on the observations made in Sparcle [2] that a 4-cycle context switch can be implemented in hardware. Note the scheduling is at thread level in our system, rather than at instruction level as done in other multithreaded systems (e.g., Tera [3], SMT [41]), and, thus, requires simpler hardware.

The Scheduler Unit is also responsible for scheduling preload (PLC) and poststore (PSC) on multiple SPs and preloaded threads on multiple EPs in superscalar implementations of our architecture (Section 5.2).

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

Fig. 4. A sample poststore code. This one corresponds to the previous example.

2. Since both EP and SP need to execute instructions, our instruction cache is assumed to be dual ported. Since instruction memory causes no coherency related problems, it may be possible to utilize separate cache memories for EP and SP. This is not unlike most Superscalar systems.

3. Following the traditional dataflow paradigm, we use I-Structure memory for arrays and other structures.

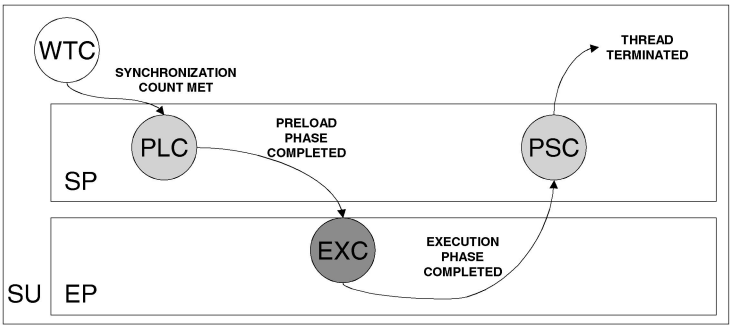


Fig. 5. Thread continuation transitions handled by the Scheduling Unit (SU).

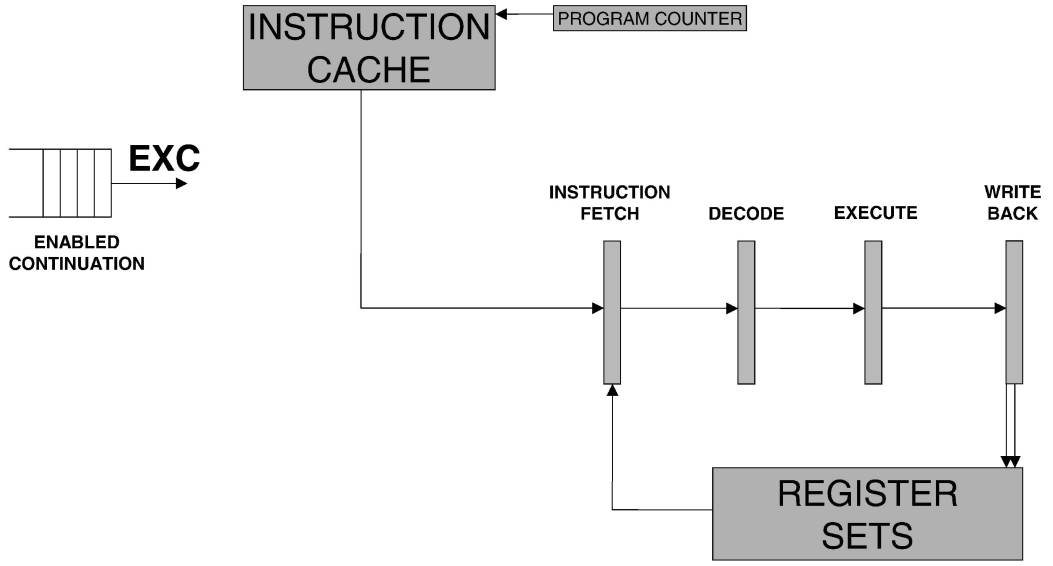


Fig. 6. General organization of Execution Pipeline (EP).

4 Methodology

We have evaluated our architecture based on execution of generated code for actual programs using our instruction level simulator.[4] The simulator used for this paper assumes a perfect cache (i.e., all memory accesses take one cycle). We have also developed a backend to Sisal [9] and used MIDC as intermediate language [32], [33] to generate code for our architecture, but still without implementing any particular optimization.

Previously, we have reported comparisons of SDF with MIPS-like architectures in [22]. In this paper, we will compare our SDF with a superscalar architecture with multiple functional units and Out-of-Order instruction issue logic as facilitated by the SimpleScalar tool set [10]. Also, we will investigate the effect of parallelism (i.e., number of enabled threads) and thread granularity (average run-lengths of the execution threads on EP) on the performance of our architecture (Sections 5.3, 5.4). We will investigate the performance gained by increasing the number of SPs and EPs (that is, Superscalar-SDF) and compare the performance with that of conventional superscalar processors containing multiple functional units (Section 5.2. The programs used for this study include a Matrix Multiply, FFT, Fibonacci, Zoom. 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; recursive Fibonacci exhibits very little parallelism (either instruction level or thread level); Zoom (a code segment of picture zooming application [36]) consists of three nested loops and a substantial amount of instruction level parallelism in the middle loop (but only small degrees of thread level parallelism).

4. We will provide the simulator and our benchmark programs to any interested reader so that our experimental data can be verified.

5 Evaluation of the Decoupled Scheduled Dataflow Architecture

In the first experiment, we have compared the execution performance of SDF (with one SP and one EP), with a superscalar processor with one Integer ALU (one integer adder and an integer multiply/divide unit) and one Floating Point ALU (one floating-point adder and a floating point multiply/divide unit). *This way SDF and the superscalar have the same number of functional units.*[5] For the superscalar,

5. Actually, the superscalar system contains four functional units, one integer adder, one integer multiply/divide, one floating point adder, one floating point multiply/divide. SDF has only two arithmetic units, one in SP and one in EP. There are no separate multiply/divide units.

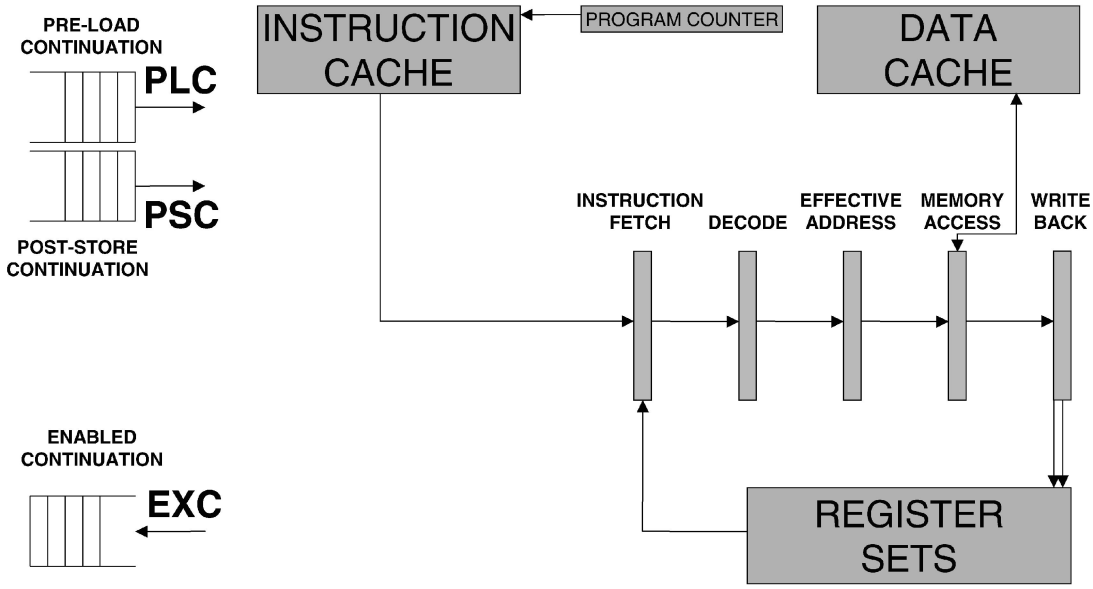


Fig. 7. The Synchronization Pipeline (SP).

TABLE 1
Superscalar Parameters for Tables 2, 3, 4, and 5

Superscalar Parameter	Value
Number of Functional Units	1 Integer Adder, 1 Integer Multiply/Divide 1 FP Adder, 1 FP Multiply/Divide
Instruction Issue Width	8
Instruction Fetch and Decode Width	8
Register Update Unit (RUU)	32
Load/Store Queue (LSQ)	32
Branch Prediction	Bimodal with 2048 entries

we will show data for both In-Order and Out-of-Order instruction issue. In all systems, we have set all instructions to take one cycle and assume perfect cache (all memory accesses are set to one cycle). For superscalar (Table 1), we have set the Instruction Fetch and Decode Width and Instruction Issue Width to 8, Register Update Unit (RUU) and Load/Store Queue (LSQ) to 32. *The last two columns of Tables 2, 3, 4, and 5 compare SDF with In-Order and Out-of-Order superscalar systems. We indicated in boldface the cases when SDF cycles result lower than both the In-Order and Out-of Order superscalar cases.*

For the Matrix Multiply program (Table 2), we have forked 10 threads to execute concurrently on the SDF. The Out-of-Order Superscalar system consistently outperforms SDF, although SDF performs better than the In-Order superscalar.

We are not surprised by this result since the SimpleScalar tool set performs extensive optimizations and dynamic instruction scheduling. SDF does not perform any dynamic instruction scheduling, eliminating complex hardware (e.g., Scoreboards or reservation stations [18]). Moreover, SimpleScalar utilizes branch prediction (the data shown uses Bimodal prediction with 2,048 entries). At present, SDF uses no branch prediction. Matrix Multiply program exhibits a large degree of instruction level parallelism and a good branch prediction is easy to achieve. Although the Matrix multiply program can be written to exhibit greater thread level parallelism, we have used a fixed number of threads (10) in this experiment. Later, we will show how the thread level parallelism can improve SDF performance (Section 5.3).

While executing FFT (Table 3), unlike for Matrix Multiply, SDF outperforms Out-of-Order superscalar only for larger input sizes (shown in bold). This is because of the available instruction-level and thread-level parallelism. For very small data sizes, Out-of-Order Superscalar system performs better than all other systems by exploiting instruction level parallelism. Very little thread level parallelism is available for such data sizes. However, for data sizes of 256 or larger, the available thread level parallelism in SDF (and the overlapped execution of SP and EP) exceeds the available instruction level parallelism, leading to a better performance by SDF. This data is in line with the studies performed on Simultaneous Multithreading systems [26], [25], which indicates that high performance is achieved by using a combination of thread level and instruction level parallelism. Fig. 8 shows this more clearly—for larger data sizes, SDF performs better than superscalar architectures.

The Recursive Fibonacci program (Table 4) exhibits very little parallelism (neither instruction level nor thread level). For very small data sizes, conventional superscalar systems

TABLE 2
Scheduled Dataflow (SDF) vs. Superscalar (SS) for the Program Matrix Multiply for Different Data Sizes

Data Size	SDF (cycles)	SS-IO (cycles)	SS-OO (cycles)	SDF vs. SS-IO speed-up	SDF vs. SS-OO speed-up
50*50	1,720,885	1,968,235	1,174,250	1.14	0.68
100*100	13,318,705	15,434,835	9,170,850	1.16	0.69
150*150	44,453,530	51,811,474	30,747,479	1.17	0.69

"IO" stands for In-Order and "OO" for Out-of-Order.

TABLE 3
Schedule Dataflow (SDF) vs. Superscalar (SS) for the Program FFT for Different Data Sizes

Data Size	SDF (cycles)	SS-IO (cycles)	SS-OO (cycles)	SDF vs. SS-IO speed-up	SDF vs. SS-OO speed-up
8	13,791	21,423	13,294	1.55	0.96
16	33,635	37,917	25,608	1.13	0.76
32	79,895	83,024	53,595	1.04	0.67
64	185,765	212,301	132,479	1.14	0.71
128	424,411	604,674	364,203	1.24	0.86
256	**955,721**	1,906,115	1,095,955	1.99	1.15
512	**2,126,583**	6,453,376	3,576,399	3.03	1.68

"IO" stands for In-Order and "OO" for Out-of-Order.

TABLE 4
Scheduled Dataflow (SDF) vs. Superscalar (SS) for the Program Fibonacci for Different Data Sizes

Data Size	SDF (cycles)	SS-IO (cycles)	SS-OO (cycles)	SDF vs. SS-IO speed-up	SDF vs. SS-OO speed-up
5	**799**	11,914	7,852	14.92	9.83
10	**8,092**	15,676	10,697	1.93	1.32
15	87,835	57,422	42,226	0.66	0.48
19	597,382	326,032	245,107	0.55	0.41

"IO" stands for In-Order and "OO" for Out-of-Order.

TABLE 5
Scheduled Dataflow (SDF) vs. Superscalar (SS) for the Program Zoom for Different Data Sizes

Data Size	SDF (cycles)	SS-IO (cycles)	SS-OO (cycles)	SDF vs. SS-IO speed-up	SDF vs. SS-OO speed-up
50*50*4	442,940	353,969	250,508	0.8	0.57
100*100*4	1,768,070	1,400,923	998,666	0.79	0.56
150*150*4	3,975,450	3,231,841	2,326,862	0.81	0.59
200*200*4	7,065,080	5,584,818	3,970,448	0.79	0.56

"IO" stands for In-Order and "OO" for Out-of-Order.

appear to incur overheads in creating recursive function calls, while SDF creates very few threads and incurs smaller overhead. As the data size increases, SDF creates too many threads, yet there is very little thread level parallelism, leading to poor performance by SDF as compared to superscalar systems. This is again in line with the general observation that multithreaded architectures perform poorly for applications with little or no thread level parallelism (and for single threaded applications).

The Zoom program (Table 5) contains substantial amounts of sequential code in the middle loop. This code allows for the exploitation of instruction level parallelism. However, it limits the amount of thread level parallelism. Moreover, in SDF, newly created threads wait for preload (and poststore) operations, causing the SP to be overloaded. As we will see later, SDF's performance improves when multiple SPs are used (see Tables 7, 8, 9, 10).

5.1 Summarizing

The data thus far confirms that any multithreaded architecture requires greater thread level parallelism to achieve good performance; superscalar architectures require greater instruction level parallelism. We feel that our nonblocking model is better suited for decoupling memory accesses from execution unit. The functional nature of our instructions eliminates the need for dynamic scheduling of instructions within a thread. Since our architecture uses two different types of pipelines (SP and EP), it is necessary to achieve a good balance of utilization between these two units. Our architecture incurs unavoidable overheads for creating threads (allocation of frames, allocation of register

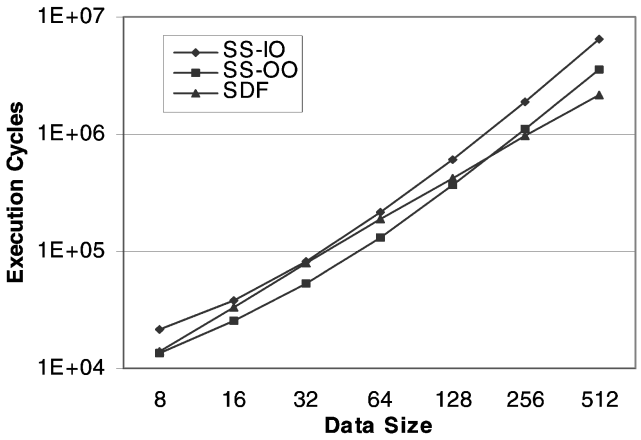


Fig. 8. Comparing SDF with a superscalar processor for FFT ("IO" stands for In-Order and "OO" for Out-of Order) for different data sizes.

contexts) and transferring threads between SP and EP (FORKEP and FORKSP instructions). At present, data can only be exchanged between threads by storing them in threads' frames (memory). These memory accesses can be avoided by storing the results of a thread directly into another thread's register context. Our experiments show that Matrix Multiply needs 16 frames with 10 parallel threads (for data shown in Table 2). For this application, we could have eliminated storing (and loading) thread's data in memory by allocating all frames directly in register sets (by providing sufficient register sets in hardware). It is our contention that the hardware savings achieved by SDF (by eliminating dynamic instruction scheduling logic) can be used to either increase the number of register sets (thus supporting greater thread-level parallelism) or add more SPs and EPs; either of which can improve the performance of SDF.

5.2 Execution Performance of SDF with Multiple SPs and EPs

In our next experiment, we have investigated the performance of SDF using multiple SPs and EPs and compared the performance with superscalar architectures using multiple Integer and Floating-Point units. We have utilized an equal number of functional units in our comparisons by setting the number of functional units in a superscalar (#Integer ALUs + #Floating Point ALUs)[6] equal to the number of SPs and EPs (#SPs + #EPs). 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 instruction fetch/decode window size and the RUU size. SDF relies primarily 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.

Tables 7, 8, 9, 10 show the results for this series of experiments. *In order to provide greater opportunities for dynamic instruction scheduling for the superscalar system, we have set the Instruction Fetch & Decode window widths to 32 and RUU to 32* (Table 6). We have observed little change in the performance (for the selected benchmarks) when the window width is increased beyond 32. We have also explored the impact of changing the RUU size. When RUU is set to 64, the performance of superscalar showed less than 5 percent improvement as compared to that with RUU set to 32.

In Table 7, we show the data for the Matrix Multiply program. 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 (shown in bold in Table 7), even when compared to complex out-of-order scheduling used by superscalar architectures.

SDF performance overtakes that of the Out-of-Order superscalar architecture with three SPs and three EPs (correspondingly, with three Integer and three FP ALUs in the Superscalar system). It should also be noted that, for the superscalar architecture, the performance improvement with increasing number of functional units scales poorly—superscalar architecture exhibits no improved performance beyond three Integer and three Floating Point ALUs. For SDF, the performance is limited by SPs—performance is improved consistently by adding more SPs.

This can more easily be seen from Fig. 9. 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 matrix multiplication with a 150*150 data size.

The next table (Table 8) shows the results for FFT. In this case, SDF outperforms Out-of-Order Superscalar for data sizes greater that 256 for all machine configurations. Once again, SDF performance scales better with added SPs than that of a superscalar when more functional units are added.

Fig. 10 shows the scalability of SDF for FFT (data size 256). Again, the x-axis shows the number of functional units (#SP + #EP for SDF; #Integer ALUs + #FP ALUs for Superscalar).

For Fibonacci (Table 9), as the number of SPs is increased, SDF compares more favorably with Out-of-Order Superscalar with a similar number of Integer units (as compared to the data in Table 4). As before, SDF performance scales better with more SPs and EPs than the superscalar case (when more functional units are added). Adding more FP ALUs in superscalar shows no improvement since Fibonacci does not utilize Floating Point arithmetic). Fig. 11 shows the scalability of SDF for Fibonacci (data size 15) more clearly. The x-axis shows the number of functional units (#SP + #EP for SDF; #Integer ALUs + #FP ALUs for superscalar).

Table 10 shows the data for the Zoom program. Once again, the performance of SDF scales better than Superscalar. With five SPs and four EPs, SDF outperforms the Out-of-Order Superscalar system with five Integer and four FP ALUs (shown in bold in Table 10), even for this program. Fig. 12 shows the scalability of SDF for Zoom (for 200*200*4) more clearly. The X-axis shows the number of functional units (#SP + #EP for SDF; #Integer ALUs + #FP ALUs for Superscalar).

6. Again, each ALU in superscalar contains separate adder and multiply/divide units. In SDF, each ALU is treated as a single unit performing all arithmetic operations.

TABLE 6
Superscalar Parameters for Tables 7, 8, 9, and 10

Superscalar Parameter	Value
Number of Functional Units	Varied
Instruction Issue Width	32
Instruction Fetch and Decode Width	32
Register Update Unit (RUU)	32
Load/Store Queue (LSQ)	32
Branch Prediction	Bimodal with 2048 entries

TABLE 7
Scheduled Dataflow (SDF) vs. Superscalar (SS) for the Program Matrix Multiply for 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	Superscalar (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	**4,440,772**
	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		18,468,409		18,468,409	

Data Size		Superscalar (cycles) 4INT ALU 3FP ALU	SDF (cycles) 4SP 3EP	Superscalar (cycles) 4INT ALU 4FP ALU	SDF (cycles) 4SP 4EP	Superscalar (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	

"IO" stands for In-Order and "OO" for Out-of-Order.

TABLE 8
Scheduled Dataflow (SDF) vs. Superscalar (SS) for the Program FFT for 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	Superscalar (cycles) 3INT ALU 2FP ALU	SDF (cycles) 3SP 2EP	Superscalar (cycles) 3INT ALU 3FP ALU	SDF (cycles) 3SP 3EP
256	IO	2,874,748	**992,129**	2,874,748	**638,705**	2,843,118	**559,665**	2,843,118	**558,353**
	OO	1,056,799		1,055,514		1,005,519		1,005,519	
512	IO	8,672,232	**1,516,660**	8,672,232	**1,418,356**	8,571,664	**1,240,948**	8,571,664	**1,238,580**
	OO	2,977,573		2,974,495		2,808,108		2,808,108	

Data Size		Superscalar (cycles) 4INT ALU 3FP ALU	SDF (cycles) 4SP 3EP	Superscalar (cycles) 4INT ALU 4FP ALU	SDF (cycles) 4SP 4EP	Superscalar (cycles) 5INT ALU 4FP ALU	SDF (cycles) 5SP 4EP	Superscalar (cycles) 5INT ALU 5FP ALU	SDF (cycles) 5SP 5EP
256	IO	2,871,928	**525,873**	2,871,928	**525,457**	2,871,928	**508,049**	2,871,928	**507,633**
	OO	1,013,417		1,013,417		1,012,031		1,012,031	
512	IO	8,639,150	**1,163,956**	8,639,150	**1,163,956**	8,639,150		8,639,150	
	OO	2,828,025		2,828,025		2,823,356		2,823,356	

"IO" stands for In-Order and "OO" for Out-of-Order.

5.2.1 Summarizing

The data for each of the benchmarks (Tables 7, 8, 9, 10) is consistent with our contention that SDF can be a viable alternative with multiple SPs and EPs to Superscalar architectures that utilize complex dynamic instruction scheduling logic. *In fact, it would be fairer to compare SDF with more functional units (#EPs + #SPs) than those in a Superscalar because of the hardware savings.* Our SPs and EPs are no more complex than a traditional functional unit used in Superscalar systems. We eliminate the complex instruction issue, register renaming, and instruction retiring logic. Scheduling of threads among available SPs and EPs is performed at thread level (instead of at instruction level, as done in Tera and SMT).

TABLE 9
Scheduled Dataflow (SDF) vs. Superscalar (SS) for the Program Fibonacci for 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	Superscalar (cycles) 3INT ALU 2FP ALU	SDF (cycles) 3SP 2EP	Superscalar (cycles) 3INT ALU 3FP ALU	SDF (cycles) 3SP 3EP
10	IO	14,677	7,408	14,677	**4,162**	14,208	**3,929**	14,208	**2,874**
	OO	7,064		7,064		6,471		6,471	
15	IO	51,580	80,802	51,580	44,033	50,566	41,481	50,566	29,463
	OO	27,967		27,967		24,439		24,439	

Data Size		Superscalar (cycles) 4INT ALU 3FP ALU	SDF (cycles) 4SP 3EP	Superscalar (cycles) 4INT ALU 4FP ALU	SDF (cycles) 4SP 4EP	Superscalar (cycles) 5INT ALU 4FP ALU	SDF (cycles) 5SP 4EP	Superscalar (cycles) 5INT ALU 5FP ALU	SDF (cycles) 5SP 5EP
10	IO	14,174	**2,764**	14,174	**2,263**	14,174	**2,193**	14,174	**1,888**
	OO	5,964		5,964		5,960		5,960	
15	IO	50,189	28,151	50,189	**22,206**	50,189	**21,415**	50,189	**17,841**
	OO	24,534		24,534		24,530		24,530	

"IO" stands for In-Order and "OO" for Out-of-Order.

TABLE 10
Scheduled Dataflow (SDF) vs. Superscalar (SS) for the Program Zoom for 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	Superscalar (cycles) 3INT ALU 2FP ALU	SDF (cycles) 3SP 2EP	Superscalar (cycles) 3INT ALU 3FP ALU	SDF (cycles) 3SP 3EP	Superscalar (cycles) 4INT ALU 3FP ALU	SDF (cycles) 4SP 3EP
100*100*4	IO	1,310,785	1,217,847	1,310,785	881,837	1,310,379	626,967	1,310,379	586,417	1,310,377	441,847
	OO	518,518		518,518		447,182		447,182		408,164	
200*200*4	IO	5,224,086	4,868,037	5,224,086	3,522,217	5,223,680	2,522,577	5,223,210	2,340,497	5,223,208	1,760,257
	OO	2,061,519		2,061,519		1,777,283		1,776,955	1,622,538		

Data Size		Superscalar (cycles) 4INT ALU 4FP ALU	SDF (cycles) 4SP 4EP	Superscalar (cycles) 5INT ALU 4FP ALU	SDF (cycles) 5SP 4EP	Superscalar (cycles) 5INT ALU 5FP ALU	SDF (cycles) 5SP 5EP	Superscalar (cycles) 6INT ALU 5FP ALU	SDF (cycles) 6SP 5EP	Superscalar (cycles) 6INT ALU 6FP ALU	SDF (cycles) 6SP 6EP
100*100*4	IO	1,310,377	440,417	1,310,377	**353,037**	1,310,377	**353,057**	1,309,907	**295,357**	1,309,907	**295,497**
	OO	408,164		398,454		398,454		398,128		398,128	
200*200*4	IO	5,223,208	1,756,957	5,223,208	**1,407,537**	5,223,208	**1,406,717**	5,223,208	**1,177,057**	5,223,208	**1,175,137**
	OO	1,622,538		1,585,528		1,585,528		1,585,528		1,585,528	

"IO" stands for In-Order and "OO" for Out-of-Order.

5.3 Effect of Thread Level Parallelism on Execution Behavior

Here, we will explore the performance benefits of increasing the thread level parallelism (i.e., number of concurrent threads) using one SP and one EP for SDF architecture. We have used the Matrix Multiply for this purpose. We have executed a 50*50 matrix multiply by varying the number of concurrent threads. Each thread has executed five (unrolled) loop iterations. In this data collection, we concentrated only on the innermost loop of Matrix Multiply, unlike previous data, where we have parallelized all three nested loops of Matrix Multiply, see Fig. 13.

As can be expected, increasing the degree of parallelism will not always decrease the number of cycles needed in a

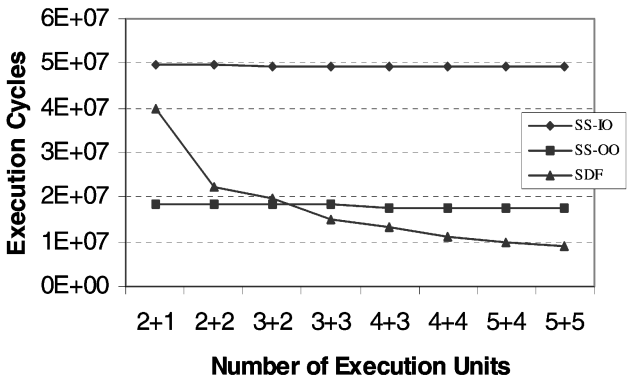


Fig. 9. Scalability of SDF over Superscalar for the program Matrix Multiply (data size 150*150).

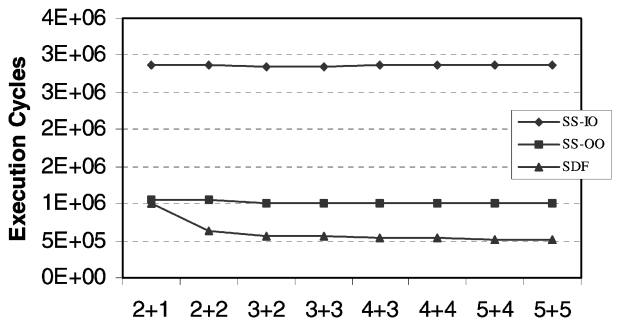


Fig. 10. Scalability of SDF over Superscalar for the program FFT (data size 256).

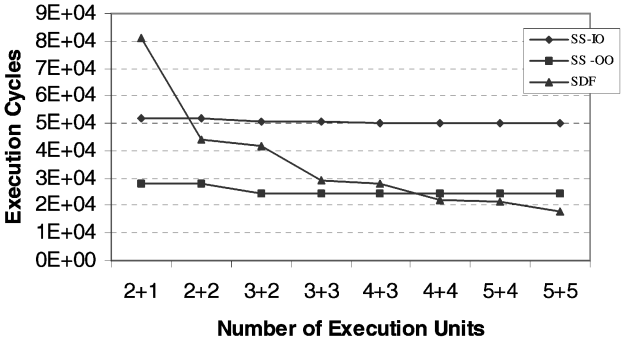


Fig. 11. Scalability of SDF over Superscalar for the program Fibonacci (data size 15).

linear fashion. This is due to the saturation of SP (reaching more than 90 percent utilization with 10 threads). As shown previously (Table 7, Fig. 9), adding additional SP and EP units (i.e., Superscalar-SDF implementation) will allow us to utilize higher levels of thread parallelism. Although not presented in this paper, we have observed very similar behavior with other data sizes for Matrix Multiply and with the other benchmarks, Fibonacci, FFT, and Zoom.

5.4 Effect of Thread Granularity on Execution Behavior

In the next experiment with Matrix Multiply, we have held the number of threads at five and varied the thread granularity by varying the number of innermost loop iterations executed by each thread (i.e., degree of unrolling). Once again, we have used one SP and one EP for this experiment and concentrated only on the innermost loop of Matrix Multiply.

The data size for Fig. 14 is 50*50. Here, the thread granularity ranged from an average of 27 instructions (12 for SP and 15 for EP), with no loop unrolling, to 51 instructions (13 for SP and 39 for EP), when each thread executed 10 unrolled loop iterations. Once again, the execution performance improves (i.e., execution time decreases) as the threads become coarser. However, the improvement becomes less significant beyond a certain granularity. Similar behavior has been observed for larger data sizes and other benchmarks. We are exploring innovative compiler optimizations utilizing static branch prediction to speculatively preload threads to increase thread run-lengths (i.e. granularities).

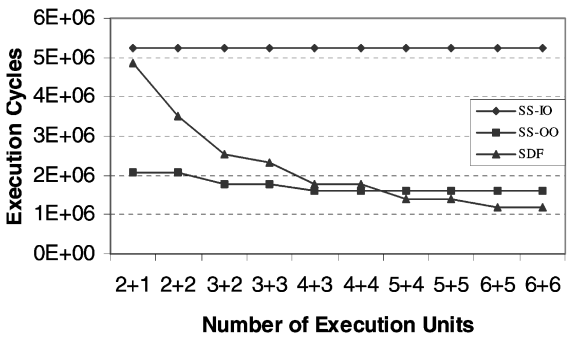


Fig. 12. Scalability of SDF over Superscalar for the program Zoom (data size 200*200*4).

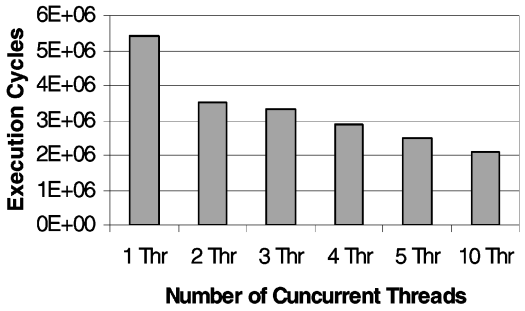


Fig. 13. Effect of thread level parallelism on SDF execution for the program Matrix Multiply (data size 50*50).

6 Conclusions and Future Work

In this paper, we have presented a nonblocking multithreaded dataflow architecture that utilizes control-flow-like scheduling of instructions. Our architecture separates memory accesses from instruction execution. Using an instruction set simulator for our decoupled Scheduled Dataflow (SDF), we have compared the execution performance of SDF with that of a superscalar with multiple functional units and aggressive Out-of-Order instruction issue logic. When the thread level parallelism is high, SDF substantially outperforms superscalar architectures (with multiple functional units) using In-Order instruction execution. SDF underperforms superscalar architectures with Out-of-Order execution, when the instruction level parallelism is high, but thread level parallelism is low. Also, the SP in SDF can be a bottleneck since threads can only be scheduled on EP after preload operations. The performance can be improved by adding more SPs. As a matter of observation, when more functional units are added, the Out-of-Order execution of superscalar architecture does not scale as well as SDF. Another factor that must be kept in mind while analyzing the data is that SDF uses no branch prediction (unlike superscalar architectures). At this time, we did not optimize our instruction set or the compiler that generated the code for the benchmarks.

SDF reduces the complexity of the processor by eliminating the need for complex logic (e.g., scoreboard or reservation stations) needed for resolving data dependencies, register renaming, Out-of-Order instruction issue, and branch predictions. The silicon area thus saved may be used to include more register-sets and registers per set to improve thread level parallelism and thread granularities or add more SPs and EPs. We are working to improve both

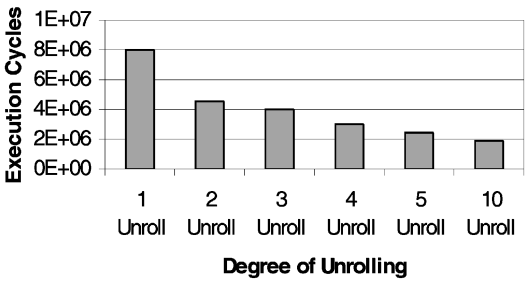


Fig. 14. Effect of thread granularity on SDF execution for the program Matrix Multiply (data size 50*50).

the instruction set and the compiler to produce more efficient executions of programs. At present, SDF uses no branch prediction, although we are planning to experiment with static branch prediction to speculatively preload data and increase run-lengths of our threads. Using compiler optimizations, speculative executions, and branch-prediction, we aim to increase the run-lengths of threads executing on EP.

While decoupled access/execute implementations are possible within the scope of conventional architectures, the multithreading model presents greater opportunities for exploiting the separation of memory accesses from the execution pipeline. We feel that, even among multithreaded alternatives, nonblocking models are more suited for the decoupled execution. 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 a clean decoupling of memory accesses into preloads and poststores. This can lead to greater data localities and relatively low cache-miss rates.

ACKNOWLEDGMENTS

This research is supported in part by the following grants from the US National Science Foundation: CCR-9796310, EIA-9805216, and EIA-9820147 and Italian grant from CNR 203.15.9/97. the authors also than the anonymous reviewers for their work.

REFERENCES

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-J. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Architecture and Performance," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 2-13, June 1995.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeoung, G. D'Souza, and M. Parkinet, "Sparcle: An Evolutionary Processor Design for Multiprocessors," *IEEE Micro,* pp. 48-61, June 1993.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The TERA Computer System," *Proc. 1990 Int'l Conf. Supercomputing,* pp. 1-6, July 1990.

[4] B.S. Ang, Arvind, and D. Chiou, "StarT—The Next Generation: Integrating Global Caches and Dataflow Architecture," Technical Report 354, Laboratory for Computer Science, Massachusetts Inst. of Technology, Aug. 1994.

[5] Arvind and K.S. Pingali, "A Dataflow Architecture with Tagged Tokens," Technical Memo 174, Laboratory for Computer Science, Massachusetts Inst. of Technology, Sept. 1980.

[6] Arvind, R.S. Nikhil, and K.S. Pingali, "I-Structures Data Structures for Parallel Computing," *ACM Trans. Programming Languages and Systems,* vol. 11, no. 4, pp. 598-632, Oct. 1989.

[7] Arvind and R.S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Trans. Computers,* vol. 39, no. 3, pp. 300-318, Mar. 1990.

[8] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Proc. Fifth ACM Symp. Principles and Practice of Parallel Programming (PoPP),* pp. 206-215, July 1995.

[9] A.D.W. Bohm, D.C. Cann, J.T. Feo, and R.R. Oldehoeft, "SISAL Reference Manual: Language Version 2.0," Technical Report CS91-118, Computer Science Dept., Colorado State Univ., 1991.

[10] D. Burger and T.M. Austin, "The SimpleScalar Tool Set Version 2.0," Technical Report #1342, Dept. of Computer Science, Univ. of Wisconsin, Madison, 1997.

[11] M. Butler et al., "Single Instruction Stream Parallelism Is Greater than Two," *Proc. 18th Int'l Symp. Computer Architecture (ISCA-18),* pp. 276-286, May 1991.

[12] D.E. Culler and G.M. Papadopoulos, "The Explicit Token Store," *J. Parallel and Distributed Computing,* vol. 10, no. 4, pp. 289-308, 1990.

[13] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Programming Languages and Systems,* vol. 13, no. 4, pp. 451-490, Oct. 1991.

[14] J.B. Dennis, "Dataflow Supercomputers," *Computer,* pp. 48-56, Nov. 1980.

[15] M. Edahiro, S. Matsushita, M. Yamashina, and N. Nishi, "Single-Chip Multiprocessor for Smart Terminals," *IEEE Micro,* vol. 20, no. 4, pp. 12-20, July 2000.

[16] R. Govindarajan, S.S. Nemawarkar, and P. LeNir, "Design and Performance Evaluation of a Multithreaded Architecture," *Proc. First High Performance Computer Architecture (HPCA-1),* pp. 298-307, Jan. 1995.

[17] W. Grunewald and T. Ungerer, "A Multithreaded Processor Design for Distributed Shared Memory System," *Proc. Int'l Conf. Advances in Parallel and Distributed Computing,* pp. 206-213, Mar. 1997.

[18] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* second ed. Morgan Kaufmann, 1996.

[19] H.H.-J. Hum et al., "A Design Study of the EARTH Multiprocessor," *Proc. Conf. Parallel Architectures and Compilation Techniques (PACT),* pp. 59-68, June 1995.

[20] R.A. Iannucci, "Toward a Dataflow/Von Neumann Hybrid Architecture," *Proc. 15th Symp. Computer Architecture (ISCA-15),* pp. 131-140, May 1990.

[21] K.M. Kavi, H.S. Kim, and A.R. Hurson, "Scheduled Dataflow Architecture: A Synchronous Execution Paradigm for Dataflow," *IASTED J. Computers and Applications,* vol. 21, no. 3, pp. 114-124, Oct. 1999.

[22] K.M. Kavi, J. Arul, and R. Giorgi, "Execution and Cache Performance of the Scheduled Dataflow Architecture," *J. Universal Computer Science,* special issue on multithreaded and chip multiprocessors, vol. 6, no. 10, pp. 948-967, Oct. 2000.

[23] V. Krishnan and J. Torrellas, "Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Trans. Computers,* vol. 48, no. 9, pp. 866-880, Sept. 1999.

[24] M. Lam and R.P. Wilson, "Limits of Control Flow on Parallelism," *Proc. 19th Int'l Symp. Computer Architecture (ISCA-19),* pp. 46-57, May 1992.

[25] J.L. Lo et al., "Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading," *ACM Trans. Computer Systems,* pp. 322-354, Aug. 1997.

[26] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen, "Instruction Level Parallelism vs. Thread Level Parallelism on Simultaneous Multi-Threading Processors," *Proc. Supercomputing '99,* http://www.supercomp.org/sc99/proceedings/papers/mitchell.pdf, 1999.

[27] S. Onder and R. Gupta, "Superscalar Execution with Direct Data Forwarding," *Proc. Int'l Conf. Parallel Architectures and Compiler Technologies (PACT-98),* pp. 130-135, Oct. 1998.

[28] G.M. Papadopoulos, "Implementation of a General Purpose Dataflow Multiprocessor," Technical Report TR-432, Laboratory for Computer Science, Massachusetts Inst. of Technology, Aug. 1988.

[29] G.M. Papadopoulos and D.E. Culler, "Monsoon: An Explicit Token-Store Architecture," *Proc. 17th Int'l Symp. Computer Architecture (ISCA-17),* pp. 82-91, May 1990.

[30] G.M. Papadopoulos and K.R. Traub, "Multithreading: A Revisionist View of Dataflow Architectures," *Proc. 18th Int'l Symp. Computer Architecture (ISCA-18),* pp. 342-351, June 1991.

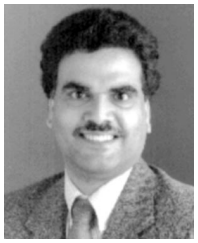
[31] S. Sakai et al., "Super-Threading: Architectural and Software Mechanisms for Optimizing Parallel Computations," *Proc. 1993 Int'l Conf. Supercomputing,* pp. 251-260, July 1993.

[32] B. Shankar, L. Roh, W. Bohm, and W. Najjar, "Control of Parallelism in Multithreaded Code," *Proc. Int'l Conf. Parallel Architectures and Compiler Techniques (PACT-95),* pp. 131-139, June 1995.

[33] B. Shankar and L. Roh, "MIDC Language Manual," technical report, CS Dept., Colorado State Univ., July 1996, http://www.cs.colostate.edu/~dataflow/papers/Manuals/manual.pdf.

[34] J.E. Smith, "Decoupled Access/Execute Computer Architectures," *Proc. Ninth Ann. Symp. Computer Architecture,* pp. 112-119, May 1982.

[35] M. Takesue, "A Unified Resource Management and Execution Control Mechanism for Dataflow Machines," *Proc. 14th Int'l Symp. Computer Architecture (ISCA-14),* pp. 90-97, June 1987.
[36] H. Terada, S. Miyata, and M. Iwata, "DDMP's: Self-Timed Super-Pipelined Data-Driven Multimedia Processor," *Proc. IEEE*, pp. 282-296, Feb. 1999.
[37] R. Thekkath and S.J. Eggers, "The Effectiveness of Multiple Hardware Contexts," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 328-337, Oct. 1994.
[38] S.A. Thoreson and A.N. Long, "A Feasibility Study of a Memory Hierarchy in Data Flow Environment," *Proc. Int'l Conf. Parallel Conf.,* pp. 356-360, June 1987.
[39] M. Tokoro, J.R. Jagannathan, and H. Sunahara, "On the Working Set Concept for Data-Flow Machines," *Proc. 10th Ann. Symp. Computer Architecture (ISCA-10),* pp. 90-97, July 1983.
[40] J.Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P.C. Yew, "The Superthreaded Processor Architecture," *IEEE Trans. Computers,* vol. 48, no. 9, pp. 881-902, Sept. 1999.
[41] D.M. Tullsen et al., "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 392-403, 1995.
[42] D.W. Wall, "Limits on Instruction-Level Parallelism," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-4),* pp. 176-188, Apr. 1991.
[43] 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,* Dec. 1999.
[44] V. Milutinovic, *Microprocessor and Multiprocessor Systems.* Wiley Int'l, 2000.



Krishna M. Kavi received the BE (electrical) degree from the Indian Institute of Science and the MS and PhD degrees (computer science and engineering) from Southern Methodist University. He is currently a professor and eminent scholar of computer engineering at the University of Alabama at Huntsville (UAH). Prior to joining UAH, he was a professor of computer science and engineering at the University of Texas at Arlington. For two years (1993-1995), he was a program manager at the National Science Foundation, managing operating systems and programming languages and compilers programs in the CCR Division. He was an IEEE Computer Society (CS) Distinguished Visitor (1989-1991), editor of the *IEEE Transactions on Computers* (1993-1997), and editor of the Computer Society Press (1987-1991). His primary research interest lies in computer systems architecture, including dataflow and multithreaded systems, memory management, operating systems, and compiler optimization. His other research interests include formal specification of concurrent processing systems, performance modeling, and evaluation, load balancing, and scheduling of parallel programs. He has published more than 125 technical papers on these topics. He is a senior member of the IEEE and a member of the ACM.

Roberto Giorgi received the MS degree in electronic engineering, summa cum laude, and the PhD degree in computer engineering, both from the University of Pisa, Italy. He is currently an assistant professor in the Department of Information Engineering, University of Siena, Italy. He was a research associate in the Department of Electrical and Computer Engineering, University of Alabama at Huntsville. His main academic interest is computer architecture and, in particular, multithreaded and multiprocessors systems. He is exploring coherence protocols, compile time optimizations, behavior of user and system code, architectural simulation for improving the performance of a wide range of applications from desktop to embedded-systems, web-servers, and e-commerce servers. He took part in the ChARM project in cooperation with VLSI Technology Inc., San Jose, California, developing part of the software used for performance evaluation of ARM-processor-based embedded systems with cache memory. He is a member of the IEEE, IEEE Computer Society, and ACM.

Joseph Arul received the BSc degree in mathematics in 1981 from Indore University, India, and the MS degree in computer science in 1994 from De Paul University, Chicago. From 1994-1995, he was a lecturer at Fu Jen Catholic University, Taiwan. Currently, he is a computer engineering PhD student at the University of Alabama at Huntsville (UAH). His current research interests are computer architecture, parallel and distributed computing, multithreaded programs, and compilers. He is a student member of the IEEE and the IEEE Computer Society and a member of the ACM.