Comparing Execution Performance of Scheduled Dataflow With RISC Processors

Krishna M. Kavi
Roberto Giorgi
And
Joseph Arul
The University of Alabama in Huntsville

Abstract

In this paper we describe a new approach to designing multithreaded architecture that can be used as the basic building blocks in high-end computing architectures. Our architecture uses non-blocking multithreaded model based on dataflow paradigm. In addition, all memory accesses are decoupled from the thread execution. Data is *pre-loaded* into the thread context (registers), and all results are *post-stored* after the completion of the thread execution. The decoupling of memory accesses from thread execution requires a separate unit to perform the necessary pre-loads and post-stores, and to control the allocation of hardware thread contexts to the enabled threads. The non-blocking nature of threads reduces the number of context switches, thus reducing the overhead in scheduling threads. Our functional execution paradigm eliminates complex hardware required for dynamic scheduling of instructions used in modern superscalar architectures. We will present our preliminary results obtained from an instruction set simulator using several benchmark programs. We compare the execution of our architecture with that of MIPS architecture as facilitated by DLX simulator.

Key Words. Multithreaded architectures, Dataflow architectures, Superscalars, Decoupled Architectures.

1. Introduction

The performance gap between processors and memory has widened in the past few years and the trend appears to continue in the foreseeable future. Multithreading has been touted as the 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 in processors that contain multiple functional units (e.g., superscalars and VLIW) or multiple processing elements (i.e., Chip Multiprocessors) [2, 10, 11, 16, 17].

It is necessary to find an appropriate multithreaded model and implementation to achieve the best possible performance. We believe that the use of non-blocking dataflow based threads are appropriate for improving the performance of superscalar architectures. Dataflow ideas are often utilized in most modern processor architectures. However, these architectures rely on conventional programming paradigms and require complex runtime transformation of the control-flow programs into dataflow programs. This necessitates complex hardware to detect data and control hazards (renaming of registers and branch prediction), reorder and issue multiple instructions.

Our architecture differs from other multithreaded architectures in two ways: i) our programming paradigm is based on dataflow, and ii) complete decoupling of all memory accesses from execution pipeline. The underlying dataflow and non-blocking models of execution permit a clean separation of memory access (which is very difficult to coordinate in other programming models). Data is pre-loaded into an enabled thread's register context prior to its scheduling on the execution pipeline. After a thread completes execution, the results are post-stored from its registers into memory. Previously we used queuing models and mean value analyses to compare our architecture with conventional RISC processors and other decoupled systems [8,9]. Now, we have developed an instruction level simulator[1]. In this paper, we report our initial execution performance of SDF using the simulator We compared our architecture with a conventional scalar RISC processors using DLX simulator [6].

2.Related Research And Background

2.1. Decoupling Memory Accesses From Execution Pipeline

Decoupling memory accesses from the execution pipeline in order to overcome the ever-increasing processor-memory communication cost was first introduced in [14]. Advances in cache memory technologies made the decoupling unnecessary at that

[1] A complete instruction set specification (for both EP and SP) can be found in [3].

time. Moreover, control-flow model of conventional architectures presented difficulties in coordinating the instructions for Access and Execute units. The gap between processor speed and average memory access time is once again the major limitation in achieving high performance. Decoupled architectures, particularly when combined with multithreaded models may again present a solution in leaping over the "memory wall". Recently, a similar concept was the major guideline in the design of Rhamma [5]. A comparison (using analytical models) of our architecture with Rhamma can be found in [8,9].

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 in applications. However, the 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, register renaming, dynamic scheduling and out-of-order instructions execution, I-structure like synchronization, non-blocking threads). 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.

There have been several hybrid architectures proposed where the dataflow scheduling was applied only at thread level (i.e., macro-dataflow) with conventional control-flow instructions comprising threads (e.g., [4, 7, 13]). In such systems, the instructions within a thread do not retain functional properties, and introduce side effects, WAW and WAR dependencies. Not preserving dataflow properties at instruction level requires complex hardware for the detection of data dependencies and dynamic scheduling of instructions. A comparison of our architecture with a hybrid architecture EARTH [7] using analytical models was reported in [8,9].

3. Decoupled Scheduled Dataflow Architecture

Our architecture consists of two processing units: Synchronization Pipeline (SP) and Execution Pipeline (EP). SP is responsible for scheduling enabled threads on EP, pre-loading thread context (i.e., registers) with data from the thread's Frame memory, and post-storing results from a completed thread's registers in Frame memories of destination threads. A thread is enabled when all its inputs are received: the number of inputs is designated by its synchronization count, and the input data is stored in its Frame memory. The EP performs thread computations including integer and floating point arithmetic operations. In this section we will describe the two processing units in more detail.

3.1. Execution Pipeline

Figure 2 shows the block diagram of the Execution Pipeline (EP).

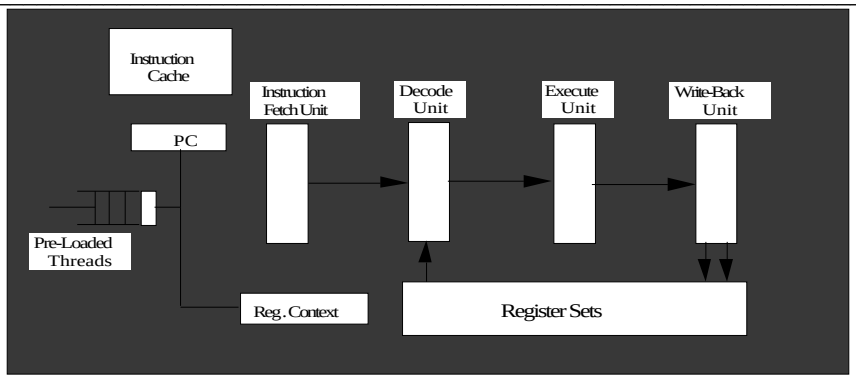

Figure 2. General Organization of Execution Pipeline (EP).

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 assured that the data for the instruction is already available in its pair of source registers (or can be forwarded within the pipeline from preceding instructions). The information in the Register context can be viewed as a part of the thread continuation: <ip, fp>, where fp refers to a register set assigned to the thread during its execution. *Decode (and register fetch) unit* obtains a pair of

[2] Since both EP and SP need to execute instructions, our instruction cache is assumed to be dual ported.

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 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 require no pipeline stalls (or context switches) due to cache misses.

3.2. Synchronization Pipeline

Figure 3 shows the organization of the primary pipeline of the Synchronization Processor (SP). Here we deal mostly with pre-load and post-store instructions. The pipeline consists of the following stages: *Instruction Fetch unit* fetches an instruction belonging to the current thread using PC. *Decode unit* decodes the instruction and fetches register operands (using Register Context). *Effective Address unit* computes effective address for memory access instructions. LOAD and STORE instructions only reference the Frame memories of threads, using a frame-pointer (FP) and an offset into the frame — both the frame-pointer and the offset are contained in registers. *Memory Access unit* completes LOAD and STORE instructions. Pursuant to a post-store, the synchronization count of a thread is decremented. *Execute unit* decrements synchronization counts. When the count becomes zero, the thread is moved to enabled list for pre-load and subsequent execution on EP. Finally, *Write-Back unit* completes LOAD (pre-load).

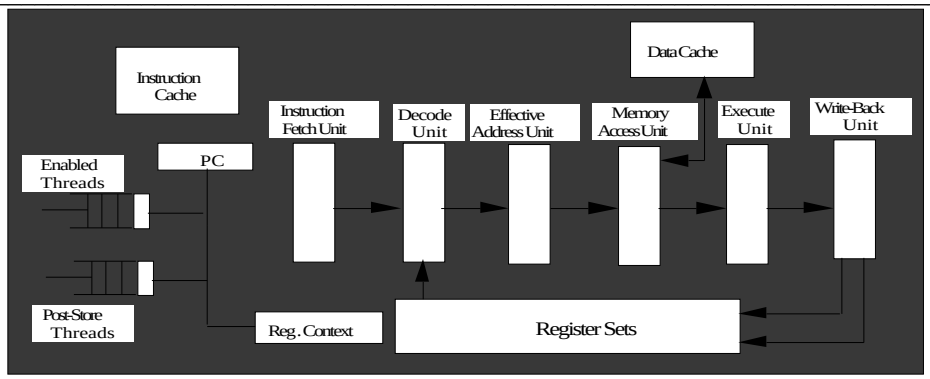

Figure 3. The Synchronization Pipeline.

In addition to accessing memory (for pre-load and post-store), Synchronization Pipeline (SP) holds thread continuations awaiting inputs and allocates register contexts for enabled threads. In our architecture a thread is created using a FALLOC instruction. FALLOC instruction creates a frame and stores instruction pointer (IP) of the thread and its synchronization count (Synch Count) indicating the number of inputs needed to enable the thread. When a thread completes its execution and "post-stores" results (performed by SP), the synchronization counts of awaiting threads are modified.

An enabled thread is scheduled by allocating a register context to it, and "pre-loading" the registers from its Frame memory. In order to speed up frame allocation, SP pre-allocates fixed sized frames for threads and maintains a stack of indexes pointing to the available frames. The Execution processor (EP) pops an index from the stack and uses it as the address of the frame (i.e., FP) in response to a FALLOC instruction. SP pushes de-allocated frames when executing FFREE instruction subsequent to post-stores of completed threads. The register sets (Reg. Context) are viewed as circular buffers for assigning (and de-allocating) to enabled threads. These policies permit for fast context switch and creation of threads. A thread moves from "pre-load" status (at SP), to "execute" status (at EP) and finishes in "post-store" status (at SP). We use FORKSP to move a thread from EP to SP and FORKEP to move a thread from SP to EP. FALLOC and FFREE take 2 cycles in our architecture. FORKEP and FORKSP take 4 cycles to complete. This number is based on the observations made in Sparcle [1] that a 4-cycle context switch can be implemented in hardware. Figure 4 shows a more complete view of the SP.

The scheduler unit is responsible for determining when a thread becomes enabled and allocating a register context to the enabled thread. Scheduler will also be responsible in scheduling preload and post-store threads on multiple SP's and preloaded threads on multiple EP's in superscalar implementations of our architecture. We are currently developing the superscalar implementation of SDF. Note the scheduling is at thread level in our system, rather than at instruction level.

Notice how a thread is identified differently during its life cycle. Initially, when a thread is created, a frame is allocated. Such a thread (called Waiting) will be identified by a Frame Pointer (FP), an Instruction Pointer (IP) that points to the first instruction of the thread, usually a pre-load instruction, and a synchronization count (Synch Count) indicating the number of inputs needed before the thread is enabled for execution. When the synchronization count becomes zero, the thread is moved to the Enabled list, following the allocation of a Register Context. At this time, the thread is identified by a FP, a Reg. Context, and IP. Once a thread completes the "pre-load" phase, it is moved to the Pre-Loaded list and handed off to the Execution Processor (EP). At this time, Register Context and the Instruction Pointer identify threads. The IP will now point to the first instruction beyond the pre-load (referring to the first executable instruction). After EP completes the execution of a thread, the thread is then moved to the Post-Store list and handed off to the SP for post-storing (by executing FORKSP instruction). At this time a Register Context and an IP identify the thread. The IP points to the first post-store instruction.

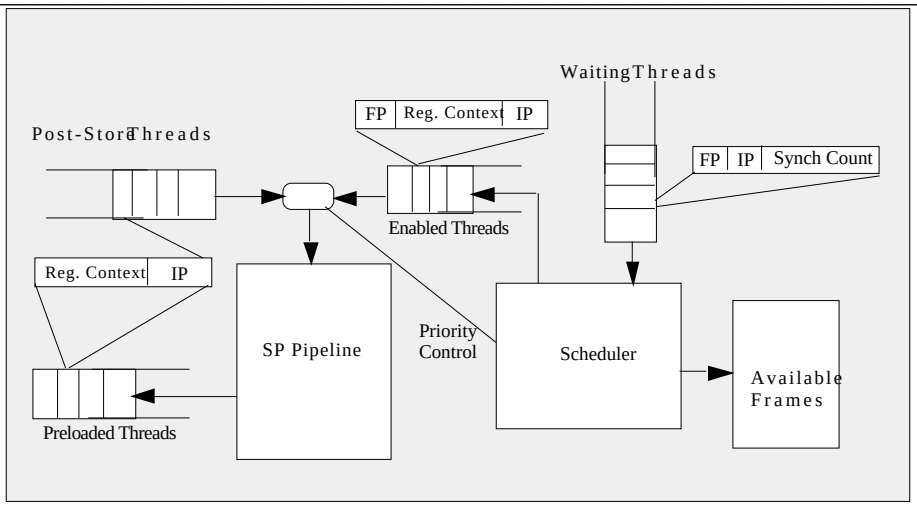

Figure 4. Overall Organization of the SP.

4. Evaluation of the Decoupled Scheduled Dataflow Architecture

Initially, we relied on analytical models and Monte Carlo simulations to compare the proposed architecture with Rhamma [5], ETS [12], a hybrid architecture [7] and conventional RISC processors [8,9]. More recently we developed an instruction level simulator for Scheduled Dataflow architecture. At present the simulator assumes a perfect cache. Using the simulator we were able to compare performance of the Scheduled Dataflow system with a single threaded RISC architecture. Our aim is to evaluate the benefits of separating memory accesses from execution pipeline: hence we use an architecture where a single pipeline executes all instructions including memory accesses to compare with our decoupled system. In the near future we will extend our studies to compare SDF with recent multithreaded and superscalar architectures.

4.1 Execution Performance Of Scheduled Dataflow.

In this section we compare the execution cycles required for Scheduled Dataflow with those for a conventional RISC system using DLX simulator [6]. The programs used for this comparison include a recursive Fibonacci program, Matrix Multiply, Livermore Kernel 5 and a code segment for picture zooming application [15]. We used dlxcc to generate DLX code in our comparisons. We equate a thread in SDF with a function in DLX -- if a SDF thread executes 5 (unrolled) loop iterations, so does the equivalent DLX function. The results are shown in Table 1.

We used a degree of 5 unrolling for Matrix multiply, Livermore Loop 5 and Zoom; we also used 5 (concurrent) threads for these 3 programs in SDF. In both platforms, we assumed one cycle per arithmetic and memory access instructions. However, if memory access requires more than one cycle (realistic caches with cache misses) we feel our multithreading will lead to even better performance than conventional single threaded system. As can be seen from Table 1, SDF system outperforms MIPS architecture when the program exhibits greater parallelism (e.g., Matrix Multiply, Zoom and

Livermore Loop 5). Livermore loop exhibits less parallelism than Matrix Multiply due to a loop carried dependency. Zoom exhibits moderate parallelism; however, a significant serial fraction (in the outer loop) exists, limiting the speed-up (Amdahl's law).

SDF underperforms when the program exhibits little parallelism (e.g. Fibonacci). This is in line with general acceptance that multithreaded architectures are not very effective for sequential (or single threaded) applications. The speed achieved for Matrix multiply really surprised us. Part of the speed up is because of the multithreading, partly due to the decoupling of memory accesses and partly due to lack of any pipeline stalls (due to the non-blocking dataflow model). Due to data dependencies encountered by DLX (from Load to ALU ops), more cycles were wasted. In addition, since SDF threads are equated to functions in DLX, and since DLX used stack for exchanging data, this may have caused some unnecessary memory accesses in DLX. It is satisfying to note that it is possible to design non-blocking, fine-grained multithreaded architectures with completely decoupled memory accesses, and achieve scalable performance. Our architecture incurs unavoidable overheads for creating threads (allocation of frames, allocation of register 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 11, 9, 8, 7, 6 when using 5, 4, 3, 2 and 1 concurrent thread, respectively. For this application, we could have eliminated storing (and loading) thread data in memory by allocating all frames directly in register sets (by providing sufficient register sets in hardware).

Table 1. Execution Behavior Of Scheduled Dataflow

Matrix Multiply(size N*N)				Zoom (PX*PY*C)				Livermore 5				Fibonacci			
N	DLX Cycles	SDF Cycles	Speed UP	Size	DLX Cycles	SDF Cycles	Speed UP	Size	DLX Cycles	SDF Cycles	Speed UP	N	DLX Cycles	SDF Cycles	Speed UP
25	966090	336153	2.87	5*5*4	10175	9661	1.05	50	87359	56859	1.54	5	615	842	0.730
50	7273390	2434753	2.99	10*10*4	40510	37421	1.08	100	354659	215579	1.65	10	7014	10035	0.699
75	2.4E+07	7938353	3.02	15*15*4	97945	83331	1.17	150	801959	476299	1.68	15	77956	111909	0.697
100	5.8E+07	18489453	3.14	20*20*4	161580	147391	1.09	200	1E+06	839019	1.70	20	864717	1E+06	0.696
				25*25*4	271175	229601	1.18	250	2E+06	1E+06	1.72	25	9590030	1E+07	0.696
				30*30*4	391150	329961	1.19	300	3E+06	2E+06	1.72	30	1.1E+08	2E+08	0.696
				35*35*4	532285	448471	1.19	350	4E+06	3E+06	1.72				
				40*40*4	645520	585131	1.10	400	6E+06	3E+06	1.73				
								450	7E+06	4E+06	1.74				

At this time we do not know if SDF performs better than a more recent RISC superscalar processor with dynamic instruction scheduling (i.e., out of order instruction issue and completion, predicated instructions). However, SDF system eliminates the need for complex hardware required for dynamic instruction scheduling. The hardware savings can be used to include additional register-sets, which can help in an increased degree of thread parallelism and thread granularities.

4.2.. Thread Level Parallelism

Here we will explore the performance benefits of increasing the thread level parallelism (i.e., number of concurrent threads). We used the Matrix Multiply for this purpose. We executed a 50*50 matrix multiply by varying the number of concurrent threads. Each thread executed five (unrolled) loop iterations. The results are shown in Figure 5. As can be expected, increasing the degree of parallelism will not always decrease the number of cycles needed in a linear fashion. This is due to the saturation of both the Synchronization and the Execution Pipeline (reaching nearly 80% utilization with 10 threads). Adding additional SP and EP units (i.e., superscalar implementation) will allow us to utilize higher thread level parallelism. The number of registers available per context also limits on how many concurrent threads can be spawned at a time. We are exploring techniques to enhance the thread level parallelism when multiple EP's and SP's are available. Although not presented in this paper, we observed very similar behavior with other data sizes for Matrix Multiply and the other benchmarks, Zoom and Livermore Loop 5.

4.3. Thread granularity.

In the next experiment with Matrix Multiply, we held the number of concurrent threads at 5, and

varied the thread granularity by varying the number of innermost loop iterations executed by each thread (i.e., degree of unrolling). The data size for Figure 6 is 50*50 matrices. Here, the thread granularity ranged form an average of 27 instructions (12 for SP and 15 for EP) with no loop unrolling, to 51 instructions (13 for EP and 39 for EP) when each thread executes ten unrolled loop iterations. Once again, the execution performance improves (i.e., execution time decreases) as the thread granularity increases. However, the improvement becomes less significant beyond certain granularity. The number of registers per thread context (currently 32 pairs) also is a limiting factor on the granularity. Our results confirm that performance of multithreaded systems can benefit both from the degree of parallelism and coarser grained threads. Because of the non-blocking nature and the decoupling of memory accesses, it may not always be possible to increase thread granularity in Decoupled Scheduled Dataflow (SDF). We are exploring innovative compiler optimizations utilizing speculative executions to increase thread run lengths.

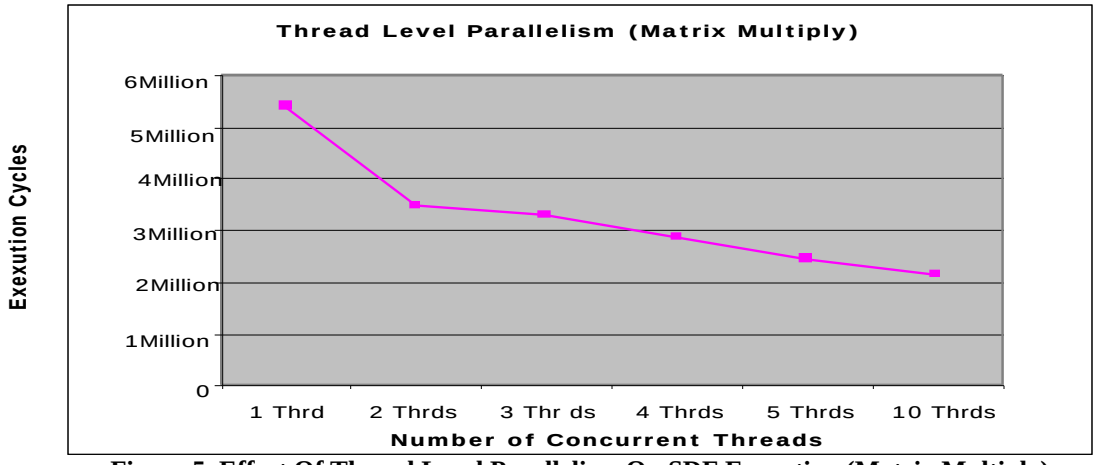

Figure 5. Effect Of Thread Level Parallelism On SDF Execution (Matrix Multiply)

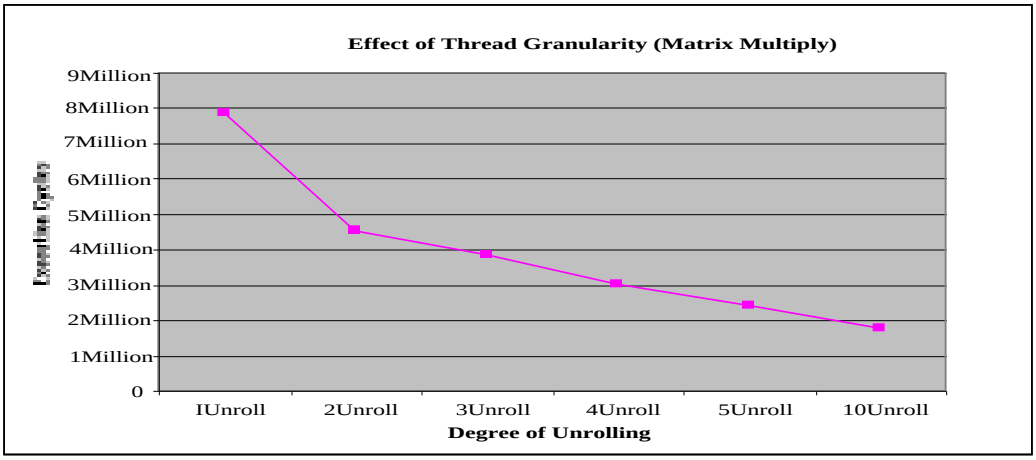

Figure 6. Effect Of Thread Granularity On SDF Execution (Matrix Multiply)

5. Conclusions

In this paper we presented a dataflow multithreaded architecture that utilizes control-flow like scheduling of instructions. Our architecture separates memory accesses from instruction execution to tolerate long latency operations. We developed an instruction set level simulator for our decoupled Scheduled Dataflow (SDF), and a backend to a Sisal compiler. Using these tools we compared the execution performance of SDF with that of a single pipelined MIPS processing system. Our results are very encouraging. When the degree of parallelism is high, SDF substantially outperforms MIPS. We also investigated the impact of increasing thread granularity and thread level parallelism. As with any multithreaded system, SDF shows performance

improvements with coarser grained threads and increased thread level parallelism. Our current architecture simulator assumes a perfect cache. We will soon incorporate realistic cache memories into our simulator.

While decoupled access/execute implementations are possible within the scope of conventional architectures, multithreading model presents greater opportunities for exploiting the separation of memory accesses from execution pipeline. We feel that, even among multithreaded alternatives, non-blocking models are more suited for the decoupled execution. In our model, threads exchange data only through the frame memories of threads. The use of frame memories for thread data permits for a clean decoupling of memory accesses into pre-loads and post-stores. This could lead to greater data localities and very low cache-miss rates.

At this time we do not know if our approach performs better than modern superscalar systems that use dynamic instruction scheduling (e.g., out of order instruction issue and completions) or other multithreaded systems such as SMT. However, our system reduces hardware complexity.

6. References

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

[2] M. Butler, et al. "Single instruction stream parallelism is greater than two", *Proc. of 18th Intl. Symposium on Computer Architecture (ISCA-18)*, pp 276-286, May 1991.

[3] R. Giorgi, K. M. Kavi and H.Y. Kim. "Scheduled Dataflow Instruction Manual", http://crash1.eb.uah.edu/~kavi /Research /sda.pdf

[4] R. Govindarajan, S.S. Nemawarkar and P; LeNir. "Design and performance evaluation of a multithreaded architecture", *Proceeding of the first High Performance Computer Architecture (HPCA-1*), Jan. 1995, pp 298-307.

[5] W. Grunewald and T. Ungerer, "A Multithreaded Processor Design for Distributed Shared Memory System," *Proc. Int'l Conf. on Advances in Parallel and Distributed Computing*, 1997.

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

[7 H.H.-J. Hum, ET. al., "A Design Study of the EARTH Multiprocessor," *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, Limassol, Cyprus, June 1995, pp. 59-68.

[8] H.S. Kim, K.M. Kavi and A.R. Hurson. "A simple non-blocking multithreaded architecture", *Proceedings of the 12th ISCA PDCS-99*, Ft. Lauderdale, FL, Aug. 18-20, 1999, pp 231-236.

[9] K.M. Kavi, H.-S. Kim, J. Arul and A.R. Hurson "A decoupled scheduled dataflow multithreaded architecture", *Proceedings of the I-SPAN-99*, Fremantle, Western Australia, June 23-25, 1999, pp 138-143.

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

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

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

[13] S. Sakai, et al, "Super-threading: Architectural and Software Mechanisms for Optimizing Parallel Computations," *Proc. of 1993 Int'l Conference on Supercomputing*, July 1993, pp. 251-260.

[14] J.E. Smith. "Decoupled Access/Execute Computer Architectures", *Proc of the 9th Annual Symp on Computer Architecture*, May 1982, pp 112-119.

[15] H. Terada, S. Miyata and M. Iwata. "DDMP's: Self-timed super-pipelined data-driven multimedia processor", *Proceedings of the IEEE*, Feb. 1999, pp. 282-296

[16] J. Y. Tsai, et al. "The Superthreaded processor architecture", IEEE Trans. on Computers, Sept. 1999, pp. 881-902.

[17] D.W. Wall. "Limits on instruction-level parallelism", *Proc of 4th Intl. Conference on Architectural support for Programming Languages and Operating Systems (ASPLOS-4)*, pp 176-188, April 1991.

Acknowledgements: This research is supported in part by NSF grants: CCR 9796310, EIA 9729889, EIA 9820147.