Journal of Universal Computer Science, vol. 6, no. 10 (2000), 948-967
submitted: 30/3/00, accepted: 29/8/00, appeared: 28/10/00

Execution and Cache Performance of the Scheduled Dataflow Architecture

Krishna Kavi
(The University of Alabama in Huntsville,
kavi@ece.uah.edu)

Joseph Arul
(The University of Alabama in Huntsville,
arulj@ece.uah.edu)

Roberto Giorgi
(Universita' di Siena, Italy
giorgi@acm.org)

Abstract: This paper presents an evaluation of our Scheduled Dataflow (SDF) Processor. Recent focus in the field of new processor architectures is mainly on VLIW (e.g. IA-64), superscalar and superspeculative architectures. This trend allows for better performance at the expense of an increased hardware complexity and a brute-force solution to the memory-wall problem. Our research substantially deviates from this trend by exploring a simpler, yet powerful execution paradigm that is based on dataflow concepts. A program is partitioned into functional execution threads, which are perfectly suited for our non-blocking multithreaded architecture. 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. 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 enabled threads.

The analytical analysis of our architecture showed that we could achieve a better performance than other classical dataflow architectures (i.e., ETS), hybrid models (e.g., EARTH) and decoupled multithreaded architectures (e.g., Rhamma processor). This paper analyzes the architecture using an instruction set level simulator for a variety of benchmark programs. We compared the execution cycles required for programs on SDF with the execution cycles required by the programs on DLX (or MIPS). Then we investigated the expected cache-memory performance by collecting address traces from programs and using a trace-driven cache simulator (Dinero-IV). We present these results in this paper.

Category: Processor Architectures, Performance of Systems.

Key Words: Multithreaded architectures, Dataflow architectures, Superscalars, Decoupled Architectures, Memory latency.

1 Introduction

Multithreading has been touted as the solution to minimize the loss of CPU cycles due to the performance gap between processors and memory, by executing several instruction streams simultaneously. Moreover there is a consensus that multithreading, in general, achieves higher instruction issue rates on processors that contain multiple functional units (e.g., superscalars and VLIW) or multiple processing elements (i.e., Chip Multiprocessors) [Butler 91], [Kavi 98a], [Krishnan 99], [Lam 92], [Tsai 99], [Wall 91].

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 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, which eliminates the need for complex runtime scheduling, thus reducing the hardware complexity, 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 accesses (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. The instruction set implements dataflow computational model, while the execution engine relies on control-flow like scheduling of instructions (thus, our instructions are *not data driven*). We have completed the definition of the instruction set and developed an instruction level simulator. We have translated several programs into our SDF instruction set. Using the simulator and the benchmark programs, we compared the execution performance of our architecture with that of conventional scalar RISC processors using DLX simulator [Hennessy 96]. The comparison is fair since both our SDF architecture and the MIPS are single-issue processors. We evaluated the expected cache performance by collecting address traces and using a trace-driven cache simulator (Dinero-IV [Edler 99]).

In Section 2 we present research that is most closely related to ours. In Section 3 we present our Scheduled Dataflow Architecture in detail. Section 4 discusses the methodology that we used in our evaluation and shows our numerical results based on real programs. Finally we present the concluding remarks in Section 5.

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 [Smith

82]. Since then the concept of cache memory has been used extensively to alleviate the memory latency problem. The gap between processor speed and average memory access time is once again a major limitation in achieving high performance. However, 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 for leaping over the "memory wall". There seems to be a growing interest in decoupling memory accesses from execution pipeline. We feel that combining the decoupled architecture with multithreading allows for a wide-range of implementations for next-generation architectures. Recently, a similar concept was the major guideline in the design of Rhamma [Grunewald 97]. A comparison of our architecture with Rhamma can be found in [Kavi 99a, 99b]. Rhamma uses conventional control-flow programming paradigm and blocking threads, hence requires many more thread context switches than our non-blocking dataflow threads. Moreover, SDF groups all Load instructions together into "preload" and all Store instructions together into "post-store". Thus SDF outperformed Rhamma in our analyses.

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. 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. It is our contention that such complexities can be eliminated if a more suitable 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:

Too fine-grained (instruction level) multithreading,
Difficulty in exploiting memory hierarchies and registers, and
Asynchronous triggering of instructions.

Many researchers have addressed the first two limitations of dataflow architectures [Kavi 95, 98b], [Papadopoulos 90, 91], [Takesue 87], [Thoreson 87], [Tokoro 83]. Our current architecture specifically addresses the third limitation.

Some researchers have proposed 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 [Govindarajan 95], [Hum 95], [Sakai 93]. In such hybrid dataflow-control flow 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, where each instruction specifies a location for the data to be stored. Our deviation in the proposed decoupled Scheduled Dataflow (SDF) system from pure dataflow is a deviation from data driven execution (or token driven execution) that is traditionally used for the implementation of "pure" dataflow processors[1]. The data-driven execution of dataflow program utilized in previous architectures required two cycles per (dyadic) instructions. By scheduling dataflow instructions (akin to control-flow execution) results in one cycle per instruction.

Using analytical models we compared SDF with hybrid architectures (e.g. EARTH [Hum 95]) that use two processors: one (Execution Processor) for executing instructions of a thread and a second processor (Synchronization Processor) to perform thread synchronizations and scheduling of threads. SDF outperformed hybrid architectures both because of the decoupling of memory accesses (not part of hybrid architectures) and because of the elimination of WAW and WAR dependencies that exist among the instruction of threads in hybrid architectures [Kavi 98a, 98b]; such dependencies can cause pipeline stalls

2.3 Explicit Token Store (ETS) Architecture

Since our architecture draws heavily from previous research on dataflow system, in general, and from the ETS model in particular [Papadopoulos 90, 91], we will describe the ETS model in some detail here. ETS uses direct matching of operands (or tokens) belonging to an instruction. In a direct matching scheme, storage (called frame) is dynamically allocated for all the tokens needed by the instructions in a code block. A code block can be viewed as a sequence of instructions comprising a loop body or a function. The actual disposition of locations within a frame is determined at compile-time; however, the actual allocation of frames is determined during run-time. In a direct matching scheme, any computation is completely described by a pointer to an instruction (IP) and a pointer to a frame (FP). The pair of pointers, <FP.IP>, called a *continuation,* corresponds to the tag part of a token. A typical instruction pointed to by an IP specifies an opcode; an offset (r) in the frame where the match of input operands for that instruction will take place; one or more displacements (dests) that define the destination instructions that will receive the result token(s); and input port (left/right) indicator that specifies the appropriate input arc for a destination instruction. Consider Figure 1 for illustration.

When a token arrives at a node (e.g., ADD), the IP part of the tag points to the instruction that contains an offset r as well as displacement(s) for the destination instruction(s). The actual matching process is achieved by checking the disposition of the slot in the Frame memory pointed to by FP+r. If the slot is empty, the value of the token is written in the slot and its presence bit is set to indicate that the slot is full. If the slot is already full (indicating a match), the value is extracted, leaving the slot empty, and the corresponding instruction is executed. The result token(s) generated from the

---

[1] It is often believed that dataflow means parallel execution. 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.

operation is communicated to the destination instruction(s) by updating the IP according to the displacement(s) encoded in the instruction (e.g., execution of the ADD operation produces two result tokens <FP.IP+1, 3.55> and <FP.IP+2, 3.55>$_L$). Instruction execution in ETS is asynchronous since an instruction is enabled immediately upon the arrival of the input operands. This token driven execution necessitates two cycles through the pipeline, per (dyadic) instruction. In our model, we schedule instructions synchronously, requiring only one cycle per instruction.

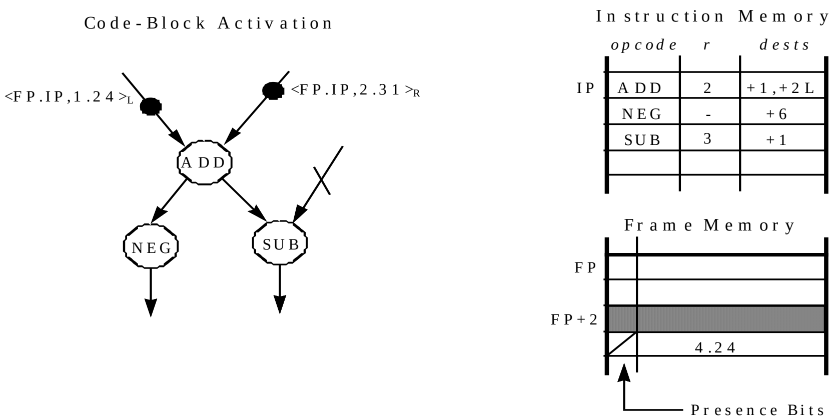

Figure 1: ETS representation of a dataflow program execution.

3 The Scheduled Dataflow Processor

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). Remember that EP executes computations of a thread using only registers.

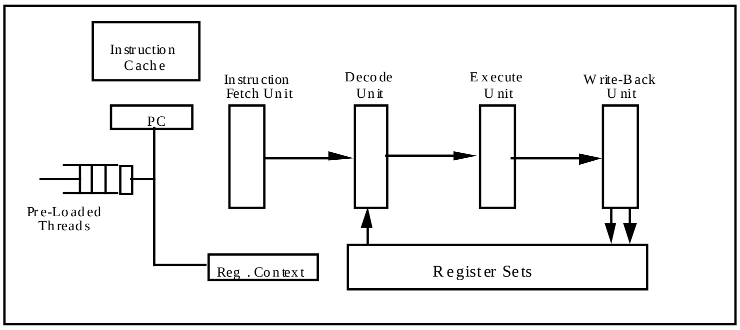

Figure 2: General Organization of Execution Pipeline (EP).

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. 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 registers that contains (up to) 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 very much 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 memory access primary pipeline of the Synchronization Processor (SP).

[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.

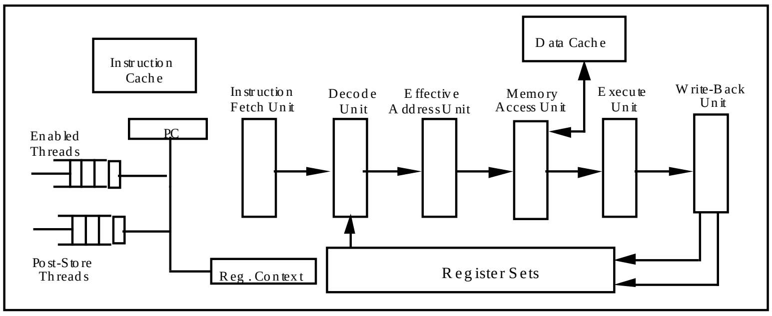

Figure 3: The Memory Access Pipeline.

Here we process 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[3] of threads, using a frame-pointer (FP) and an offset into the frame; both of which 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. Finally, Write-Back unit completes LOAD (pre-load).

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 decremented.

An enabled thread (when the Synch Count becomes zero) 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 after finishing 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 switches and thread creations. 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

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

the observations made in Sparcle [Agarwal 93] that a 4-cycle context switch can be implemented in hardware. Figure 4 shows a more complete view of the SP.

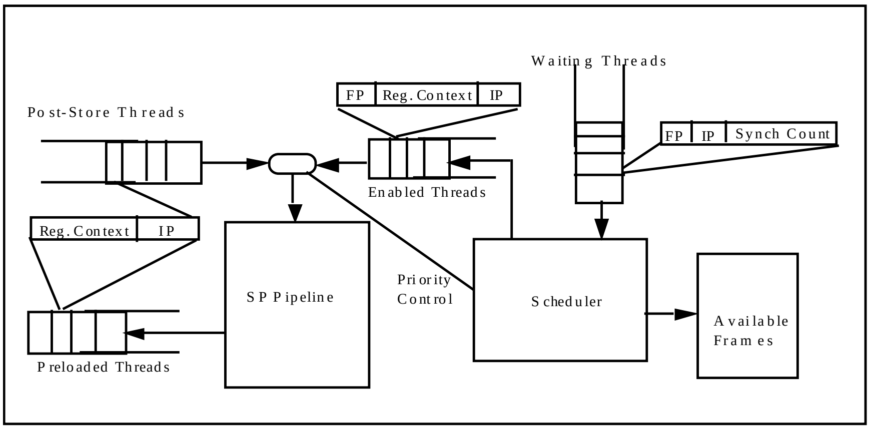

Figure 4: Overall Organization of the SP

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 SPs and preloaded threads on multiple EPs in superscalar implementations of our architecture. We are currently developing the superscalar implementation of SDF. Note that the scheduling is at thread level in our system, rather than at instruction level as done in other multithreaded systems (e.g., Tera, SMT), and thus requires simpler hardware.

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 a 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 IP 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 . At this time a Register Context and an IP identify the thread. The IP points to the first post-store instruction.

3.3 Instruction set architecture

The latest version of SDF instruction set can be found in [Giorgi 99]. Our instructions (Figure 5) are very similar to those of ETS model [Papadopoulos 90, 91]. The difference lies in the specification of destinations for the result generated by an instruction. In ETS, the destinations refer to the destination instructions (and the instruction specifies a memory location where operands for that instruction are matched). In scheduled dataflow the destinations refer directly to the operand locations (one of a pair of source registers) of the destination instructions. This change eliminates the need for fetching an instruction twice for dyadic instructions as in ETS. This change also permits the detection of RAW data dependencies among instructions in the Execution Pipeline and the application of data forwarding scheme in pipelines for sending results directly to the successor instructions. The result forwarding is not applicable in ETS since instructions are token-driven (execute asynchronously), and an instruction is not allowed to enter the execution pipeline until both operands were generated and written into the operand memory.

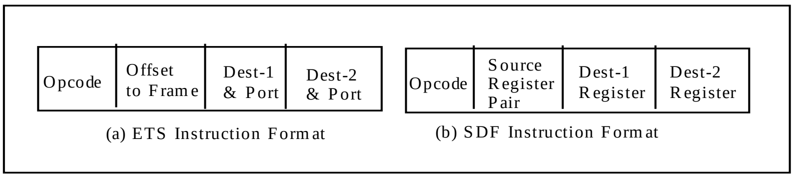

Figure 5: Instruction Format

3.4 Programming Example.

To describe the "scheduling" of instructions in our architecture, we show how our SDF code for Figure 1 may look like. We will view each frame memory location used for matching tokens in ETS as a pair of registers -- a pair consists of even-odd registers. For example, RR6 refers to registers R6 and R7 within a specified thread context. The two source operands destined for a SDF instruction are stored in the pair of registers assigned to that instruction -- data is stored in either the left or right half of a register pair by a predecessor instruction. Unlike in ETS, in our architecture, an instruction is not scheduled for execution immediately when the operands are matched. Instead, operands are saved in the register-pair associated with the instruction and the enabled instruction is scheduled for execution at a later time.

```
ADD     RR2, R4, R6
NEG     RR4, R9, R12
SUB     RR6  R14, R17
```

Assuming that registers R2 and R3 contain the source operands for ADD, when scheduled, this instruction adds the contents of these two registers and stores the result in R4 and R6. Register R4 is one (only one) of the source operands for NEG instruction. Likewise the operands for SUB are stored in the pair R6, R7. Registers R9, R12, R14, R17 indicate the destinations for the results generated by NEG and SUB instructions which are not shown in Figure 1. Note that these instructions still retain the functional nature of dataflow -- data flows from instruction to instruction and there are no write-after-read (WAR or conceptually equivalent anti-) and write-after-write (WAW or equivalent output-) dependencies. Our deviation is from token driven models of previous dataflow implementations. We use "instruction driven" paradigm by scheduling instructions.

The code shown above is for the Execution Pipeline (EP). The Synchronization Processor (SP) is responsible for scheduling enabled threads on EP, pre-loading thread's 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.

To illustrate the preload concept, consider the code segment of Figure 1 and the SDF code shown previously. Assume that the code block of Figure 1(viewed as a thread) receives the two inputs for ADD from other threads. Each thread will be associated with a frame and the inputs to the thread are saved in the frame until the thread is enabled for execution (based on its synchronization count, as described later). When enabled, a register context is allocated to the thread and the input data for the thread from its frame memory is "preloaded" into its registers.

```
LOAD        RFP| 2, R2
LOAD        RFP| 3, R3
LOAD        RF P| 4, R32
LOAD        RFP | 5, R35
```

Assuming that the inputs for the thread (or ADD instruction) are stored in its frame (RFP) at offsets 2 and 3, the first tow LOAD instructions preload te thread with required data. Consider that the result generated by SUB in our code example (in R17) is needed by some other thread. The last two LOAD instructions save the frame pointer and offset for returning the results when the thread completes its execution.

```
STORE       R17, R32|R35
```

This instruction transfers (or post-stores) the result of the current thread (i.e., from SUB, in R17) to a frame pointed to by R32 at a frame-offset contained in R35.

3.5 Code partitioning

The SDF assembly code is the product of our compiler (SDFC). The compiler takes care of partitioning the high-level source code in order to create the SDF threads. Each thread consist of three portions: pre-load code, execute code and post-store-code [Fig. 6].

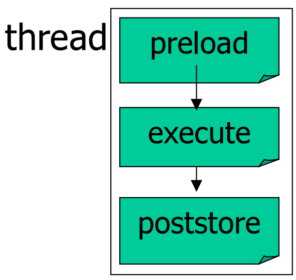

Figure 6: The three code portions of an SDF thread.

When a thread is created (using FALLOC), a frame allocated for storing the inputs of the thread. An instruction pointer (IP) indicating the first executable instruction of the thread and a synchronization count indicating the number of inputs needed before the thread becomes enabled for execution are stored in the allocated frame. Once a thread receives all the necessary inputs, the thread is allocated a register context. The pre-load code then moves the data from a thread's frame memory into its registers. The execute portion will perform computations using only the registers, while the post-store code will store the thread's results in other threads' frames.

4 Evaluation of the Decoupled Scheduled Dataflow Architecture

Previously, we relied on analytical models and Monte Carlo simulations to compare the proposed architecture with Rhamma, ETS, EARTH and conventional RISC processors [Kavi 99a-b]. In this work we evaluate our architecture based on the execution of complete programs. We developed an instruction level simulator for Scheduled Dataflow architecture. At present the simulator assumes a perfect cache. However, we will show the cache behavior of our architecture by using address traces from our simulator. Concurrent with the simulator, we have also developed a backend to a Sisal [Bohm 91], and used MIDC as intermediate language [Shankar 95, 96] to generate code for our architecture[4].

Using the simulator we were able to compare the performance of the Scheduled Dataflow system with a single threaded RISC architecture. We also investigated the effect of parallelism (i.e., number of enabled threads), thread granularity (average run-lengths of the execution threads on EP) on the performance of our architecture. Using address traces from our simulator, we investigated the expected cache behavior

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 [Hennessy 96]. The programs used for this comparison include a Matrix Multiply, Livermore Kernel 5, Fibonacci and a code segment for picture zooming application [Terada 99]. We used generated code for complete programs. We used dlxcc to generate DLX code in our comparisons. For both DLX and SDF we used a degree of 5 loop unrolling for Matrix Multiply, Livermore Loop 5 and Zoom. Since DLX is single threaded, only one thread was used for all programs. We used 5 threads for Scheduled Dataflow when executing Matrix Multiply, Livermore Loop 5 and Zoom programs. The results are shown in Table 1.

---

[4] At this time our backend only generates partial code. We extend this with hand-coding to generate complete programs for SDF simulator. The backend does not perform any optimization (and produces rather poor code).

	Matrix Multply				**Livermore 5**		
N	DLX	SDF	Speed	Loop=N	DLX	SDF	Speed
	Cycles	Cycles	UP		Cycles	Cycles	Up
25*25	966090	306702	3.150	50	87359	56859	1.536
50*50	7273390	2159780	3.368	100	354659	215579	1.645
75*75	24464440	6976908	3.506	150	801959	476299	1.684
100*100	57891740	16175586	3.579	200	1429259	839019	1.703
				250	2236559	1303739	1.715
				300	3223859	1870459	1.724
				350	4391159	2911789	1.508
				400	5738459	3309899	1.734
				450	7265759	4182619	1.737
	Fibonacci				**Zoom**		
N	DLX	SDF	Speed	N	DLX	SDF	Speed
	Cycles	Cycles	UP		Cycles	Cycles	UP
5	615	842	0.7304	5,5,4	10175	9661	1.0532
10	7014	10035	0.699	10,10,4	40510	37421	1.0825
15	77956	111909	0.6966	15,15,4	97945	83331	1.1754
20	864717	1241716	0.6964	20,20,4	161580	147391	1.0963
25	9590030	13771467	0.6964	25,25,4	271175	229601	1.1811
30	1.06E+08	1.53E+08	0.6964	30,30,4	391150	329961	1.1854
				35,35,4	532285	448471	1.1869
				40,40,4	645520	585131	1.1032

Table 1: Execution Behavior Of Scheduled Dataflow

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 that 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). Livermore loop exhibits less parallelism than Matrix Multiply due to a loop carried dependency. Zoom exhibits moderate parallelism, but contains a significant sequential fraction degrading the parallelism (as per Amdhal's law). Fibonacci contains no thread-level parallellims and hence our architecture under-performs DLX. 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 for 5, 4, 3, 2 and one 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).

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 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). 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 7. 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 for other data sizes with Matrix Multiply and for the other benchmarks, Zoom and Livermore Loop 5.

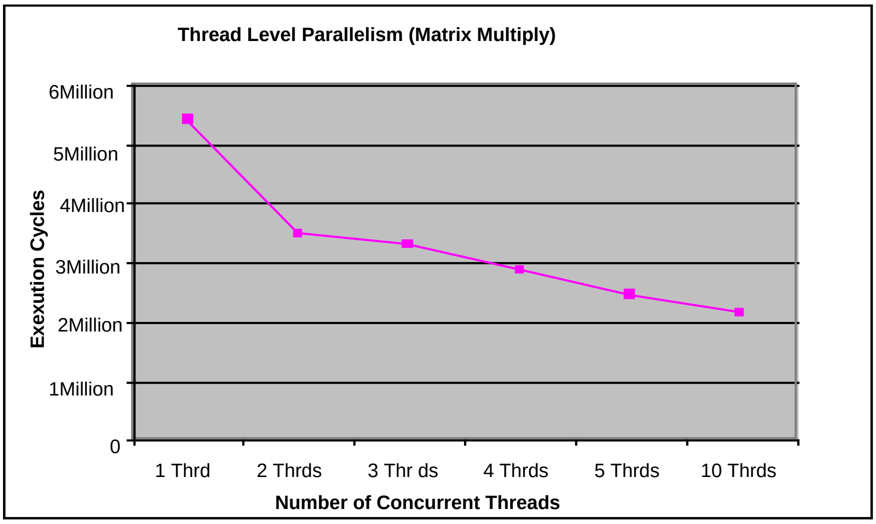

Figure 7: Effect Of Thread Level Parallelism On SDF Execution (Matrix Multiply)

4.3 Effect Of Thread Granularity On Execution Behavior

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 8 is 50*50. 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 SP 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. The number of registers per thread context (currently 32 pairs) is also 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.

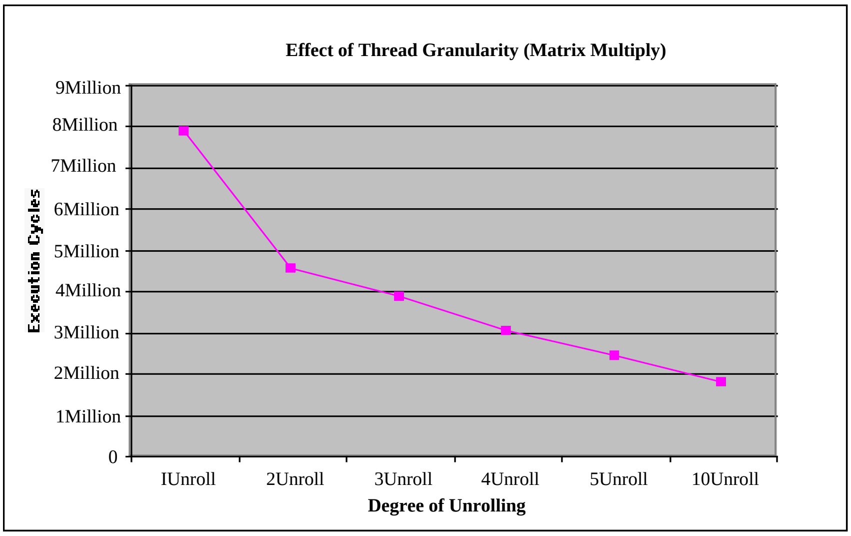

Figure 8: Effect Of Thread Granularity On SDF Execution (Matrix Multiply)

4.4 Utilization of the two processing units.

It may be natural to wonder if there exists a workload imbalance between the two separate processing units (EP and SP) in our system. We collected the utilization of EP and SP for a variety of benchmarks and input data sizes. Figure 9 shows the average utilization rates for 4 benchmarks (the benchmark LGR is a program used heavily for testing the correctness of the code generated by our Sisal compiler; it consists of a variety of loops, conditional statements and case statements). As can be seen, at least in our current environment, both EP and SP handle reasonably balanced workloads. This is not conclusive data since at present we assume a perfect cache. In the near future, we will collect utilization rates for EP and SP using a realistic cache.

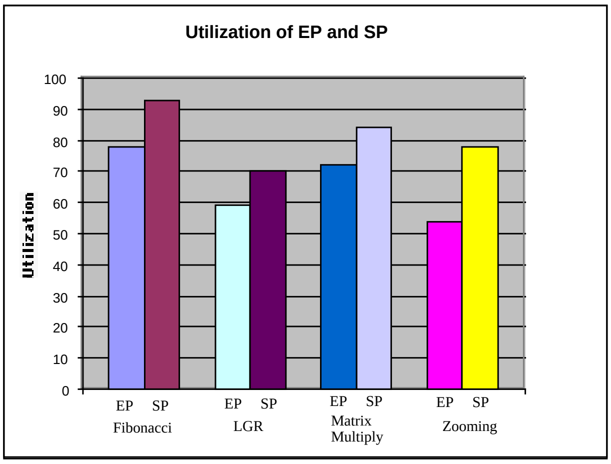

Figure 9: Average Utilization Of The Two Hardware Units (SP and EP)

4.5 Cache Behavior of Scheduled Dataflow.

At present our instruction set simulator does not include cache memories. In this section, we compare the expected cache behavior of SDF programs with that of DLX. For this purpose we generated address traces on both systems and used Dinero-IV [Edler99] to generate cache behaviors.

In Table 2, we used 5 SDF threads for Matrix Multiply, and Livermore Loop 5 programs, but a single thread for Fibonacci program. The cache behavior for SDF programs is very comparable to that of DLX programs. For the data in Table 2, we used direct mapped cache with 256K bytes and a block size of 64 bytes. SDF cache behavior is similar to cache memories in conventional systems when the cache parameters (like associativity, block size, and cache size) are changed. The best cache behavior is observed when the block size equals the frame size. We feel that cache pre-fetching is more effective in our architecture; since the input data from a thread's frame is pre-loaded into the thread's context, the frame can be pre-fetched. Once preloaded, the frame is freed and can be allocated for the next thread to be created.

Matrix Multiply						
N	DLX Refs	DLX Misses	DLX Miss Rate	SDF Refs	SDF Misses	SDF Miss Rate
25	286714	61	0.0002	156470	382	0.0024
50	1627928	237	0.00015	1094360	614	0.0006
75	5462503	530	0.000097	3526250	1010	0.0003
100	12910828	940	0.000073	8164640	1558	0.0002

Livermore						
Loop=N	DLX Refs	DLX Misses	DLX Miss Rate	SDF Refs	SDF Misses	SDF Miss Rate
50	22429	8	0.00036	24177	22	0.00091
100	88829	13	0.000146	91913	31	0.000337
150	199229	17	8.53E-05	203249	40	0.000197
200	353629	22	6.22E-05	358185	49	0.000137
250	550458	27	4.90E-05	556721	58	0.000104
300	794429	32	4.03E-05	798857	67	8.39E-05
350	1080829	36	3.33E-05	1084593	76	7.01E-05
400	1411229	41	2.91E-05	1413929	88	6.22E-05
450	1785629	46	2.58E-05	1786865	97	5.43E-05

Fibonacci						
N	DLX Refs	DLX Misses	DLX Miss Rate	SDF Refs	SDF Misses	SDF Miss Rate
5	260	5	0.019	134	8	0.06
10	3014	10	0.003	1702	13	0.008
15	33546	14	4E-04	19076	18	9E-04
20	372152	18	5E-05	211758	23	1E-04
25	4127350	23	6E-05	2E+06	28	1E-05
30	40975448	27	7E-07	3E+07	33	1E-06

Table 2: Cache Behavior

4.6 Separate Data And I-Structure Caches.

In our architecture, I-structure elements and Frames are mapped to different areas of memory. Using a single cache for both the frame data (data cache) and the I-structures (arrays) cause more conflict misses. Following our previous work [Kavi 95, 98b] where we have shown the benefits of using separate I-structure cache for ETS, here we investigated the use of a separate I-structure cache. The data is shown in Table 3.

N	SDF Refs	Unified Misses	Matrix Multiply Unified Miss Rate	I-Struct Misses	I-Struc Miss Rate	Frame Misses
25	156470	382	0.0024	81	0.0005	11
50	1094360	614	0.0006	319	0.0003	11
75	3526250	1010	0.0003	3132	0.0009	11
100	8164640	1558	0.0002	6215	0.0008	11

N	SDF Refs	Unified Misses	Livermore Loops Unified Miss Rate	I-Struct Misses	I-Struc Miss Rate	Frame Misses
50	24177	22	0.00091	12	0.0005	10
100	91913	31	0.000337	21	0.0002	10
150	203249	40	0.00097	30	0.0001	10
200	358185	49	0.000137	39	0.0001	10
250	556721	58	0.000104	48	8.62E-05	10
300	798857	67	8.39E-05	57	7.14E-05	10
350	1084593	76	7.01E-03	66	6.09E-05	10
400	1413929	88	6.22E-05	78	5.52E-05	10
450	1786865	97	5.43E-05	87	4.87E-05	10

Table 3: Effect Of A Separate I-Structure Cache

The table shows data only for Matrix Multiply and Livermore Loop 5 (both with 5 threads). For unified case, we used a single 256K cache (64byte blocks); for split case, we used 128K I-structure cache and 128K frame cache. The I-structure miss behavior is somewhat erratic for Matrix Multiply, because the program accesses rows of one matrix and columns of another matrix, and the strides have caused more conflict misses for certain data sizes. It should be noted that data cache (used for thread frames) encounters no conflict misses. This behavior can be attributed to our "stack" of frames allocation described previously. The misses indicate the maximum number of frames needed by the program during its execution. Reusing recently freed frames eliminates cache misses. As previously mentioned, for 5 threads, Matrix Multiply requires a maximum of 11 frames (while Livermore needs 10 frames), and these frames could be eliminated by allocating register sets to threads on creation.

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. However, we investigated the expected cache behavior of SDF program by collected address traces and using a trace-driven cache simulator (Dinero-IV). The results indicate that SDF programs incur no more (often fewer) cache misses than a traditional RISC processor. Using separate caches for I-structure memory and frame memories further reduces the number of cache misses encountered by SDF programs.

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 (all other data is provided through I-structure memory). The use of frame memories for thread data permits a clean decoupling of memory accesses into pre-loads and post-stores. This can 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, we strongly believe that the use of dataflow instructions reduces the complexity of the processor by eliminating the need for complex logic (e.g., scoreboard or Tomasulo's reservation stations [Hennessy 96]) needed for resolving data dependencies, register renaming, out-of-order instruction scheduling and branch predictions. The silicon area saved may be used to include more register-sets and registers per set to improve thread level parallelism and thread granularities. Moreover, our current instruction set and the compiler are not optimized. We are working to improve both the instruction set and the compiler to produce more efficient executions of programs. We will soon develop quantitative comparisons of our architecture with conventional scalar and superscalar architectures for a wider range of benchmark programs (including SPEC-2000 programs).

Acknowledgments

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

References

[Agarwal 93] Agarwal, et. Al.: "Sparcle: An evolutionary processor design for multiprocessors", *IEEE Micro*, pp 48-61, June 1993.

[Bohm 91] A. D. W. Bohm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft: "SISAL Reference Manual: language version 2.0", Tech, Report CS91-118, Computer Science Dept., Colorado State University.

[Butler 91] 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.

[Cuppu 99] V. Cuppu, B. Jacob, B. Davis and T. Mudge: "A performance comparison of contemporary DRAM architectures", *Proc of the Intl. Symposium on Computer Architecture (ISCA-26)*, pp 222-233, May 1999.

[Edler 99] Jan Edler and Mark D. Hill: "DineroIV Trace-Driven Uniprocessor Cache Simulator", http://www.cs.wisc.edu/~markhill/DineroIV

[Giorgi 99] R. Giorgi, K. M. Kavi and H.Y. Kim: "Scheduled Dataflow Instruction Manual", Dept. of Electrical and Computer Engineering, UAH. http://crash1.eb.uah.edu/~kavi /Research/sda.pdf

[Govindarajan 95] 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.

[Grunewald 97] W. Grunewald, T. Ungerer: "A Multithreaded Processor Design for Distributed Shared Memory System," *Proc. Int'l Conf. on Advances in Parallel and Distributed Computing*, 1997.

[Hennessy 96] J.L. Hennessy, and D.A. Patterson: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publisher, 1996.

[Hum 95]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.

[Kavi 99a] K.M. Kavi. H.S. Kim and A.R. Hurson: "Scheduled dataflow architecture: A synchronous execution paradigm for dataflow", *IASTED Journal of Computers and Applications*. Vol. 21, No. 3 (Oct. 1999), pp 114-124.

[Kavi 99b] K.M. Kavi, H.-S. Kim, J. Arul and A.R. Hurson: "A decoupled scheduled dataflow multithreaded architecture", *Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN-99),* Fremantle, Western Australia, June 23-25, 1999, pp 138-143.

[Kavi 98a] K.M. Kavi, B. Lee and A.R. Hurson: "Multithreaded systems: A survey", *Advances in Computers,* Academic Press, 1998, pp 287-327.

[Kavi 98b] K.M. Kavi and A.R. Hurson: "Design of cache memories in dataflow architectures", *Euromicro Journal on Systems Architecture.* Vol. 44, No. 9-10, June 1998, pp 657-674.

[Kavi 95] K.M. Kavi, et. al. "Design of cache memories for multi-threaded dataflow architecture", *Proceedings of the 22nd Intl. Symp. on Computer Architecture* (ISCA-22), June 1995, St. Margherita Ligure, Italy, pp. 253-264.

[Krishnan 99] V. Krishnan and J. Torrellas: "A chip-multiprocessor architecture with speculative multithreading", IEEE Trans. on Computers, Sept. 1999, pp.866-880.

[Lam 92] 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.

[Papadopoulos 91] G.M. Papadopoulos and K.R. Traub: (1991). "Multithreading: A Revisionist View of Dataflow Architectures," *Proceedings of the 18th International Symposium on Computer Architecture (ISCA-18)*, pp. 342-351.

[Papadopoulos 90] 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.

[Sakai 93] 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.

[Shankar 96] B. Shankar and L. Roh: "MIDC Language manual", Tech Report, CS Dept., Colorado State University, July 1996, http://www.cs.colostate.edu/~dataflow/papers/ Manuals/manual.pdf.

[Shankar 95] B. Shankar, L. Roh, W. Bohm and W. Najjar: "Control of parallelism in multithreaded code", *Proc of the Intl Conference on Parallel Architectures and Compiler Techniques (PACT-95*), June 1995. http://www.cs.colostate.edu /~dataflow/papers/pact95b.pdf.

[Smith 82] Smith, J.E: “Decoupled Access/Execute Computer Architectures*”, Proc of the 9th Annual Symp on Computer Architecture*, May 1982, pp 112-119.

[Takesue 87] M. Takesue: "A unified resource management and execution control mechanism for Dataflow Machines". *Proc. 14th Annl. Intl. Symp. on Computer Architecture(ISCA-14)*, June 1987, pp. 90-97.

[Terada 99] 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

[Thekkath 94] R. Thekkath and S.J. Eggers: ``The effectiveness of multiple hardware contexts," *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.328--337, October 1994.

[Thoreson 87] S.A. Thoreson and A.N. Long; "A Feasibility study of a Memory Hierarchy in Data Flow Environment". *Proc. Intl. Conference on Parallel Conference*, June 1987, pp. 356-360.

[Tokoro 83] M. Tokoro, J.R. Jagannathan and H. Sunahara: "On the working set concept for data-flow machines", *Proc. 10th Annul. Intl. Symp. on Computer Architecture (ISCA-10)*, July 1983, pp. 90-97.

[Tsai 99] J. Y. Tsai, J. Huang, C. Amlo, D. Lilja, and P. C. Yew: “The Superthreaded processor architecture”, IEEE Trans. on Computers, Sept. 1999, pp. 881-902.

[Wall 91] 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.