SCHEDULED DATAFLOW ARCHITECTURE: A SYNCHRONOUS EXECUTION PARADIGM FOR DATAFLOW[†]

Krishna M. Kavi Hyong-Shik Kim
Department of Electrical and Computer Engineering
University of Alabama in Huntsville
Huntsville, AL 35899

Ali R. Hurson
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802

Abstract

Recent trends in technology are widening the performance gap between memory and processors. Multithreading has been touted as a possible solution to minimize the loss of CPU cycles due to memory latency, by executing several instruction streams simultaneously. In this paper, we propose a new multithreaded dataflow architecture that uses RISC like pipelines and control-flow like scheduling of dataflow instructions, but retains functional properties of the dataflow model. In addition, our Scheduled Dataflow architecture utilizes two separate hardware units for the execution of threads – decoupling memory accesses from pipeline execution. We present performance data obtained using queuing analyses of the proposed architecture. Our analysis investigated the impact of fine-grained vs. coarse-grained threads, number of hardware contexts, and decoupling memory accesses from pipeline execution in our architecture.

Keywords: Dataflow architecture, multithreading, Explicit Token Store, multiple hardware contexts

[†]This research is funded in part by NSF grants, MIP-9796310 and MIP-9622836.

1 Introduction

Even though the dataflow model and architectures 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, the actual implementations of the model have failed to deliver the promised performance. Several features of the dataflow computational model, however, have found their place in modern processor architectures and compiler technology (e.g., SSA, register renaming, dynamic scheduling and out-of-order instructions execution, I-structure like synchronization, non-blocking threads). Most modern processors utilize complex hardware to bring the execution engine closer to an idealized dataflow engine (for detecting data hazards and renaming of registers, for instruction reordering and issuing multiple instructions, branch prediction and predicated branches). It is our contention that such complexities can be eliminated if a more suitable implementation of the dataflow model can be discovered. We feel that the primary limitations of the pure dataflow model that prevented commercially viable implementations are:

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

Many researchers have addressed the first two limitations of dataflow architectures[1, 2, 3, 4, 5]. The benefits of cache memories within the context of Explicit Token Store (ETS) dataflow paradigm were presented in [2]. There have been several research projects that demonstrated how coarser grained threads can be utilized within the dataflow execution model. In this paper, we propose a new dataflow architecture that addresses the third limitation. In this new model, we will deviate from the asynchronous triggering of dataflow instructions (implied by token-driven systems), and schedule instructions for synchronous execution. 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., [6], [7], [8]). In such systems, the execution of instructions within a thread do not retain the functional properties of dataflow, and introduce side-effects, WAW (or output) and WAR (or anti) dependencies. Not preserving dataflow properties at instruction level requires complex hardware for the detection of data dependencies and dynamic scheduling of instructions. In our system, the instructions within a thread still retain dataflow properties, and thus eliminate the need for complex hardware.

1.1 Overview of Dataflow And Explicit Token Store Architecture

In the dynamic dataflow implementations, the results generated by an instruction are "tagged" by its destination instruction's address. The tagged data (known as tokens) circulate in the system awaiting their matches. Tokens destined for the same instruction will have identical tags; tokens destined to different activations (e.g., different iterations) of the same instruction will have different tags. In a *direct matching* scheme (used in ETS[9]), storage (called an *activation frame*) is allocated for all the tokens needed by the instructions of each iteration of a code block. A code block can be viewed as a sequence of instructions comprising a loop body or a function. A computation is completely described by a pointer to an instruction (IP) and a pointer to an activation frame (FP). The pair of pointers, <FP, IP>, is called a *continuation* and corresponds to the *tag* part of a token. A typical instruction pointed to by an IP specifies an *offset*(R) in the activation frame where the match of input operands for that instruction will take place. When a new token arrives, the IP part of the tag is used to obtain the offset(R). Examining the operand location in the activation frame at FP+R, it can be discovered if a match is made or if the new token should wait for its match. A hardware implementation of the ETS architecture is shown in Figure 1. For a more detailed description of ETS, see [2] or [9].

The following describes the functionality of the various pipeline stages in Figure 1.

1. Instruction Fetch: The incoming token's instruction pointer(IP) is used to access an instruction in the local instruction cache. For unary (one-operand) instructions, the next two stages are skipped and the instruction is executed in ALU.
2. Address Decode: The effective address(FP+R) of the operand memory location is computed (the offset R is obtained from the instruction and FP from tag of the token).
3. Operand Fetch: The presence bit in the operand cache location is examined. If the bit is reset the data value is stored; otherwise a match occurs leading to a read of the previously stored value.
4. ALU: On a match the ALU executes the opcode, with the value retrieved from the operand memory and the value contained in the token. One or two tags are computed by adding the destination offsets to the instruction pointer.
5. Token Form: The result value and destination tags are packaged into tokens and the tokens are written to the token queue.

1.2 Limitations of ETS Architecture

While ETS architecture permits the use of memory hierarchies (for activation frames), the execution of instructions are asynchronous: an instruction is enabled (only and immediately) when its operands are generated by predecessor instructions. A two operand instruction

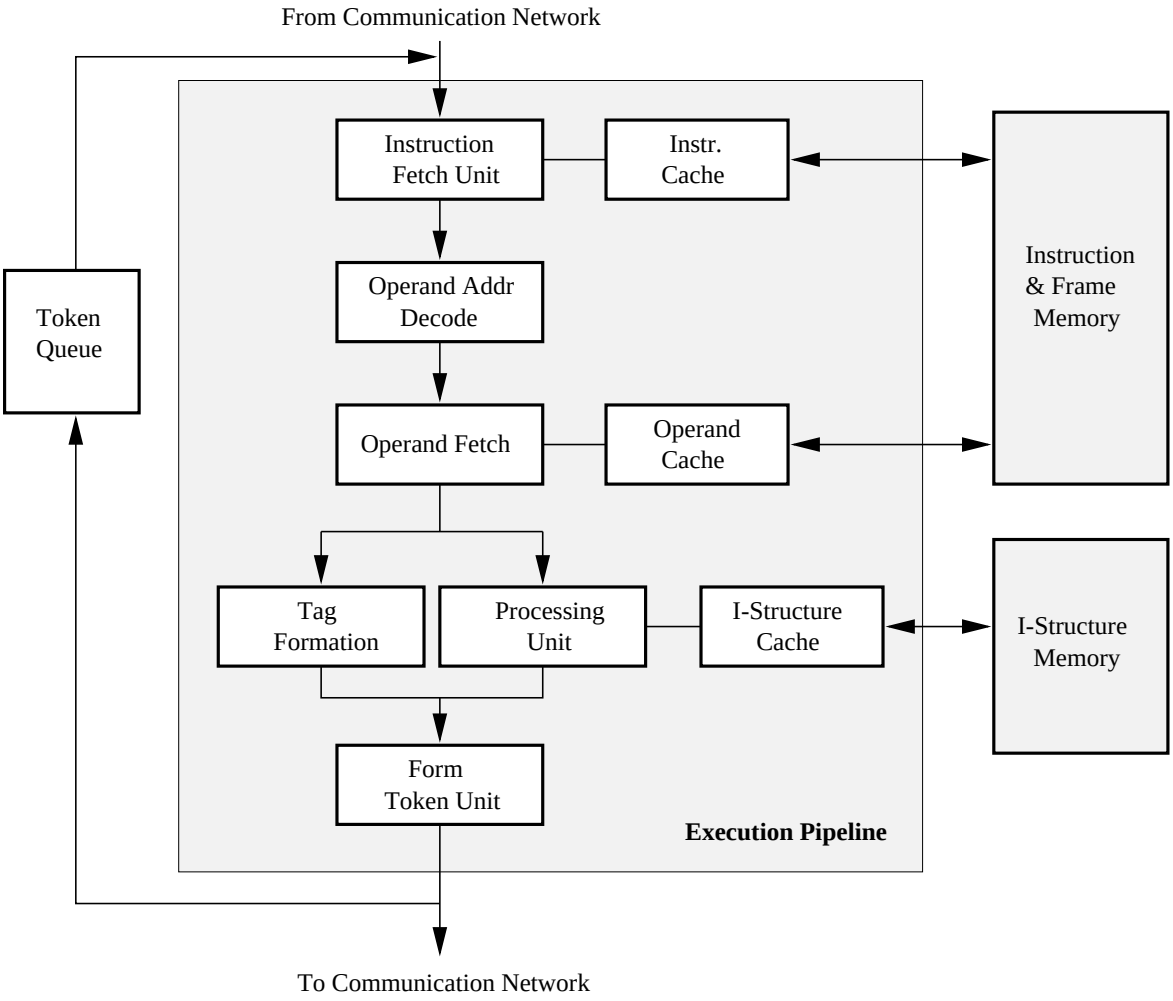


Figure 1: An organization of a pure-dataflow processing element

requires two separate cycles through the pipeline, corresponding to the two tokens containing the operands. Even if registers are used for matching operands, conventional techniques such as result-forwarding (where the results of an instruction are directly supplied to a dependent instruction) cannot be incorporated into the ETS pipeline.

1.3 Hybrid Architectures

Hybrid dataflow/control flow organizations have been proposed by several researchers ([6], [10], [8]). In most of these systems, coarse-grained threads represent macro dataflow nodes while each thread includes conventional load/store instructions. In one such proposed system, EARTH[10], two processors are used for the execution of macro dataflow threads. One processor, Execution Unit (EU) behaves like a traditional RISC processor executing instruc-

tions belonging to a thread. The second processor, Synchronization Unit (SU) is responsible for scheduling of threads on EU, remote memory accesses and thread synchronization. Earth uses non-blocking threads. *It should be noted that EU executes all instructions within a thread, including load/store.* This is one of the differences between our Scheduled Dataflow architecture and EARTH. A more fundamental difference lies in the instructions of a thread – instructions in our architecture retain functional properties of dataflow while those of EARTH are control-flow instructions.

2 Scheduled Dataflow Architecture

Techniques for increasing locality in dataflow have relied on defining an (expected) order in which instructions are enabled ([2], [4], [5]). We feel that it should be possible to define an architecture that executes instructions in the prescribed order instead of executing them as soon as the data is available. Compile time analysis on the source program can be used to define an expected order in which instructions may be executed, even if the data is already available for these instructions. We will call such a system as *Scheduled Dataflow*. The processor for our architecture can be designed with the pipeline stages as depicted in Figure 2 (The figure does not show all the data paths for the Synchronization Processor).

The various stages of Execution Pipeline (EP) are outlined below.

1. Instruction Fetch: The instruction fetch behaves like traditional fetch, relying on a program counter to fetch the next instruction. The Context register along with the PC can be viewed as a part of the thread id: <FP, IP>.
2. Operand Fetch: This unit fetches a double word from a register file that contains the two operands for the instruction. Each instruction specifies an offset(R) that refers to a pair of registers where its operands are stored by its predecessor instructions (see Section 2.1 for more details on the instruction formats). Thus, a read port should supply a double word.
3. Execute: The execute executes the instruction and sends the results to write-back unit along with the destination addresses (identifying the registers for the destination instructions).
4. Write-back: This unit writes up to two values to the register file; the two values may go to two different locations in the register file. This necessitates 2 write ports to the register file.

Other units of Figure 2 will be explained later. Figure 3 shows the organization of the operand registers. Note that unlike ETS, the operand locations should permit the saving of both tokens belonging to an instruction waiting to be scheduled, since instructions may not be

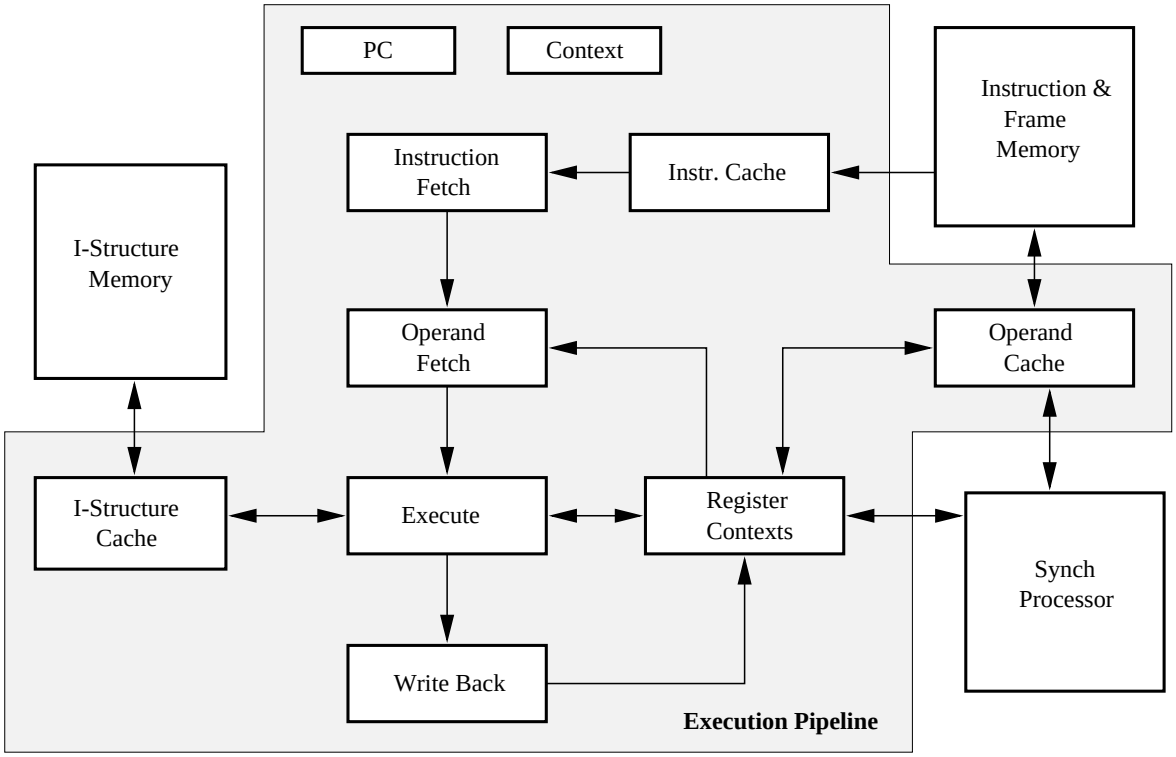


Figure 2: General organization of Scheduled Dataflow architecture

scheduled for execution immediately upon the arrival of the second token. The presence bits associated with operand locations are used only to catch exceptions from improper scheduling of instructions (i.e., attempt to execute an instruction before the availability of operands).

As can be seen, this execution pipeline described above, behaves very much like conventional RISC pipelines while retaining the primary dataflow properties; functional nature, side-effect freedom, and non-blocking threads. The functional and side-effect free nature of dataflow eliminates the need for complex hardware (e.g., scoreboard or reservation stations of Tomasulo's method) for detecting write-after-read (WAR) and write-after-write (WAW) dependencies and register renaming. The non-blocking nature of our thread model and the use of a separate processor (Synchronization Processor, SP) for thread synchronization and memory accesses (see subsections 2.2 and 2.3 for more details) eliminate unnecessary thread context switches on long latency operations or cache misses. Our architecture does not prevent superscalar or multiple instruction issue implementations for the Execution Pipeline (EP). *However, unlike modern RISC processors, our architecture eliminates the need for instruction reordering at execution time; the execution order is completely determined at compile time.*

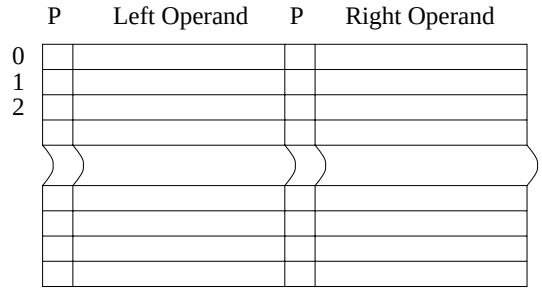


Figure 3: Operand registers in Scheduled Dataflow architecture

Op Code	Offset(R)	Dest-Instr-1 and Port	Dest-Instr-2 and Port

(a) ETS Instruction Format

Op Code	Offset(R)	Dest-Data-1 and Port	Dest-Data-2 and Port

(b) Scheduled Dataflow Instruction Format

Figure 4: Instruction Formats

2.1 Instruction Formats

Relative to the ETS model[9], the instruction format for the proposed Scheduled Dataflow architecture requires only minor changes. The difference lies in the specification of destinations for the results generated by an instruction (Figure 4). In ETS, the destinations refer to the destination instructions (i.e., IP values); in Scheduled Dataflow the destinations refer to the operand locations of the destination instructions (i.e., offset value into activation frames or register contexts). This change also permits the detection of RAW data dependencies among instruction in the execution pipeline and the use of *result forwarding* so that results from an instruction can be sent directly to dependent instructions. The result forwarding is not applicable in ETS dataflow since instructions are token driven, implying that an instruction is not allowed to enter the execution pipeline until the operands were generated and written into the operand memory. *It should be remembered that each operand memory location (or register) of Scheduled Dataflow consists of a pair of values. Thus, the offset(R) of an instruction refers to a pair of registers containing the two source operands of that (binary) instruction.*

2.2 Separate Synchronization Processor

Using multiple hardware units for the coordination and execution of instructions is not new. Some of the earlier designs attempted to use a separate hardware unit for accessing memory, decoupling the memory accesses from pipeline execution[11]. More recently, separate hardware units have been proposed to handle the synchronization among threads in multithreaded architectures (e.g., Alewife[12], StartT-NG[13], EARTH[10], PL/PS[14]). We follow this tradition and propose two hardware units for the Scheduled Dataflow (see Figure 2 above – although the figure does not show all data paths for Synchronization Processor). One of the hardware units (EP) will be similar to conventional RISC pipelines as described previously. The other hardware unit (SP) is responsible for accessing memory to load the initial operands of enabled threads into registers (i.e., preload) and store the results produced by threads from registers (i.e., post-store); for maintaining synchronization counts for threads and scheduling enabled threads (including allocation of register contexts and placing the enabled thread on the ready queue of the execution unit). Our approach more closely resembles the dual processors of EARTH[10] – with the addition of preload and post-store operations.

2.3 Operand Registers

To fully utilize the multithreading capabilities, we propose multiple register sets. Each thread is allocated a register file (context or activation frame). As mentioned earlier, each register consists of a pair of locations and registers must be designed with two write ports and one double-word read port. Initial values and inputs from other threads are preloaded into a thread's context by the Synchronization Processor (SP). The thread's results will be left in its registers and the Synchronization processor either "post-stores" results in operand cache or uses them towards the synchronization requirements of awaiting threads. This is similar to a non-blocking architecture described in [14], eliminating all data memory (or cache) accesses during the execution of threads by the EP.

2.4 Operand Memory Reuse

In ETS the operand locations used for matching operands of an instruction can be reused for matching operands other instructions with careful analysis of the data dependencies (see for example in [15]). This reuse can lead to smaller activation frames for threads. Since in Scheduled Dataflow, instructions of a thread are executed sequentially, there will be more opportunities to reuse operand registers for more than one instruction. *This in turn can lead to even smaller thread contexts and the possibility of more hardware contexts on chip.*

2.5 Example Code Segment

In order to provide a better insight into the proposed architecture, we present a code segment for the Fibonacci function in Figure 5. C code for the function is shown for reference. The function is translated into two code blocks – one makes the two recursive function calls and the other sums the results from the two forked functions. The non-blocking nature of our architecture requires the creation of a new code-block to sum the results. Each code block includes an a preload thread (to access memory and load initial values into the thread's context), one or more post-store threads, and execution threads. A typical binary operand instruction specifies an op code, a single "double word" source register, and up to two destination registers as shown in Figure 4. A complete description of the instruction set with several program examples can be found in [16].

Figure 5(b) shows the code segment that will be executed by the EP, while Figure 5(c) shows code for SP. The postfixes, "`pre`", "`post`" and "`main`" indicate the preload thread, the post-store thread, and the main thread respectively. The extensions, "`.l`" and "`.r`" are devised to designate either the left or the right half of a "double-word" register. Every thread finishes its execution by either enabling the next thread (on either EP on SP) or deallocating the frame. An enabled thread is scheduled only when its synchronization requirements (including data from other threads) are met.

Since the example code segment has not been written for number-crunching purpose, only five binary operand instructions are found at EP code[1] – `eq`, `gt`, two `sub`'s, and `add`. Their two operands are made ready at a double word single register before being accessed. Even though two destination registers are allowed, only one destination is needed in this example.

The right column shows that SP preloads two pairs of operands from the frame into two double word registers, `R1` and `R2`, at both preload threads, and post-stores values into individual registers instead of double word registers at post-store threads. The second code block has two post-store threads, and only of the two threads will be executed depending on where the control flows (i.e., branch decision).

3 An Analytical Model For Evaluation

There have been many analytical formulations to predict the performance of multithreaded programs on conventional architecture (see for example in [17], [18]). In this paper, we will use a closed-form queuing network model to compare the performance of Scheduled Dataflow with conventional processors, ETS-like dataflow architecture and hybrid systems

[1] The authors believe that numeric application could utilize a double word source registers more effectively than this example.

```
int f(int i)
{
    if (i > 1) return f(i-1)+f(i-2);
    else if (i == 1) return 1;
    else return 0;
}
```

(a) C code

```
0main:   mkval 1 R2.r
         gt R2 R3.l
         brnz R3.l 0main_2
         eq R2 R4.l
         brnz R4.l 0main_1
0main_0: mkval 0 R15.l
         switch.s 0post_0
0main_1: mkval 1 R15.l
         switch.s 0post_0
0main_2: falloc R3.l 0pre 2
         falloc R4.l 0pre 2
         falloc R5.l 1pre 3
         select.l R2 R6.l R7.l
         mkval 1 R6.r
         sub R6 R8.l
         mkval 2 R7.r
         sub R7 R9.l
         mkval 2.l R5.r
         mktag R5 R10.l
         mkval 2.r R5.r
         mktag R5 R11.l
         switch.s 0post_2

1main:   add R2 R15.l
         switch.s 1post
```

(b) EP code

```
0pre:    load R1 Rfp 1
         load R2 Rfp 2
         switch.p 0main

0post_0: store R15.l R1.l 0
         ffree

0post_2: store R10.l R3.l 1.l
         store R8.l R3.l 2.l
         store R11.l R4.l 1.l
         store R9.l R4.l 2.l
         store R1.l R5.l 1.l
         ffree

1pre:    load R1 Rfp 1
         load R2 Rfp 2
         switch.p 1main

1post:   store R15.l R1.l 0
         ffree
```

(c) SP code

Figure 5: Example code segment

utilizing separate processors for thread execution and thread scheduling (e.g., EARTH[10]). It may be instructive to explore the factors that impact our model. Consider the number of cycles needed to execute a program with $N \times R$ instructions, where N is the number of threads and R is the average number of instructions per thread (i.e., run-length). In a conventional architecture (assuming no instruction reordering or multiple issue), the number of cycles needed to execute the program depends on the number of instructions per thread, the average CPI that accounts for cache misses (ignoring other pipeline stalls), context switching overhead to switch between threads, and the number of threads. Thus, in addition to N and R, the factors that influence the performance include f_m (fraction of instructions which are memory reference instructions), T_m (average memory access time including cache misses) and T_c (context switch time). In ETS, two operand instructions require two cycles per instructions and one operand instruction requires one cycle through the pipeline. In addition, since the operands (for two operand instructions) must be stored in the operand cache, we need to account for delays due to cache misses. One operand instructions do not involve an access to the operand memory. Thus, the factors that influence the performance of ETS include, total number of instructions ($N \times R$), the fraction of two operand instructions (f_2) and the average memory access time (T_m). Since ETS uses pure dataflow model, each instruction can be viewed as a separate thread, and the thread switching cost is zero (as each instruction carries its own context). In practical implementation of ETS (e.g., Monsoon), a context switch is required to change from one activation frame to another, particularly when only finite number of activation frames can be held in the operand memory. For hybrid dataflow systems (such as EARTH), since there exist two separate execution units, some overlap between thread execution and thread synchronization (and scheduling) can be achieved. The factors that influence performance of the Execution Unit (EU) include N, R, f_m (fraction of instructions which are memory reference instructions), T_m (average memory access time including cache misses), while T_s (thread synchronization, scheduling and context switch time) impacts the performance of the Synchronization Unit (SU). If insufficient parallelism is available in the program to completely overlap the execution of threads (in EU) with synchronization of thread related activities (in SU), hybrid systems incur idle cycles.

In Scheduled Dataflow, thread context is preloaded by the Synchronization Processor (SP), eliminating all memory accesses during the execution of a thread. f_m (the fraction of instructions that are memory access instructions), T_m (average memory access time and T_s (thread synchronization and scheduling) effect the performance of the SP, while the performance of EP is influenced by N, R and the context switch between threads (n_p). If insufficient parallelism is available in the program to completely overlap the execution of threads with the preloading and synchronization delays, Scheduled Dataflow will incur idle cycles during

Table 1: Notation

symbol	meaning
T_p	pipeline execution time of a single instruction
T_m	memory access time including cache miss
T_s	synchronization and scheduling overhead
T_c	context switch overhead
f_2	fraction of instructions with two operands
f_m	fraction of memory access instructions (e.g., preload/post-store)
n_p	number of wasted pipeline cycles on thread switch (set to 5)
N	number of runnable threads in a system
R	average run length of a thread
U	average utilization rate

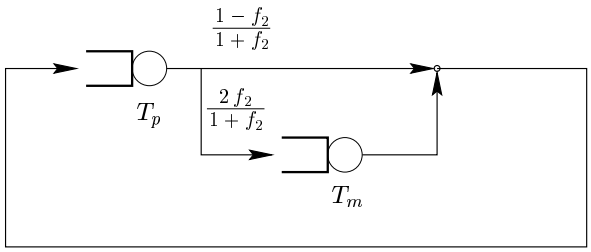
the execution of the program. It should be noted that the context switching overhead(n_p) is at most equal to the number of pipeline stages. This is because, a new thread can be fed into the execution pipeline, soon after the previous thread completes execution (as indicated by "`switch`" instruction). It may be possible to reduce the overhead by initiating a new thread as soon as the "`switch`" to a new context instruction is fetched and decoded. It should also be noted that, in our evaluation of hybrid architectures (such as EARTH), we have lumped the context switch into the synchronization overhead. In practice, the combined overhead of synchronization, scheduling and context switching can be very significant.

Figure 6 shows closed queuing networks for ETS, EARTH-like hybrid architectures, and Scheduled Dataflow. The workload for conventional architecture, ETS, hybrid architecture and Scheduled Dataflow will be respectively N, $(N \times R) \times (1 + f_2)$,[2] N and N. Table 1 defines the symbols.

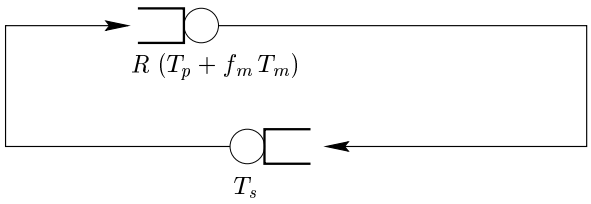
4 Performance Analysis

It is straightforward to derive throughput, utilization, average queue length, etc., from the queuing model using Mean Value Analysis. In this paper, we are interested in comparing ETS and hybrid architectures with Scheduled Dataflow in terms of total execution times. In addition, we are also interested in the effects of run-lengths (coarse vs. fine-grained), number of threads (hence hardware contexts), number of loads/stores, and synchronization overhead

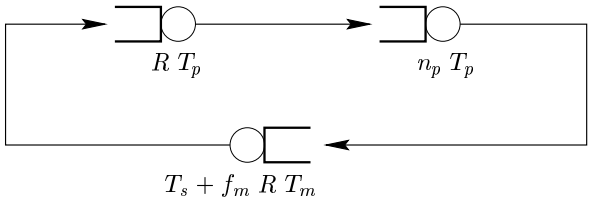
[2]Remember that $1 + f_2$ number of tokens are needed for each instruction on the average, thus making the execution time $1 + f_2$ times total number of executed instructions ($N \times R$). Therefore, the probability of two-operand instruction being executed is represented as $\frac{2\ f_2}{1 + f_2}$.



(a) ETS



(b) EARTH-like hybrid architecture



(c) Scheduled Dataflow

Figure 6: Queuing networks

on the performance of the Scheduled Dataflow. While we do not have real benchmarks for our evaluation, all parameters used are based on either published data, our observations based on hand-coded programs or our observations based on architectural differences.

4.1 Total Execution Time

To compare the execution time of Scheduled Dataflow with that of the other architectures, we have varied f_2 (i.e., fraction of two-operand instructions in ETS) from 30% to 50%, and the synchronization overhead (i.e., T_s) for both EARTH-like hybrid architecture and Scheduled Dataflow from 4 to 8. We also varied the context switch overhead for conventional architecture from 5 to 10. We set the fraction of load/store instructions (f_m) to 35% (using data from [19]). It should be noted that the synchronization overhead (T_s) can be as small as 4 cycles[3]

[3]The typical synchronization and scheduling requires:

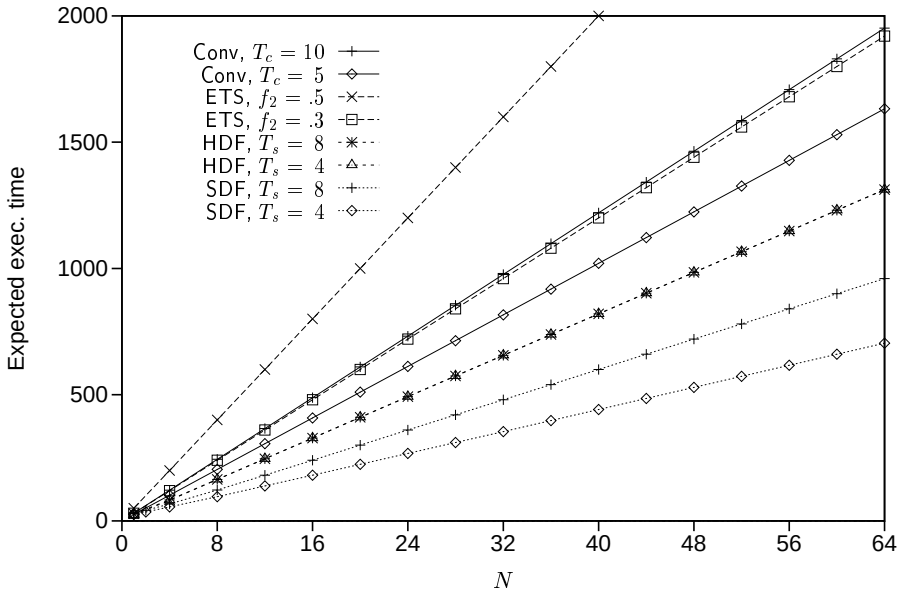
(as indicated in Sparcle[20]). The number of cycles lost between thread switches (n_p) is set to 5 (based on the number of pipeline stages of Scheduled Dataflow system shown in Figure 2). Figure 7 shows the results for different number of threads (N) and run-lengths (R). "HDF" represents hybrid architecture. Note that for ETS, R is always 1 and the number of threads is equal to $N \times R$.

We assumed a CPI (T_p) of 1 cycle, and different memory access times (T_m) for each architecture. We set the memory access time that takes into account cache miss rates and miss penalties (T_m) of Scheduled Dataflow to 2.0. It was previously observed that (data) cache miss rates for ETS are typically higher than those for conventional architectures[2]. It is our contention that the (data) cache miss rates for Scheduled Dataflow will be lower than those of a conventional architecture. This results from the preloading and poststoring of thread contexts which facilitate for better prefetching and data placement possibilities. With these assertions, we set the cache miss rate of ETS to four times as high as that of Scheduled Dataflow (i.e., $T_m = 5.0$ for ETS), Cache miss rates for the other architectures are assumed to be twice that for Scheduled Dataflow (i.e., $T_m = 3.0$). (In section 4.3, we will compare the performance of each architecture by varying memory access time (T_m) from 1.0 through 6.0. Even with the same memory access times and cache miss rates, Scheduled Dataflow performs better than the other architectures.)

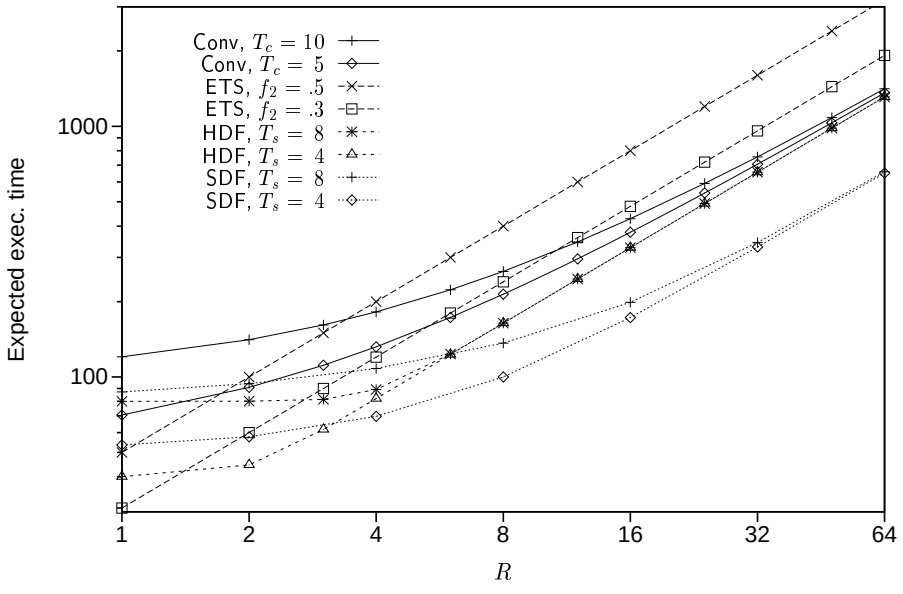
Although the experiment with other values of the various parameters was conducted, we present only two sets of data for each architecture. All results exhibit similar tendencies. Scheduled Dataflow performs better than other architectures unless R is very small (less than 4) or there is very little parallelism (N is less than 2) – this may be difficult to observe from the figure because of the clustering of data near the origin. SDF's performance is superior to the other systems even when the synchronization delays for Scheduled Dataflow are as high as 8. Scheduled Dataflow incurs a fixed overhead even when $N = 0$, since two separate processors are used.

From Figure 7(b) it can be observed that for large R (> 16), the synchronization overhead has negligible impact because the thread execution time (in EP) is completely overlapped by the synchronization and scheduling of threads (in SP). For very small R (< 4), HDF and ETS perform better than the proposed SDF. For $R = 10$ and $N = 16$ (and $T_s = 8$), Scheduled Dataflow requires 26.78% less execution time than hybrid dataflow, 41.14% less than conventional architecture and 49.97% less than ETS. For larger N (i.e., more fine grained

1. post-store results from a thread (already included in f_m).
2. If the synchronization count of a thread is zero, obtain next <FP,IP>.
3. Set FP to the first instruction of the thread.
4. Preload initial values of thread into its local registers (already included in f_m).

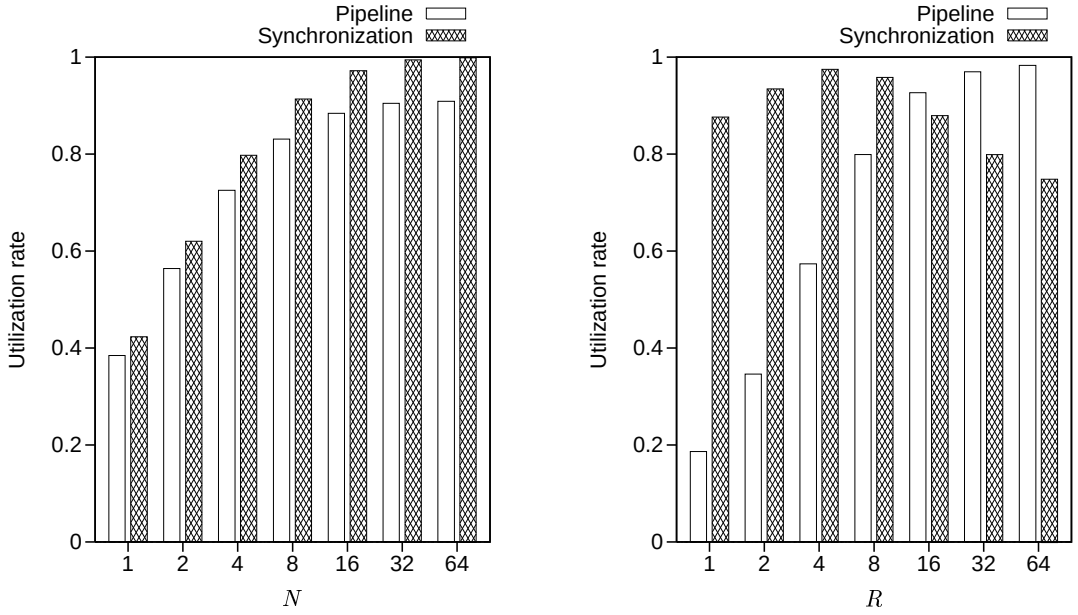


(a) $R = 10$



(b) $N = 10$

Figure 7: Analysis on expected execution time



(a) $R = 10$ (b) $N = 10$

Figure 8: Utilization of Scheduled Dataflow

parallelism), Scheduled Dataflow performs even better. For larger R (i.e., coarser grained threads), the performance differences between SDF, HDF and conventional architectures is less pronounced.

4.2 Utilization

We wanted to investigate the effects of thread granularity and the number of threads(or register contexts) on the utilization of the two separate functional units in Scheduled Dataflow (viz., EP and SP). Figure 8 shows the results. For this experiment we chose $f_m = 35\%$, $T_s = 4$. All other parameters are the same as those for Figure 7. The results indicate that the synchronization unit is not a bottleneck for thread lengths (R) greater than 8. Also, very high utilization can be achieved for modest sized threads ($R = 10$) and a modest number of threads ($N > 8$). This implies that it is feasible to design fine-grained dataflow systems with very small number of register contexts.

4.3 Effect of Memory Accesses

We wanted to investigate the impact of memory access times on the performance. We varied the average number of cycles needed for memory access (T_m) from 1.0 to 6.0. This includes the effect of cache misses and miss penalties. The results are shown in Figure 7.

Here, we set $R = 10$ and $N = 10$.[4] The figure indicates that memory access delays have the most significant impact on ETS. In Scheduled Dataflow memory accesses are overlapped with thread execution. For both conventional architectures and hybrid systems, the impact of memory access time is more linear. The figure also shows that the performance of Scheduled Dataflow is better than the other architecture even when we assume the same cache miss rates and memory access times.

We also studied the significance of the load/store instructions (for preload/post-store) on the performance of Scheduled Dataflow (Figure 10). As expected, with more load/store instructions, the overall execution time increases for SDF. This is because SP is more heavily loaded than EP. This can be alleviated by more parallelism (larger N). For HDF, since the EU performs load/store, the increase in execution times are more linear. The same behavior is found for conventional architecture, too.

4.4 Thread Granularity

In order to investigate the impact of the thread granularity (i.e., R) on the performance of Scheduled Dataflow, we compared the total execution times of Scheduled Dataflow with the other architectures for various values of R. Note that $R = 1$ for ETS. For a fair comparison, we kept $N \times R$ constant in each experiment to reflect equal amount of work on all architectures. The results are shown in Figure 11. We varied total number of instructions and included 2 sets of data in this paper. Results from other values show very similar trends.

The right-most values in the graphs reflect the case when $N = 1$. Except when R is small (less than 8), Scheduled Dataflow outperforms the other architectures, for all values of f_2 (fraction of two operand instructions in ETS), T_c (context switch overhead in conventional architecture) and T_s (synchronization overhead in hybrid architecture and Scheduled Dataflow). The graphs also reflect that optimal performance (in Scheduled Dataflow) is achieved only when a balance between the number of threads and thread granularity is achieved. The experiment indicates that coarse-grained threads are not necessary for high performance (although longer threads tolerate more synchronization and preload/post-store overheads). As expected hybrid architecture performs better than ETS and conventional architecture because of the use of two separate hardware units (EU and SU); *the proposed SDF performs better than hybrid systems, since SDF not only uses two separate hardware units (SP and EP), but also decouples memory accesses from Execution Pipeline.*

[4]Note that previous figures show results using different cache miss rates for the various architectures as described in Section 4.1. Here we compare the architectures for a range of memory access times. As can be observed, even when the memory access times are the same, Scheduled Dataflow performs better than the other architectures.

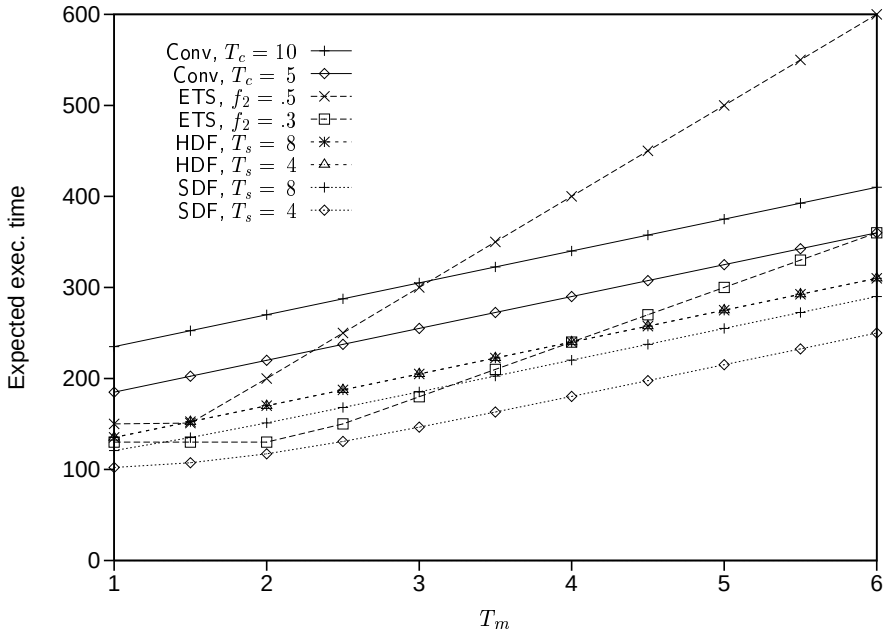


Figure 9: Effect of memory access time

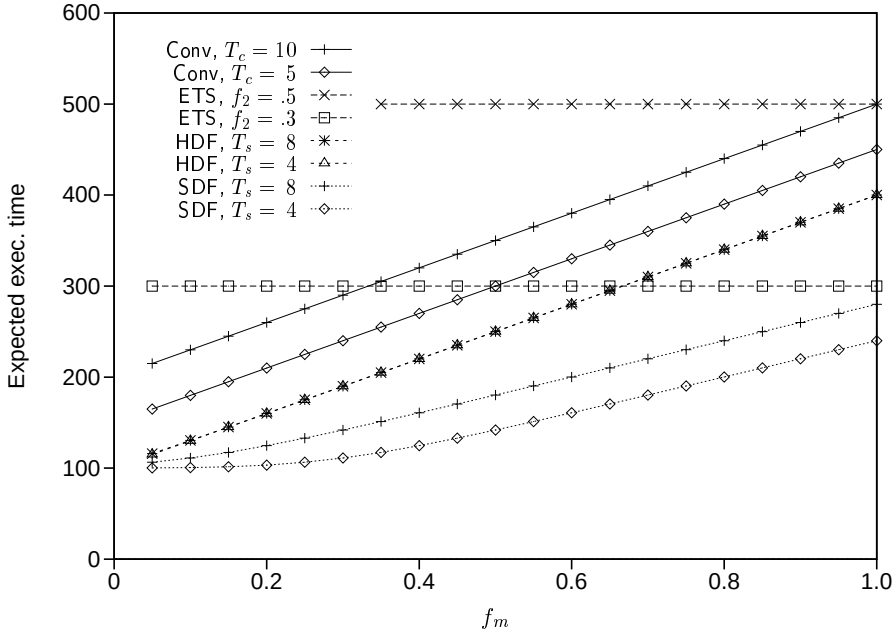
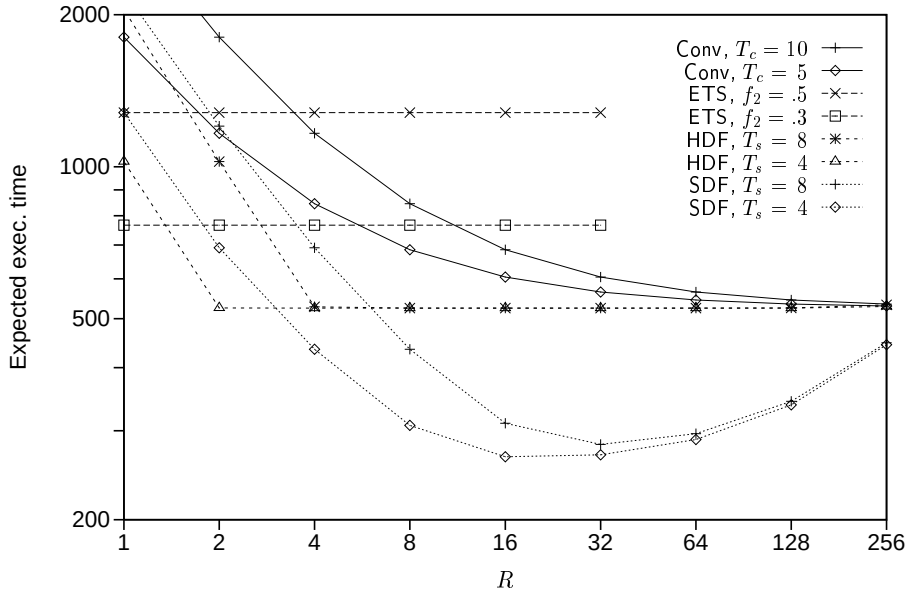
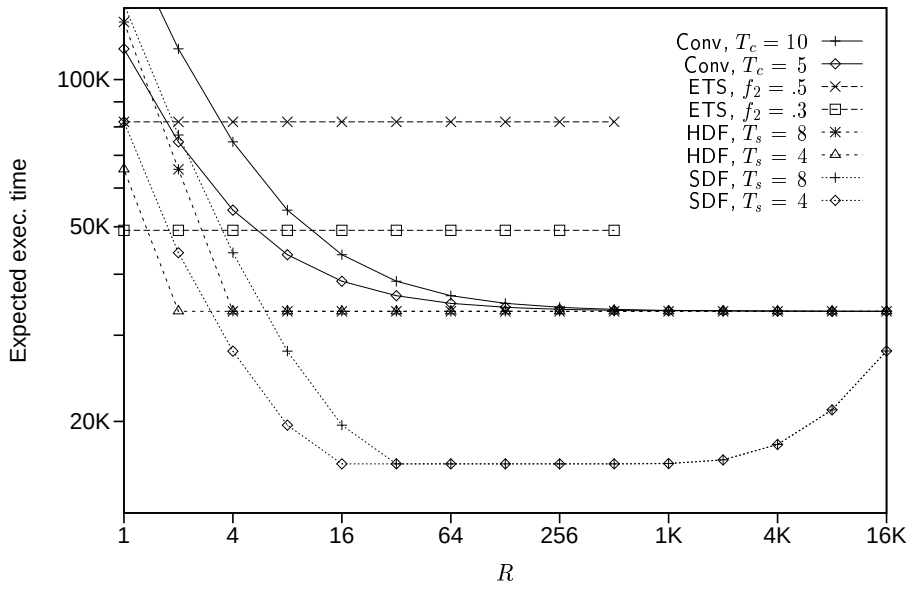


Figure 10: Effect of number of load and store instructions

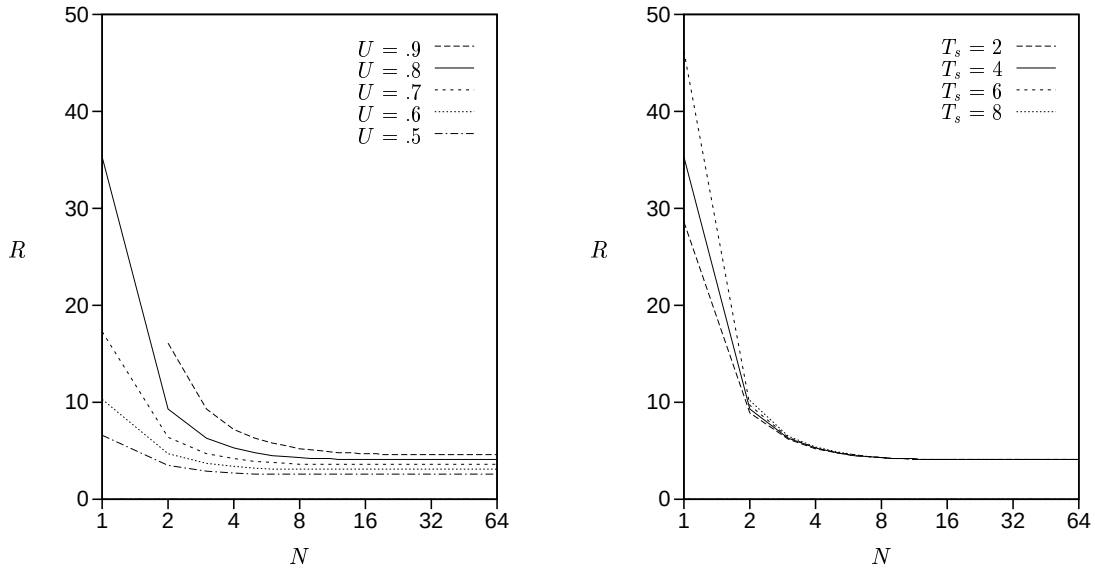


(a) $N \cdot R = 256$



(b) $N \cdot R = 16384$

Figure 11: Effect of run length



(a) $T_s = 4$ (b) $U = 0.8$

Figure 12: Required number of hardware contexts

4.5 Hardware Contexts

Although the previous figure (Figure 11) gives some clues on the number of threads (hence number of hardware contexts) required to achieve optimal performance, we conducted an experiment to investigate the number of hardware contexts (N) needed to achieve high utilization of the pipeline unit. The results shown in Figure 12 indicate that even for a modest number of threads ($N = 6$), and a modest thread granularity ($R = 8$), very high utilization ($> 85\%$) can be achieved. Longer threads are needed to tolerate higher synchronization overheads (Figure 12(b)). This reiterates our belief that it is possible to design fine-grained dataflow systems with a modest number of hardware contexts. For example with run-lengths of 8, 90% utilization is achieved with 6 hardware contexts (even when $T_s = 8$). More contexts do not increase the utilization.

4.6 Utilization of Synchronization Unit and Pipeline Unit

The next issue of interest is the utilization of the two hardware units (SP and EP) in Scheduled Dataflow. We wanted to find when the Synchronization Processor (SP) limits the system performance. We varied the number of threads (with fixed R) and run-lengths (with fixed N), as the synchronization overhead (T_s) is varied from 1 to 10 – reflected by the x-axis (y-axis shows utilization). The results are shown in Figure 13. The utilization of the

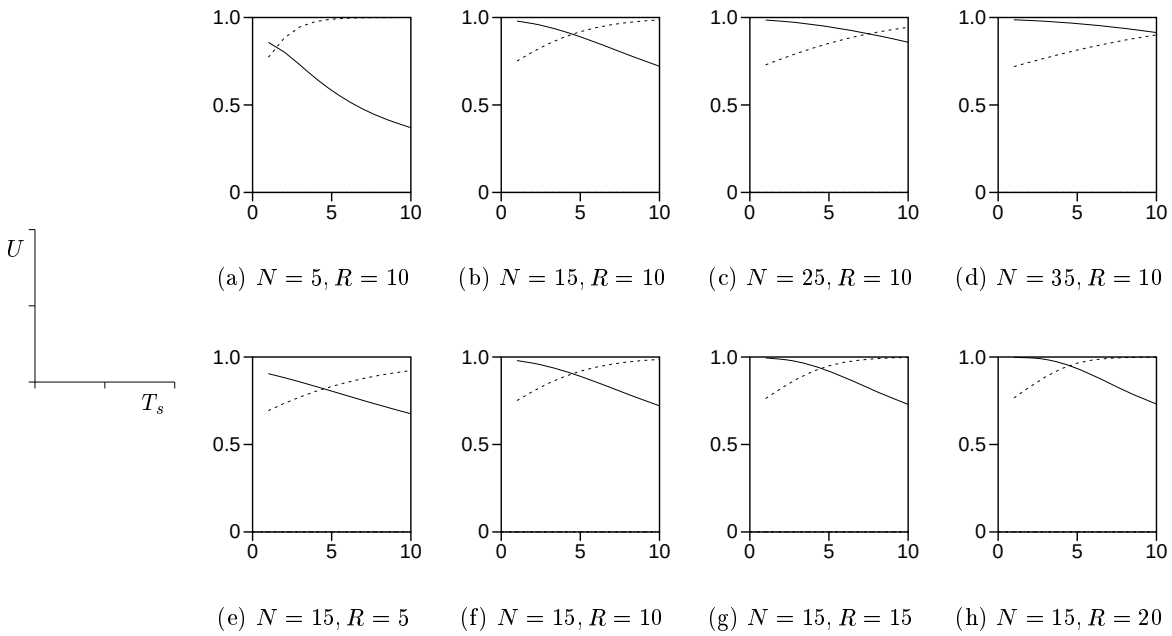
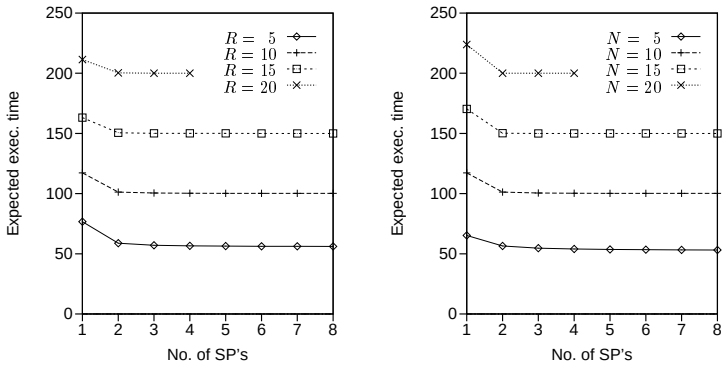


Figure 13: Effect of T_s on system saturation of Scheduled Dataflow

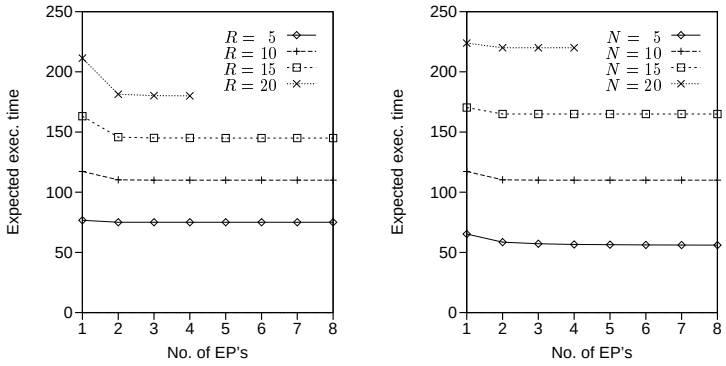
Execution Pipeline (solid lines) decreases while the utilization of the Synchronization Processor(dotted lines) increases with increasing values of T_s. Except for large T_s, Synchronization Processor is not a bottleneck. The figure also implies that it is better to increase the number of contexts (N) instead of the granularity of threads (R), when T_s is large.

4.7 Utilization of Multiple Functional Units

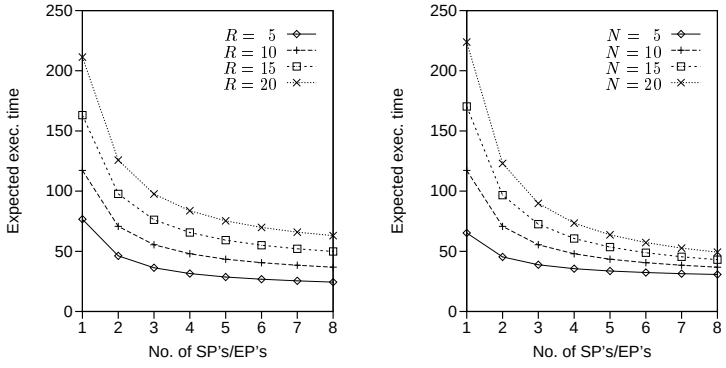
Considering that multiple functional units are commonly used in today's high-performance processors, we measured the potential performance gain when our Scheduled Dataflow adopts multiple functional units. Figure 14 shows three different configurations: only Synchronization Processors are replicated, only Execution Pipelines are replicated, both units are replicated. The performance improvement is most remarkable when both units are replicated. Replication of SP's only, at least for small replication factors, shows performance gains both for larger R (coarser grained threads) and larger N (more parallelism). Replication of only EP's affects the performance only for larger N. These results are to be expected since larger N more heavily loads EP, and smaller N more heavily loads the SP. Figure 15 investigates the effect of multiple functional units on the number of thread contexts. Even with a replication factor of 8, 35 hardware contexts are sufficient for 80% utilization rate (that is about 4 or 5 contexts per pipeline unit).



(a) multiple SP's only



(b) multiple EP's only



(c) multiple SP's/EP's

Figure 14: Effect of multiple functional units

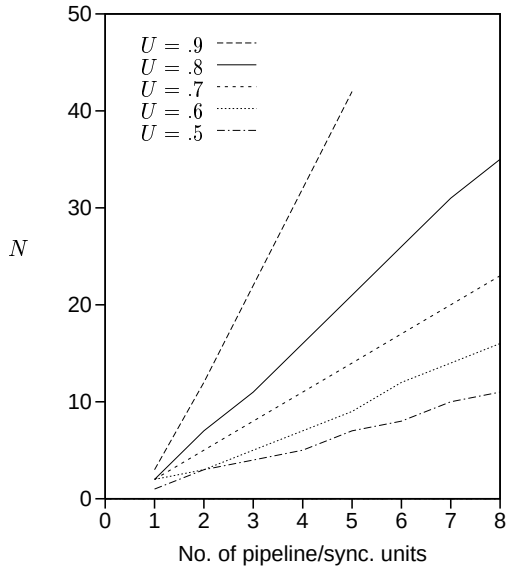


Figure 15: Required number of hardware contexts for multiple functional units

5 Conclusion

Our research is centered around the investigation of approaches to implement dataflow architectures that can become viable alternatives to control-flow systems. We also hope to study the impact of decoupled hardware units on the performance of dataflow and hybrid multithreaded architectures. In this paper we presented a dataflow architecture that deviates from traditional token-driven model, and schedules instructions sequentially, akin to control-flow model. Unlike other hybrid architectures (e.g., [6], [7], [8]) where the dataflow properties were applied only at thread level (i.e., macro-dataflow) with conventional control-flow instructions within threads (along with side-effects, WAW and WAR dependencies), our model retains dataflow properties even at the instruction level. We used a simple closed-form queuing model to analyze the proposed architecture. The results indicate that it is possible to develop control-flow like execution of dataflow architectures, which outperform token-driven dataflow models. The results also indicate that when a separate hardware unit is used for synchronization, scheduling of threads, preloading and post-storing thread data into hardware contexts, one can tolerate synchronization overheads and memory latencies. Most importantly, in such systems, high utilization is achieved with fine-grained threads and a small number of hardware thread contexts. The impact of multiple functional units (i.e., superscalars and multiple issue systems) on the number of hardware contexts (hence the complexity of hardware) is also investigated. Due to the non-blocking and fine-grained nature of our threads, the proposed architecture achieves very high utilization even with very few

hardware contexts.

The closed form reflects an idealized situation and assumes that as threads leave the system an equal number of new threads enter the system. In reality, there may be a variable number of idle cycles between the completion of one thread and the initiation of a new thread due to synchronization requirements (in addition to context switching and preload/post-store delays). In order to analyze the architecture in a more realistic light, we are in the process of developing detailed, instruction level simulations of the proposed architecture and empirically evaluate the proposed architecture. We will use various benchmark programs available in Sisal language for the evaluations.

References

[1] A. R. Hurson, K. M. Kavi & B. Lee. Cache memories in Dataflow architectures, *IEEE Parallel and Distributed Technology*, 1996, 50–64.

[2] K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham, & P. Shanmugam, Design of cache memories for multi-threaded dataflow architecture, *Proceedings of the 22nd Int'l Symp. on Computer Architecture (ISCA-22)*, 1995, St. Margherita Ligure, Italy, 253–264.

[3] M. Takesue, A unified resource management and execution control mechanism for Dataflow Machines, *Proc. 14th Annl. Int'l Symp. on Computer Architecture*, 1987, 90–97.

[4] S. A. Thoreson & A. N. Long, A Feasibility study of a Memory Hierarchy in Data Flow Environment, *Proc. of Int'l Conference on Parallel Conference*, 1987, 356–360.

[5] M. Tokoro, J. R. Jagannathan & H. Sunahara, On the working set concept for data-flow machines, *Proc. of 10th Int'l Symp. on Computer Architecture*, 1983, 90–97.

[6] R. Govindarajan, S. S. Namawarkar & P. LeNir, Design and performance evaluation of a multithreaded architecture, *Proc. of the HPCA-1*, 1995, 298–307.

[7] H. H.-J. Hum, *The super-actor machine: A hybrid dataflow/von Neumann architecture*, doctoral diss., McGill University, Montreal, Canada, 1992.

[8] S. Sakai, K. Okamoto, H. Matsuoka, H. Hirono, Y. Kodama, & M. Sato, Super-threading: Architectural and software mechanisms for optimizing parallel computations, *Proc. of 1993 Int'l Conference on Supercomputing*, 1993, 251–260.

[9] G.M. Papadopolous & D.E. Culler, Monsoon: an Explicit Token-Store Architecture, *The 17th Int'l Symp. on Computer Architecture*, 1990, 82–90.

[10] H. H.-J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, & Y. Zhu, A design study of the EARTH multiprocessor, *Proc. of the Conference on Parallel Architectures and Compilation Techniques*,

Limassol, Cyprus, 1995, 59–68.

[11] J. E. Smith, Decoupled Access/Execute Computer Architectures, *Proc of the 9th Annual Symp. on Computer Architecture*, 1982, 112–119.

[12] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, & D. Yeung, The MIT Alewife machine: Architecture and performance, *Proc. of 22nd Int'l Symp. on Computer Architecture (ISCA-22)*, 1995, 2–13.

[13] D. Chiou, B. S. Ang, B. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. E. Hicks, & A. Boughton, StarT-NG: Delivering seamless parallel computing, *Proc. of the first Int'l EURO-PAR conference*, 1995, 101–116.

[14] K. M. Kavi, D. Levine & A. R. Hurson, PL/PS: A non-blocking multithreaded architecture, *Proc. of the Fifth International Conference on Advanced Computing (ADCOMP '97)*, Madras, India, 1997.

[15] K. M. Kavi & A. R. Hurson, Investigation of operand memory reuse in a dynamic dataflow architecture, *Proceedings of the High Performance Computing Symposium 96)*, New Orleans, Louisiana, 1996, 288–295.

[16] H.-S. Kim, Instruction set architecture of Scheduled Dataflow, *Technical Report, Dept. of Electrical and Computer Engineering, University of Alabama in Huntsville*, 1998.

[17] A. Agarwal, Performance tradeoffs in multithreaded processors, *IEEE Transactions on Parallel and Distributed Systems*, 3(5), 525–539, 1992.

[18] D. E. Culler, Multithreading: Fundamental limits, potential gains and alternatives, *Proc. of Supercomputing 91, workshop on Multithreading*, 1992.

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

[20] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, & D. Yeung, Sparcle: An evolutionary processor design for multiprocessors, *IEEE Micro*, 1993, 48–61.