A Technology-Scalable Multithreaded Architecture

Clecio D. Lima[1], Kentaro Sano[1], Hiroaki Kobayashi[1], Tadao Nakamura[1] and Michael J. Flynn[2]
[1] Graduate School of Information Sciences, Tohoku University
Aramaki Aza Aoba 01, Aoba-ku, Sendai, 980-8579, Japan
{clecio, kentah, koba, nakamura}@archi.is.tohoku.ac.jp
[2] Computer Systems Laboratory, Stanford University
Gates Hall 334, Stanford, California 94305
{flynn@umunhum.stanford.edu}

Abstract—
Advances in integrated circuit technology have offered an increasing transistor density with a continuous performance improvement. Soon, it will be possible to integrate a billion of transistors on a chip running at very high speeds. At this level of integration, however, physical constraints related to wire delay will become dominant, requiring microprocessors to be more partitioned and use short wires for on-chip communication. On the other hand, effective parallel processing by taking advantage of the large number of transistors will be challenging.

In this research, we propose the Shift Architecture, a multithreaded paradigm that maps statically scheduled threads onto multiple functional units. Communication is based on shift register files and restricted to contiguous functional units, requiring reduced wire lengths. Threads are dynamically interleaved on a cycle-by-cycle basis, to maintain high processor utilization. We describe the basic concepts of our approach. A preliminary evaluation shows that this architecture has the potential for achieving high instruction throughput for multithreaded benchmarks.

Keywords— **Wire Delay, Multithreading, Interleaving.**

I. INTRODUCTION

For the past decade, microprocessors' performance improved continuously at a rate of more than fifty percent a year. These improvements are basically due to two factors. First, clock speed has been increasing fast, both by scaling CMOS technology and by deeper pipelining at higher clock rates. Second, several techniques have been used to exploit instruction-level parallelism (ILP) and efficiently utilize the ever- increasing number of transistors on a chip. Within a decade, it will be possible to integrate a billion of transistors on a chip, potentially running at speeds over 3GHz.

However, achieving high performance in future microprocessors will be challenging because wire delays will limit the ability of microprocessors to improve throughput. As CMOS technology improves, wire delay become paramount in comparison with logic delay, and therefore will limit the fraction of chip reachable in a single clock cycle. Thus, long wires can drastically affect the processor cycle time if it happens to be in the critical path. Global structures usually present in conventional architectures, such as register files, crossbars and issue windows, will also be affect by wire delays and circuit complexity and therefore are candidates to become a bottleneck. In order to avoid these physical limitations, future microprocessors must be partitioned into several small independent logic blocks. Communication should be limited to local resources, instead of global resource sharing. The architecture, as well as the compiler and the OS must be aware of the on-chip communication latencies.

Another major challenge to improve performance in future is related to extracting parallelism. Current microprocessors mostly use the superscalar and the VLIW approaches to exploit ILP from a single thread of control. In superscalar architectures, the processor performs data dependence checking at run-time. The amount of ILP, however, is limited by the issue window, whose complexity is proportional to the square of the number of entries [JOH 91]. In VLIW architectures, data dependence analysis is performed at compile-time and therefore requires a much simpler hardware. This approach, however, does not handle well dynamic events, such as cache misses. A stall caused by any one of the suboperations causes all functional units to stall. These approaches also suffer from the limited amount of inherent ILP in single threads of control. Therefore, it is natural to consider exploiting parallelism from multiple threads of control.

So far, designers have emphasized in either increasing clock speed or increasing instruction throughput by improving ILP. However, future generation of microprocessors will require both strategies to be considered.

In this paper, we present a new architectural paradigm called the Shift Architecture, which takes into consideration the physical constraints of technology scaling and, at the same time, efficiently exploits parallelism. The physical constraints are minimized primarily by partitioning a conventional global register file into several independent register files. Each functional unit has its private set of registers

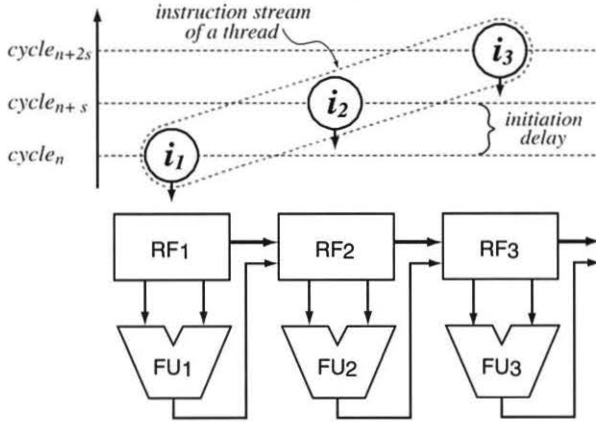

Fig. 1 Functional units arrangement and instruction scheduling

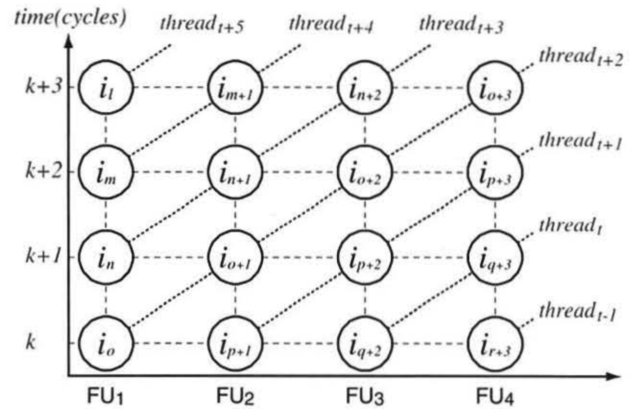

Fig. 2 Thread Interleaving

to access and on-chip communication is done mainly by shifting register contents to the consecutive functional unit. This reduces the complexity of register files as well as avoids long wires for communication.

In our approach, different threads are interleaved on a cycle-by-cycle basis, allowing fast switching between threads and fast pipelines to be built, since no data dependences have to be solved inside the pipeline. Parallelism is exploited by executing several threads simultaneously.

Section II describes the basic concepts of the Shift Architecture. Section III presents our preliminary evaluation. Section IV presents our simulation results. In Section V, we present some of the related works. Finally, we summarize and present our concluding remarks in Section VI.

II. THE BASIC CONCEPTS

In the Shift Architecture, functional units (FUs) are arranged in a 1-dimensional array. Each functional unit has a dedicated register file (RF), as showed in Figure 1.

To avoid the use of long wires and complex crossbars, the communication between FUs is done only through consecutive RFs. The contents of a RF as well as the results from its corresponding FU are placed into the consecutive register file every clock cycle. The compiler statically schedules instructions and guarantees that operands will be available at the correct place to be executed by the appropriate instruction. In the example illustrated in Fig. 1, for example, instruction i_1 is assigned to FU_1. After operands are fetched from registers, the whole content of RF_1 is shifted to RF_2. The result from FU_1 is also placed in RF_2. Thus, all the operands will be available for instruction i_2 to use in FU_2, for i_3 in FU_3 and so on. For an individual thread, the appearance of a single and static register file is maintained. This simplifies scheduling since a thread accesses a register file as if it was dedicated to that thread. Moreover, threads do not need to compete for access.

The delay between the executions of two consecutive instructions from the same thread is called *initiation delay*. This initiation delay must be long enough to allow the data to be shifted from one register file to the next, and short enough to permit a fast execution of a thread. To avoid using a pipeline stage devoted to data shifting, we can perform the shift of data simultaneously with the execution of an instruction in a FU.

The compiler statically partitions a program into threads. Each thread performs a round of computation on a set of data, which contains no data dependencies with the other threads.

At run-time, the hardware scheduling mechanism interleaves several threads to exploit inter-thread parallelism and maintains high utilization of functional units. The arbitration for functional unit usage is performed on a cycle-by-cycle interleaving basis. Every cycle, the processor dynamically switches to a new thread and maps its instructions to be executed onto the multiple functional units.

Interleaving different threads on a cycle-by-cycle basis has two main advantages. First, it allows a fast switching between threads, since no context switching determination has to be performed. Second, since there are no dependencies between instructions of different threads, stalls due to pipeline dependencies can be avoided if enough threads are available for fetching. This leads to a fast pipeline since complex forwarding paths are not necessary ([BOO 92], [LAU 94]). Figure 2 demonstrates how multiple threads are dynamically interleaved across the multiple FUs. Each node in the grid represents an instruction been executed. Rows show instructions that are executed at the same time, whereas columns show instructions that execute in the same functional unit. In this example, an initiation delay of 1 clock cycle is assumed.

A. Instruction Fetch

Once a thread starts execution in the first functional unit all the consecutive units should execute instructions of the same thread in subsequent cycles. To simplify scheduling and the organization of the instruction memory system, a

block of instructions is fetched from a thread every clock cycle. Each block should contain as many instructions as the number of functional units available. These instructions are primarily stored in the instruction buffer IB_1, which corresponds to the first functional unit in the arrangement. The first instruction of the block is fetched from IB_1.The remaining instructions are shifted to consecutive IBs in the same fashion as the data shifting. Giving the small size required for such buffers, we expect that fetching an instruction from each of them is fast and trivial.

The fetch unit should be smart about which thread it fetches, selecting from those which can offer the most immediate benefit. This selection is done by means of a thread management unit (TMU), a table that contains several program counters, one for each thread context. The main responsibility of the TMU is to maintain the control of thread states and to provide a fast fetching. When a cache miss or branch misprediction is detected, the state of a thread is asserted to *not-ready-to-fetch*, and will not be eligible to be fetched until it recovers from the penalty, and then becomes *ready-to-fetch*. Threads are selected among ready contexts in a round-robin fashion.

B. Multiple-cycle Instructions

So far, combining the data shifting and the interleaving techniques in a cycle-by-cycle basis may look simple. However, such execution model only works under the assumption that every instruction can be executed in a single clock cycle. Instructions that take multiple cycles to execute, such as multiplication or division, would cause all the functional units (to its right) to stall. To avoid these stalls, we propose the inclusion of dedicated multiple-cycle units that could be used in parallel with single-cycle functional units. A possible implementation for the Shift Architecture including these units is illustrated in Figure 3. The processor is organized as a collection of *processing elements* (PEs), each of which contains dedicated functional units for multiplication, division, floating-point operations, several single-cycle units, as well as the corresponding register files.

Details of the scheduling of multiple-cycle instructions will be discussed in a future paper.

C. Memory Access

Functional units mainly depend on the data stored in register files. However, although less frequent, accesses to data memory may become a bottleneck, due to the long memory latencies usually implied. Therefore, the design of an adequate memory access system becomes another key issue in the implementation of our architecture. The simplest possible implementation would be to stall the whole thread after a (shared) memory reference. This stalled thread could not be fetched again until the memory access is performed. In fact, this model was implemented in the Denelcor HEP ([SMI 81]) and MASA ([HAL 88]), which used a cycle-by-cycle context switching mechanism. However, stalling at every memory access showed to severely degrade performance, since it requires a very large number of threads to hide the memory latency.

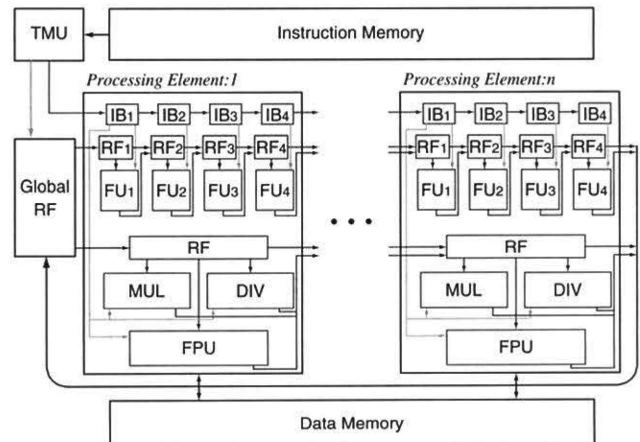

Fig. 3 A possible implementation for the Shift Architecture

Techniques as the grouping of load instructions described in [BOO 92] could reduce the number of memory accesses. However, to effectively obtain high performance, the alternative is preventing the processor from the long-latency of a shared memory by using caches. The Shift Architecture, however, imposes additional difficulties to caching. Exploiting locality seems to be more difficult given the large number of threads that access a cache, potentially resulting in high cache miss rates. Providing efficient cache coherence and exploiting data sharing among threads at compiler level will be necessary and are of high priority in our future work.

D. Branch prediction

A thread may contain unconditional and conditional jumps among its instructions. Unconditional jumps shall cause functional units subsequent to the one containing the jump instruction to stall. In the case of conditional jumps, however, the branch to be taken can be predicted. When a misprediction is detected, the consecutive functional units must stall. At this time, however, the following instruction of the same thread is already been fetched. Thus, the stalling should be done by locking the consecutive register file and avoid any of the following instruction to perform a register write.

Finally, the program counter of the thread incurring the misprediction penalty must be updated in the thread management unit.

III. Preliminary Evaluation

Using multiple threads has basically two objectives in the Shift Architecture – to fill all the processing elements pipeline stages and to hide memory latency. In this evalua-

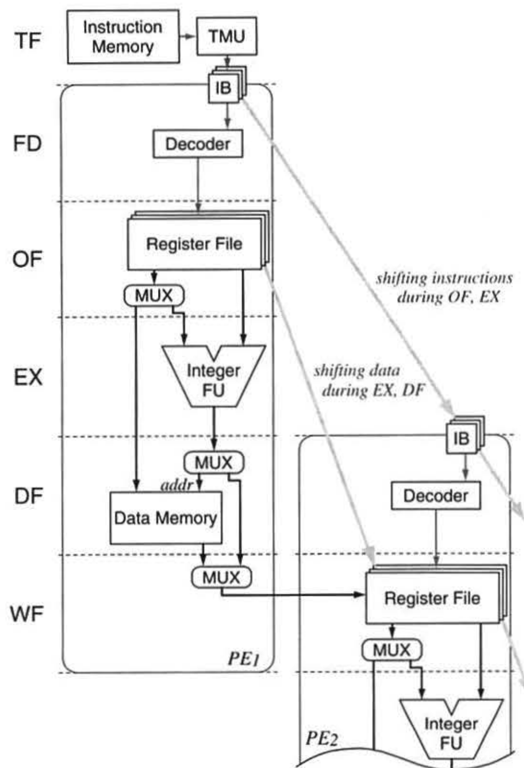

Fig. 4 Pipeline stages

tion, however, we examine only the aspects of program behavior that affect the utilization of the processing elements. We simulate a simplified implementation of the Shift Architecture, and examine the behavior of some multithreaded benchmarks.

Several assumptions are made in order to simplify our evaluation. A processing element contains only one functional unit and corresponding register files. This functional unit is able to perform integer, single-clock cycle operations. The TMU is assumed to be able to handle any number of threads in a single clock cycle. Instruction and data memory are considered large enough to hold instructions and data of the benchmark programs. Each register file is assumed to contain three 16-register banks.

In addition, to avoid the influence of long memory latencies in our results, we assume a memory reference to occur within a single clock cycle. This assumption allows us to concentrate only on the processing utilization aspect. Moreover, benchmarks were carefully written to contain none or very few memory access instructions.

A. *Pipeline Implementation*

The simplified implementation used in our simulations is described here. Figure 4 illustrates the pipeline stages in two consecutive processing elements. Each processing element consists of five pipeline stages: instruction fetch and decode (FD), operand fetch (OF), execution (EX), data fetch (DF) and write forward (WF). A special pipeline stage – the thread fetch (TF) stage - is required in order to fetch a block of instructions to the first processing element.

A.1 Thread fetch (TF)

In this stage, a block of instructions is fetched from the instruction memory. The block to be fetched is determined by the TMU, which provides the current program counter (PC) of an active thread. The instruction fetch unit fetches enough instructions to keep all the functional units busy. For example, n instructions should be fetched in a configuration of n processing elements, starting from that indicated by the PC. These instructions are stored in the instruction buffer.

A.2 Instruction fetch and decode (FD)

During this stage, an instruction is fetched from the instruction buffer and then decoded. The remaining instructions are shifted to consecutive units. Because instructions are shifted, the buffer will always contain the same number of instructions, unless a thread contains blank instructions or the TMU was not able to schedule any thread at that cycle. This simplicity makes its operation fast, giving room to perform both instruction fetch and decode in the same cycle.

A.3 Operand fetch (OF)

In this stage, the source operands required by the instruction are read. Most of these operands will be available since most of the data dependencies are satisfied during scheduling. The only data dependences not solved by the scheduling mechanism are those involving long memory latencies. However, these dependences will not be considered in this simulation.

A.4 Execution (EX)

In the execution stage, integer operations are performed. We assume that any integer operation can be performed in a single clock cycle.

A.5 Data fetch (DF)

During this stage, memory is accessed during this stage. We assume a single clock cycle for local memory references.

A.6 Write forward (WF)

Finally, the results are written to the destination register. Since this destination register is located in the register file of the next PE, we call this stage write-forward stage, instead of the conventional write-back stage.

A.7 Instructions and Data Shifting

The instruction shifting between instruction buffers of contiguous PEs is performed during the stages OF and EX.

Instruction buffers have to keep their contents for three stages: FD, OF and EX stages, due to the initiation delay. This behavior requires each instruction buffer to be implemented with three banks, as showed in Figure 4. These banks are alternately used by different threads, to avoid pipeline stalls.

The data shifting between the register files of contiguous PEs is performed simultaneously with the stages EX and DF. Three banks are also necessary per register file, for the same reason as for the instructions buffers.

B. Simulation

We developed a simulator in order to evaluate the behavior of the implementation described above. Our simulator is execution-driven and models the pipelines stages as well as the instruction and data shifting. Basically, it takes the assembly code of a multithreaded benchmark as input, executes the program and counts the number of instructions executed per clock cycle. The experiment was performed for three architectural configurations with four, six and eight processing elements, respectively.

B.1 Benchmarks

We selected two simple algorithms as benchmarks: the Fast Fourier Transform (FFT) and MMT, for the large amount of thread-level parallelism available. These benchmarks were written in assembly language using the MIPS instruction set. The partitioning into threads was done manually by following the natural flow of the algorithms.

The FFT algorithm performs the butterfly computation described in [BRI 97]. At each stage of the algorithm, multiple threads may perform a number of computation independently. However, because the result of a stage serves as input for the subsequent one, threads of different stages are required to communicate with each other. This communication is done through the memory. We executed this benchmark for 16 and 32 elements as inputs.

The MMT algorithm performs the multiplication of two matrices. Each thread consists basically of a row by column operation, and therefore no inter-thread communication is required. We performed this algorithm for matrixes of 10x10 and 20x20.

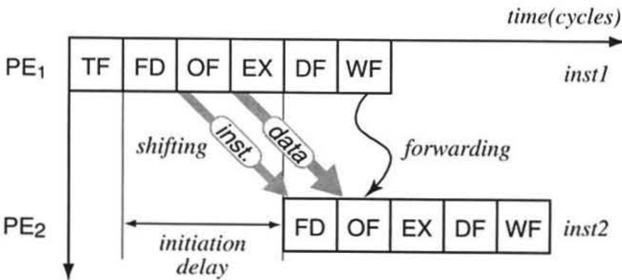

Fig. 5 Interaction between two successive instructions

C. Mathematical analysis

Figure 5 demonstrates two successive instructions of the same thread, mapped respectively to PE1 and PE2. The content of the register file of PE1 is shifted to the register file of PE2 during the execution stage. Instruction 1 writes its results directly to the register file of PE2, to be used by instruction 2. Since instruction 2 cannot read operands before receiving the results from the previous instruction, an initiation delay of three clock cycles is necessary. We assumed that a register write occurs in the first half of a clock cycle and the operand fetch occurs in the second half.

It is important to notice that once a thread is fetched, it cannot be fetched again until its last instruction leaves the last stage of the last processing element. Therefore, a single threaded program is expected to perform poorly. The minimum number of threads (nt) required to fully utilize all the processors is, in this implementation, given by (1).

$$nt = (id \times np) + id \tag{1}$$

where np is the number of processing elements and id is the initiation delay.

However, having enough threads does not guarantee high utilization. Another factor that influences in the utilization of the processing elements is the number of instructions in a thread. Threads in which the number of instructions is a multiple of the number of processing elements will be able to fill all the pipeline stages, while other threads not. We define completeness (c) of a thread as its ability to fill the processing elements. It can be represented mathematically as a function of np and the number of instructions (ni) of a thread.

For $ni = k \cdot np$, where k is an integer, the completeness is given by (2).

$$c = 1 \tag{2}$$

For the case where $ni \neq k \cdot np$, the thread completeness should be less than 1, indicating that the thread will not be able to fill some of the PEs with instructions during its execution. For example, consider a thread with 3 instructions, to be executed on a configuration with 4 PEs. In this case, the thread completeness would be c=3/4=0.75.

Now consider a thread with 7 instructions in the same configuration. In this case, the thread would be fetched in 2 different blocks, containing 4 and 3 instructions, respectively. The first block has completeness c=1. The second block has completeness c=0.75, as in the previous example. Therefore, the thread completeness would be c=(1+0.75)/2=0.875.

For $ni \neq k \cdot np$, in a general form, the completeness is given by (3).

$$c = \frac{ni}{ni + np - ni \bmod np} \tag{3}$$

IV. RESULTS

A factor of primary importance to effectively utilize the processing elements is the number of threads ready to be fetched every clock cycle.

Table 1 compares the total number of generated threads with the average number of ready-to-fetch threads for all benchmarks and architectural configurations. In all benchmarks, the number of threads that can be fetched every cycle is smaller than the total number of threads generated, and decreases as the number of processing elements increases. This is basically because the number of threads currently occupying a pipeline (and therefore is not eligible for fetching) increases with the number of processing units.

For the two types of MMT, the number of threads available for fetching is superior to the minimum number of required threads. This is due to the threads in MMT being independent of each other. Except for those threads currently in execution, all the remaining threads can be fetched at any time.

For FFT, however, the number of threads that can be fetched is usually smaller than the required. This is because threads performing a stage of the FFT algorithm depend on the results of the previous stage. This dependence causes many threads to wait, even if these threads are not occupying the processing elements.

Using the average number of ready-for-fetch threads to estimate the utilization of processing elements fails when there is a large variation in the distribution of threads in a program. Some portions of a program can contain a large number of threads, while other portions can be mostly serial. The benchmarks, used in our experiments, however, presented a constant distribution of threads due to the nature of the algorithms.

The other factor that can influence in the utilization of the processing units is the completeness of threads. Table 2 shows the average size of a thread and the completeness in each benchmark. In general, thread completeness tends to increase as thread size increases, as can be observed in the table. However, varying the number of processing elements also causes thread completeness to oscillate, suggesting that the size of a thread be adjusted to the number of processing elements. In particular, this effect is stronger for threads of short size, as can be observed for the FFT benchmarks.

TABLE I

AVERAGE NUMBER OF READY THREADS AND THREAD SIZE. THE NUMBERS IN PARENTHESIS SHOW THE NUMBER OF THREADS REQUIRED FOR FULL UTILIZATION

Benchmark	Average number of ready-to-fetch threads			Total number of threads
	4 PEs(15)	6 PEs(21)	8 PEs(27)	
FFT-16	14.6	13.4	12.4	60
FFT-32	26.8	25.4	24.2	150
MMT-10x10	67.5	62.5	56.5	100
MMT-20x20	285	265.5	242.5	400

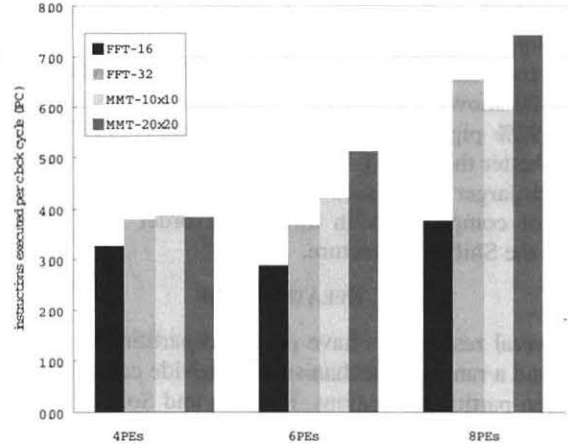

Fig. 6 Instructions executed per cycle

Figure 6 shows the number of instructions executed per clock cycle (IPC) for each benchmark. FFT-16 presents the poorest performance among the benchmarks in all experiments. This is due to the small size of the benchmark and consequently the small number of threads that could be extracted and kept ready-to-fetch.

FFT-16 and FFT-32 perform better on a 4 PEs configuration than on a configuration of 6 PEs. This can be explained by the lower thread completeness of these benchmarks for 6-PEs. The effect of low thread completeness can also be observed for MMT-10 and MMT-20. Although these benchmarks offer a large amount of thread-level parallelism, increasing in IPC is not proportional to the number of PEs because of their unbalanced thread completeness. In particular, they perform poorer than expected for a configuration of 6-PEs.

As expected, the highest utilization for all benchmarks was obtained for a configuration of 4-PEs, due to the small number of threads required. For this case, IPC ranges from

TABLE II

AVERAGE THREAD SIZE AND THREAD COMPLETENESS

Benchmark	Average Thread Completeness (*c*)			Average thread size (instructions)
	4 PEs	6 PEs	8 PEs	
FFT-16	0.96	0.86	0.96	15.4
FFT-32	0.96	0.86	0.96	15.4
MMT-10x10	1.00	0.93	0.87	28
MMT-20x20	0.98	0.98	0.92	59

3.25 to 3.92, indicating that threads are able to fill nearly every stage of the pipeline.

For the configuration of 8-PEs, FFT-32, MMT-10 and MMT-20 shows IPC ranging from 5.98 to 7.42, indicating up to 92% pipeline utilization. Interestingly, FFT-32 performs better than MMT-10, although the number of threads is much larger in the second. This again shows the importance of completeness in threads, in order to effectively utilize the Shift Architecture.

V. RELATED WORK

Several researchers have proposed partitioned architectures and a range of mechanisms to provide communication between partitioned regions. Franklin and Sohi ([FRA 92], [SOH 95]) proposed the *Multiscalar*, a collection of processing units with a sequencer that assigns threads to be executed in parallel. In this architecture, the processing units are connected by a unidirectional ring, which is used to forward information from one unit to the next. Special tag bits containing information of forwarding are determined at compile-time.

In [KEC 92], Kecker and Dally proposed a technique called *Processor Coupling*, where threads are dynamically scheduled to be executed in different processing units. Inter-thread communication is done by explicit instructions, which allows threads to allocate results directly in each other's register files. This technique was implemented as an experimental multicomputer, the *M-Machine* ([FIL 95]).

Waingold ([WAI 97]) proposed the *RAW* architecture, a distributed execution model with extensive software support. In the RAW architecture, several simplified ALUs work on multiple compiler generated instruction streams and communicate through a point-to-point interconnect.

Rotenberg ([ROT 97]) proposed the *Trace Processor*, an architecture where multiple processing elements execute different traces of a program, passing data across a common register bus.

Hammond ([HAM 97]) described a chip multiprocessor (CMP), in which each single processor is assigned to a single thread, generated at compile time and scheduled with run-time support. Inter-thread communication is basically performed through a shared memory.

Finally, in [TSA 97] Tsai and Yew proposed the *Superthreaded* architecture. By implementing a dedicated communication unit in each processing unit, this approach allows recurrence data and memory address reservations to be forwarded to consecutive units, and then execute data-dependent threads in parallel.

The main difference between these and the Shift Architecture approach is the way a thread is scheduled. While in conventional multithreaded architectures a single thread is assigned to a specific processing element, our approach executes a thread along all the available units.

VI. CONCLUSIONS

In this paper, we have proposed the Shift Architecture, a technology-driven multithreaded architecture that uses a new scheduling mechanism based on shift register files. We have described the basics of this mechanism and evaluated some of the conditions required for achieving high performance.

The physical characteristics of this mechanism permit two evident conclusions. First, since communication is restricted to contiguous functional units, no global wires will be necessary. Second, each register file is small and requires only a single read and a single write ports, in contrast with the huge multi-ported register file of conventional architectures. Because of these two features, we believe our proposal to be less passive to the negative effect of long wires in future generations of transistor technology. In our model, a limiting factor could be the large number of registers is required. However, this fact will not pose a problem, because of the large number of transistors expected in future generations.

A primary difficulty for this architecture, indeed, is achieving high utilization. To keep functional units busy, we have provided a dynamic thread interleaving mechanism. Our preliminary evaluation results show that achieving high instruction throughput in the Shift Architecture requires a large number of threads available for fetching every clock cycle. The number of ready-to-fetch threads depends on the structure of a program. Our simulations showed that inter-thread data dependence can degrade the performance of some of the simulated benchmarks. This effected, however, could be hidden if more threads were available. Moreover, our architecture could execute independent programs or processes in parallel. In this case inter-thread dependences would be minimized.

Thread completeness also showed to have influence in performance, suggesting that compilers not only need to generate threads, but to adjust the size of threads to the number of functional units. These adjustments, however, become more complicate when branch decisions and memory references are present. These effects are being examined in our current studies.

Our current work also includes analyzing the effect of multiple-cycle instructions in scheduling, as well as providing an efficient mechanism for memory access and inter-thread communication. As a future work, we intend to develop compiler techniques to support our model. Another research direction is to evaluate the impact of the distributed register communication in terms of clock speed improvement and power consumption.

REFERENCES

[BOO 97] BOOTHE, B.; RANADE, A. Improved Multithreding Techniques for Hiding Communication Latency in Multiprocessors. In *Proceedings of the 19th Interna-*

tional Symposium on Computer Architecture, pages 214-223, Queensland, Australia, May 1992.

[BRI 97] BRIGHAM, Oran E. *Fast Fourier Transform and Its Applications*. New Jersey: Prentice-Hall, 1997. 416p.

[FIL 92] FILLO Marco; KECKLER, Stephen W.; DALLY William J.; CARTER Nicholas P.; CHANG Andrew; GUREVICH Yevgeny, LEE, Whay S. The M-Machine Multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146-156, November 1995.

[FRA 92] FRANKLIN M. *The Multiscalar Architecture*.

[HAL 94] HALSTEAD, R. H. Jr.; FUJITA, Tetsuya. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 443-451, 1992.Madison: UWM, 1995 (PhD. Thesis).

[HAM 97] HAMMOND, L.; NAYFEH, B. A.; OLUKOTUN, K. A Single-Chip Multiprocessor. *IEEE Computer*, pages 79-85, September 1997.

[JOH 91] JOHNSON, Mike. *Superscalar Microprocessor Design*. New Jersey: Prentice-Hall, 1991. 288p.

[KEC 92] KECKLER, Stephen W.; DALLY William. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 202-213, Queensland, Australia, May 1992.

[LAU 94] LAUDON, J et al: Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations In *Proceedings of the International Conference* on ASPLOS, October 1994.

[ROT 97] ROTENBERG, E.; JACOBSON, Q.; SAZEIDS, Y.; SMITH, J. Trace Processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138-148, December 1997.

[SMI 81] SMITH, B. J. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 288, pages 241-248, 1981.

[SOH 95] SOHI, G. S.; BREACH, S. E.; VIJAYKUMAR, T. N. Multiscalar Processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414-425, June 1995.

[TSA 97] TSAI, J-Y.; HUANG, J.; AMLO, C.; YEW, P-C. The Superthreaded Processor Architecture. *IEEE Transactions on Computers*, pages 881-902, September 1999.

[WAI 97] WAINGOLD, E.; TAYLOR M.; SRIKRISHNA, D.; SARKAR, V.; LEE W.; LEE, V.; KIM, J; FRANK, M.; FINCH, P.; BARUA, R.; BABB J.; AMARSINGHE, S.; AGARWAL, A. Baring it all to software: RAW Machines. *IEEE Computer*, pages 86-93, September 1997.