First IEEE Symposium on High-Performance Computer Architecture, January 22–25, 1995, Raleigh, USA, pp.318–325

The Effects of STEF in Finely Parallel Multithreaded Processors

Yamin Li and Wanming Chu

Computer Architecture Laboratory
The University of Aizu
Aizu-Wakamatsu, 965-80 Japan

Abstract

The throughput of a multiple-pipelined processor suffers due to lack of sufficient instructions to make multiple pipelines busy and due to delays associated with pipeline dependencies. Finely Parallel Multithreaded Processor (FPMP) architectures try to solve these problems by dispatching multiple instructions from multiple instruction threads in parallel. This paper proposes an analytic model which is used to quantify the advantage of FPMP architectures. The effects of four important parameters in FPMP, S, T, E, and F, (STEF) will be evaluated. Unlike previous analytic models of multithreaded architecture, the model presented here concerns the performance of multiple pipelines. It deals not only with pipeline dependencies but also with structure conflicts. The model accepts the configuration parameters of a FPMP, the distribution of instruction types, and the distribution of interlock delay cycles. The model provides a quick performance prediction and a quick utilization prediction which are helpful in the processor design.

Keywords and phrases: finely parallel multithreading, interleaved dispatching, round robin scheduling, pipeline dependency, structure conflict, speed-up ratio, utilization of pipelines.

1 Introduction

Pipelining has been widely used in designing processors for exploiting the parallelism of operations. The potential speed-up of pipelining is equal to the number of pipeline stages used. This advantage encourages engineers to use deeper and deeper pipelines in designing high-performance processors. But, this ideal speed-up is rarely achieved in practice due to the delays associated with pipeline dependencies and memory access latencies. NOOP instructions (no operation, i.e., pipeline bubbles) will be inserted into the delay cycles.

An approach of improving pipeline utilization is to realize concurrent multithreading in the processor. Such processors dispatch instructions from different threads on every clock cycle to tolerate the delays. An instruction thread is defined as a set of instructions belonging to a particular context that can be executed independently of other instruction threads [1]. Because there is no dependency between instructions belonging to different threads, pipeline bubbles due to pipeline dependencies or processor stalls due to memory latencies can be prevented [4]. This cycle-by-cycle interleaved processor is called *Finely Concurrent Multithreaded Processor* (FCMP).

Contrast to FCMP, the *Coarsely Concurrent Multithreaded Processor* (CCMP) dispatches instructions from one thread. When a remote memory access is encountered, the processor rapidly switches to another thread [3]. CCMP can give a single thread better performance but has worse capability of tolerating instruction dependencies than FCMP. Both the FCMP and CCMP dispatch instructions sequentially. This is one of the main features of the concurrent multithreaded processors. The performance of this type processors is influenced mainly by the instruction dependencies.

The concurrent multithreading is efficient when (1) the system has large memory-access latencies, (2) the processor has a deep pipeline, and (3) the context switch overhead is low. In order to speed-up the context switch, generally, multiple register sets and special data paths are needed to serve multiple threads. Since only one thread uses its register set at a given time, the concurrent multithreaded processors result in a low utilization of multiple register sets. Furthermore, because of the advances in circuit technologies, most high-performance processors were designed with multiple functional units (multiple execution pipelines) to execute different type of instructions. For example, six functional units (ALU, Shifter, Load/Store Unit, Branch Unit, Floating-Point Adder and Multiplier) can accept six instructions in one clock cycle. In this case, the concurrent multithreaded processor also result in a low utilization of multiple pipelines.

Allocating multiple thread slots in a single processor, to realize multiple instruction threads to be executed simultaneously, is a solution for improving the utilization of multiple pipelines [1, 2]. The multiple threads may be generated from a single program or from multiple programs. Thus, the MIMD parallel processing will be realized on a single processor. Such processor is called parallel multithreaded processor. Similarly, there are *Finely Parallel Multithreaded Processor* (FPMP) and *Coarsely Parallel Multithreaded Processor* (CPMP). In the CPMP, during a period, the number of scheduled threads is equal to the number of thread slots. Whenever a remote memory access from a thread is encountered, the processor suspends that thread and schedules a new thread. On the other hand, the FPMP dispatches instructions from different resident threads (usually more than thread slots) on every clock cycle. Multiple execution pipelines and multiple register files are needed for executing multiple instructions in parallel. The performance of this type of processors is influenced not only by instruction dependencies but also by structure conflicts.

When the number of thread slots is given, using more functional units in a single processor will improve the processor performance, but will result in a low functional unit utilization. On the other hand, using fewer functional units will improve utilization but at the cost of reduced performance. A compromise between processor performance and functional unit utilization must be made for the processor design.

Dubey et.al. [5] proposed an analytic model for evaluating the FCMP architecture. The model presented by them is limited to predicting the performance of a processor with one pipeline and one thread slot. Because, at most one instruction can be issued on every clock cycle, no structure conflict exists. The processor performance is affected only by the distribution of instruction interlock delay. When the processor has sufficient instruction threads for interleaved scheduling (for example, when the number of threads is equal to or larger than the maximal number of cycles required by execution pipeline stage), the processor utilization is said to be 100%. But when the processor has six independent execution pipelines, the average pipeline utilization is only 16.7%.

This paper proposes an analytic model which is used to quantify the utilization of multiple pipelines in FPMP architectures with multiple thread slots. The model deals not only with pipeline dependencies but also with structure conflicts. The effects of four important parameters S, T, E, and F $(STEF)$ will be evaluated where S is the the number of thread slots, T is the number of instruction threads, E is the maximal number of cycles required by execution stage, and F is the number of functional units. The model accepts a general distribution for the interlock delays with multiple latencies same as in [5] and a general distribution for the different type of instructions which will be dispatched to different pipelines. The model predicts the utilization of multiple pipelines for different configurations, for example, for different number of thread slots and different number of resident threads. The model can also be used to quantify the speed-up ratio of the FPMP architecture compared to the FCMP architecture.

The paper is organized as follows. Section 2 introduces the FPMP architecture. Section 3 describes the analytic model. Section 4 takes four examples to discuss the effects of STEF. The final section concludes the paper.

2 A FPMP Architecture

Figure 1 shows a typical FPMP architecture. Differing from a conventional pipelined processor, FPMP contains state information of *T* threads. Each thread has its own program counter, status register, and register file. *F* functional units serve as *F* independent execution pipelines to support multiple instruction executions. There are *S* thread slots used for instruction dispatching. A *Thread Dispatch Unit* (TDU) selects *S* threads from *T* threads in an interleaved fashion. On every clock cycle, up to *S* instructions can be dispatched.

A separate instruction cache is provided for each of thread slots. In order to facilitate the interleaved thread selection, *T* instruction threads are distributed to *S* instruction caches equally. Each instruction cache contains on average of *T/S* instruction threads.

An *Instruction Scheduling Unit* (ISU) schedules the *S* instructions and issues them to the functional units if there are neither structure conflicts among *S* instructions nor instruction dependencies with previously issued instructions within a thread.

If two or more instructions require the same functional unit on the same clock cycle, then structure conflict occurs. The ISU selects one instruction to issue to the functional unit if dispatched instructions cause structure conflicts. A simple instruction scheduling strategy, $round\ robin$, is employed. Referring to Figure 2, a unique priority (`PRi`) is assigned to each thread slot (not thread). The instruction with highest priority will be considered first (`REQ1`). If the source operands are not available (`READY1`) for the instruction, an instruction from next thread slot (`REQ2` and `READY2`) will be examined. This procedure is repeated until a ready instruction is found (`ISSUEi`) or all the thread slots are examined. In order to give thread slots equal opportunity to be scheduled, the priorities are rotated on every cycle or several cycles (`R_CLK`).

Availability of source operand is checked by using the *scoreboard* mechanism. If the scoreboard bits of the source operands are cleared, a ready instruction is found. Then the source operands are read out from the correct register file and destination register's scoreboard bit is set. The scoreboard bit will be cleared at the final clock cycle of execu-

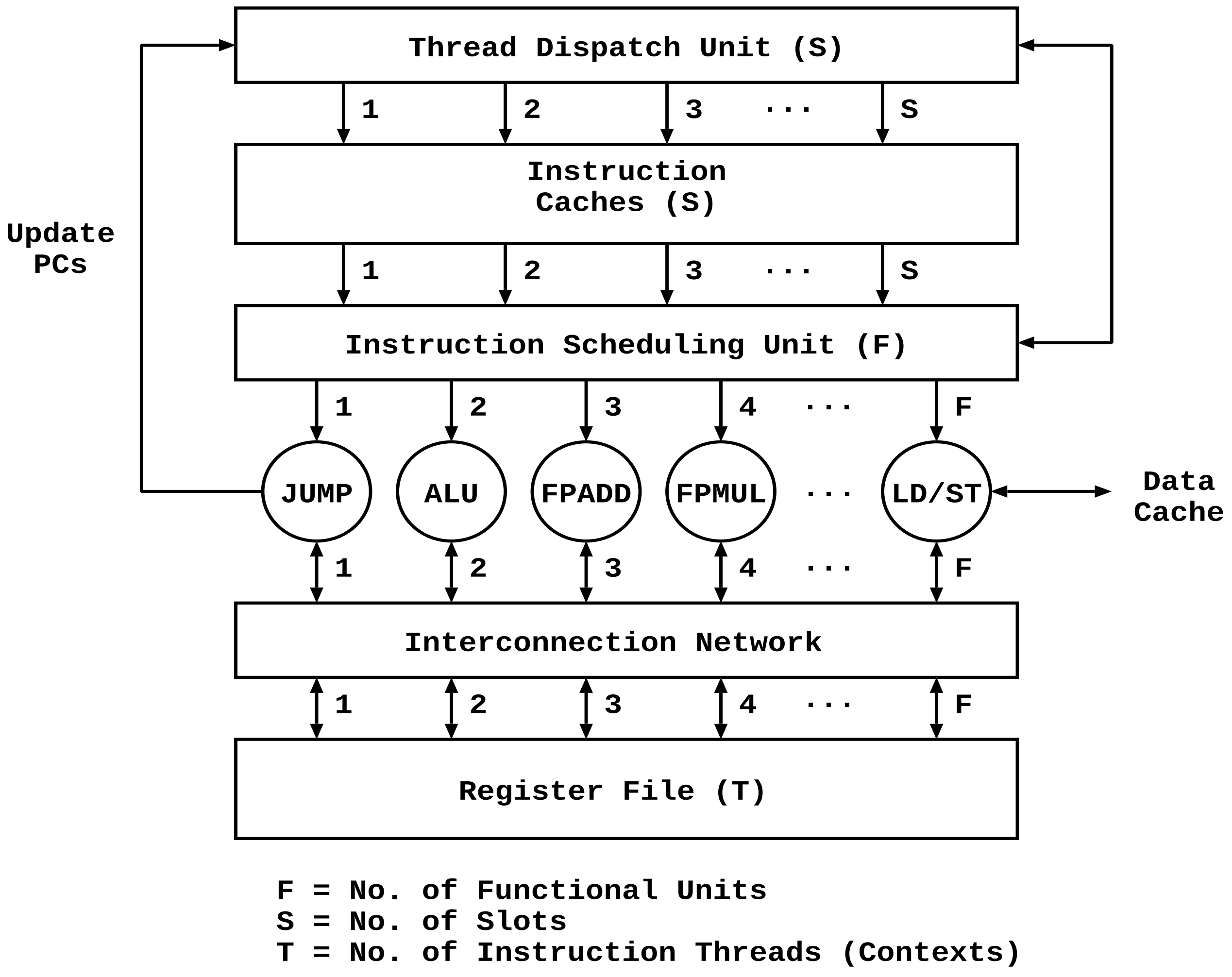

Figure 1: A finely parallel multithreaded processor architecture

tion stage. Thus the scoreboard bits could prevent incorrect data from entering into the pipeline.

The ISU is provided for each functional unit and FIFO registers for each of thread slot are provided in the ISU. Un-issued instructions will be held in FIFO, waiting for scheduling in the following clock cycle. The TDU is informed to stop fetching instructions from corresponding thread slots in the following clock cycle. Because the next instruction to the un-issued instruction is being fetched, the FIFO must have at least two registers for the thread slot. The total number of FIFO registers is $2 * S * F$, where S is the number of thread slots and F is the number of functional units. The functional units carry out the desired data operations, and the results are written back into register file.

An *Interconnection Network* (IN) is needed between the register files and the functional units. From the programmer's point of view, this physical FPMP is equal to S logical FCMPs and each FCMP executes *T/S* threads concurrently.

3 The Analytic Model

In this section, we will propose an analytic model for predicting processor-performance improvement and functional unit utilization for the finely parallel multithreaded processor architecture.

According to the FPMP architecture described in Section 2, we make the following assumptions:

- There are *T* instruction threads and all the threads are identical.
- There are *S* thread slots that can dispatch instructions simultaneously, and for fast instruction fetching, a

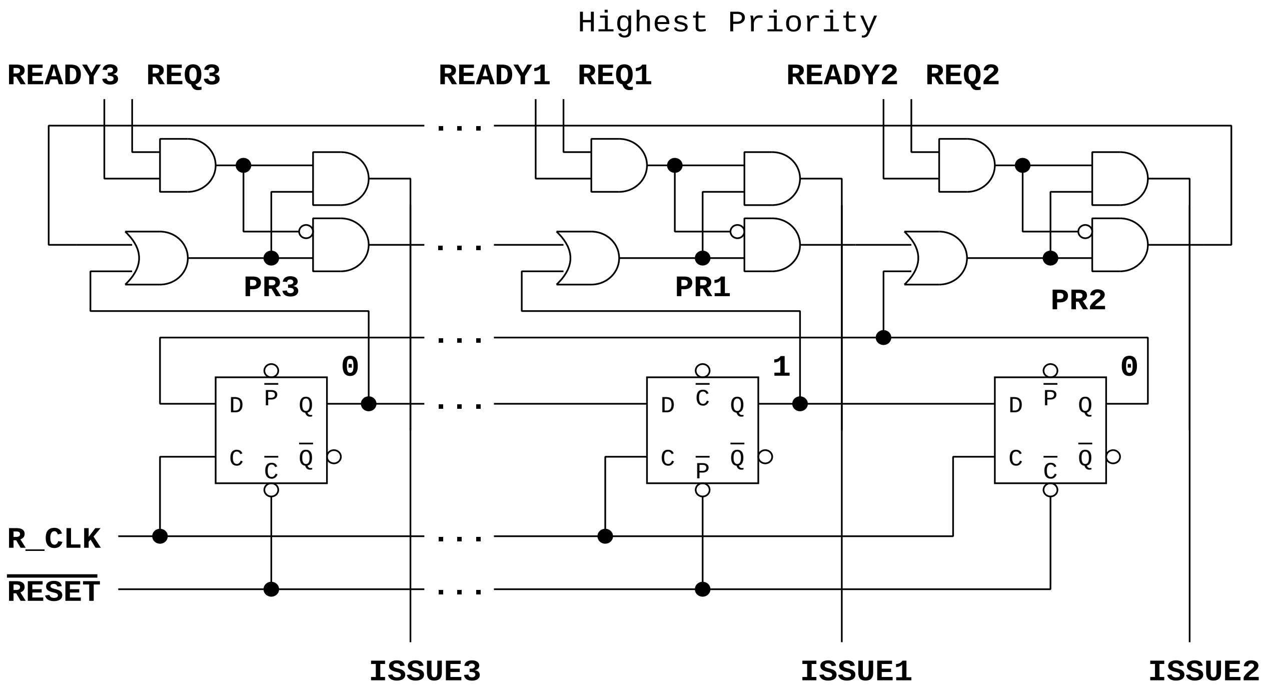

Figure 2: A round robin circuit for instruction dispatching

separated instruction cache is provided for each thread slot.

- The processor has more instruction threads than thread slots and the instruction threads are distributed equally to instruction caches: each cache holds average *T/S* threads.
- The *T/S* instruction threads are interleaved and one instruction is fetched from the selected thread per clock cycle.
- *F* functional units are provided to serve as *S* execution pipelines.
- All the functional units are effectively pipelined and are capable of accepting a new instruction in every cycle.
- All the instructions are divided into *F* classes: the jth class instructions will be executed on the jth functional unit *(j=1,2,...,F)*.
- The percentage of occurrences of the jth class instructions in dynamic instruction stream is ρ_j *(j=1,2,...,F)*.
- The distribution of interlock delays is described by the probability vector $p = (p_1, p_2, ..., p_E)$, where p_j is the fraction of instructions that have an interlock delay of *j-1* cycles after they were scheduled, *E-1* is maximum of interlock delay cycles, i.e., *E* is the maximal number of cycles required by execution pipeline stage.

First of all, consider a conventional processor: *T*=1 and *S*=1. The *CPI* (cycles per instruction) estimate can be easily obtained [5] as

$$CPI = 1 + \sum_{j=1}^{E} p_j * (j-1). \qquad (1)$$

For example, a *E*=4 processor has following distribution of interlock delays. 20% of instructions have a interlock delay of three cycles. 10% of instructions have a interlock delay of two cycles and 30% of instructions have a interlock delay of one cycle. The remaining instructions require no interlock delay. The *CPI* of the processor is $1 + 0.4 * 0 + 0.3 * 1 + 0.1 * 2 + 0.2 * 3 = 2.1$, as given by Equ. 1. The number of real executed instruction-per-cycle is 1/2.1=0.476, i.e., the processor utilization is 47.6%.

Now, consider that there are two instruction threads, *T*=2, that share the processor pipeline. Two threads are scheduled alternately, i.e., the *interval* of dispatching instructions from a thread is two cycles (Figure 3). Because independent instructions will be inserted into the instructions of a thread, the suffering of interlock delay will be alleviated for each thread. For example, in the *E*=4 processor mentioned above, 20% of instructions have a new interlock delay of one instruction cycle, as explained in Figure 3(b).

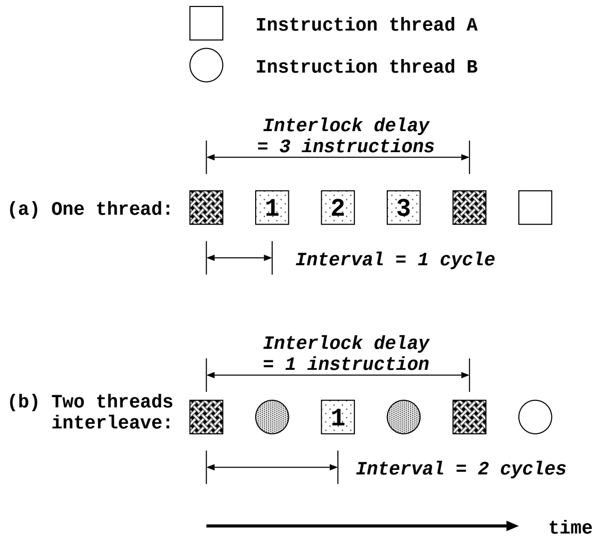

Figure 3: The interlock delay in a two-thread processor

Generally, in a processor with *T* instruction threads, the new value of interlock delays will be changed from $(j-1)$ cycles to $(j/T-1)$ cycles. Therefore, we can obtain the new *CPI*, denoted by C_T, for a *T-thread* processor as

$$C_T = 1 + \sum_{j=1}^{E} p_j * max(0, (\frac{j}{T} - 1)). \qquad (2)$$

In the mentioned *E*=4, *T*=2 processor, the C_2 is $1 + 0.4*0 + 0.3*0 + 0.1*0.5 + 0.2*1 = 1.25$. The number of real executed instructions per cycle is 1/1.25=0.8. The processor utilization increased from 47.6% to 80.0%, i.e., 68% of improvement was achieved. Note that in the $T \geq E$ processor, C_T reaches the minimum of one cycle, i.e., the processor utilization is 100%. However, if the processor has seven independent execution pipelines ($F = 7$), (ALU, Shifter, Load/Store Unit, Branch Unit, Floating-Point Adder, Multiplier, and Divider for instance), the total average utilization of all the pipelines is $(100/7)\% = 14.3\%$.

For the $S > 1$ processor, maximal S instructions can be dispatched and issued to F execution pipelines if neither instruction dependencies nor structure conflicts exist. The structure conflicts must be considered in the $S > 1$ processor. According to the assumptions described in the beginning of this section, when the number of the jth class instructions to be dispatched on the same cycle is less than or equal to one, there will be no structure conflicts. Hence, the probability of structure conflict $Pc_j(S)$ for the jth functional unit is

$$Pc_j(S) = 1 - \sum_{i=0}^{1} \frac{S!}{i!(S-i)!} \rho_j^i (1-\rho_j)^{S-i},. \qquad (3)$$

where i is the number of the jth class instructions to be dispatched on the same cycle, $\rho_j^i (1-\rho_j)^{S-i}$ is the probability of the case in which i instructions out of S instructions belong to the jth class, and $\frac{S!}{i!(S-i)!}$ is the binomial coefficient.

If the number of the jth class instructions to be dispatched on the same cycle is larger than one, at most one instruction can be executed. The maximal average number of the jth class instructions $\xi_j(S)$, which are dispatched to jth functional unit, can be calculated by Equ. 4.

$$\xi_j(S) = \sum_{i=1}^{S} \frac{S!}{i!(S-i)!} \rho_j^i (1-\rho_j)^{S-i} * 1 = 1 - (1-\rho_j)^S. \qquad (4)$$

The total number of instructions N, which are dispatched from S thread slots, can be calculated by considering all the instruction classes:

$$N = \sum_{j=1}^{F} \xi_j(S) = \sum_{j=1}^{F} (1 - (1-\rho_j)^S). \qquad (5)$$

We assume that the N instructions come from S thread slots equally. Thus, each thread slot dispatches N/S instructions. Note that $N/S \leq 1$. Because each thread slot has T/S instruction threads and dispatches N/S instructions per cycle, the interval of dispatching instructions from a thread is equal to $(T/S)/(N/S)$, i.e., there are $n = T/N$ virtual threads in a thread slot. In this case, similar to Equ. 2, the CPI for those N instructions, denoted by C_n, should be calculated by Equ. 6.

$$C_n = 1 + \sum_{j=1}^{E} p_j * max(0, (\frac{j*N}{T} - 1)). \qquad (6)$$

The total number of real executed instructions I can be calculated by dividing N by C_n, $I = N/C_n$, and the total average utilization of all the pipelines μ can be obtained by dividing I by F, as shown in Equ. 7.

$$\mu = \frac{N}{F * C_n}. \qquad (7)$$

The performance improvement is measured by the speed-up ratio ν which is defined as the ratio of execution time required by $S > 1$-slot parallel multithreaded execution to those by $S = 1$-slot concurrent multithreaded execution. The execution time of a given program can be expressed as the product of three terms: $i * c * t$, where i is the number of instructions required, c is the average number of cycles per instruction, and t is the time per cycle. Note that ν will asymptotically approach maximal F as T and S increase.

$$\nu = \frac{i * C_T * t}{i * (C_n/N) * t} = N * \frac{C_T}{C_n}. \qquad (8)$$

4 Examples and validation

As mentioned in Section 3, in the finely parallel multithreaded processor, there are four parameters that influence the processor performance and utilization of functional units. The four parameters are denoted by S, T, E, and F as explained in the following. S is the the number of thread slots that affects the capability of dispatching instructions per cycle. T is the number of instruction threads and it affects the interval cycles of threads interleaving. E is the maximal number of cycles required by execution stage and it affects the interlock delay cycles. F is the number of functional units that affects the structure conflicts.

In the following examples, we assume that $\rho_j = 100\%/F$ for $j = 1, 2, ..., F$ and $p_j = 100\%/E$ for $j = 1, 2, ..., E$. The first example (Figure 4 and 5) shows the effects of S and T then *F*=6 and *E*=4. Figure 4 shows the speed-up ratio calculated from Equ. 8 and Figure 5 shows the utilization of functional units calculated from Equ. 7. By increasing the number of thread slots S, the speed-up and the average utilization improves, but when the processor has four thread slots, significant improvement cannot be obtained by further increasing the number of thread slots. Also, we found that increasing the number of instruction threads T cannot always result in performance improvement and utilization improvement.

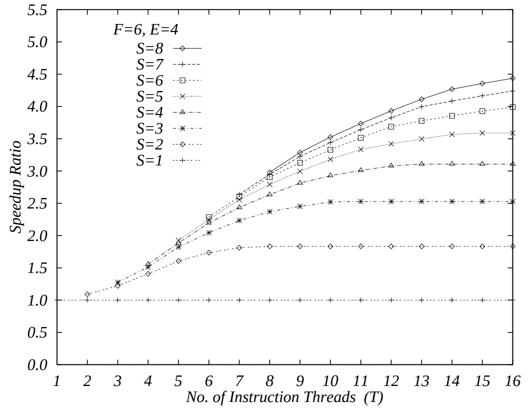

Figure 4: Speed-Up ratios for *F*=6 and *E*=4 processor

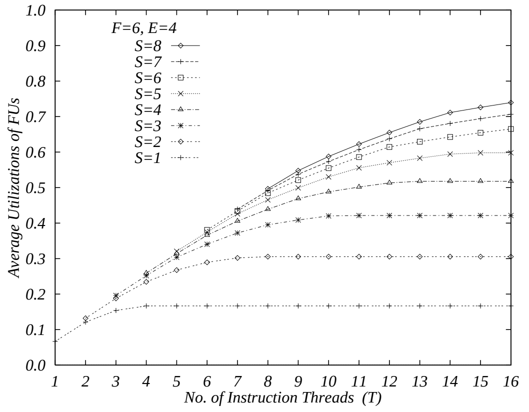

Figure 5: FU utilization for *F*=6 and *E*=4 processor

The second example (Figure 6 and 7) shows the effects of F and E, when $S = 6$ and $T = 12$, on speed-up ratio and average utilization. When E is large, increasing the number of functional units does not result in increased speed-up. In this case, by increasing the number of functional units, no performance improvement will be obtained (Figure 6), but the utilization of functional units will be decreased (Figure 7). Note that the case $E = 1$ and the case $E = 2$ have the same speed-up ratios and the same utilizations of functional units because each thread slot has T/S=2 instruction threads that can be interleaved to tolerate delays.

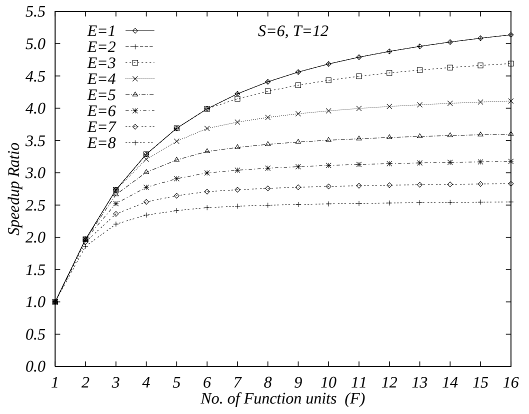

Figure 6: Speed-Up ratios for *S*=6 and *T*=12 processor

The third example (Figure 8 and 9) shows the effects of T and E, when $S = 6$ and $F = 6$. Note that there are upper bounds of speed-up ratio and average utilization. The upper bounds will be reached quickly when E is small. In this case, increasing the number of instruction threads does not result in increased speed-up and increased utilization.

The fourth example (Figure 10 and 11) shows the ef-

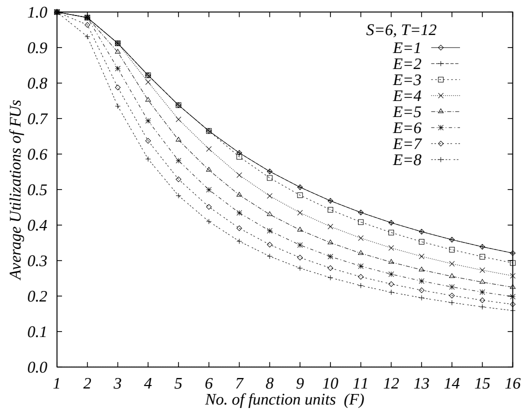

Figure 7: FU utilization for *S*=6 and *T*=12 processor

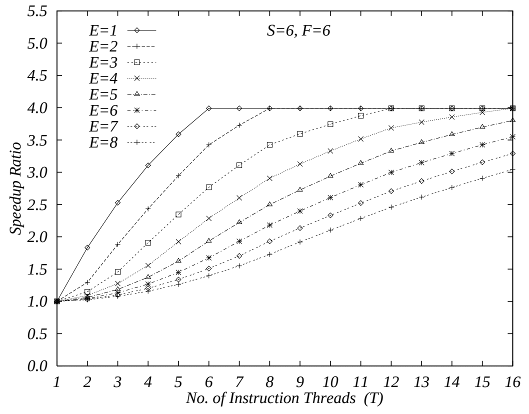

Figure 8: Speed-Up ratios for *S*=6 and *F*=6 processor

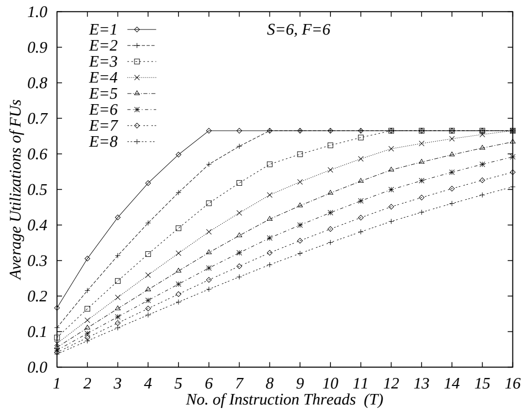

Figure 9: FU utilization for *S*=6 and *F*=6 processor

fects of S and E, when $F = 6$ and $T = 16$, on speed-up ratio and average utilization. In the case of $S = 4$ and E has a large value, by further increasing the number of thread slots, no performance improvement and utilization improvement will be achieved.

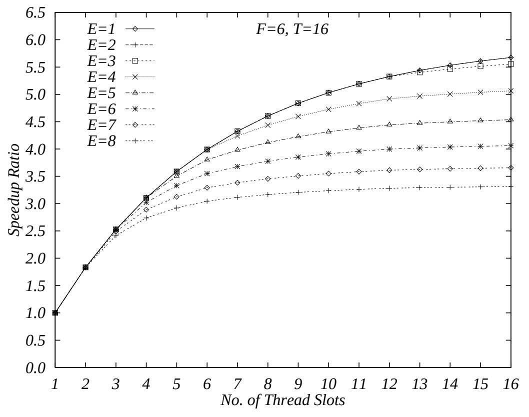

Figure 10: Speed-Up ratios for *F*=6 and *T*=16 processor

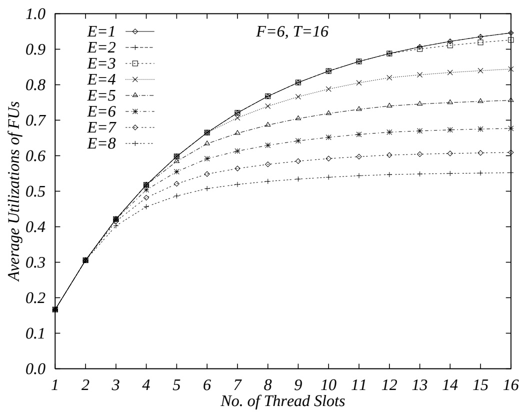

Figure 11: FU utilization for *F*=6 and *T*=16 processor

For the model validation, we choose a texture mapping program, that maps a texture pattern onto a 3D object's perspective projection in screen space. The main operations of the program are like as

$$u = \frac{ax + by + c}{\alpha x + \beta y + \gamma}, v = \frac{dx + ey + f}{\alpha x + \beta y + \gamma},$$
$$frame\_buffer[x][y] = texture\_pattern[u][v].$$

Seven functional units are arranged for executing the program.

- Integer unit performs integer arithmetic and logic operations.
- Load/store unit performs memory access.
- Branch unit evaluates condition codes and transfers control to new address.

- Floating-point adder performs floating-point add, subtract, and comparison.
- Floating-point multiplier performs floating-point multiply.
- Floating-point divider performs floating-point division.
- Floating-point convert unit performs floating-point/integer data type conversions.

We assume that all the functional units are capable of receiving a new instruction per cycle but have different execution cycles as following [6, 7]. The integer unit and the branch unit have one execution cycle. The load/store unit has two execution cycles when the data cache hit. The floating-point adder, multiplier, and convert unit have three execution cycles. The floating-point divider has thirteen execution cycles.

The program is compiled to assembly code and the multiple code streams are used as inputs to a FPMP simulator. As mentioned in Section 2, the processor has a separated instruction cache for each thread slot and a data cache for all the thread slots. The multiple streams are distributed equally to the instruction caches. In order to simplify the simulation, we assume that the cache accesses always hit.

The simulated results of speed-up ratio on texture mapping program are almost equal to the results generated by using analytic model, as shown in Figure 12. We get the instruction distribution ρ_j for $j = 1, 2, ..., F$ from the dynamic execution stream. As for the distribution of interlock delay, we assume that the numbers of interlock delay cycles are uniformly distributed over 0 to $E_j - 1$, where E_j is the number of clock cycles required by execution pipeline stage of *j*th functional unit for $j = 1, 2, ..., F$. Hence, we get the distribution of interlock delay p_i for $i = 1, 2, ..., max(E_1, E_2, ..., E_F)$ from following expression.

$$p_i = \sum_{j=1}^{F} [\frac{\rho_j}{E_j} ||(i \leq E_j)].$$

5 Conclusion

In this paper, we presented a *Finely Parallel Multithreaded Processor* (FPMP) architecture. The FPMP realizes multiple *Finely Concurrent Multithreaded Processors* (FCMPs) in a single processor environment. The FPMP contains multiple functional units and multiple thread slots. Each thread slot has multiple instruction threads and the threads are interleaved for dispatching. In order to evaluate the FPMP architecture, we proposed an analytic model that provides a quick prediction for performance improvement and a quick prediction for average utilization of multiple functional units. The model deals not only with instruction dependencies but also with structure conflicts. The effects of four important parameters in FPMP, $STEF$, were evaluated and the analytic model is validated by simulation.

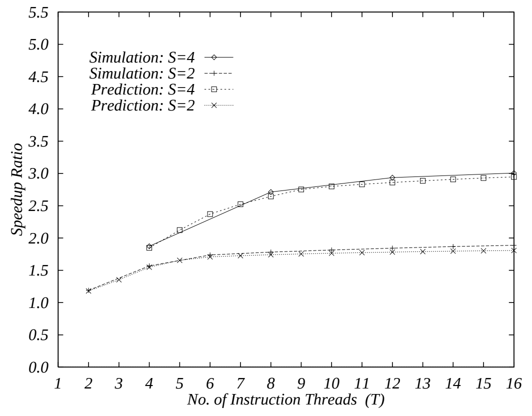

Figure 12: Simulated speed-up ratio for texture mapping program

References

[1] R. Guru Prasadh and Chuan-lin Wu, “A benchmark evaluation of a multi-threaded RISC processor architecture,” in *Proc. of the 20th Intl. Conf. on Parallel Processing*, 1991.

[2] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizaki, A. Nishimura, Y. Nakase, and T. Nishizawa, “An elementary processor architecture with simultaneous instruction issuing from multiple threads,” in *Proc. of the 19th Annual Intl. Conf. on Computer Architecture*, 1992.

[3] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz, “APRIL: a processor architecture for multiprocessing,” in *Proc. of the 19th Annual Intl. Conf. on Computer Architecture*, 1990.

[4] D. C. McCrackin, “Eliminating interlocks in deeply pipelined processors by delay enforced multistreaming,” *IEEE Trans. on Computers*, vol.40, No.10, Oct. 1991.

[5] P. K. Dubey, A. Krishna, and M. J. Flynn, “Analytical modeling of multithreaded pipelined performance,” in *Proc. of the 27th Annual Hawaii Intl. Conf. on System Sciences*, 1994.

[6] “M88110: Second generation RISC microprocessor user manual,” Motorola Inc., 1991.

[7] K. Diefendorff and M. Allen,“Organization of the Motorola 88110 Superscalar RISC Microprocessor,” in *IEEE MICRO*, April 1992.