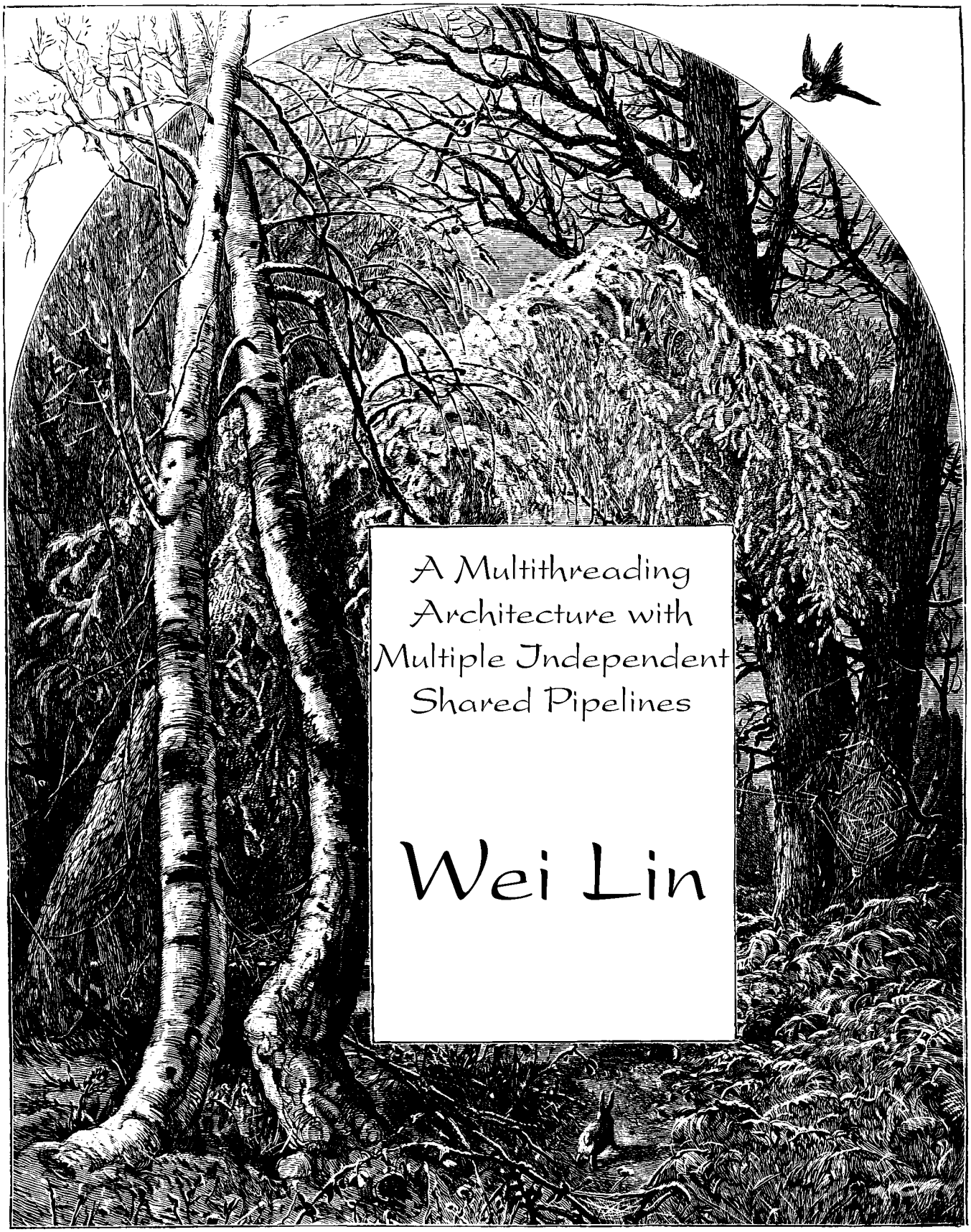


A Multithreading Architecture with Multiple Independent Shared Pipelines

Wei Lin

A Multithreading Architecture with Multiple Independent Shared Pipelines

Wei Lin
Peter M. Maurer
Computer Science Department, University of South Florida,
Tampa, FL 33620

Email: wlin@eggo.csee.usf.edu maurer@screamer.csee.usf.edu

Abstract

PSAM consists of a number of multithreaded pipeline processors to support parallel computation. Thread control is implemented using a distributed approach, in which each processor can independently initiate and terminate a thread. This thread control mechanism is based on dataflow model, and allows the degree of multiprocessing to vary with time. Although the basic processor is von Neumann- based, the overhead to support parallelism is very low.

Keywords: Multithreaded Architecture, Multithreading, Multiprocessors,Shared Pipelines,Dataflow

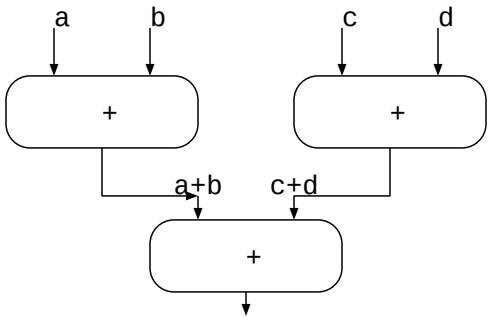
1. Introduction

A great amount of work has been done in building both fine-grained and coarse-grained parallel machines. One appealing idea is that of the multithreaded architecture[10,12,21]. It has been shown that sharing several active instruction threads on a pipeline can reduce the number of pipeline holes and increase the system throughput[11,18]. In order to support parallelism on a multithreaded machine, an efficient thread switching method must be provided. Conventional multithreaded machines[10,20] are based on static thread control, in which thread initiation and termination are decided statically by the compiler or operating system. The static approach lacks flexibility, and therefore tends to restrict parallelism. Furthermore thread switching may require a large amount of overhead.

As an alternative, there are dynamic thread control mechanisms[1,4] which are motivated by the dataflow computation model[5]. In this model, a loop-free computation can be represented as an acyclic graph in which any node can be fired as soon as each of its input arcs contains a token. Such graphs exhibit much inherent parallelism. In fact, it is theoretically possible for all instructions in the program to execute simultaneously, restricted only by the flow of data between instructions. However, the conventional von Neumann architecture lacks mechanisms to support this kind of fine-grained parallelism. Pure data flow machines[9], on the other hand, are quite costly to implement. Therefore, as an alternative architecture to support dataflow computation, the multithreaded architecture has received much attention in recent years [1,2,3,4,14,15]. In dynamic multithreaded machines, two special instructions are introduced to map the dataflow graph to multiple threads which can be run on an enhanced von Neumann machine[1]. One of these instructions provides the functionality to activate another instruction thread, while the other provides the reverse operation. By using these two instructions the system can allow the number of active threads to vary with time.

In the SAM architecture[1], these two special instructions are called *split* and *merge*. The Split instruction resembles a conventional conditional branch, but actually has two successors instead

of one. After the execution of a Split, execution continues at the branch address and with the instruction following the Split. The Merge instruction reverses the operation of the Split instruction. The Merge operation has a single operand which is usually initialized to zero. When the Merge instruction is executed and its operand is zero, it sets the operand to 1, and terminates the execution of the thread that executed it. When its operand is non-zero, it is reset to zero and execution of the thread continues with the instruction following the merge. The split and merge instructions allow the degree of parallelism to vary with time, just as does a dataflow architecture. Figure 1 shows an example of mapping a dataflow graph into two parallel threads by using *split* and *merge*.



a)Dataflow graph of e=(a+b)+(c+d).

```
1    split L1
2    add  a,b,t1
3    merge x
4    jump L2
5 L1:add  c,d,t2
6    merge x
7 L2:add  t1,t2,3
```

b) Evaluating e=(a+b)+(c+d) by split and merge.

Fig.1. Evaluation of an expression.

Compared with dataflow machines, multithreaded machines need explicit control for the parallelism, and thus require more overhead than the ideal dataflow machine. However this extra cost is quite small. As an enhanced von Neumann machine, a multithreaded pipeline architecture can avoid many difficulties encountered by dataflow machines[6].

The architecture in this paper is called the PSAM architecture. It supports a number of dynamic multithreaded processors that work in parallel. Each processor element has a multithreaded pipeline, and is based on the SAM architecture[16]. In SAM, the pipeline is shared by a set of instruction threads which are controlled by a thread queue. All threads are executed in parallel inside the pipeline, and are served on a round-robin basis. As shown in [16], when a sufficient number of threads are active, the pipeline remains full, provided that the machine has a high cache hit rate.

2. The architecture of SAM processor

The architecture of the SAM processor is described in [16]. In order to make the paper self-contained, we outline the key issues of this processor architecture in this section.

Two unique features of the SAM processor are the thread queue and the short memory. The thread queue is used to control the multiple threads, and the short memory provides an efficient way to support multiple thread environments. The storage interface unit is similar to that of most contemporary microprocessors with a few additional features. A storage operation queue, which buffers all main memory accesses issued by cpu, is integrated into the interface unit. The hardware of SAM is reasonably simple, so that the whole structure can be fabricated on a single chip. On the other hand, certain features of the architecture and instruction set make it possible to use several chips in parallel to support multiple shared pipelines working simultaneously.

The Thread Queue

Since in one SAM processor there are several threads executing simultaneously, each with its own program and data, some mechanism is needed to contain the information pertaining to a thread and to allow the various threads to be serviced. In the SAM architecture the thread queue is used for this purpose. The thread queue consists of a set of frames, each of which defines a thread. A frame structure is shown in figure 2. To support a thread, four registers are needed. The PC serves as program counter pointing to the next instruction to be fetched for this thread. The SP is the same as the stack pointer used in a conventional machine. (Note that except under exceptional conditions, stacks cannot be shared between threads.) The EP is an environment pointer, which will be explained more fully below. The PSW contains all the information about the current state of the thread, such as priority and condition codes. Another two 16-bit registers (PID and SID) are used to contain the identification number of the thread.

The state bits indicate the different states of the thread. *Ready* means that the thread can issue a new instruction to the pipeline. *Running* means that an instruction is being fetched from the thread. After the instruction is fetched, the thread will go back to ready state, so if there is only

one pipeline, at any given time, only one item in the thread queue can be in the running state. *Wait* indicates the thread is waiting for some event and it can not currently be executed. *Sleep* indicates that the thread in this frame is being dumped to main memory. As soon as the contents of the frame are sent to the storage queue, this slot can be used for a new thread (i.e., its state becomes *empty*). *Suspended* is similar to *wait*, and shows the thread can not proceed at the moment. It is used to avoid busy waiting during process synchronization. *Awaken* means a dumped thread is being restored into the cpu.

Figure 2. The frame in the thread queue.

The thread queue is divided into two parts. One part is in the cpu (local queue) and the other is in the main memory (dumped queue). Only those threads in the CPU can be used to select instructions for execution. The swapping of threads between the local thread queue and the main memory is discussed in section 4. The fetch cycle processes the local queue in a circular fashion. A token cycles through the thread queue, advancing one slot every clock cycle. Wait, Empty, Suspend and Sleep slots are skipped without wasting a clock cycle. The thread queue is one of the most important parts of this model. During every clock cycle a thread will be selected and, the six registers will be pumped into the pipeline. When a thread is generated it will be assigned to a slot in the thread queue. When a thread is merged or dequeued, the slot will be set to empty.

In contrast to many multithreaded architectures, SAM serves each active thread in a round-robin fashion. This is much easier to implement in hardware than other scheduling strategies[11]. Based on the thread queue mechanism and the short memory structure which will be explained below, switching threads has almost no cost in time. It also has been shown that under a careful design, round-robin can produce optimum throughput[16].

Short Memory

Another fundamental issue for a shared pipeline architecture is how to support multiple execution environments. Generally speaking an execution environment is a set of temporary memory locations used to distinguish one computation from another, even if the same code is being executed by two different processes or threads. In conventional machine, a set of cpu registers is used for this purpose. When an operating system switches from one process to another, or when subroutine is called, the cpu registers need to be saved. However, this approach cannot be used in the SAM multithreaded environment, because the environment may change with every clock cycle. Another approach is to use cache instead of a register file. Since the translation speed and the complexity of a pure data cache will affect system performance and cost, it was considered impractical for environment management[17]. Therefore, the SAM architecture uses a concept called *short memory* to solve this problem.

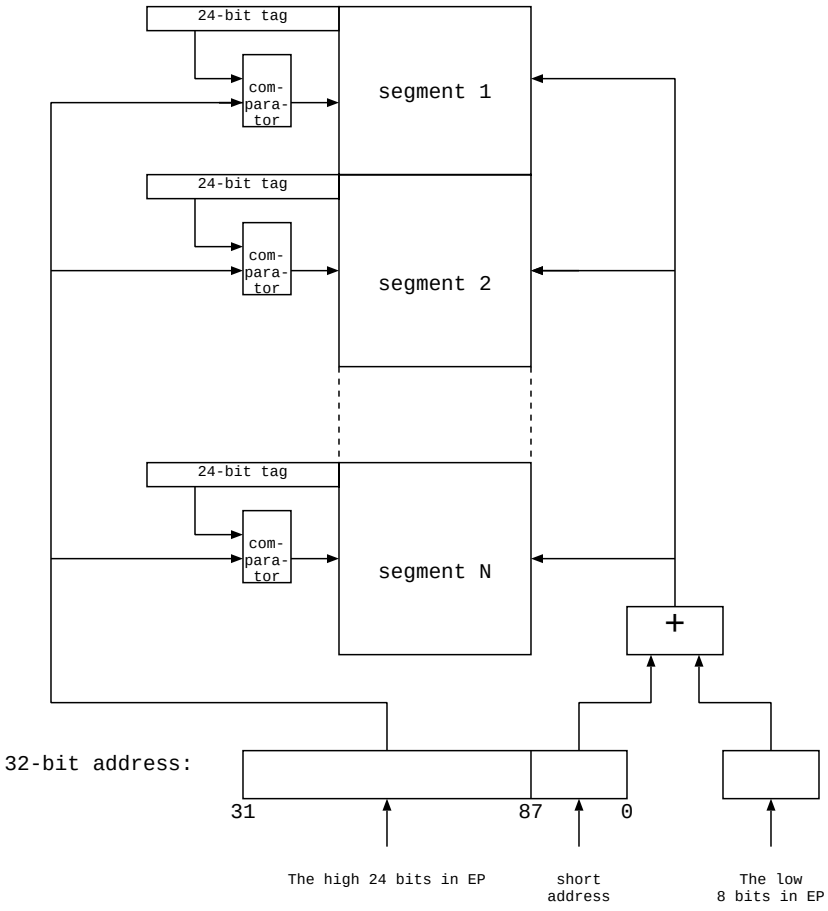


Figure 3. CPU short memory organization.

Instead of using registers, SAM has a small cache-like memory inside the cpu, which is organized as illustrated in Figure 3. Short memory consists of several blocks, each of which has 256 words. Each block is associated with a 24-bit tag. Each thread in SAM has a 256-word block of memory called its short address space. In addition to the program counter, each thread has an environment pointer (EP) that points to its short address space. This register is passed from stage to stage in the pipeline instead of passing an entire register file. Most instructions, as shown in figure 4, contain at least one short memory address, thus the EP will contain useful information for most instructions. A special instruction is provided for changing the contents of the EP, thus allowing the program to determine whether two threads execute in the same or different environments. Furthermore, a single thread can use several different EP pointers (one at a time) to effectively increase the size of its short memory beyond the 256 byte limit.

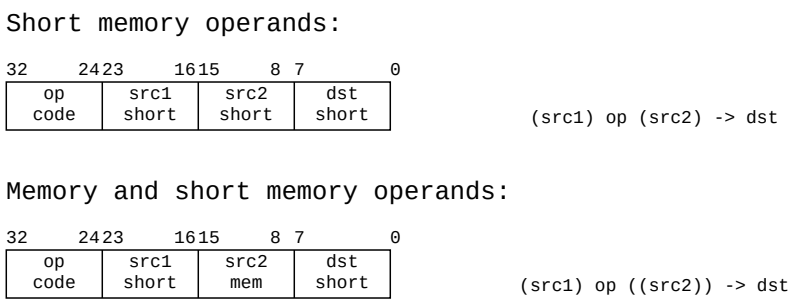


Figure 4. Instruction format

In the SAM architecture, operands can be fetched from the short memory in one clock cycle, and dual ports allows two operands to be fetched simultaneously. The short memory is on-chip, which allows operands to be fetched without going through a comparatively lengthy bus protocol. The full address of a short-memory operand is computed by adding the short address to the contents of the EP. As shown in figure 3, there are two steps in addressing short memory locations. The first step is to locate the correct segment, which is accomplished by comparing the high order bits of the full address with the segment tags of the short memory. The tag matching is fully associative. The second step is to locate the position in the segment using the eight low-order bits of the full address. These bits are *added* to the short memory address found in the instruction rather than

simply being concatenated to the 24-bit tag. This allows one short memory segment to be shared by several threads. These two steps can be carried out simultaneously. Instructions support two types of addresses: short addresses and long addresses. A short address refers to the short memory in the cpu, while a long address refers to a location in main memory. Each thread can use a portion of or the entire cpu short memory. It is also possible to partition short-memory blocks among threads that do not require 256 bytes of short memory.

The SAM short-memory can work in two modes. In first mode, the short memory is not transparent to the programmer, and works somewhat like a register file. A thread can allocate a block of short memory by setting the EP to a new value. As long as there is a free block in the short memory, it will be assigned to a thread by setting the its tag to the new value of the EP. Any read-reference to a short memory address that precedes the first write operation to that address will produce garbage. The value retrieved by the read operation will not necessarily be the same as that found at the corresponding location in the main memory.

Compared with the conventional register file approach[3,10], this strategy has the following advantages:

1) There is no need to swap the register files when switching the threads.

2) The size of the short memory space used for a thread is flexible, while in the conventional approach the size of the register file is fixed and every thread must have the same number of registers whether it needs them or not.

If the system is working in register file mode, dumping a thread to memory requires the saving of the short memory block associated with the thread.

In transparent mode, two additional features are added. The first is an auto-memory-load operation will be issued if an instruction is trying to read a short memory location which has not yet been written. The memory address for this load operation is obtained from the EP and the short address. This feature is the same as that in real cache. The second feature is that the short memory write operation, for some selected instructions, performs a "write through", i.e., the corresponding memory location will be written.

```
EXAMPLE: A short SAM program that adds the contents in memory location
         1000 and 2000, and store the result in 1002.

     1) Under the register file mode:

        SETEP 0;            set EP to 0.
        MOVIWS 1000, 0;     move 1000 to location 0 in a short memory
                            block.
        MOVIWS 2000, 1;     move 2000 to location 1 in the block.
        MOVIWS 1002, 2;     move 1002 to location 2 in the block.
        MOVLIS 0,0,3;       move the content in memory location 1000
                            to 3 in the short memory block. ((0))->3
        MOVLIS 1,0,4;       move the content in memory location 2000
                            to 4 in the short memory block. ((1))->4
        ADD 3,4,5;          add the contents in 3 and 4 and store in 5
        MOVSIL 5,0,2;       move the content in 5 to the memory
                            location addressed by the content of 2.


     2) Under the cache mode:

        SETEP 1000;         set EP to 1000.
        MOVIWS 2000,3;      move 2000 to location 3 in the block.
        MOVLIS 3,0,1;       move the content in memory location 2000
                            to 1 in the short memory block, with a write
                            through, i.e., (2000)->1001.
        ADD 0,1,2;          ((EP)+0)+((EP)+1)->(EP)+2.
```

Figure 5. Example for different short memory modes

As shown in the example in figure 5, the transparent mode leads to much simpler programming. Another obvious advantage for cache mode is that it can totally eliminate saving the short memory contents when the thread is dumped, provided that the program is written appropriately. On the other hand, write-through may cause a great deal of extra memory traffic, even if the thread is eventually dumped, since every write, including those for temporary variables, must be recorded in the memory. This may have a detrimental effect on system performance if cache mode is used indiscriminately [16]. Consequently, we restrict the write-through to a selected instruction subset, in order to minimize unnecessary writes.

2. The multi-processor architecture model

A PSAM architecture is defined by $\langle P, M, M_0, W, C \rangle$.

$P = \{P_1, P_2, ..., P_n\}$. is a group of processor elements. n is the number of processors in the system. Each $P_i, i = 1, 2, ..., n$, can be described by a pair: $\langle n_i, \pi \rangle$, in which n_i is the number of the

active threads in the local thread queue and π is a partition of the short memory for n_i threads.

M = $\{M_0, M_1, M_2, ..., M_{m-1}\}$ stands for a set of memory modules, where m is the total number of memory modules in the system.

M_0 is a special memory module, which is connected to all the processors. M_0 contains a memory processor, which is used to handle some special instructions. The purpose of using this module is to solve the thread synchronization problem. All the synchronization operations, such as merge, and critical section protection, will be executed in this module.

C = $\{C_1, C_2, ..., C_n\} \subseteq 2^M$, defines the memory sharing among the different processors. C_i, i=1,2,...,n, is a subset of M, which contains all the memory modules that can be directly accessed by processor P_i.

W describes the characteristics of the interconnection network among the processors and the memory modules. It is defined as follows:

$$W_{(m+n)*(m+n)} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} & w_{1,n+1} & \cdots & w_{1,n+m} \\ w_{2,1} & w_{2,2} & ... & w_{2,n} & w_{2,n+1} & \cdots & w_{2,n+m} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,n} & w_{n,n+1} & \cdots & w_{n,n+m} \\ w_{n+1,1} & w_{n+1,2} & \cdots & w_{n+1,n} & w_{n+1,n+1} & \cdots & w_{n+1,n+m} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ w_{n+m,1} & w_{n+m,2} & \cdots & w_{n+m,n} & w_{n+m,n+1} & \cdots & w_{n+m,n+m} \end{bmatrix}$$

The rows and columns from 1 to n stand for n processors, and n+1 to n+m for m memory modules.

$w_{i,j}$ is the time delay for the transmission from i to j. There is no direct connection between any two processors. So, $w_{i,i} = 0$, and $w_{i,j} = \infty$, for i $\neq$ j, and i,j = 1,2,...,n.

Figure 6 shows the structure of this parallel model.

The communication between two threads can be implemented in two levels in PSAM:

1) through shared short memory;

2) through shared main memory.

The first has very little cost. The second level suffers from both the memory and interconnection delay. If a thread is accessing a shared location in the main memory, SAM can set the its status to

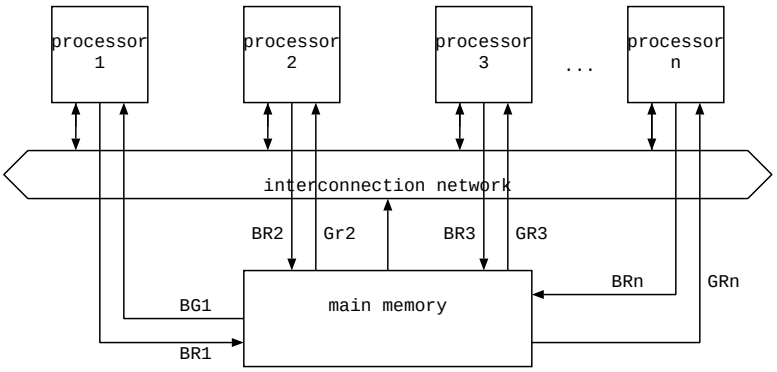


Figure 6. PSAM Architecture.

Wait so that no clock cycle will be wasted. When the memory access finished, the thread can be reactivated.

Computation is based on a distributed protocol. Thread control is implemented by *split* and *merge* instructions. The *split* will generate a new thread. If the local thread queue is full then a thread must be dumped to M_0. Any processor executing a *merge* instruction will send a signal to M_0. M_0 will perform the operation and signal the result to the processor. Based on this return information, the processor will terminate or continue the current thread. Any processor with empty slots in its local thread queue can pick up a dumped thread in M_0.

At the start, multiple threads can be initiated on several processors simultaneously. As an example, figure 7 shows a dataflow graph with several parallel paths executed on different processors.

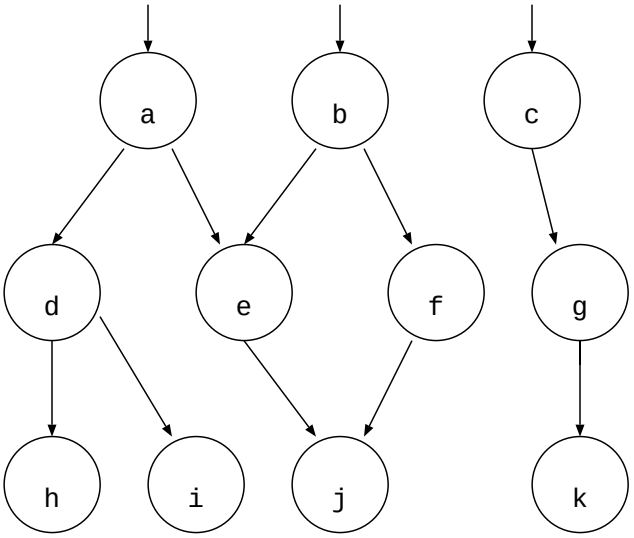
When a new thread is initiated, two things may occur:

1) If the local thread queue is not full, then the new thread can be immediately placed in an empty slot. In this case the cost, denoted by C_0, is small.

2) If the local thread queue is full, then a currently running thread is selected to be dumped. In this case, the thread information as well as its portion of short memory need to be stored in M_0. This cost is denoted by C_1.

Based on the above model, the throughput of the system can be calculated by the following formula:

$$\eta = \frac{\sum_{i=1}^{n} \eta_i}{n},$$




```
Program:

   At PE1:              At PE3:
      execute a;           execute c;
      split E;             execute g;
      execute d;           execute k;
      split i;             ......
      execute h;
       ......           E: merge e;
                           execute e;
   At PE2:             J: merge j;
      execute b;           execute j;
      split E;             ......
      execute f;
      goto J;
      ......
```

Figure 7. Execution a parallel program on 3 PEs.

η_i is the throughput of the ith PEs.

Suppose a job can be parallelized into X threads, which are divided into several groups, $\langle k_1, k_2, \cdots, k_n \rangle$ each of which is mapped to a PE. Then the speed of parallel execution of PSAM is given by

$$s = \sum_{i=1}^{n} \frac{1}{\tau_i} * \frac{k_i}{n_i}$$

in which τ_i is the instruction dispatch interval as defined in [16], n_i is the total number of active threads in the PE_i. s is the number of instructions per clock cycle.

The initiation and distribution of these threads will incur a certain amount of cost. The quantitative measure for this cost is difficult. In the following, we discuss a simplified case.

Theorem 1: *Suppose all the PEs are the same, and their local thread queues have the same length. The probability of that PEi's local thread queue is full, prob(k), is a constant, X, when the system is stable. During this stable period if an algorithm has N parallel threads and the data sharing does not affect the speed of the computation, then the cost of parallelizing N threads is given by the following formula:*

$$C = NC_0 + (C_1 - C_0)NX$$

PROOF: Let q_i, i=1,2,$\cdots$,n, be the probability of the thread i being dumped, where n is the number of processor elements in the system.

$$q_i = \sum_{k=1}^{n} [prob(k)\frac{1}{n}] = X.$$

$$\overline{q_i} = \sum_{k=1}^{n} [(1 - prob(k))\frac{1}{n}] = 1 - X.$$

The total cost

$$C = \sum_{i=1}^{N} [q_i C_1 + \overline{q_i} C_0] \quad = \sum_{i=1}^{N} [XC_1 + (1-X)C_0] \quad = NC_0 + NX(C_1 - C_0)$$

C_0 is contributed by the execution of *split* and *merge* instructions. $C_1(S, w_{n+1,i}, w_{i,n+1})$ is a function of S, the length of the short memory to be saved and the communication delays, $w_{n+1,i}, w_{i,n+1}, i = 1, 2, ..., n$. The details will be given in the next section.

The theorem shows the cost will linearly increase with the number of parallel threads, under the assumption that the thread full rate is a constant. This assumption may not be true when the number of active threads and the processors in the system is not large. In fact, when the number of active threads is small the cost function is slower than a linear function, when the number grows, it will approach a linear function.

4. Implementation Issues on Multithreading of Multiple Pipelines

Thread Swapping

The *split* and *merge* mechanism theoretically allows unlimited parallelism in the execution. However, the thread queue size in the cpu can not be unlimited. Hence the two part structure of the thread queue is introduced. In order to support this structure, a swapping protocol is required.

There are two steps required to dump or restore a thread. The first step is to save or load the frame in the thread queue related to the thread. The second step is to save or load the short memory space for the given thread. The time required for these operations is roughly comparable to the time required for a conventional process switch, and in some cases can be completed much more quickly.

The hardware used for this purpose in SAM includes three special registers. The register *Qtail* points the end of the thread queue in the memory. The *Qhead* serves as a pointer to the head of the part of the queue in the memory. Whenever a thread is dumped, *Qtail* is incremented by one; whenever a thread is restored from the memory, *Qhead* is decremented by one. The increment and decrement of *Qhead* and *Qtail* is on a module basis. Similarly, another pair of registers *Dhead* and *Dtail* is used to point to where to load and restore the data of the dumped short memory and where to store the short memory contents of the thread. All of these registers are instruction addressable, and can be set by a system command. All the decrement and increment operations on the registers are performed by the hardware in one clock cycle. In addition, two fields are added in PSW. They are *offset* and *count* as shown in figure 8. The *offset* field shows the starting address in short memory space that needs to be saved during dumping, while *count* indicates the number of words needed to be saved. As with the other fields, these two fields can be set by an instruction.

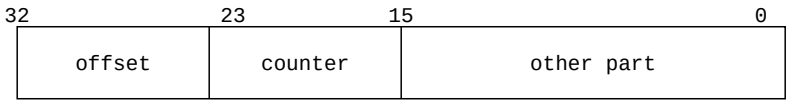


Figure 8. Additional fields in PSW.

There are several possible approaches to implement dumping and restoring in a multithreaded environment. One approach is by hardware, somewhat like an interrupt response in a conventional machine. As soon as a dump begins, the cpu could insert a block of operations to save the thread-queue frame and the short memory. However, SAM does not take this approach. The reason is that inserting a block of consecutive memory operations into the memory queue could seriously degrade pipeline performance. A second approach is to use an interrupt subroutine to do all the saving and loading. This will use less hardware but take more time than the pure hardware approach. It might also degrade the pipeline performance somewhat, as will be shown later. The third approach, which is adopted by SAM, combines the hardware and software approach, and yields better performance. (Note that even in the worst case, the effect on performance will be no worse than that which would be experienced on a conventional single-threading processor.)

In the cpu local thread queue, one slot is reserved for dumping. This thread is normally in the inactive state. When a *split* instruction or an interrupt[17] causes an overflow in the thread queue in cpu, this reserved thread is activated, and executes the program shown in figure 8. The first five instructions save the frame registers, and the rest store the short memory into the main memory. The information about size and the address of the short memory can be found in PSW. Since the information required to execute these operations is all inside the cpu, it would be redundant to put these instructions in memory and then reload them. The hardware itself generates these instructions. To do this, a 3-bit counter and a state bit are needed for each slot. When a thread-dump is activated, during the first five cycles (counter 1 to 5) the hardware will issue the first five instructions illustrated in figure 9(a) to save the frame registers. Then it will proceed to save the contents of the short memory. (This is only necessary in register-file mode.) Each write to main memory will be treated as an instruction. The counter field in the PSW will be decremented for each write until the size becomes zero. For each word of short memory to be dumped, the *Dtail* register is used to generate the main memory address. The final instruction is *deq*, which will reinitialize the slot and inactivate it.

```
STO ID;
STO EP;
STO SP;
STO PC;
STO PSW;

STO S;
......
STO S;

DEQ;
```

(a) The program for dumping a thread.

```
LOD PSW;
LOD PC;
LOD SP;
LOD EP;
LOD ID;

LOD S;
.......
LOD S;

normal execution
```

(b) The program for restoring a thread.

Figure 9. Dumping and restoring a thread.

Loading a thread is somewhat different from dumping a thread. when there is an empty slot and there is a thread in the memory portion of the queue, the hardware will set the empty slot to the *awaken* state. Slots in the *awaken* state will be scheduled as if they were in the *ready* state. While

in the *Awaken* state, the thread will first execute the program shown in figure 8(b). It will load the frame and short memory contents. Then it will change the *awaken* state to the *ready* state.

There are two protocols used in the SAM architecture which correspond to the the two modes of the short memory. In the register file mode, both frame registers and short memory need to be saved or loaded. In the write through cache mode, only frame registers need to be saved, and short memory will be reloaded when a cache miss occurs. It seems that a write through cache has a big advantage over the register file in this respect. However, further analysis, as shown below, reveals that there are drawbacks to this scheme.

We define the *instruction dispatch interval* to be the minimum number of clock cycles required before the next valid instruction can be dispatched. As presented in [16], the instruction dispatch delay caused by a cache miss is

$$\tau_c^i = (a-1)x[p_c(l-p_d)+p_d] + lx/2.$$

Where a is number of active threads in the thread queue, and b is the number of the stages in the pipe. The probability of a cache miss for the thread i at time t is p_c^i(t), while p_d^i(t) is the probability of a data coherence problem occurring in the thread i at time t. The cache block size is denoted by l, while x is the number of clock cycles required for a memory operation.

The instruction dispatch delay caused by a data coherence problem is

$$\tau_d^i = (a-1)x[p_c(l-p_d)+p_d] + x$$

The instruction dispatch delay caused by a branch is

$$\tau_d^i = \text{b}$$

For round robin the worst-case dispatch interval for a thread i, is given by the following formula.

$$\tau^i(t) = max\{\tau_c^i(t), \tau_b^i(t), \tau_d^i(t)\}$$

When thread dumping and restoring occurs, these parameters become as follows.

Theorem 2: *Let $v_c^i(t), v_d^i(t), v_b^i(t)$ be the dispatch delays caused by cache misses, data coherence and branches for a thread i, when thread dumping or restoring is taking place. Then*

$$v_c^i = (a-2)x[p_c(l-p_d)+p_d] + x + lx/2.$$

$$v_d^i = (a-2)x[p_c(l-p_d)+p_d] + 2x$$

$$\tau_d^i = b.$$

The proof of the theorem is straightforward based on the formulas in [16]. Due to the limited space, we skip the proof.

Definition *For a given thread i, the dispatch delay factor is defined as* $r^i(a) = \lfloor \tau^i/a \rfloor$.

In the case of dumping or restoring a thread, it is denoted as $R^i(a) = \lfloor v^i/m \rfloor$.

In round-robin scheduling, $r^i(a)$ identifies when the next instruction can be dispatched. If $r^i(a) = 0$, then an instruction can be dispatched during the next clock cycle.

Theorem 3 *Based on the above thread swapping protocol, when a goes to infinity,*

$$R^i(a) = r^i(a).$$

PROOF:

$$R^i(a) = r^i(a) + x(1 - [p_c(l-p_d)+p_d])/a$$

$$\lim_{a\to\infty} R^i(a) = \lim_{a\to\infty} r^i(a).$$

When $p_c(l-p_d)+p_d \leq 1$ and processor has enough active threads[16], theorem 2 shows that the register file based thread swapping protocol may not affect the pipeline performance. So the total cost for swapping is 10+2S cycles, where S is the size of short memory. No memory space space is required, except that is used for store the frame contents and short memory.

In transparent mode, the swapping operation is much simpler if a write through is used. However, write through will greatly increase the p_d, which will affect the important condition: $p_c(l-p_d)+p_d < 1$. In that case, the overall performance could be greatly degraded[16]. Although we restrict the write through to certain instructions to reduce the unnecessary memory accesses, this selected write through may not be able to cover all applications.

Another problem with thread swapping is the execution of *split* instructions. During the swapping operation, *split* must be blocked. Since there is only a limited number of threads active in the cpu thread queue and the swapping procedure for the thread frame is very short, a split buffer of a limited size in the cpu can be used to buffer the *split* operation during the swapping. An alternative solution is to inhibit all the *split* execution during the thread swapping. This will affect the pipeline performance, and has not been counted in the above theorem. However, since the *split* is not a high frequency instruction, its effect will be minimal and can be ignored. The dumped threads can be dispatched to other pipeline processors as long as they have empty slots in their cpu thread queues. The cost for dumping, as defined in previous section, equals

$$C_1(S, w_{i,n+1}, w_{n+1,i}) = (5 + S)(w_{i,n+1} + w_{n+1,i})$$

Main Memory Organization and Interconnection Network

In PSAM all processors and the memory modules are connected to a high speed interconnection network. A special memory module M_0 will be used for all important system information, such as the thread queue, etc. There is a special system path connecting all the PEs and the basic memory unit, M_0. In contrast to the conventional machine, this system path is controlled by the main memory unit most of time. Each processor that wants to use the system path needs to make a request to the basic memory unit before using the system path. The request and grant are transferred through the BR and BG lines as shown in figure 6.

The thread queue in PSAM is distributed among all the processors and the shared main memory. It is of fundamental importance to keep thread generation and termination consistent. To do this, some special arrangement in the memory structure is needed. The basic memory module M_0 consists of two parts: storage part and a memory processor. The memory processor contains the *Qhead, Qtail, Dhead* and *Dtail* as mentioned in the above section. The idea of using memory processor is to provide a mechanism for distributed processing the *merge* and *deque* instructions, protection of critical sections, and thread switching among the different pipelines.

The Implementation of Merge

Compared with the *split* instruction, the *merge* is much more difficult to implement. Several approaches[17] have been considered. In multiple pipelines all the merge operands must be sharable by all *merge* instructions issued by any thread in an arbitrary pipeline. When a *merge* is executed by a pipeline, the following events will occur:

1) The processor executing the *merge* will request bus control, and set the thread state to *waiting*

.

2) If the request for the bus is granted, the processor will send the merge operand and a merge signal to the memory processor. If the bus is busy, the merge operand will be buffered in the storage unit and wait for bus cycle. (note: it will not be buffered in the storage queue, because merge operation has the highest priority in bus access.)

3) As soon as the memory processor get the merge signal and the merge operand, it will check the state of the merge operand in the main memory, and send an "on" or "off" signal to the processor.

4) If the processor gets an "on" response from the memory processor, it will set the *waiting* thread back to the ready state. If the response is "off", the processor will erase the *waiting* thread.

Two Levels of Thread Control

Although the PSAM model provides the dynamic thread control at the hardware level, the software level control may help to reduce the overhead as stated in theorem 1, and thus increase the speed. In addition to the ordinary split instruction, two extended types are proposed in PSAM. Both of these instructions can reduce C_1.

The first is a split without short memory dumping. If the thread generated by this split is dumped, its short memory will not be saved and this thread can be set back to the current PE only when it is restored later. This can protect short memory sharing.

The second is a split that will cause the generated thread be dumped to the memory directly and be re-scheduled to another PE. This instruction supports coarse-grain parallelism. The programmer can use this to initiate an independent process on another PE.

In consequence, the M_0 should be able to restore these threads to the appropriate hosts. That is another reason why PSAM needs to use a special memory module with a memory processor.

4. Conclusions

The PSAM architecture supports multiple SAM processors working in parallel with minor cost. The distributed thread queue architecture, short memory concept, and memory processor provide an efficient thread switching and synchronization mechanism. Programming on PSAM is close to conventional programming. It can efficiently support general purpose computation in parallel.

Compared with dataflow machine, PSAM architecture has a much lower cost both in terms of hardware cost and the research and development required to create a fully functional hardware model. PSAM provides an efficient architecture to support dataflow computation, while its hardware structure is almost as simple as that of a conventional RISC machine[6,7]. The cost of synchronization among the threads is simple and fast. The only overhead is the split and merge instructions themselves. Another advantage over the dataflow machine is that since the hardware architecture is still an enhanced Von Neumann architecture, the memory management techniques that have been successfully used in most existing commercial machines are feasible in the PSAM architecture. In addition, the purely sequential program is supported by the PSAM architecture, and if desired, the PSAM could be run as if it were an ordinary sequential processor. Thus it is possible to maintain software compatibility with existing sequential machines.

Cache miss rate or thrashing is one of the major problem in the multithreaded machines. The cache miss rate grows when the number of active threads increases[20,18]. When a large number of irrelevant threads are running theoretically the whole virtual memory space may be active. Thus the thrashing is unavoidable. However in our dynamic multithreaded machine the computation is based on dataflow graph. Under this model the active threads on one pipeline and its cache are relevant. In many cases they are the same code running on different data. With the help of

good programming and the thread scheduling policy, the cache miss rate may be lower than that in conventional machines.

A common problem in multiple processor system is cache consistency. Although PSAM contain multiple caches, consistency problems can be avoided by careful programming. Since SAM provides two addressing modes: short address and long address[17], users can use long addresses (main memory addresses) for shared data. Therefore, all reads or writes these locations will go to main memory. This might reduce the speed of execution. However, since in a single SAM processor multiple threads are active simultaneously in the pipeline, any thread waiting for memory access will be set to the *waiting* state and the other active threads will fill up the time slot. So the overall system throughput will not be degraded.

Another important possibility is this that the SAM architecture may be able to reduce a large amount of operating system overhead. We believe that the thread queue structure and short memory concept provide better support than the conventional sequential machine for multitasking and multithreading.

The PSAM is an on-going research project. Many approaches are still under investigation. We are aiming at a high performance, low cost, general purpose multiple processor system.

Reference

1. P.M. Maurer, Mapping the Dataflow Computation Model into an Enhanced Von Neumann processor, Proceedings of International Conference on Parallel Processing, 1988,

2. H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads, The 19th ACM International Symposium on Computer Architecture, May 1992, pp.136-145.

3. M. Sato, Y. Kodama, S Sakai, Y. Yamaguchi, and Y. Koumura, Thread-based Programming for the EM-4 Hybrid Dataflow Machine, The 19th ACM International Symposium on Computer Architecture, May 1992, pp.146-155.

4. R.S. Nikhil, Arvind, Can dataflow Subsume von Neumann Computing?, The 16th ACM Annual International Symposium on Computer Architecture, May 1989, pp.262-272.

5. J.B. Dennis, Data flow supercomputers, IEEE Computer Magazine, Nov. 1980, pp.48-56.

6. D.D. Gajski, D.A. Padua, D.J. Kuck, and R.H. Kuhn, Second opinion on data flow machines and languages, IEEE Computer Magazine, Feb. 1982, pp.489-500.

7. D. Patterson, C.H. Sequin, A VLSI RISC, IEEE Computer, Vol.15, No.9, 1982, pp.8-22.

8. J.L. Hennessy, VLSI Processor Architecture, IEEE Transactions on Computers, Vol. C-33, No.12, 1984, pp.1221-1246.

9. Arvind, L. Bic and T. Ungerer, Evolution of Data-Flow Computers, Advanced Topics in Data-Flow Computing, Prentice Hall, 1991, pp.3-34.

10. B.J. Smith, Architecture and Applications of the HEP Multiprocessor Computer System, Real Time Signal Processing IV, Proceedings of SPIE, 1981, pp.241-248.

11. M.K. Farrens and A.R. Pleszkun, Strategies for Achieving Improved Processor Throughput, The 18th ACM International Symposium on Computer Architecture, May 1991, pp362-369.

12. W.J. Kaminsky and E.S. Davidson, Developing a Multiple-Instruction-Stream Single-Chip Processor, Computer, Dec. 1979, pp.66-76.

13. R.F. Cmelik, et al, An Analysis of MIPS and SPARC Instruction Set Utilizat ion on the SPEC Benchmarks, Proceeding of the 4th Architectural Support for Programming Lan guages and Operating Systems, April 1991.

14. R.S. Nikhil, G.M. Papadopoulos, and Arvind, *T: A Multithreaded Massively Parallel Architecture, The 19th ACM International Symposium on Computer Architecture, May 1992, pp.156-167.

15. G.M. Papadopoulos and R. Traub, Multithreading: A Revisionist View of Dataflow Architecture, The 18th ACM International Symposium on Computer Architecture, May 1991, pp.342-351.

16. Wei Lin, Peter M. Maurer, SAM: A Multithreaded Pipeline Architecture for Dataflow Computing, Technical Report, Computer Science Department of USF, August 1992.

17. D.R. Ditzel, H.R. McLellan, "Register Allocation for Free: the C Machine Stack Cache," Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 48-54.

18. Anat Agarwal, "Performance Trandeoffs in Multithreaded Processors", IEEE Trans. on Parallel and Distributed Systems, Vol.3, No.5, Sept. 1992, pp525-539.

19. R.H. Halstead and T. Fujita, " MASA: A multithreaded processor architecture for parallel symbolic computing", Proc. 15th Annual International Symposium on Computer Architecture, June, 1988. pp443-451.

20. Anant Agarwal, Mark Horowitz and John Hennessy, "An Analytical Cache Model", ACM trans. on Computer Systems, Vol.7, No.2, May 1989, pp.184-215.

21. Cail Alverson, Robert Alverson, et. al, "Exploiting Hetergeneous Parallelism on a Multithreaded Multiprocessor", Proc. of International Conference on Supercomputing, 1992.