LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Synchronization and Pipeline Design for a Multithreaded Massively Parallel Computer

Computation Structures Group Memo 343-1
March 1992

Shuichi Sakai

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Synchronization and Pipeline Design for a Multithreaded Massively Parallel Computer *

Shuichi Sakai †

Computation Structure Group
Laboratory for Computer Science
Massachusetts Institute of Technology

January 27, 1995

Abstract

This paper examines two basic functions in a massively parallel computer - synchronizations and pipelining - and proposes efficient implementations. The data-driven synchronization mechanisms which currently exist are carefully analyzed from the viewpoint of efficiency and hardware complexity, and the optimized synchronization mechanism is proposed. The pipeline structure for a massively parallel computer containing the proposed synchronization is presented. Performance improvement methods for this pipeline are proposed, cost-effectiveness of the proposed method is considered, and related issues are listed. Lastly, future problems including software issues are presented.

1 Introduction

The main stream of research and development of dataflow architectures is being shifted from pure dataflow computers toward multithreaded computers. This tide includes : (1) the activities at MIT, from TTDA [2] to the Hybrid Architecture [5] and Monsoon [8][13], and from Monsoon to *T [15]; (2) the activities in ETL, from the SIGMA-1 [4] to the EM-4 [7], and from the EM-4 to the EM-5 [12]; and the activities in Sandia National Laboratory, from the Epsilon-1 to the Epsilon-2 [9]. Clearly these multithreaded architectures still maximally exploit the advantages of dataflow architectures; they tolerate long latencies by cheap local synchronization, and they are suited to the programming model which

*This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

†also belongs to the Computer Architecture Section, Computer Science Division, Electotechnical Laboratory, 1-1-4, Umezono, Tsukuba, Ibaraki 305, Japan. Email address: sakai@abp.lcs.mit.edu

naturally extracts maximum parallelism. In addition, they are also advantageous for local sequential computation, since they include a von Neumann architecture (typically, a RISC architecture) which efficiently executes sequential threads with a set of registers and an advanced-control pipeline.

The concept of multithreading is not exclusive to the extension of dataflow architectures. For instance, the Denelcor HEP [1] and the Tera Computing System [6] are multithreaded computers in the sense that they execute and control multiple threads in a single pipeline. Dally's J-machine [16] does not interleave multiple threads, but it can switch between threads very quickly; thus, we can say that it actually supports the multithreaded computation. In addition, Dally's new machine, called the M-machine, has a mechanism of thread-interleaving [17] where many threads can exist inside a processor chip at the same time.

In all of these machines, the common and central design decisions are how to design an efficient synchronization mechanism and how to pipeline the machine? These two functions seriously influence the performance of the multithreaded system.

This paper examines the functions of synchronization and pipelining, and proposes ultimately efficient mechanisms for them. Firstly, the data-driven synchronization mechanisms which were already proposed are carefully reconsidered from the viewpoint of efficiency and hardware complexity, and the new synchronization mechanism is proposed (Section 2). Secondly, the pipeline structure for a processing element of a massively parallel computer including the proposed synchronization is presented (Section 3). Thirdly, methods to improve performance proposed this pipeline are proposed (Section 4). In Section 4, cost-effectiveness of synchronization and pipelining are also considered, and related issues are listed. Lastly, future problems including software issues are presented.

The mechanisms proposed here can be implemented using the current VLSI technology. We can provide an ultra-high performance massively parallel system such that the communication/synchronization performance is strictly comparable to the computation performance, and the latency of each processor (time from packet input to result packet output) is fairly short, e.g. less than or equal to 5 RISC clocks.

2 Optimizing Data Driven Synchronization

2.1 Data-Driven Synchronizations

This subsection summarizes the data-driven synchronizations which have already been proposed and reexamines their merits and defects. [1]

Earlier version of dynamic dataflow machines used associative mechanisms, such as hashing mechanisms, to implement data-driven synchronization (token matching). For instance, Manchester Dataflow Machine [3] uses a parallel hash table to provide a pseudo-associative access for token matching. Another example is the SIGMA-1 [4] system whose processor contains a chained hash mechanism. Matching by hashing has the following advantages.

[1] Here, the author concentrates on the data-driven synchronizations on a dynamic dataflow model. We have another model, *a static model*. Although a modern "static architecture" [10] is also a challenging architecture, this paper does not include the architectural optimization of it.

- Efficient use of memory.
- Flexibility to implement different matching schemes; sticky token matching for example.

On the other hand, there are several drawbacks.

- Complexity.
- In case of a hash miss, pipeline bubbles are generated or an exception occurs.
- Difficulty to understand what is happening.

In the next generation dataflow machines, these defects were eliminated by introducing a frame-based matching. The explicit token store (ETS) in Monsoon [8], the direct matching in the EM-4 [7] and the direct match in the Epsilon [9] are such mechanisms.

Here the former two are briefly described and criticized in preparation for proposing a new scheme of a data-driven synchronization.

(1) Explicit Token Store in Monsoon

Figure 1 illustrates how matching is operated in Monsoon. The method by which tokens find their partners is called the Explicit Token Store (ETS) [8].

In Monsoon, an area for matching is exclusively reserved for a single function instance. This area is called a frame and a frame is allocated at the function invocation time. Matching occurs at a certain word in the frame.

In ETS, every token has a field of IP and FP. The IP is an instruction pointer which points the address of the instruction invoked by this packet, and the FP is a frame pointer which points the top of the frame. As each instruction has a field which indicates the offset of matching, the matching address can be calculated by adding FP and the offset (Figure 1). Each data word in the frame is associated with a flag memory word where the synchronization flag is stored. In the case of a dyadic matching, Monsoon test-and-sets this flag in a single cycle. If the partner has already arrived, it clears the flag, reads the partner data and executes an instruction. If the partner has not yet arrived, then the token data will be stored in the memory word of matching address and the flag is set.

At the end of the function execution, the frame is released for a future reuse.

The advantages of ETS are as follows.

- It does not need any associative mechanisms.
- It does not generate a pipeline bubble caused by a hash miss.
- The location of the matching is written in the instruction, so debugging and tracing are quite easy.
- A frame word can be reused even by another instruction in the same function instance, if there is an ordering between two nodes.

However, ETS has a few defects as described below.

- An instruction fetch is always necessary, even if the matching fails.

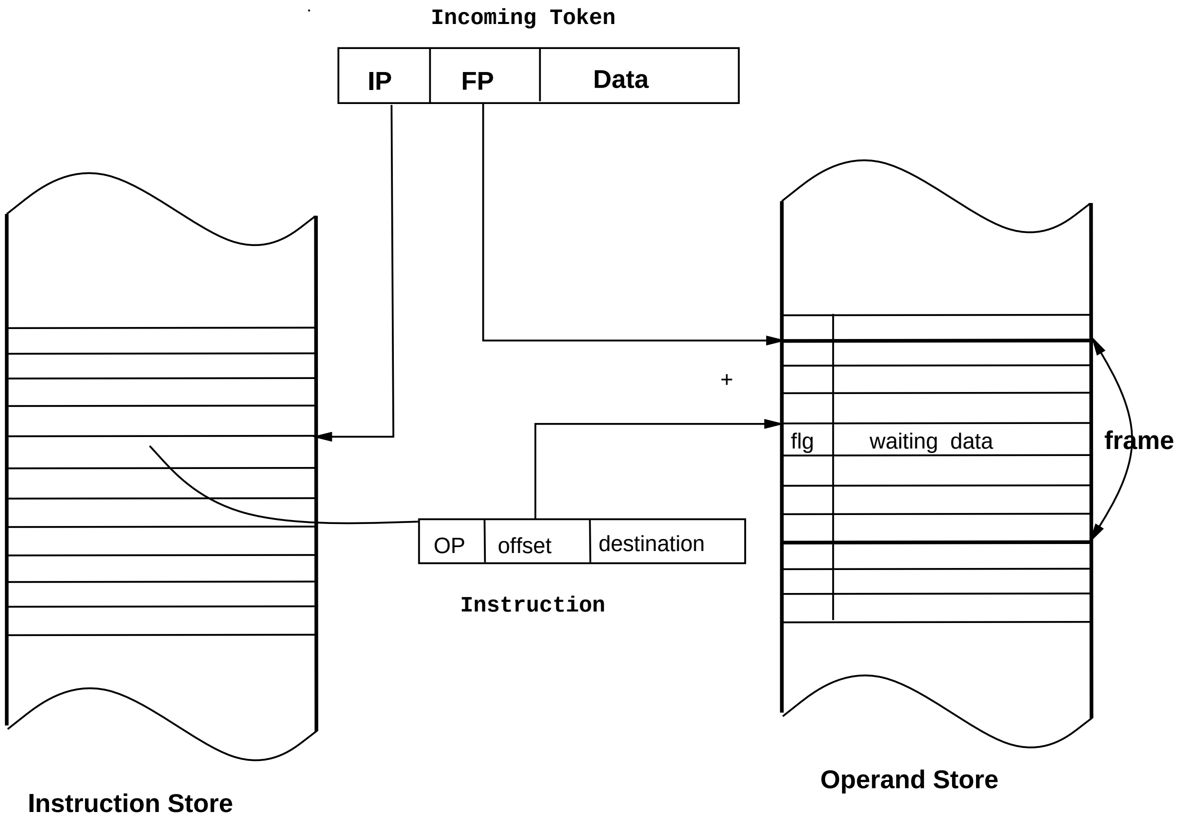

Figure 1: Explicit Token Store.

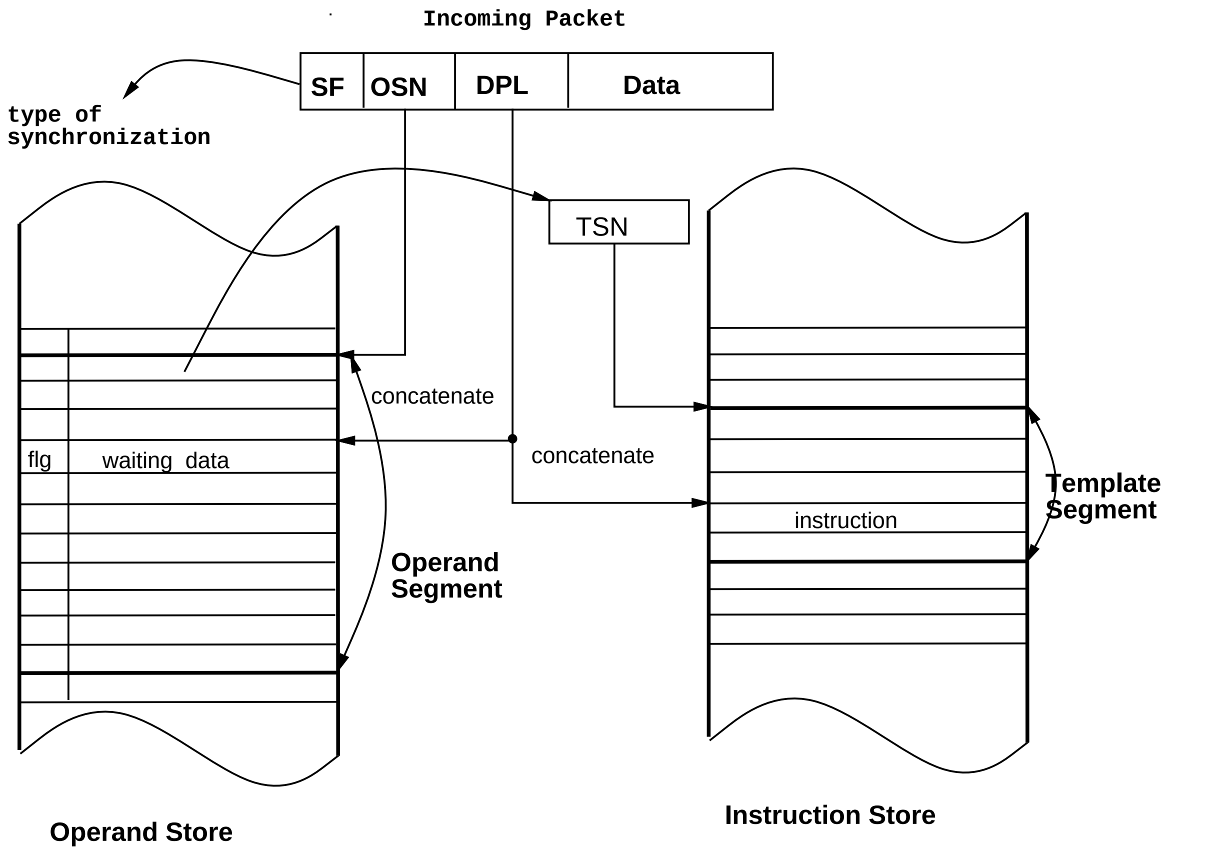

Figure 2: Direct Matching.

- An instruction fetch and matching must be serialized, and the former always occurs before the latter occurs. This makes pipeline length longer and thus causes a long turnaround.

These two defects are the result of storing the frame offset with the instruction; it is necessary to fetch the instruction offset pair to find the match location. (2) Direct

Matching in EM-4

The direct matching in EM-4 [7] is also a data-driven synchronization without using associative mechanisms.

Figure 2 shows the direct matching scheme.

Similar to ETS, a matching area is exclusively reserved for a single function instance. This area is called an operand segment while the code block corresponding to it is called a template segment. Matching occurs at a certain word in the frame. At the function invocation time, one operand segment is reserved for the new function instance and the top address of the template segment is written to the top address word of the operand segment. The top address of the template segment is called TSN, template segment number.

In direct matching, every token contains two fields, OSN and DPL. OSN is an operand segment number (same as a frame pointer in ETS) which points the top of the operand segment. The DPL serves both a displacement of the instruction and that of the matching. This displacement is common for both matching and an instruction fetch, i.e. the matching offset in the operand segment is the same as the instruction offset in the template segment.

Each packet also has a field of SF, a *synchronization flag*. This field indicates the type of the synchronization. In the EM-4, the SF contains two bits representing four synchronization types: monadic, dyadic left, dyadic right and immediate.

The matching address is produced simply by concatenating the OSN and the DPL. Instruction address is derived by concatenating the TSN, which is fetched from the top word of the operand segment, and the DPL (Figure 2).

Each data word in an operand segment includes a synchronization flag. A dyadic matching in the EM-4 is performed by a test-and-set of the flag. If the flag has already been set, the partner data will be read, the flag will be cleared, and the instruction will be executed. Otherwise the arriving data will be stored there and the flag will be set.

At the end of function execution, the operand segment is released for a future reuse.

The advantages of the direct matching are as follows.

- It does not need any associative mechanisms.
- It does not generate a pipeline bubble caused by a hash miss.
- An instruction fetch is not necessary, if the matching fails. This solves the first defect of ETS.
- After a TSN fetch, matching and an instruction fetch can be performed in parallel. This partly solves the second defect of ETS[2].
- The location of the matching is completely written in the packet, so debugging and tracing are quite easy.

However, the direct matching has a few drawbacks as described below.

- A TSN fetch is necessary before an instruction fetch. Although these can easily be pipelined, this makes the turnaround one cycle longer.
- The reuse of a frame word within the same function instance is impossible, since it is bound to a single instruction which has the same offset as the matching word.
- At the time of a function invocation, a TSN must be written to the first word of the operand segment. This overhead is usually negligible, since it happens once per one function. If the function body is quite small, this may be a problem.
- Each instruction must have exactly one frame slot. This reduces the memory efficiency.

[2] In the EM-4, matching and an instruction fetch are not operated in parallel, because of its narrow memory bandwidth. However, it has to be emphasized that, in principle, the direct matching can provide parallel operations of matching and an instruction fetch, if the memory bandwidth is sufficiently high or if the Harvard architecture is adopted.

Both in Monsoon and in the EM-4, a dyadic matching is performed only in a single clock cycle. Memory read-modify-write is thus carried out in a single clock cycle and the pipeline pitch of these two machines cannot be shorter than it.

After the development of these machines, more innovative computers are proposed and now being built by the same groups. One is *T [15] and the other is the EM-5 [12]. In these two machines, the data-driven synchronizations are improved so as to eliminate the defects listed above.

(3) *T message synchronization

The data-driven synchronization in *T is based on ETS, but is more flexible. For thread-based computation, cases where more than two messages must be synchronized are the common case. To perform synchronization of n-messages efficiently and flexibly, *T will provide a general join instruction at the starting/resuming time of each thread. Each message in *T includes the FP and IP, just like ETS in Monsoon. The join instruction reads a flag (or a counter), updates it, and performs a conditional branch if the synchronization is completed. This method makes the turnaround longer, but gains flexibility. The details of this instruction and the details of the *T architecture have not been presented yet, so this paper will not closely examine the data-driven mechanisms in *T.

(4) Advanced Direct Matching in the EM-5

The EM-5 adopts an improved synchronization method, called an *advanced direct matching*. Unlike the previous mechanisms, the matching and the instruction fetch are completely independent within the advanced direct matching.

This method is based on the direct matching. Each token packet contains the OSN and the displacement in the operand segment (ODPL). Furthermore, it contains the TSN and the displacement of the template segment (TDPL). There are thus two offset fields, so the address of the instruction is completely independent of the matching.

The advanced direct matching uses the SF field in each packet which indicates the type of the synchronization.

Figure 3 illustrates the advanced direct matching in the EM-5. At a function's invocation time, a frame is allocated for it. When each token arrives at a processor, an instruction fetch occurs using TSN and TDPL. At the same time, matching occurs using OSN and ODPL.

As the packet in the advanced direct matching holds all information which are necessary for an instruction fetch and matching, it is a superset of both ETS and the direct matching, i.e. it receives all the advantages which are listed in (1) and in (2). In addition, the advanced direct matching has the following advantages.

- An instruction fetch is not necessary for matching. A token whose partner has not arrived yet does not fetch an instruction.
- a matching and a speculative instruction fetch can be performed completely in parallel, which shortens the turnaround.
- It is not necessary to write TSN at a function invocation.

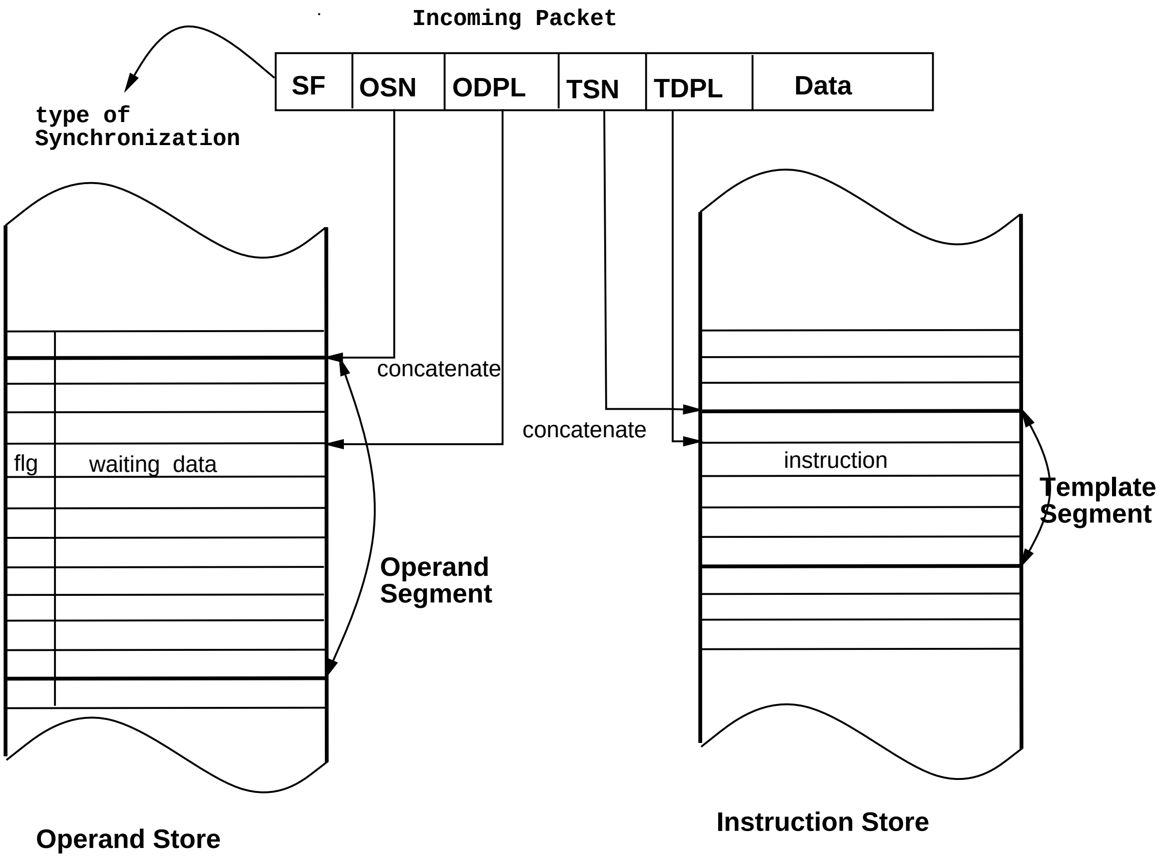

Figure 3: Advanced Direct Matching.

- There are no TSN fetches.

These advantages are derived at the cost of the additional address field in a packet, i.e. an additional offset and a TSN field. In each instruction which outputs a packet, there is also an additional field containing the TDPL. Whether this cost is reasonable or not will be discussed in Section 4.

2.2 Optimized Synchronization

Before proposing a new method, we have to understand the differences between the synchronization requirements in an original dataflow architecture, and those in a multithreaded architecture.

The additional requirements (or conditions) for "multithreaded" synchronization are as follows.

1. In a multithreaded architecture, multi-way join occurs more frequently. This is because a long thread may need a lot of data for all of its operations. In addition, the size of messages may not be fixed, i.e. the amount of data included in a message will vary from one to many, depending on the system's data distribution method and its operand passing method.

2. Multithreaded architectures may not need as many data-driven synchronizations as the conventional dataflow architectures. This is because a node inside a thread does not need data matching.

3. To exploit locality, a cache and a register set are introduced for a fast single-thread execution. Synchronization should be comparable in the speed to these local storages.

The first condition may force us to use a counter-type join logic in addition to (or instead of) the traditional dyadic matching logic. In addition, we have to prepare multiple data words for waiting data storage at one synchronization. The second condition will make us reconsider the cost-effectiveness of synchronization. If a synchronization occurs usually once per ten thousand cycles, we may not need a special mechanism for it. The third condition will force us to make a short-pitch pipelined implementation of synchronization. It will also force us to perform the synchronization in parallel with an instruction fetch and a decode.

In this subsection, an ultimately efficient synchronization is proposed. Discussions on cost-effectiveness will be made in Section 4.

Figure 4 shows the optimized data-driven synchronization for massively parallel multithreaded computers.

Each message (or each packet) contains the (TSN, TDPL) pair (i.e. instruction address), and the (OSN, ODPL) pair[3], just like the advanced direct matching shown in 2.1. In addition, the packet has a synchronization flag field (SF) which indicates the type of the synchronization. This flag field has typically three bits, i.e. larger than that in the advanced direct matching, representing (1)a monadic synchronization, (2)dyadic left

[3] All the addresses are virtualized, so there is an address translation mechanism for each cache.

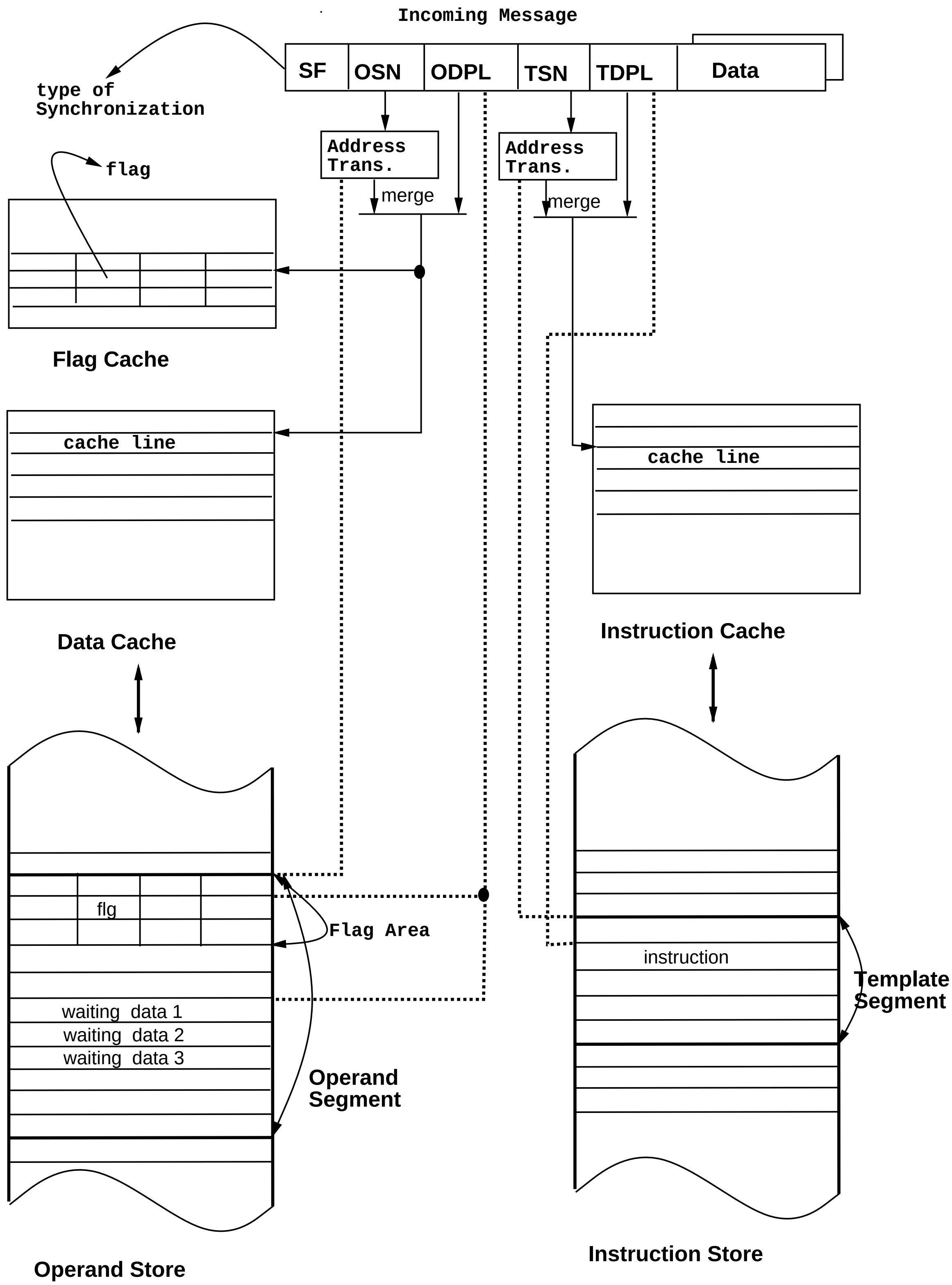

Figure 4: Optimized Data-Driven Synchronization.

data, (3)dyadic right data, (4)a synchronization for more than two messages, (5)a signal synchronization, (6)dyadic data with an immediate left or (7) dyadic data with an immediate right. We may consider a larger field for this flag for representing a sticky token, representing a triadic synchronization, etc.

We use an instruction cache and a data cache respectively, to shorten local memory latency. These will be separate on-chip-caches just as in a typical RISC chip. In addition, it is advantageous to have another independent cache, a *flag cache*, to speedup synchronization.

Suppose a single clock cycle means time enough for a cache read (or write). When a message comes into the synchronization part in a processor, an instruction can immediately be fetched. The operational sequence of a instruction fetch and a decode are completely independent of that of the synchronization until the execution occurs. We can thus concentrate on the data-driven synchronization.

Actually there are five types of data-driven synchronizations according to the SF. We describe the mechanism for each case.

1. Monadic Synchronization
   In case of monadic operations or in case that a single message contains all data which are necessary for certain thread execution. Synchronizations are not necessary.

   Besides fetching an instruction and decoding it, nothing happens before the execution. Note that all the synchronization stages of a pipeline should be bypassed to reduce latency.

2. Dyadic Synchronization
   A dyadic synchronization is the same as the dyadic matching in conventional dataflow machines. Although this can be implemented by a multiway join described below, this should be distinctively implemented, since dyadic synchronizations occur so frequently that its efficiency influences the overall performance of the machine considerably.

   The operations of a dyadic synchronization are (1)read a flag from a flag cache (if it fails, a cache line is read from the main memory), (2)check whether the flag shows the existence of the partner or not, (3)clear the flag if the partner exists or to set the flag if it does not exist, (4)read data from the data cache (if it fails, a cache line is read from the main memory) whose address is (OSN, ODPL), and (5)write message data to the data cache whose address is (OSN, ODPL) if the partner does not exist.

   These operations can be parallelized, i.e. (1) and (4) are parallelized, and (3) and (5) are parallelized. So the synchronization sequence is:

   clock 1 : (1) and (4)

   clock 2 : (3) and (5) following (2)

   As (2) is simple and the address has been held since (1), (2) and (3) can be performed in a single cycle.

   To make a complete pipeline, it is necessary to have a dual port flag cache and a dual port data cache.

3. Multiway Join (≥ 3)
   There are two ways to synchronize more than two messages. One is to make a tree of dyadic synchronizations. Each node of the tree performs a dyadic synchronization and a store of message data to certain words of the frame. This methods requires no additional hardware. However, it needs many instruction execution cycles and additional packet flow, since there may be many intermediate nodes dedicated to the synchronization.

   To perform the multiway join more efficiently and more flexibly, it is necessary to use a join-counter logic. In this case, each flag is used for a counter indicating the number of messages which have already arrived or the number of messages which will have to arrive before execution.

   The operations of a multiway join with this counter are (1)read a flag from a flag cache (if it fails, a cache line is read from the main memory), (2)increment the counter and compare the result with some constant (or to decrement the counter and compare the result with zero), (3)clear the flag if the partner exists or to set the flag to the new value derived from (2) if it does not exist, (4)read waiting data from the data cache (if it fails, a cache line is read from the main memory). (5)write message data to the data cache (if it fails, a cache line is read from the main memory).

   Here, (4) can be performed by the executor, not by the synchronization logic. However, the speculative data fetch will be effective, since (4) can be performed in parallel with the flag memory write.

   The operations here can also be parallelized. The synchronization sequence is:

   clock 1: (1) and (5)

   clock 2: (2) and (4)

   clock 3: (3) and (4)

   clock 4 and after: (4), if necessary

   In case of the multiway join, the counter update and branch are more time-consuming than the flag-check and branch in the dyadic synchronization. This is the reason why (2) takes one independent clock cycle.

   If the size of the message data is more than one, (5) will take more than one cycles. In this case, data write occurs in clock 2 or after clock 2. This will delay the sequence of (4).

4. Signal Synchronization
   The operations of a signal synchronization with a lot of tokens are similar to those of the multiway join. However, it does not need to access a data memory at all.

   What happens in this case is a sequential operation of (1)(2)(3) of the multiway join.

5. Counter Overflow
   In a multiway join (also in a signal synchronization), the counter overflow may

occur when the number of messages which should be synchronized at the same point exceeds the number which can be represented by the flag.

In this case, a join-tree is constructed each node of which performs a smaller join.

One question is the size of the flag. If the flag is too small, we have to make a high tree which causes overhead. If the flag is too large, memory efficiency is degraded. For instance, if the flag is four-bit wide, then it can perform 16 way join with a single node, 256 way join with a two-layer tree, etc. Although four bits seem sufficiently for a multi-way join, a good size must be experimentally determined.

Note that the synchronization does not use instructions at all. Instructions such as Post, Start, Join and Next [15] do not exist here. This reduces the execution cycles of a processor. The synchronization for a certain thread can totally be pipelined with the other threads. This highly increases the system throughput. It also can be parallelized with the instruction fetch and the decode for the same message. This highly decreases the processing latency.

The synchronization sequence described above can also be pipelined, which is described in 4.3.

The advantages of this optimized synchronization are as follows.

1. Efficiency
   The proposed method is ultimately efficient, because of the followings: (1)it provides highly parallel and pipelinable operation mechanisms: parallel execution of instruction fetch/decode and synchronization, parallel execution of flag memory accesses and data memory accesses, and pipelined execution between multiple threads; (2)it provides five kinds of synchronization mechanisms which can optimally be adopted for the synchronization type, and (3)it does not need any instructions which will increase the latency and a whole cycle time.

2. Flexibility
   The proposed method provides five kinds of synchronization. Especially the counter type synchronization makes it fairly flexible for complex synchronizations.

3. Simplicity
   As each message has both the address of instruction and the synchronization, this mechanism simplifies the synchronization itself and eases the understanding of what is happening in the synchronization stages. To design and to debug are both much easier than the other data-driven methods.

3 Pipeline Design

3.1 Basic Methodologies

Conventional dataflow machines and the HEP[1] have a pipeline where multiple threads share its slots at the same time. The pipeline of this type is called a *circular pipeline*[3].

The circular pipeline has several advantages as follows.

- It exploits the parallel activities maximally in a single pipeline. There is no overhead of a context switching.

- It achieves the natural and efficient combination of computation and communication by providing continuous operations from the network to the instruction executor and from the instruction executor to the network. There are no interruptions between a network and a processor.

- It can provide an excellent throughput without preparing complex interlocking mechanisms and branch prediction mechanisms.

Because of these advantages, the circular pipeline has been regarded as an excellent structure for parallel computation. Two of the representative dataflow machines, the SIGMA-1[4] and Monsoon[8] have this type of pipeline.

Figure 5 shows the pipeline structure of Monsoon. It has eight stages each of which can be occupied by an independent thread in a completely non-blocking way.

However, there are several drawbacks in a "pure" circular pipeline [11].

- The circular pipeline does not perform an advanced control. Advanced control here means a look-ahead control containing an instruction prefetch.

- There may be a lot of empty slots in the circular pipeline if there is not enough parallelism.

- Throughput of the circular pipeline is limited by the speed of global data handling such as a packet flow and main memory access.

- Resource management is inefficient with the circular pipeline, since it has to lock the pipeline resources for a series of indivisible operations which cannot be interleaved.

Briefly, the circular pipeline is not suited to a local sequential execution nor is it suited to a less parallel execution.

To overcome these defects is one of the most essential reasons of proposing multithreaded computers. Multithreaded computers typically have a RISC-type von Neumann pipeline in addition to the the packet-based circular pipeline.

For instance, Figure 6 illustrates the pipeline structure of the EM-4 [7]. The processor of the EM-4 has two-layered nested pipeline, i.e. the packet-based circular pipeline includes the RISC-pipeline. The former performs a multithreaded pipelining as fast as or faster than a conventional dataflow pipeline, while the latter performs a register-based advanced-control pipelining as fast as a conventional RISC pipeline. In addition, the multithreaded pipeline of the EM-4 has several bypassing routes which shortens a latency of a packet with a simple request.

The EM-5[12] will have a faster and more complex pipeline based on the EM-4 layered-pipeline, i.e. the matching stage will be faster because of the introduction of the advanced direct matching, and the execution part will perform integer operations, floating operations and load/store in parallel[4].

[4]The basic concept of the EM-5 pipeline design is the same as the EM-4, so this paper will not describe its structure more closely.

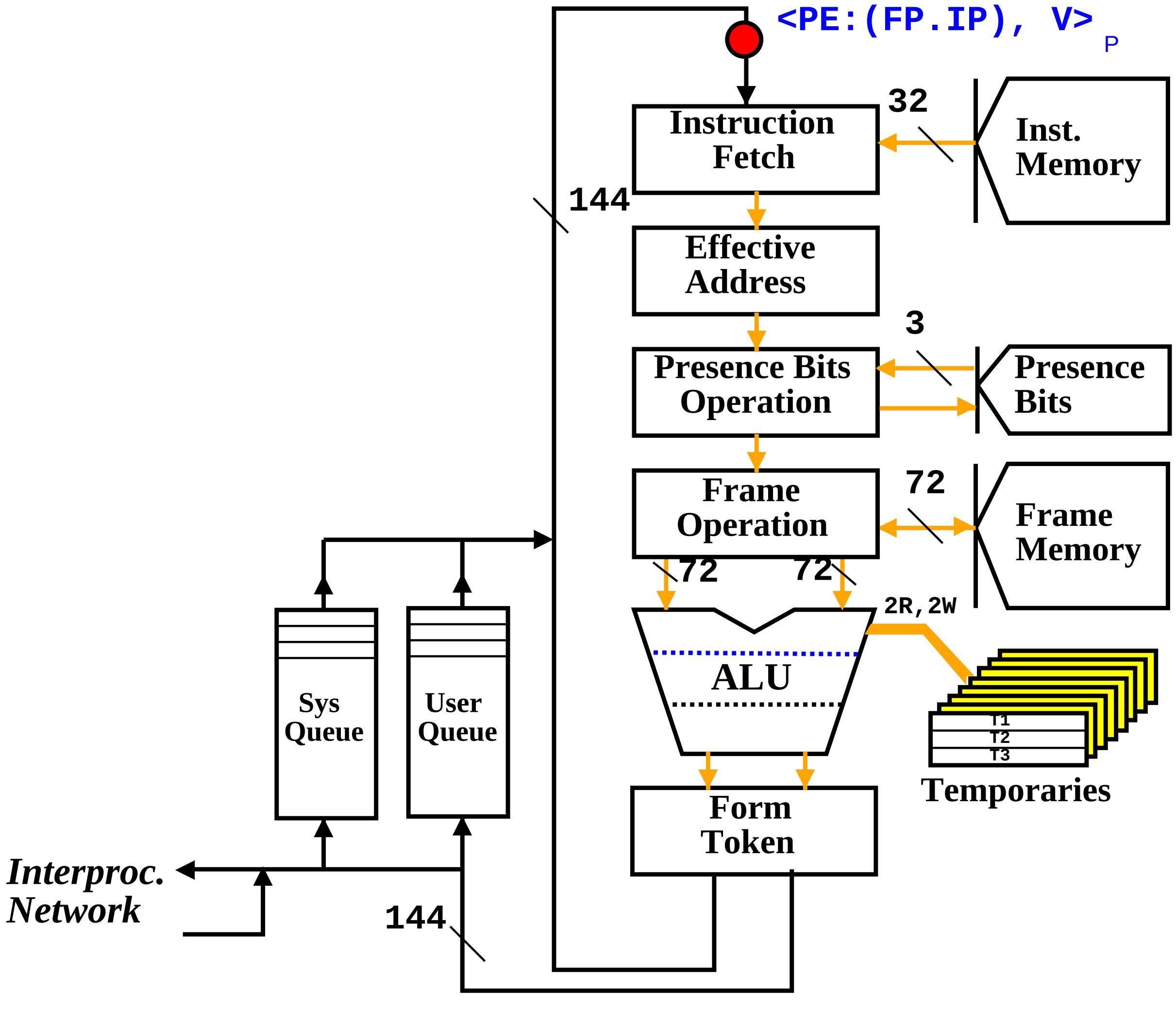

Figure 5: Pipeline Organization of Monsoon.

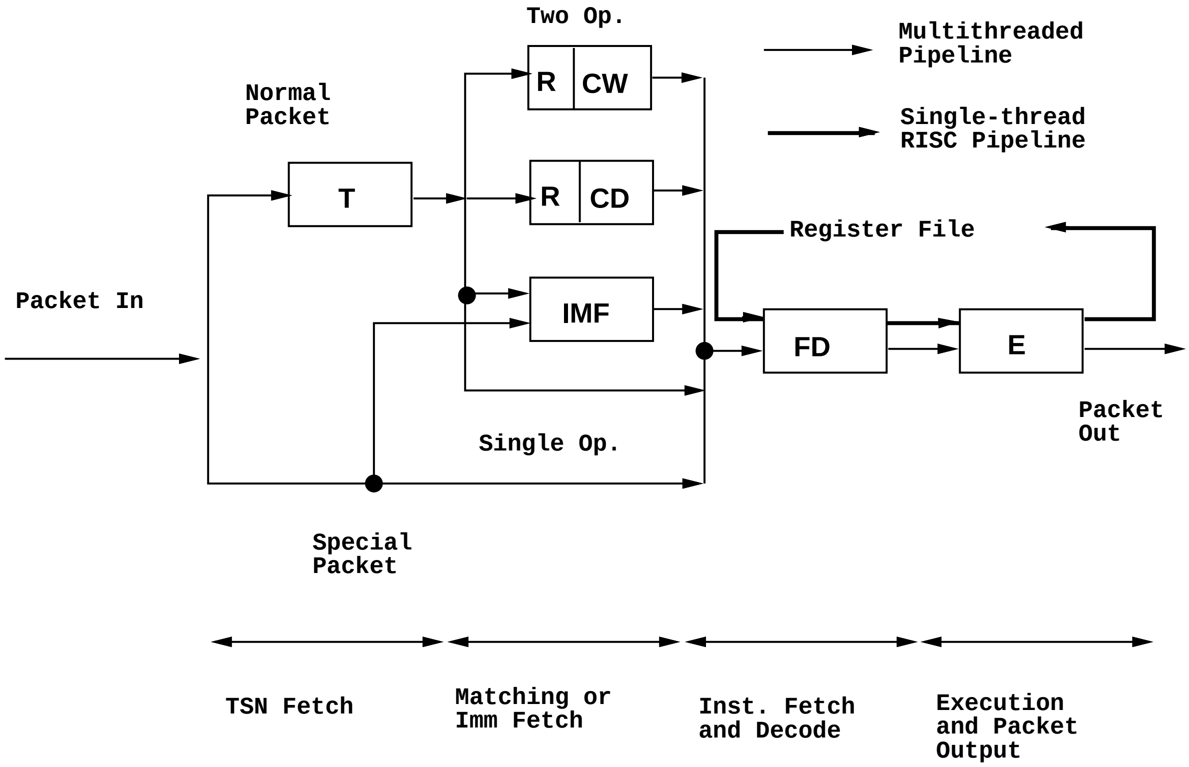

Figure 6: Pipeline Organization of the EM-4.

The *T [15] will also have a RISC pipeline inside its processor. Actually, the *T processor is an extension of a MC88110, a commercial RISC processor which will provide the peak performance of 100 MFLOPS for double precision floating point calculations. The details of the *T pipeline structure have not been presented yet, so this paper will not examine the *T pipeline closely; however, we have to note that it performs both the continuous data-flow between a network and an instruction executor, and the local RISC execution.

3.2 Optimized Pipeline

The previous architectures have shown that the pipeline for a multithreaded processor should be a combination of two types of pipelines. One is a multithreaded circular pipeline and the other is a RISC type single-threaded pipeline with an advanced control. The former should be fairly fast with the optimized data-driven synchronization stages proposed in 2.2.

Before proposing an ultimately efficient pipeline for a multithreaded computer, we have to show the requirements for a pipeline of multithreaded computers.

1. High throughput for both multithreaded execution and single-threaded execution
   In the normal case, the throughput should be one result per one RISC clock which indicates the pitch for both pipelines.

2. Short turnaround from message input to result write to message output, and from instruction fetch to result write
   For single-thread execution, the turnaround of operations inside a processor must be equal to or less than a normal RISC processor. For multithread execution, result message should be output as soon as possible after the input message enters.

To meet the first requirement, a pipeline pitch must be as short as possible.

Cache read and write should be pipelined. There are three independent caches to implement the optimized synchronization described in 2.2. Among those, the flag cache is accessed by the read-flag stage and write-flag stage. The data cache is accessed by the read-data stages, write-data stages and, possibly, load/store stages [5]. To achieve the short pitch pipeline, the former cache must be dual-port and the latter will preferably be dual- or tri-ported, in order to avoid memory-request contentions at the same time.

The executor should also be highly pipelined as is shown in the current RISC processors, e.g. floating point pipelines and load/store pipelines.

To meet the latter requirement, intra-processor parallelism should be pursued as much as possible.

As was already shown in the optimization of synchronizations, the synchronization and the instruction fetch/decode should be operated in parallel. In the executor, many operations should be executed in parallel, e.g. superscaler execution of integer instructions and parallel execution of floating point instructions and integer instructions. In addition, for a multithreaded computer, generating packets should be carried out in parallel with the instruction execution.

[5] Note that the frame is used as a working area for the executor as well as a synchronization area.

To meet the latter requirement, bypassing mechanisms are sometimes necessary. For instance, remote fetch of the data should not be performed by a long pipeline.

Figure 7 illustrates the optimized pipeline for an element of a massively parallel multithreaded computer.

When a message enters, the processor performs the multithreaded pipeline as follows. The first stages of the pipeline carry out a synchronization and an instruction fetch and an instruction decode as was stated in 2.2. There were several kinds of parallelism in it, i.e. parallelism among the flag handling, the data read/write and the instruction fetch/decode. In addition there are some bypassing routes. Continuously the latter stages execute the instruction invoked by the message. If the thread contains more than one instruction, an instruction fetch for the second instruction occurs while the decode of the first instruction is carried out. Third instruction fetch occurs while the first one is executed and the second one is in the decode stage.

The instructions can be issued in parallel in the executor as long as the operand data exist and the functional unit is available. Intra-processor Parallelism can be extracted in this way, just as in the current RISC processor. In order to fetch instructions for incoming messages in parallel with the previous thread execution, the instruction cache should also be dual-ported.

Thread execution proceeds in this way. The packet turnaround time (the time from packet input to the first packet output, if the first instruction is a packet output instruction) is (1)4 clocks for a monadic node, (2)4 clocks for a dyadic node and (3)4 +(s - 2) clocks for a s-adic node if the processor and the network have no other activities. The typical throughput is n instructions per clock where n is an average execution parallelism utilized in a processor.

There are three queues illustrated in Figure 7: the Input Message Queue, the Continuation Queue and the Output Message Queue. These queues absorb the speed gap between the network and the synchronization part, between the synchronization part and the execution part, and between the execution part and the network. To omit one or two queues is possible, depending on the typical throughput of each part.

4 Discussions

4.1 Architectural Tuning

(1) Priority Control

Although the proposed pipeline is fairly efficient, there are cases where a certain thread should be executed before the current thread execution is finished. For example, suppose that a certain remote fetch is critical for the whole computation time. If the remote fetch message is blocked at the entrance of the executor by a long thread, the overall performance will be degraded.

To solve this situation, two ways can be considered: independent execution or preemption. In the former, the processor may have an extra pipeline (including extra registers and data paths for executing) which is dedicated to the priority operations. The I-structure handling of *T uses this [15]. In the latter, the processor should have a fairly fast context switching mechanism. To exploit the ultimate pipeline shown in Section 3, the overhead

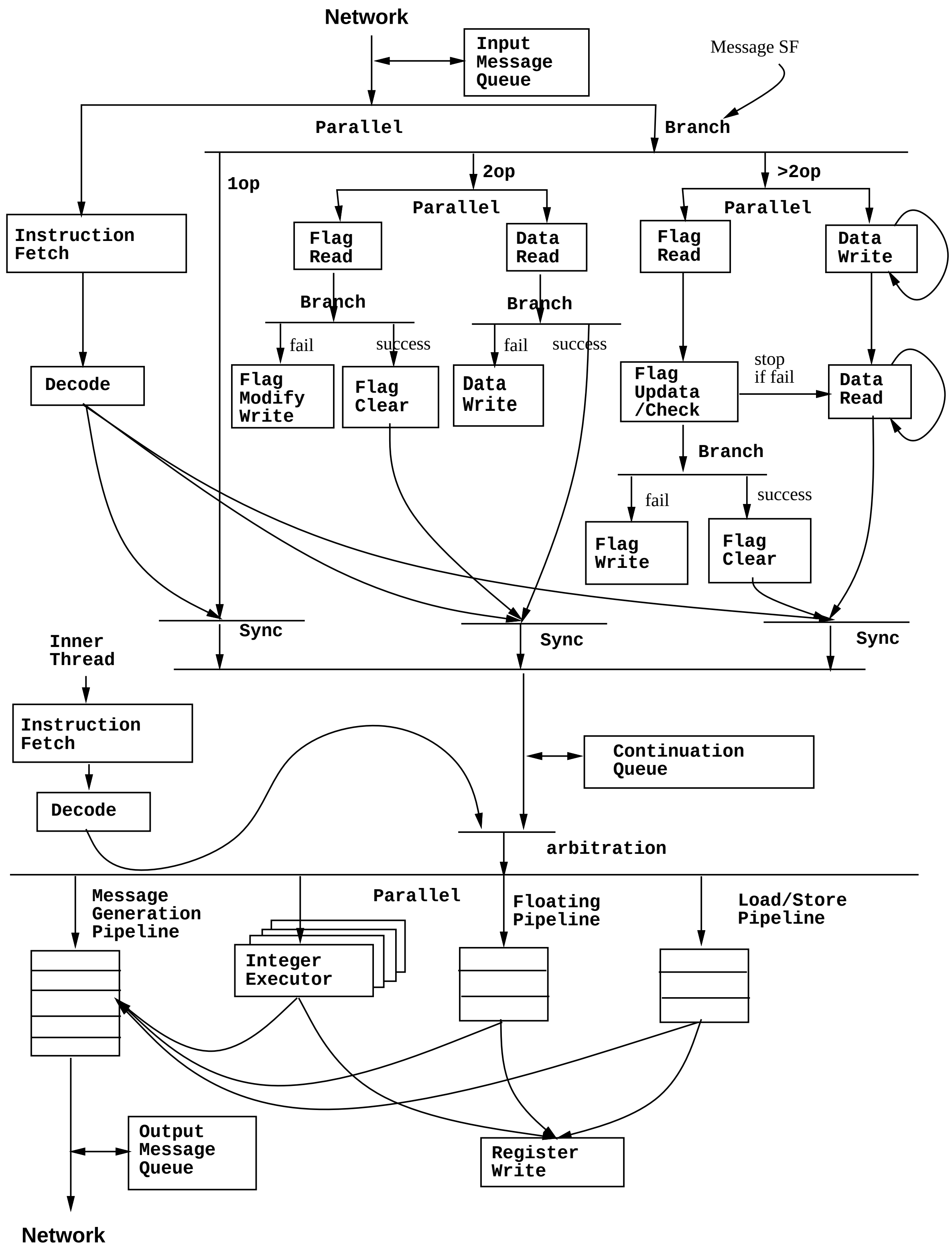

Figure 7: Optimized Pipeline.

of switching context must be zero or one clock cycle. The instruction insertion mechanism with multiple register sets [18] will be adopted for this.

In either case, efficient priority queues are necessary in a message waiting section and sometimes in an interconnection network.

(2) Load/Store Overhead between Synchronization and Execution

In the pipeline described in 3.3, all message data must be stored before execution. However, if the synchronization succeeds, they can be sent directly to the registers of the execution unit, instead of store and load after the synchronization.

(3) More Flexible Thread-Interleaving

In the previous discussions, the author did not mention about more complex situations where we can adopt a much finer control of threads. This can improve performance with fairly small amount of control logic.

Here is a simple example. A floating point pipeline can be shared by more than two threads at the same time.

Another example is to execute parallel activities between threads in the multiple execution units (e.g. integer units, a floating point unit, a load/store unit).

4.2 Cost-Effectiveness

If threads are usually long, then the instruction executions can be pipelined in a single thread in most of the computation time. In this case, people may say that the synchronization mechanism and the pipeline proposed here may be an "overdesign".

However, as for the number of transistors, these mechanisms cost fairly small. The direct matching in the EM-4 only costed 3,610 CMOS gates [7], which are relatively small in comparison with the other units, e.g. integer units. The proposed mechanisms will need slightly larger amounts of hardware, since the pipeline pitch is about as half as that of the EM-4 and thus additional pipeline registers will be necessary; however, it is clearly not a great deal of a VLSI area. The logic is quite simple, since there are no interlocks necessary for coordinating more than one threads. In addition, no interrupt mechanisms exist instead of the error handlings and exceptions.

The extra hardware will also be required for implementing a multiport cache: multiple buses, multi-set decoders and arbitration logic. However, these are simple circuits and all the buses are inside the chip. This part thus does not affect the pin limitation of VLSI design.

The last question is the bandwidth of an interconnection network and a memory. The message needs extra fields for the proposed synchronization method: in comparison with ETS, this needs the frame offset and the SF. However, in a multithreaded computer, this field can be fairly small, since synchronizations in a frame are much less frequent than in a conventional dataflow computers, and thus the address space necessary for a synchronization is small.

An extra field for a TDPL in an instruction memory is also a small problem for the same reason.

In addition, because of highly extracting intra-processor parallelism, the thread execution time becomes short in this architecture. Therefore, to shorten a pipeline is significant for performance, and we must have synchronization mechanisms with fairly short turnaround.

Therefore, we got the conclusion that the proposed synchronization and the proposed pipeline are both implementable at a reasonable cost; they are cost-effective in VLSI design of a multithreaded computer.

4.3 Related Issues

Here we concentrate on a local data-driven synchronization. However, we have to optimize the I-structure/M-structure type synchronization for a large data structure handling. In addition, a multithreaded computer should have a fast global synchronization mechanism in case of a barrier in a program. These issues are orthogonal to the issues described in this paper. They can be independently pursued and added to the pipeline described in 3.3.

Optimization of the cache structure is another significant issue, since the pipeline cannot work well if cache misses are frequent. Scheduling techniques to increase the cache hit ratio without degrading the parallelism must be considered closely.

In an actual machine, virtual memory mechanisms should be implemented [6]. To optimize the physical distribution of data influences the performance of the pipeline.

5 Conclusion

This paper made a proposal about two fundamental mechanisms for a multithreaded massively parallel computer: synchronizations and pipelining. The synchronization proposed here is based on frame and is a highly parallel and pipelinable. It provides one synchronization per each RISC clock cycle and its latency is typically four cycles if the target words are in caches.

With this synchronization mechanism, pipelines can be ultimately optimized. This paper showed such an optimized pipeline in a processor of a multithreaded massively parallel computer. The operations are highly parallelized (matching and an instruction fetch/decode are operated in parallel, instruction execution and packet generation are in parallel, etc.) and the pipeline pitch is ultimately shortened to a cache cycle.

Future problems are : (1) quantitative evaluations of the proposed mechanisms, by analyses based on a queueing model and by simulations; (2) close consideration of the related issues, including virtual memory/cache design, priority handling mechanisms, task partitioning and optimization of granularity, work load balancing, data distribution, I/O and other scheduling problems; and (3) feasibility study by constructing a prototype system and examinations of its efficiency and cost-effectiveness, with state-of-the-art VLSI implementation and practical application programs.

[6]Here, how to implement the virtual memory is not examined. It must be closely considered, combined with the implementation of caches.

Acknowledgement

The author wishes to thank Professor Arvind, Professor Gregory M. Papadopoulos and Dr. George A. Boughton for supporting this research, and the members of Computation Structure Group in MIT-LCS for the fruitful discussions. He also would like to express his sincere appreciations to Professor Dennis and Dr. Nikhil of DEC for his helpful comments.

References

[1] Smith, B.J.: A Pipelined, Shared Resource MIMD Computer, Proc. of 1978 ICPP, pp.6-8 (1978).

[2] Arvind, Kathail, V., Pingali, K.: A Data-flow Architecture with Tagged Tokens, Tech. Rep. TR-174, Lab. Computer Science, MIT (1980).

[3] Gurd, J., Kirkham, C.C. and Watson, I.: The Manchester Prototype Dataflow Computer, CACM, Vol.21, No.1, pp.34-53 (1985).

[4] Hiraki, K., Nishida, K., Sekiguchi, S., Shimada, T. and Yuba, T.: The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems, Journal of Information Processing, Vol.10, No.4, pp.219-226 (1987).

[5] Iannucci, R.A.: Toward a Dataflow/von Neumann Hybrid Architecture, Proc. of 15th ISCA, pp.131-140 (1988).

[6] Thistle, M.R. and Smith, B.J.: A Processor Architecture for Horizon, Proc. of IEEE Supercomputing Conference, pp.35-41 (1988).

[7] Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y. and Yuba, T.: An Architecture of a Single Chip Dataflow Processor, Proc. of 16th ISCA, pp.46-53 (1989).

[8] Papadopoulos, G.M. and Culler, D.E.: Monsoon: An Explicit Token Store Architecture, Proc. of 17th ISCA, pp.82-91 (1990).

[9] Grafe, V.G., Hoch, J.E., Davidson, G.S., Holmes, V.P., Davenport, D.M. and Steele, K.M.: The Epsilon Project, in Advanced Topics in Data-Flow Computing, Chapter 6, pp.175-205, Prentice Hall (1991).

[10] Dennis, J.B.: The Evolution of "Static" Data-Flow Architecture, in Advanced Topics in Data-Flow Computing, Chapter 2, pp.35-91, Prentice Hall (1991).

[11] Sakai, S., Hiraki, K., Yamaguchi, Y., Kodama, Y. and Yuba, T.: Pipeline Optimization of a Data-Flow Machine, in Advanced Topics in Data-Flow Computing, Chapter 8, pp.225-246, Prentice Hall (1991).

[12] Sakai, S., Kodama,Y. and Yamaguchi, Y.: Architectural Design of a Parallel Supercomputer EM-5, Proc. of JSPP91, pp.149-156 (1991).

[13] Papadopoulos, G.M. and Traub, K.R.: Multithreading: A Revisionist View of Dataflow Architecture, Proc. of 18th ISCA, pp.342-351 (1991).

[14] Sato, M., Kodama, Y., Sakai, S., Yamaguchi, Y. and Koumura, Y.: Thread-based Programming for the EM-4 Hybrid Dataflow Machine, to appear in the 19th ISCA (1992).

[15] Nikhil, R.S., Papadopoulos, G.M. and Arvind: *T: A Multithreaded Massively Parallel Architecture, to appear in the 19th ISCA (1992).

[16] Dally, W., Chien, A., Fiske, S., Horwat, W., Keen, J., Larivee, M., Lethin, R., Nuth, P. and Wills, S.: The J-Machine: A Fine-Grain Concurrent Computer, Proc. of IFIP 89, pp.1147-1153 (1989).

[17] Kechler, S.W. and Dally, W.J.: Processor Coupling: Integrating Compile Time and Runtime Parallelism, to appear in the 19th ISCA (1992).

[18] Toda, K., Nishida, K., Uchibori, Y., Sakai, S. and Shimada, T.: Parallel Multi-Context Architecture with High-Speed Synchronization Mechanism, Proc. of 5th IPPS, pp.336-343 (1991).