LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Virtual Memory
on a
Dataflow Computer

Computation Structures Group Memo 289
July 13, 1988

Kenneth M. Steele
Richard Mark Soley

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 N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Virtual Memory on a Dataflow Computer

Kenneth M. Steele Richard Mark Soley

July 13, 1988

1 Overview

Dataflow architectures present an exciting solution to the problem of parallel processing. Current dataflow execution models, such as the Tagged-Token Dataflow Architecture[2], have been heavily modelled and studied in order to find the amount of parallelism exploited by their compile-time and execution-time approach to automatic parallelism extraction[4]. These largely successful studies have recently led to a new dataflow execution model called Monsoon[7].

One of the features of the planned Monsoon machine is a large physical memory. However, no provisions have been made for a virtual memory function in the machine; nor was a virtual memory extension ever designed into the older Tagged-Token architecture.

Virtual memory is, however, a fact of life in commercial applications on current machines. The different approach to program execution taken by dataflow architecture advocates does not lessen the voracious need for memory and address space found in current applications; rather, there is ample evidence that memory requirements will actually increase[3]. Virtual memory schemes, embodied in segmentation and paging systems found in many extant machines, solve many different problems:

- Expansion of the addressable memory of an application by simulated main-memory usage of secondary storage.
- Simplified automatic allocation of memory and fewer memory "holes;" in general, more flexible allocation schemes.
- Allowance for better programming methodologies as well as memory protection schemes, such as segmentation á la Multics[8] or read-only pages.
- Miscellaneous tie-ins to page-not-available support hardware, such as automatic blocking on I/O-wait paging.

In order to address this missing feature of planned dataflow architectures, we present an approach to the support of automatic, hardware-supported virtual memory address translation. We introduce this feature within the context of the planned Monsoon hardware; this adds the requirement that the design must peacefully coexist with the Monsoon design philosophy.

One of the strengths of dataflow is the ability to hide latency; the address translation required by virtual memory schemes is yet another form of latency, and thus should fit into the architectural model. The context switching required by "faulted" address translations (*i.e.*, page-not-available, page-out-of-main-memory or segment-not-present) also does not present a major burden on the Monsoon execution approach, as the processor as planned switches contexts on every pipeline beat (instruction introduction) anyway. Virtual memory fault handling will be treated like any other context switch.

We shall narrow our focus to the handling of virtual address translation page faults to simplify the presentation of the paper. However, the method used to handle page faults is certainly applicable to other virtual memory schemes (such as segmentation), possibly as well as other types of machine faults.

2 Goals

The goal is to provide virtual memory on Monsoon efficiently and with as few changes to the planned system as possible. The restrictions presented by Monsoon include

- Non-faulting pipeline.
- One instruction issue per pipeline beat.
- One memory access per input token (read, write or exchange).
- One pass through the pipeline per instruction.
- Maximum of two output tokens per instruction.

The first requirement is the most important, as it greatly simplifies the design of the pipeline. Serial computers handle page faults by "trapping" to a fault handler. This fault handler fixes the fault (by loading the missing virtual page into a physical memory page frame) and then restarts the faulted instruction. By creating a new thread of control to handle the fault, dataflow computers can avoid faulting their pipelines. This has the added advantage that the fault handler and other non-faulted threads of control execute in parallel with unfaulted threads of the same computation. The effects of the other two requirements will be discussed in the implementation section.

The following goals are based on efficiency and performance. They include

- The ability to handle multiple outstanding faults.
- Fault handler execution should overlap with program execution.

A virtual memory fault will only suspend the execution of one thread of control; faulting and non-faulting execution can occur in any of the other threads. It is nonetheless desirable that faults be handled as quickly as possible (to reduce the critical path of the faulting thread), and not be required to wait for the previous fault to be fixed. This requires handling multiple outstanding faults. Since dataflow architectures imply leniency in the order of execution of instructions, faults can be handled in any order. This allows optimizations based on the order of fault handling to minimize latency. For example, pages can be read from the disk in order of their tracks location on a disk.

The final requirement is that all of the graphs must continue to enforce the *well-behaved dataflow graph* concept[5]. Current program schemata used in compiling Id[6] implicitly insure that

- Initially, the program graph is empty (contains no tokens, either waiting to be matched or executing).
- Given exactly one token on each input of each procedure, the procedure will produce exactly one token on each output.
- Once output tokens have appeared on every output of a procedure, the procedure's graph is again empty (contains no tokens). This is the so-called *self-cleaning* property of dataflow graphs.

These properties are central to the way in which the current dataflow architecture's memory and process resources are controlled, as well as to the approach to debugging taken by the machine. The importance of maintaining these assumptions is therefore paramount.

3 Implementation

Obviously, regardless of the technology for handling page faults, every virtual address must be converted to a physical address in order to execute a program. Thereafter, whenever the virtual addresses specified by a token do not map to physically present memory, a fault occurs. The relevant data in a faulted token is rerouted to a fault handler, which loads the specified virtual memory page into a physical memory page frame. The faulted instruction is then re-entered into the pipeline for execution. A primary design point is the maintenance of instruction semantics despite virtual memory behavior: the program should execute as if all of virtual memory were physical memory, except (potentially) slower.

3.1 Detection of Faults

Translation of a virtual address to a physical address is done by looking up a portion of the virtual address in a table. A portion of the translation table is kept in a cache called a Translation Look aside Buffer (TLB). A fault occurs when the virtual address is not found in the TLB. There are two cases under which faults occur, including

- The virtual page is already loaded in physical memory, but the translation is not loaded in the TLB. This is handled by loading the TLB with the correct translation value.
- The virtual page is not resident in the physical memory. The data can loaded from a number of possible sources: disk, another processor's memory, etc. The TLB is updated after the page frame is loaded.

In Monsoon, two TLB's need to be added to the pipeline. One is used to translate virtual instruction addresses, while the other is used to translate virtual effective data addresses found by combining information from the instruction and the token.

3.2 Redirection of Tokens

When a fault occurs the execution of the faulted thread must be suspended until the fault can be corrected. In a serial computer, the instruction pointer of the faulted instruction is saved (typically on a stack), and the CPU executes a fault handler to correct the fault. After fault handling, the faulted instruction is restarted (*i.e.*, the suspended thread is restarted beginning at the faulting instruction).

On dataflow architectures, every instruction in the machine is suspended until it receives its input tokens. Since the latency of a token is not specified, it is possible to redirect a faulted token to a fault handler and then back to the thread of control that caused the fault. The program can only detect this action through its longer latency.

When a fault is detected by a TLB, the instruction for the faulted token is replaced with a fault trap instruction on entry to the instruction execution pipeline stages. This *fault trap instruction* splits the original token's tag and data and forms two new tokens with these as the data. The tags for the new tokens are the entry points of the *primitive context*. It is important to note that the instruction that caused the fault is not executed and does not change any of the state of the machine.

Monsoon's restriction of only one ALU operation per cycle forces the use of this primitive context. A series of instructions cannot be executed in the ALU except by running a program graph.

3.3 Fault handler Primitive Context

The need for a "primitive" context, always kept available by the operating system, arises from this restriction; it is impossible to issue an extra instruction during fault handling to request an execution context from the resource control software. Therefore, we have a special mechanism for executing small graphs of code to link up faulting code blocks with system fault handlers. This is analogous to page fault handling under sequential machine operating systems, in which (generally) the page fault handler often "shares" the context of the faulting thread of execution in order to do its work.

The fault handler primitive context graph, shown in Figure 1, receives the tag and data portions of a faulted token from the fault trap instruction. The primitive context allocates a new "real" (*i.e.*, non-primitive) context in which to run the fault handler. The fault handler graph, see Figure 2, is called with the two parts of the faulted tokens as its arguments.

As pointed out, the primitive context is needed so that a destination for the faulted token can be generated in one pass through the ALU and Form Token sections of the pipeline. The primitive context is used only to allocate a new real context for the fault handler, because the primitive context can handle only one fault at a time and is a limited resource. See Section 4.2. By allocating a new context, many fault handlers can run in parallel to handle multiple faults. Section 5.1 expands on these ideas. Refer to section 3.6 for a discussion of handling a fault while the primitive context is busy.

3.4 Fault Handler

The fault handler is the code responsible for correcting a fault and restarting the faulting execution thread. The fault handler can be written in Id, and can run on the local processor

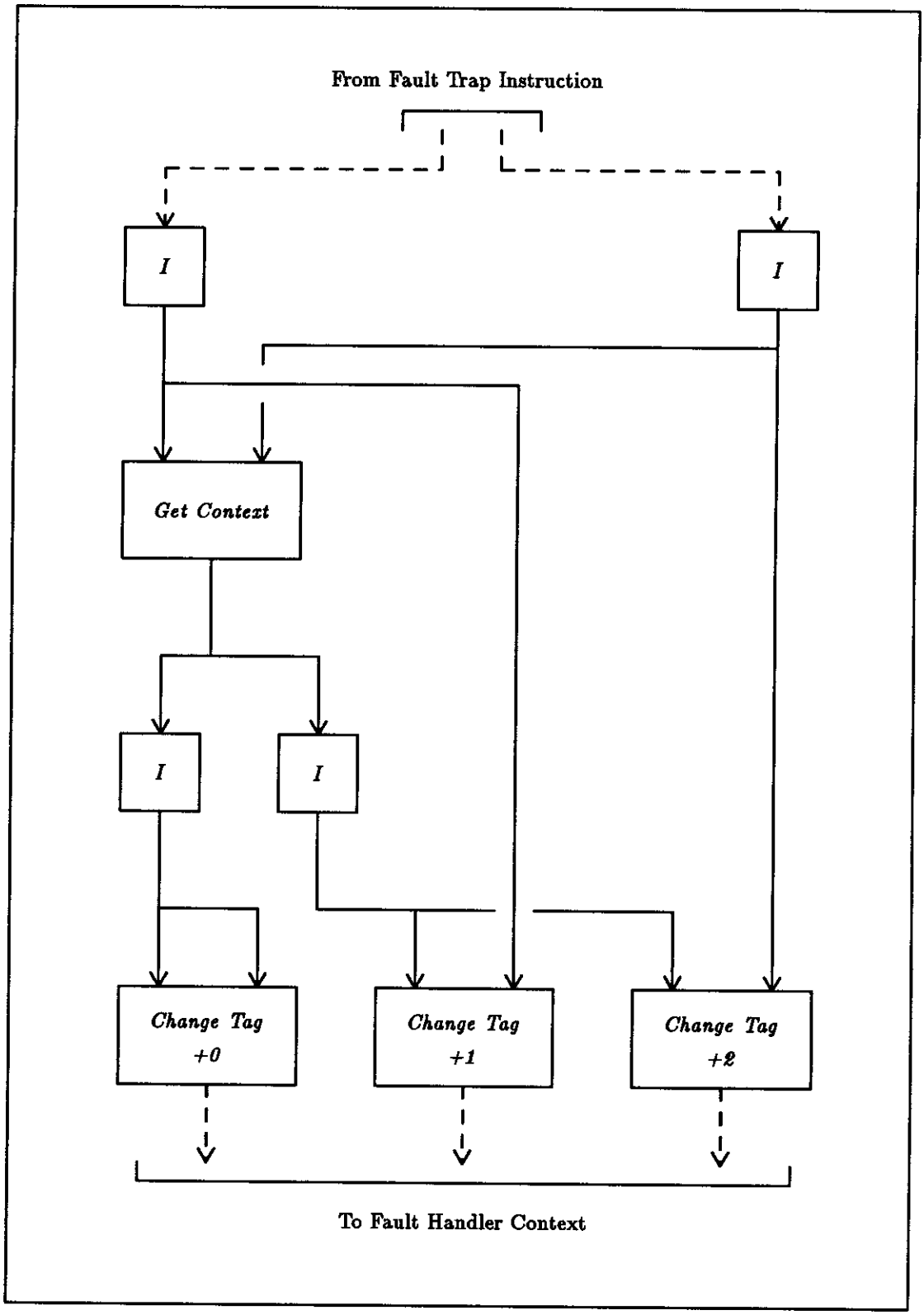

Figure 1: Primitive Fault Handler Context Graph

or on a separate I/O controller CPU. The general format of the fault handler graph is shown in figure 2. The fault handler takes as arguments the tag and data of the faulting token.

There are four basic responsibilities of the fault handler. These include

- To release the primitive context as soon as possible, so that other faults can use the primitive context. The primitive context can be released as soon as all of the arguments to the fault handler have been received, because these are the only outputs of the primitive context. At this point the primitive context graph will be empty of tokens.
- The fault handler must determine what caused the fault (*i.e.*, either an instruction or data TLB miss), and fix the fault. The fix could involve simply loading a new translation value into the TLB, or swapping memory in from an external source.
- After the fault has been fixed, the fault handler reissues the token that faulted (*in the context of the faulting execution thread*) thereby restarting the thread that faulted.
- The fault handler releases its own context. Normally, Id code blocks do not release their own contexts; rather, the code block that created a context is generally responsible for returning the context for re-use. However, the primitive context that allocated the fault handler's context has already been released, due to the need to allow reuse of primitive contexts quickly. Therefore, we simply make the fault handler release its own context.

There are special considerations in writing the fault handlers, as more than one fault handler can be executing in parallel. The faults could be unrelated or result from a common cause. The unrelated faults will need to access common data structures that will need to be locked. The related faults will need to notice if another fault handler has already fixed or started to fix the fault. Managers[1] or other nondeterministic features[9] will be required to accomplish this.

3.5 Restarting Instructions

In Monsoon an instruction is restarted by reforming the token destined for the instruction, and reissuing that token. Instructions can always be restarted, since no state is changed before a fault can occur. The first place in the Monsoon pipeline that any state is modified is the Presence Bits stage, since the instruction cache is read-only. The presence bits are read, modified and written back. Therefore, if a fault occurs it will happen while fetching the instruction or reading the presence bits; *i.e.*, *before* they are modified. The fault trap instruction which replaces a faulted instruction does not access the presence bits.

3.6 Token Recirculation

If a fault occurs while the primitive context is in use (indicated by a hardware *inhibit fault* flag) then the faulting token is simply recirculated. To prevent the token from immediately faulting again it should be directed to a lower priority token queue. Each time a faulted token is recirculated adds an overhead cost of a full cycle; however, faults should not occur

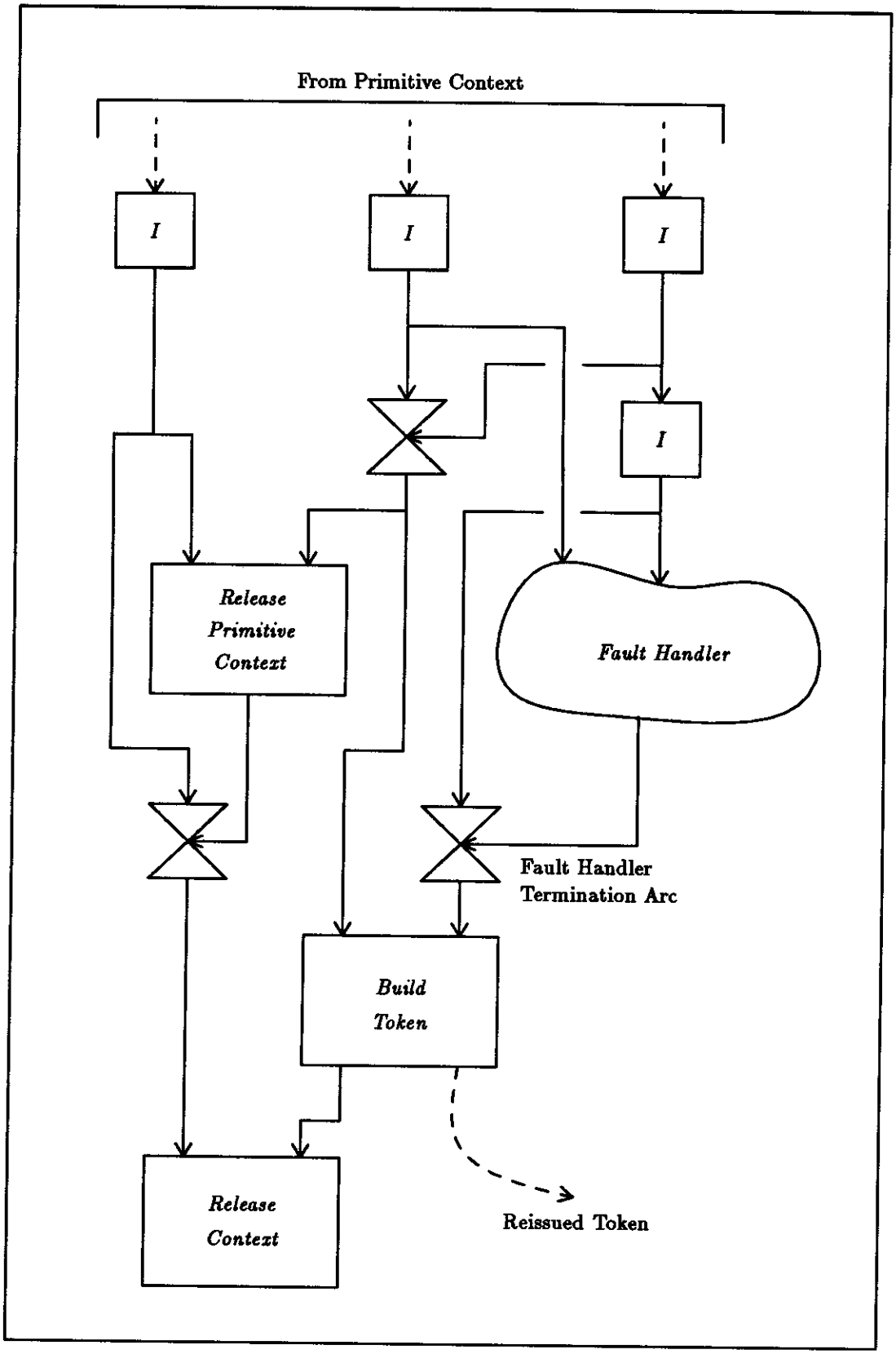

Figure 2: Outline of Virtual Memory Fault Handler Program Graph

frequently and the primitive context is kept as short as possible and released as soon as possible. See Section 5.1 for alternatives.

In Monsoon, the fault trap instruction checks the aforementioned fault inhibit bit. If the flag is not set, then the faulted token is sent to the fault handler primitive context and the flag is set. If the flag is set, then the faulted token is reformed and emitted onto a low priority token stack. An alternative would be to implement a separate fault stack; then the fault handler could pop the fault stack after it released the primitive context.

3.7 Avoiding Pipeline Faults

Faults in a pipeline are difficult to handle because not only the instruction that caused the fault is effected but all of the instructions in the pipeline. This presents extra complications in a dataflow computer because the order of instruction in the pipeline is determined at run time. Therefore, avoiding faulting the pipeline is very desirable.

In our virtual memory scheme, faults are handled not by faulting the pipeline but by changing only the instruction that caused the fault. This is a crucial, though confusing point; an *address translation (software) fault* does not cause a *pipeline (hardware) fault*.

3.8 The Fault Trap Instruction

An instruction that causes a translation fault is replaced with the Fault Trap instruction. The purpose of this instruction is to send the faulted token to a primitive context fault handler. This is done by the Form Token section of the pipeline, which can generate two tokens per pipe beat. One of the tokens has the tag of the faulted token as its data and its tag is the first entry point of the primitive context. The other token is similar except that its data is the data of the faulted token; it is sent to the second entry point of the primitive context.

A fault can occur in either the instruction address TLB or the effective address TLB stages of the pipeline. For all of the stages following the stage in which the fault occurred, until the Form Token stage, the Fault Trap instruction acts as a No-Op. It does not access the presence bits or the data cache. This implies that a Fault Trap instruction cannot cause a fault.

4 Implications and Modifications

Implementing virtual memory has implications for both the Monsoon architecture and the programs executed on it. The changes for Monsoon are discussed below. Section 4.6 talks about the effects virtual memory will have on programs.

Effort has been made to change the Monsoon architecture as little as possible. The changes that are needed are discussed in the following sections.

4.1 Translation Look Aside Buffers

In Monsoon there are two addresses that will need to be translated to execute instructions. These include

- The instruction address. This is taken directly from the token entering the instruction pipeline.
- The effective data address. This is formed by combining information from the active token and the instruction referenced by the token (*i.e.*, from the program memory).

Monsoon is a pipelined machine; therefore, as noted before, two separate translation stages (each with its own TLB) are needed. The first translation stage is active just before instruction lookup, as it is necessary to find the physical memory address of the instruction before reading it from memory. The second translation stage is active just following the effective address generation stage. Figure 3 shows the basic Monsoon pipeline with the new translation stages and TLB's.

The translation stages and TLB's are the greatest hardware change to the currently planned Monsoon architecture, as they represent a lengthening of the pipeline and considerable extra hardware. Direct access to modify TLB entries will be needed by the fault handlers and other system software.

It is important to note that dataflow processors may require larger TLB's than similar von Neumann architectures. This is because of the requirement of causing a page fault whenever a virtual-to-physical mapping is not found in the TLB, even though the virtual page addressed might reside in physical memory.

4.2 Fault Inhibit Flag

The fault inhibit flag is a hardware flag which stores the state of the primitive context used to handle faults. This flag is checked in the Form Token section of the pipeline by the fault trap instruction to determine whether to send to token to the primitive context or recirculate it.

When the flag is set the primitive context is in use; faulted tokens are recirculated. When the flag is cleared, the primitive context is ready to accept a faulted token.

The flag will require a small amount of hardware. It is the first piece of state stored in the CPU. Instructions to set, clear and read the value of the flag will be needed. The flag will be set when a token is sent to the primitive context; it will be explicitly cleared by the fault handler to release the primitive context.

4.3 Form Token Stage

The Form Token section will need the ability to generate different tokens based on the state of the fault inhibit flag. Also, a register in the Form Token stage will hold the tag of the entry point of the fault handler primitive context, referred to as the fault tag. The fault tag is used to form the tokens sent to the primitive context.

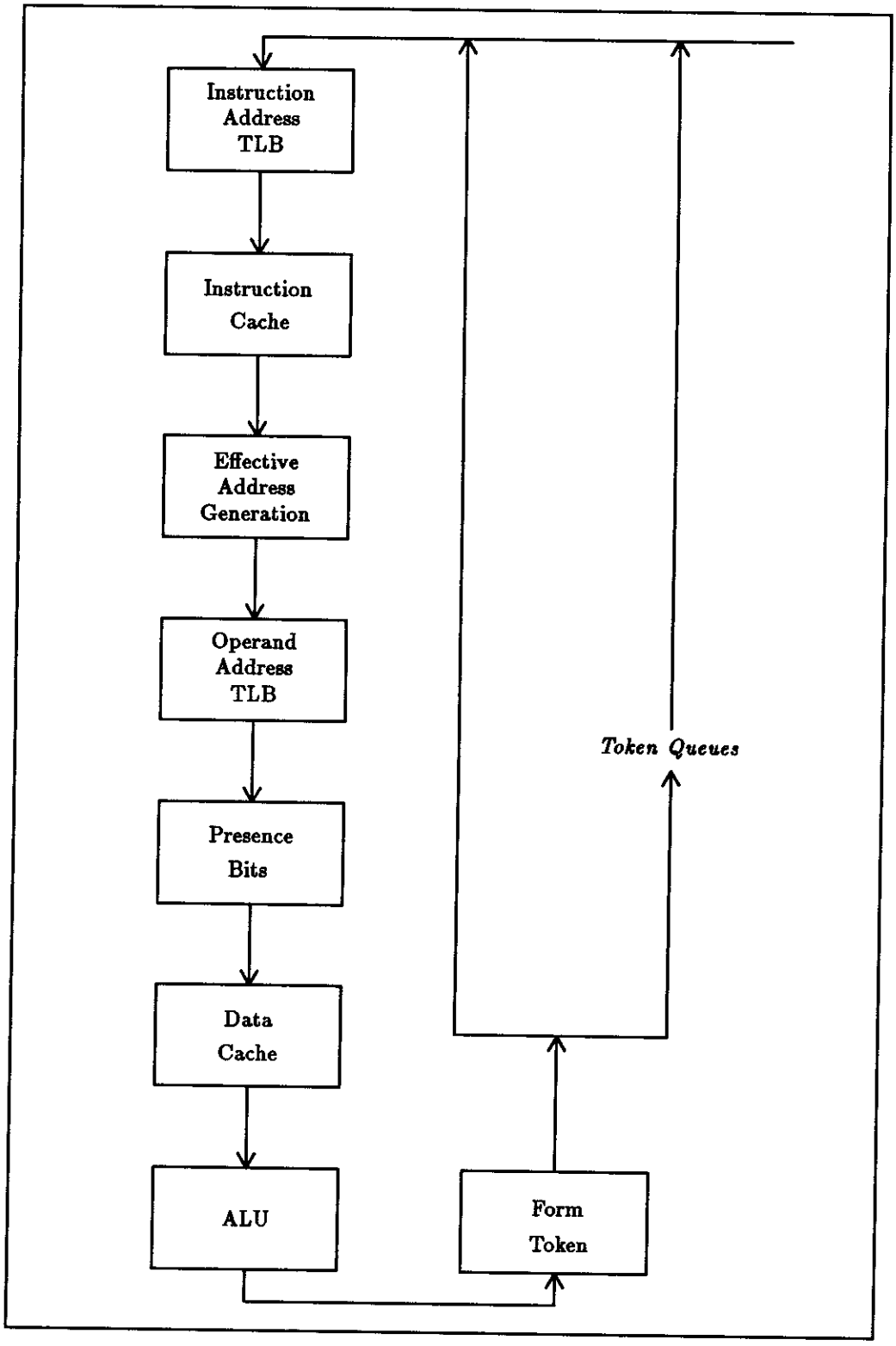

Figure 3: Modified Monsoon Pipeline with TLB Lookup

4.4 Primitive Contexts

A primitive context is exactly like any other context, except that it is allocated at system initialization and is always resident in memory. Its address is therefore permanently fixed for all code block executions. The fault inhibit flag is used as a special form of context allocation to keep different faults from using the primitive context at the same time.

Two consecutive instructions can cause faults. This is a problem because only one fault can use the primitive fault handler context at a time. To avoid this problem, a fault inhibit flag is used in the pipeline to lock the primitive context while it is being used to start the fault handler in a new context. A token that causes a fault while the primitive context is busy is reformed and recirculated. It will fault again later when the primitive context should be free. Once the fault handler determines that the primitive context's program graph is clean it releases the primitive context by clearing the fault inhibit flag.

More than one primitive context can be used. See Section 5.1.

4.5 DMA devices

The device that moves data between physical memory and secondary storage will need to generate acknowledgement tokens when the transfer has been completed. This is equivalent to the use of interrupts in serial computer systems. The fault handler will be suspended waiting for the acknowledgement token.

If the fault handler is run on an I/O processor then it will generate a token to acknowledge that it has handled the fault. This token is then used to reissue the faulted token and to release the fault handler context.

4.6 Program Implications

The results generated by a program will not be effected by using virtual memory except that larger programs can be executed. In particular, the ideal semantics of the graph language (and more important, the source language) will *not* be affected by any of the changes outlined in this paper. How virtual memory may effect the execution time of a program is discussed in the following sections.

4.7 Extra Pipeline Stages

Adding TLB's will add two stages to the pipeline. This will not effect the rate at which instructions are issued (*i.e.*, the throughput), but it *will* lengthen the critical path of programs. Assuming the depth of the pipeline without virtual memory is d, then the critical path will be lengthened by

$$\frac{d+2}{d}$$

For the pipeline shown in figure 3 this is a 33% increase in minimum instruction latency.

If caches are used, it may be possible to overlap the virtual address translation with the cache access, thus eliminating the extra two steps. This common technique would return the pipeline to it original depth. It of course requires that the memory caches thus constructed would use *virtual* keys, rather than translated *physical* keys.

4.8 Fault Handling Overhead

As is the case with all code blocks without dependencies, the fault handler will be executed in parallel with the parts of the program not effected by the fault. This will use instruction cycles that might otherwise have been used by the program, or might have been wasted idling. Again, it is important to note that virtual memory address translation is a form of increased memory latency, and that dataflow architectures inherently cover latency well[2].

4.9 I/O Processors

An alternative is to run the fault handler on an I/O processor dedicated to handle faults and I/O devices. The I/O processor could be a standard microprocessor and could support several Monsoon PE's.

The handling of page faults is itself a source of parallelism in a system. For higher performance, separate I/O processors could handle faults, either directly accepting tokens from faulting instructions or just handling requests to swap page frame contents with backing store contents. These processors would then restart the computation by reinserting the faulting token into a standard processor.

4.10 Page Tables

The primitive context, fault handler graph (code block) and first level page tables will need to be locked down in memory, reducing the amount of physical memory available for page frames containing other system and user data. As in any other virtual memory scheme, the pages of the second level page tables may be paged exactly like any other pages. If a separate I/O processor is used to handle page faults then the page tables can be kept in the memory of the I/O processor.

5 Future Directions

5.1 Multiple Primitive Contexts

Multiple primitive contexts can be used to allow faults to be handled more quickly or to trap to different fault handlers. For example, one primitive context for instruction fetch faults and another for data address faults. Some extra hardware is needed to implement this feature.

One fault inhibit bit is needed for each primitive context. For multiple primitive contexts handling the same type of fault, hardware is needed to determine which, if any, of the fault inhibit flags are not set, so that a fault can be directed to the associated primitive context.

In this case the graph for the primitive contexts can be shared, with each having its own activation frame.

Having more than one primitive context handling the same type of fault reduces the chances that a faulted token will be recirculated instead of being sent to a primitive context. With two primitive contexts, the first two faults are always handled and it takes a third fault, during the time the first two primitive contexts are busy, to cause the token to be recirculated.

The fault handler needs to know which primitive context it was called by so that it can release the correct fault inhibit flag. This is done by passing another argument from the primitive context to the fault handler.

5.2 Caches

Design of the cache and virtual memory architecture are closely related. For a paged system the size of the virtual memory page is related to the size and associativity of the cache. There is also the choice between a virtual or physical cache. With a properly sized physical cache it is possible to do the TLB and cache look ups in parallel. See Section 4.7.

References

[1] Arvind and J. Dean Brock. Resource managers in functional programming. *Journal of Parallel and Distributed Computing*, 1(1), January 1984.

[2] Arvind and David E. Culler. Dataflow architectures. In *Annual Review of Computer Science, Volume 1*, pages 225–253. Annual Reviews, Inc., Palo Alto, California, 1986.

[3] Arvind and David E. Culler. Managing resources in a parallel machine. In *Fifth Generation Computer Architectures*, pages 103–121. Elsevier Science Publishers B.V., 1986.

[4] Arvind, Michael Dertouzos, and Robert A. Iannucci. A multiprocessor emulation facility. Technical Report 302, M.I.T. Laboratory for Computer Science, 1983.

[5] Arvind and Rishiyur S. Nikhil. Executing a program on the m.i.t. tagged-token dataflow architecture. Computation Structures Group Memo 271, M.I.T. Laboratory for Computer Science, 1987.

[6] R. S. Nikhil and Arvind. Id Nouveau. Computation Structures Group Memo 265, M.I.T. Laboratory for Computer Science, 1986.

[7] Gregory M. Papadopoulos. An engineering implementation of the TTDA. Computation Structures Group Memo 270, M.I.T. Laboratory for Computer Science, February 1987.

[8] Jerome H. Saltzer. Protection and the control of information sharing in multics. In *ACM Fourth Symposium on Operating Systems Principles*, October 1973.

[9] Richard Mark Soley. *On the Efficient Exploitation of Speculation Under Dataflow Paradigms of Control.* PhD thesis, M.I.T., Expected 1988.