Proposed Model for Multithreading and Event Handling in the Aries Architecture

J.P. Grossman

1 Overview

At the heart of the Aries architecture is a multithreaded processor which must support efficient processing in the presence of frequent context switches and asynchronous events. In this document we propose the use of **writeback buffers** to support efficient multithreading. Writeback buffers are a mechanism which permit multiple threads of execution to co-exist in the functional unit pipelines while allowing only a single thread to access the register file on any given cycle. We argue for a new model of traps and executions, collectively referred to as **events**, which treats them as local rather than global phenomena. We propose a hardware/software organization in which one hardware context is reserved for a software event handler. The handler polls a hardware **event queue** for events and deals with them using short non-blocking sequences of instructions. Potentially blocking memory operations are avoided by converting them to an explicitly split-phase form in which the return event invokes a continuation. [**October 19, 2000**: writeback buffers are probably a bad idea, but I've left the section in there for completeness].

2 Multithreading

A single thread of execution typically cannot issue instructions on every cycle due to cache misses and branch mispredictions. This problem is compounded in architectures which do not provide branch prediction or data caches (such as the Aries architecture). Hardware multithreading can recover many of these lost cycles and improve silicone efficiency by allowing different threads to issue instructions on consecutive cycles, thus allowing bubbles in one thread to be filled by another. An additional potential benefit is the elimination of the context switch overhead normally required to service asynchronous interrupts. If a hardware context is reserved for handling events, then no state needs to be saved when an interrupt occurs or restored when the handler exits, and both context switches occur in a single cycle.

The benefits of hardware multithreading are compelling, but must be weighed against two significant costs:

1. Cache and/or local memory must be shared between a number of different threads
2. Registers must be duplicated for each hardware context

There is no good solution to the first problem. A dangerous temptation is to increase the size of the cache/local memory to compensate for the extra threads. However, to do this is to forget that the goal of multithreading is to provide improved silicon efficiency. If both the registers and memory are to be duplicated, then one might as well improve performance significantly at a small additional cost by using multiple single-threaded processors rather than a single multithreaded one. A hardware designer completely blind to reality might then propose the next logical step which is to add some additional registers to each of these processors to make them multithreaded… lather, rinse, repeat.

The second cost is not quite as large as it seems. Although the registers must be duplicated, the dominant cost in a multi-ported register file is not registers but wires. A register file with m read ports and n write ports requires $m+n$ control lines per register and $m+n$ bit lines per bit (Figure 1a). A naïve approach to duplicating the register file k times is to simply duplicate all the control lines, resulting in a register file k times as large (Figure 1b). However, if only one thread can issue instructions on a given cycle then only one thread can read from the register file on a given cycle. Hence, m read control lines can be shared by k registers, so the number of control lines required is only $m+kn$ per register (Figure 1c). Typically $m \approx 2n$, so this reduces the number of control lines from $3kn$ to $(2+k)n$. For $k = 4$, this represents a factor of two savings in area.

We can further reduce the area overhead by imposing the restriction that on a given cycle at most one thread can read or write the register file. This allows

Project Aries Technical Memo ARIES-TM-07
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA

both read and write control lines to be shared by k registers, eliminating most area overhead (Figure 1d). However, this restriction implies that all pipelines must be drained before performing a context switch, since otherwise instructions from one thread may complete and need to write to the register file on the same cycle that another thread is reading from the register file. Consequently, much of the advantage of hardware multithreading is lost since the original motivation was to be able to fill small (say 1-5 cycle) pipeline bubbles.

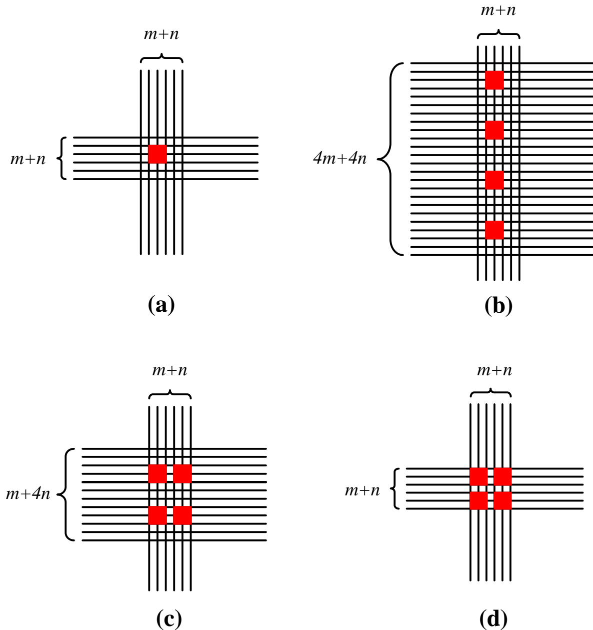

Figure 1: Read and write ports

2.1 Writeback Buffers

It is possible to share both read and write control lines without sacrificing single-cycle context switches by placing **writeback buffers** at the end of each functional unit pipeline. There is one writeback buffer per hardware context in each pipeline. The purpose of the buffers is to temporarily store results belonging to one thread while a different thread has control of the register file.

Each buffer is structured as a FIFO queue. Results enter the queue as they are produced, and they exit the queue when the corresponding thread has control of the register file. If the correct thread has control when the result is produced and the queue is empty, then the writeback buffer may be bypassed altogether.

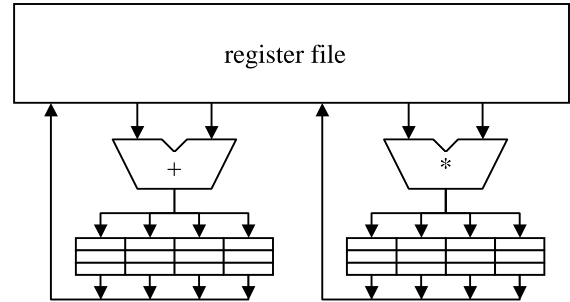

Figure 2: Writeback buffers

A key observation is that only small queues are required; in particular, the number of entries in each queue should be the same as the number of stages in the corresponding pipeline. If a pipeline has t stages then the pipeline and buffers combined would never contain more than t instructions belonging to a single thread. To see this, note that when a $(t+1)^{th}$ instruction is issued the thread has control of the register file and at least one instruction will be ready to write back (since the pipeline has t stages). Hence, on the same cycle an instruction will write back and exit the pipeline/buffer, so the number of instructions in progress will never exceed t.

2.2 Alternate Approach

The motivation for writeback buffers is to allow write control lines to be shared, reducing the number of control lines per register from $m+kn$ to $m+n$. An alternate approach to reducing the number of control lines is to attempt to minimze n. Indeed, we see that there is much more to be gained from reducing the number of write ports than from reducing the number of read ports. These considerations can help guide the search for a reasonable register file-functional unit connectivity pattern (a search which must be undertaken regardless since it is probably impractical to provide full connectivity to four or more functional units).

It is difficult to say a-priori which approach is better. An informed decision will require simulated performance comparisons of various register file-functional unit connectivities. If good performance can be achieved with only one or two write ports, the writeback buffers are probably undesirable. If more write ports are required, then control line sharing may provide enough of an area savings to justify the implementation of writeback buffers.

3 Events

Historically, interrupts, traps and exceptions, which we collectively refer to as events, have been viewed

as catastrophic occurrences which bring the entire system to a halt. The active thread is suspended at a well-defined point and swapped out. Control is transferred to a handler which services the event; when the handler finishes it swaps in a user thread (the same one that was previously running or a different one). This is a global model in that whatever part of the system generates the event, the effects are immediately visible everywhere, and no computation is allowed to proceed until the event has been atomically serviced. While this model is conceptually simple and convenient for debugging, it has a number of drawbacks:

- Complex hardware is required to maintain the illusion of precise interrupts in out-or-order, superscalar or VLIW processors
- Stringent restrictions are placed on the compiler concerning the reordering of instructions that might generate exceptions
- Useful computation at various pipeline stages must be aborted when an event occurs
- When an asynchronous event occurs, it is necessary to abort or complete all active instructions before the handler can take over. This includes long latency and/or non-local operations such as remote reads in a shared memory machine.

An alternate model for events which avoids these problems is to treat them as local phenomena which affect, and are visible to, only those instructions/hardware components which directly depend on the hardware/software operation that caused the event. As an example of the difference between the global and local models, consider the program flow graph shown in Figure 3, and suppose that the highlighted instruction generates an exception. In the global model, there is a strict division of instructions into two sets: those that precede the faulting instruction in program order, and those that do not (Figure 3a). The hardware must support the semantics that at the time the exception handler begins execution, all instructions in the first set have completed and none of the instructions in the second set have been initiated. In the local model, only those instructions which have true data dependencies on the faulting instruction are guaranteed to be uninitiated (Figure 3b). All other instructions are unaffected by the exception, and the handler cannot make any assumptions about their states.

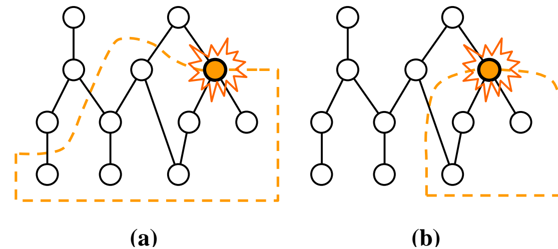

Figure 3: Global (a) vs. local (b) exceptions

The local model is better suited to parallel and distributed computing, in which the execution of a single thread may be physically distributed across the machine. Another example of the difference between the two models is provided by memory traps. In the Aries architecture, tag bits are associated with every 128 bit memory word which can cause a trap when that word is accessed. With a global exception model, a thread would have to stall on every remote memory reference. If a remote memory reference completes successfully, the thread is allowed to continue. If a trap is raised, this information must be returned to the processor where the thread is preempted by the trap handler. With a local exception model, a thread may continue processing while waiting for a remote memory reference to complete. If the reference causes a trap, the trap is serviced on the remote node, independent of the thread that caused it, and the trap handler completes the memory request manually. This is transparent to the thread; the entire sequence is indistinguishable from an unusually long-latency memory operation.

Local events are much easier to implement in hardware, they allow the compiler to produce better code and they improve overall performance. However, they also eliminate the precise interrupts which can be so useful for debugging. This is unfortunate, but in our opinion local events are a necessary step in the evolution of computer architecture. The sooner we accept this and start investigating novel debugging paradigms, the less painful it will be in the long run.

3.1 Event Handling in the Aries Architecture

The challenge of architecting an event handling mechanism is to simultaneously address all of the following issues:

- The overhead of invoking an event handler should be as low as possible
- Specific events must be correctly mapped to specific event handlers

- Events which occur while a handler is already running must be handled correctly
- The hardware must be able to deal with events caused by event handlers

Imagine an ideal processor with an infinite number of hardware contexts available for handling events. In this processor, every event can be handled immediately by a new context. No state needs to be saved, so there is no context switching overhead. There are always more contexts available, so there is no need to worry about nested events, i.e. events that are caused by a handler or that occur asynchronously while a handler is running.

We can achieve a similar effect with only a single event-handling context by maintaining an **event queue** of events waiting to be handled. A privileged thread runs in an endless loop, polling for events and handling them when they occur (Figure 4). A special polling instruction is provided which suspends the context when the event queue is empty. For the most part, this arrangement appears to the rest of the processor as an infinite number of event handling contexts, where some events take longer to handle than others. It also has the advantage that no hardware "trap vectors" are required since the task of starting up the correct handler is performed by software. However, it forces events to be handled in sequential order, which raises the possibility of deadlock. For example, if an event handler causes a page fault on a read, then it will stall indefinitely waiting for the read to complete, and the page fault will never be serviced.

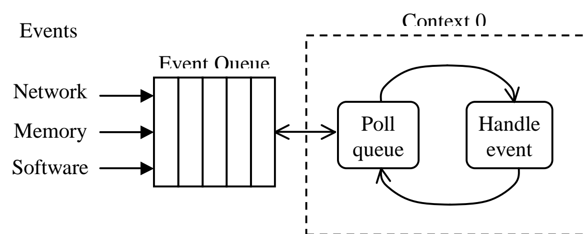

Figure 4: Event queue and event handler context

We can eliminate the possibility of deadlock by guaranteeing (in software) that event handlers do not block and finish executing in a finite (preferably small) amount of time. This ensures that forward progress can always be made independent of the pattern of events which occur. For the most part this is easy to do; software exceptions can be avoided through a combination of careful programming and masking, preventing the handler from blocking unexpectedly. The difficulty lies in performing memory operations, since every memory reference can potentially generate a page fault. Before presenting a solution to this problem, we first review the Aries memory model.

3.2 Memory Operations in the Aries Architecture

In a conventional system, memory references take a fixed amount of time and can be performed atomically in a pre-determined number of cycles. In a distributed shared-memory system this is not the case, and remote references can take an arbitrarily long time to complete. Because of this, memory references become split phase operations; a request is placed on the network (or sent directly to local memory), and some (potentially large) number of cycles later a reply is sent back to the processor. If the thread which issued the request needs the result long before it is available, it is desirable to be able to swap the thread out to memory so that the processor may continue to perform useful computation. However, this introduces the complication that when a memory reply arrives at a processor, the target thread may not be in one of the active hardware contexts.

In the Aries architecture, memory requests include a *return address* to which the reply is sent. Return addresses consist of a processor ID, a processor-dependent thread ID, and the name of the destination register specified by the memory instruction that generated the request. If the thread ID matches one of the active contexts, the result is stored directly in the appropriate register, and the "present" bit for that register is set. If the thread ID does not match any of the active contexts, then an event is generated so that the reply is handled by software.

3.3 Split Phase Memory Operations and Continuations

We can take advantage of the mechanism described in the preceding section to convert potentially blocking memory operations into an explicitly split-phase form. All that is needed is a privileged instruction which allows the event handler thread to set the contents of its thread ID register. We also assume that replies to context 0 memory requests are always placed on the event queue (there are several easy ways to do this). A memory operation then consists of the following steps:

1. Save the event handler's state to local memory
2. Replace the thread ID register with a pointer to this state

3. Initiate the memory operation
4. Jump back to the start of the event handling loop to handle the next event

Eventually, the event handler will pick up the reply to this memory operation which will include the pointer to the saved state. This pointer serves as a continuation and allows event processing to resume where it left off by restoring the state and then jumping to the instruction following the jump-back (4, above).

The only apparent problem with this approach is that it requires two sets of memory operations; saving the handler's state to memory, and then retrieving it when the reply event is processed. Clearly these memory operations cannot be handled in the same way, or we end up with an infinite recursion. Note that the memory operations are local; the handler can therefore explicitly check to see if the required page is in-core before attempting to save/restore the state. If it is, then it is safe to proceed as no page fault will be generated. If it isn't, then the handler can initiate the required page transfers and simply wait for them to complete. This is drastic as it has the effect of freezing the processor to wait for one or more page transfers, but it ought to be rare if one or two pages are reserved specifically for saving/restoring handler state, and if the kernel makes an attempt to keep a certain percentage of pages free at all times (as all good little kernels do).

3.4 Finite Queue Size

Another important difference between the ideal infinite context model and the more practical event queue model is that finite event queues can potentially fill up. This can have serious performance consequences, as it can cause any combination of hardware thread contexts, memory banks and the network interface to freeze up waiting to place an event on the queue. It is therefore important to choose the size of the event queues so that the probability of being filled is small. It is also necessary to prove that there is no possibility for system deadlock when many or all event queues fill up simultaneously.