The Hamal Processor-Memory Node

J.P. Grossman

1 Overview

The Hamal processor-memory node can be broken down into five distinct components: a processor, data memory, instruction memory, a network interface and a controller (Figure 1). The processor is a 128 bit multithreaded VLIW processor with predicated execution and single cycle context interleaving, and can be used to make coffee [Grossman00]. The embedded-DRAM data memory includes SRAM page tables, is virtually addressed, and supports simple atomic memory operations. The read-only instruction memory consists of single-cycle access SRAM and also includes hardware page tables. The network interface manages three distinct types of transactions: remote memory requests (including replies), forks, and page transfers. The controller primarily serves as an arbiter for the many producers and consumers of data, but also contains some state for controlling memory requests, forks, page transfers, and context loading.

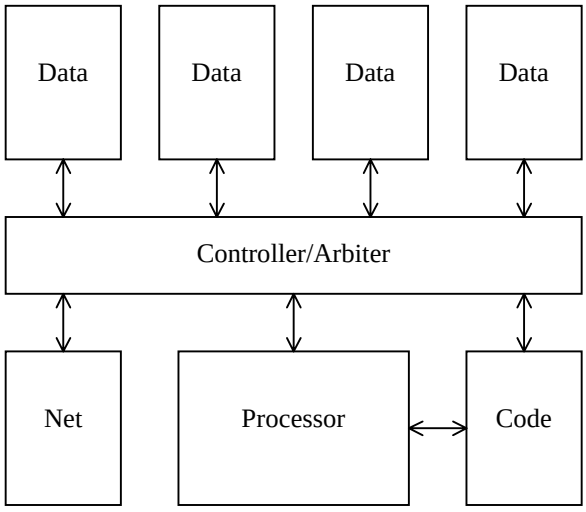

Figure 1: Node overview

This document describes each of these components in detail. It is assumed that the reader is familiar with the Hamal ISA.

2 Processor

Figure 2 shows the structure of the processor. The processor contains four hardware contexts. Each context has its own trace control and instruction queue (shown explicitly), registers, and memory request table (not shows explicitly).

Project Aries Technical Memo ARIES-TM-10
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA

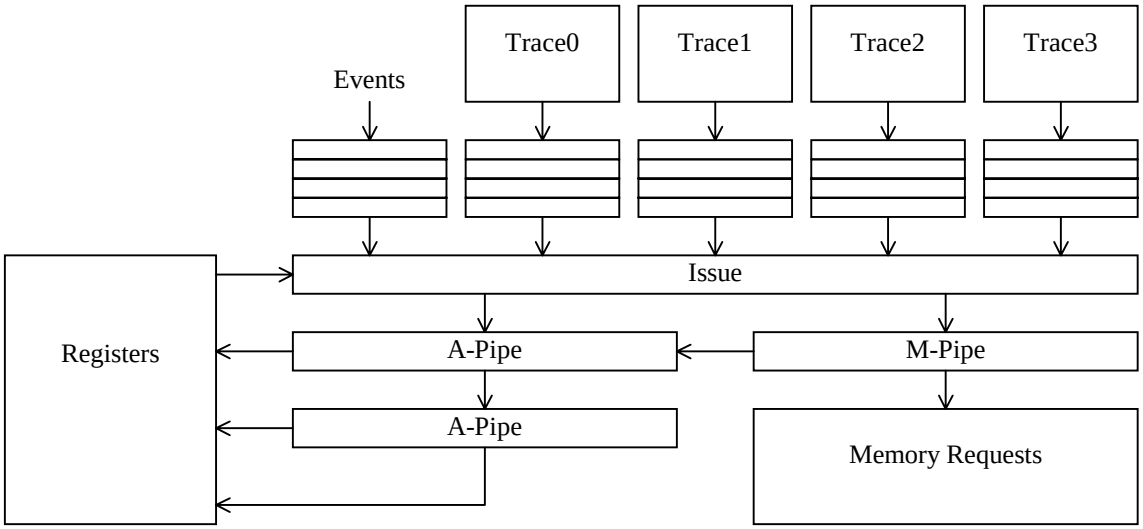

Figure 2: Processor

2.1 Trace Control

Each context has its own trace control logic. This logic is responsible for fetching the next instruction group from memory and executing the control portion of this group (if any). The control instruction may depend on one or more of bp, tr, and a predicate register. If one of the dependencies cannot be resolved then the trace controller stalls and re-attempts to execute the control instruction on the next cycle. Once the control instruction has been executed, the trace pointer is updated and the instruction group is placed on the FIFO issue queue. In order to support traps, the previous and updated trace pointers are placed on the queue along with the instruction group. On each cycle, all four trace controllers operate in parallel. The only restriction is that only one of them may access instruction memory on a given cycle; permission to do so is granted in a round-robin fashion.

If the trace controller encounters either a bad instruction or an invalid trace address, an event is placed on the instruction queue. This event causes a trap to take place once it reaches the M-Pipe (see section 2.7).

2.2 Issue

The issue stage of the pipeline consists of choosing a context to issue, removing an instruction group from that context's instruction queue, and fetching the source operands for the instruction group. A different context may issue on each cycle. When more than one context is able to issue, priority is given to context 0, and the remaining contexts are allowed to issue in a round-robin manner.

A context is ready to issue when all of the following are true:

1. There is an instruction group in the queue
2. The instruction group's dependencies are not busy
3. The instruction group does not require a resource which is in use. This applies only to the *fork*, *cload*, *cstore* and *pgout* instructions.
4. If the memory instruction is *wait*, there are no outstanding memory requests
5. If the memory instruction is a memory request, there is room in the context's request table to drain the instruction from the M-Pipe
6. If the arithmetic instruction is *poll*, there is an event in the event queue

2.3 A-Pipe

Arithmetic instructions each take 1-3 cycles to execute. In the absence of a writeback conflict, instructions write their result to the register file on the same cycle in which they complete. The result is also available to issuing instructions via bypasses on the same cycle. However, only one arithmetic writback may occur on a given cycle. If two instructions complete in the same cycle, then the older instruction is allowed to write back and the younger instruction is advanced through the A-Pipe. Note that single cycle instructions provide their result on the cycle after they are issued and thus never cause issue delays.

2.4 M-Pipe

The first M-Pipe execution stage consists of address computation or comparison. For address computation instructions with no actual memory request, the writeback occurs immediately and the instruction is removed from the pipeline. Otherwise the instruction progresses to the second pipeline stage.

In the M-Pipe second pipeline stage an attempt is made to enter the memory request into a context-specific request table and pass it on to the node controller. This may fail if:

- A previous request which was rejected by the node controller is being re-attempted
- The request has a consistency conflict with another request already in the table
- The table is full
- The event high-water level has been reached

In all three cases, the memory request is placed in a smaller context-specific buffer (which only contains two entries, enough to guarantee that all issued memory instructions can be drained from the shared pipeline) and re-attempted on a subsequent cycle.

2.5 Busy bits

All valid destination registers have associated busy bits which are used for dependency interlocks. This includes all 128 32-bit general purpose registers, the 31 predicate registers, bp, tr, tv and s0. When an instruction that modifies one of these registers enters the first execution stage, the corresponding busy bit is set. When the register is written to, the busy bit is cleared. Additionally, busy bits may be set explicitly by the *busy* instruction.

Note that busy bits alone are not sufficient for the trace controller to correctly detect dependencies, since a control instruction may depend on a register which is mutated by an instruction sitting in the instruction queue. Thus, an additional set of pre-busy bits is associated with bp, tr and the predicates. These pre-busy bits indicate that there is an instruction in the instruction queue or at the issue stage which writes to the corresponding register. When checking dependencies, the trace controller inspects both busy and pre-busy bits.

2.6 Predicated Execution

Each instruction may be independently predicated on the value (true or false) of one of the 31 predicates. For control instructions, predication is performed by the trace controller. For arithmetic and memory instructions, predication occurs at the first execution stage. The predicate is treated as a source operand and fetched along with the other operands at the issue stage. If an instruction is squashed, the busy bit for its destination is not set (this is why busy bits are set at the first execution stage rather than at the issue stage; it avoids having to set and then clear busy bits when an instruction with a destination is squashed).

2.7 Traps

Traps, also referred to as thread events, all occur at the first execution stage of the M-Pipe. In particular, there are no arithmetic traps. This simplifies the hardware design by defining a precise commit point for instructions (which is the same as the predication commit point) and avoiding the possibility of multiple traps on a single cycle. When a trap occurs, the event registers are set with the appropriate data, the instruction queue for that context is cleared, the trace pointer is set to the trap vector (or the context 0 trap vector if the given context's trap vector is invalid), and tr is set to the address of the next instruction.

2.8 Context Dribbling

The general purpose register file has five read ports for servicing the issuing context, and also a sixth read port for the *dribble context*. In response to a *fork* or *cstore* instruction, one of the four contexts is set to be the dribble context. On subsequent cycles, one word at a time may be dribbled from the context to the controller in order to service the fork/cstore.

2.9 Interface to the Controller

The processor communicates with the node controller via the following inputs and outputs:

2.9.1 Instruction Memory Busy Bit

A single bit input indicates that the instruction memory is busy. This occurs during a page-in or a memory operation. When this bit is high, no trace controller is allowed to access instruction memory.

2.9.2 Data Out

This output is used for both memory requests and context dribbling. With the exception of *fork*, all memory requests are passed on to the node controller in a single cycle. For *fork*, the first cycle places the address in the address field and the mask, predicates and context address in the data field. The general purpose registers are then dribbled out in subsequent cycles. *cstore* is converted by the processor to a series of *store128* instructions, so the controller never actually sees a *cstore*. The output must be valid at the start of the clock cycle. Later in the cycle, a single bit input indicates whether or not the request was accepted by the controller.

There are potentially four different sources of memory requests for the data output: the second M-Pipe execution stage, a context memory request buffer, a failed memory request from a previous cycle, and the dribble context. Priority is given to requests in the following order:

1. Failed context 0 request
2. context 0 buffer
3. M-Pipe context 0
4. Failed non-context 0 request
5. non-context 0 buffer
6. M-Pipe non-context 0

In the absence of a failed non-context 0 request, the dribble context has the lowest priority. However, if a non-context 0 request fails, the dribble context is given higher priority than the failed-context 0 request and lower priority than an M-Pipe context 0 request.

2.9.3 Data In

A single path is used for all incoming data. The input consists of a valid bit, a tagged 128 bit data word, and a 64 bit return address. The upper 54 bits of the return address specify the destination thread swap address; if this does not match any of the currently executing threads an event is generated. The next two bits identify the type of the data. For memory request replies with a destination, bits 0-7 identify the destination register. For memory request replies without a destination, bits 0-2 give an index into the outstanding request table. For context data (this applies to *cload* and *fork*), bits 0-5 indicate which 128 bit word of context data is being loaded. Contexts are activated when the highest context word is loaded (forks just supply zero as data for this word). Incoming data must always be accepted by the processor; there is no busy output bit.

2.9.4 Events

A fairly wide event input allows the controller to pass on memory and network events to the processor to be added to the event queue. The processor does *not* need to accept events; a single busy output bit indicates to the controller than no event will be received on the current cycle. Additionally, an event high-water bit informs the controller that the event queue is filling up. When this bit is set, the controller will not accept network events, joins, or remote

memory requests. At the same time, the processor will suppress all non-context 0 memory requests. This ensures that the event queue will never fill up, so that local memory and hence context 0 can always make forward progress.

There is one tricky point. Context 0 will often need to page in data to service memory events. When a page is loaded it generates an EV_PG_IN. This event cannot be suppressed because it is essential for forward progress to be made. However, there could potentially be a large number of page-ins in progress, and the resulting large number of EV_PG_IN events could overwhelm the event queue. To solve this problem, the event queue is actually divided into two separate queues: a regular event queue, and an EV_PG_IN event queue. This latter queue has 64 entries (to allow for up to 64 simultaneous page-ins), but is small because each entry is only 64 bits (as opposed to 512 bits for a general event). Events in this queue are given priority for poll instructions. The event high-water signal is generated based solely upon the state of the regular event queue.

There are three potential sources of events: the event input, the data input (EV_REPLY), and processor events. They are given priority in that exact order (only one new event per cycle is allowed). Processor events may be arbitrarily suppressed; the offending context simply waits for a chance to submit its event. If an EV_REPLY is suppressed, the reply is placed in a special buffer, then on the next cycle the event busy output bit is set (to prevent the controller from supplying another event) and the reply event is placed on the event queue.

3 Data Memory

The Hamal processor-memory node contains four banks of data memory. Each bank consists of 1Mb of embedded DRAM divided into 128 1KB pages, and an SRAM hardware page table with one entry per physical page. Each page table entry contains the virtual base address of the page (32 bits), status bits (32), and P, T, U, V bits for each of the 64 words in the page (256 bits). Thus, the total size of the page table is 128 x 320 = 40Kb = 5KB.

Memory instructions are directly executed by a small controller in the memory bank. Each instruction consists of a memory opcode, 8 uv bits, 3 invert bits (for atomic memory operations – see ISA document), and a single no-trap bit which specifies that the T trap bit should be ignored. When an instruction is completed, the memory bank asserts an output valid bit and returns to the controller either nothing, return data, or an event.

3.1 Paging

Data memory responds to a page-out instruction in 66 consecutive cycles. On the first two cycles, the data output contains the P, T, U and V bits for the page. On the subsequent 64 cycles, the data output contains the 64 words in the page. The node controller must accept each of these outputs immediately and forward them to the network interface.

A page-in acts like a page-out in reverse. On the first cycle, the node controller presents the page-in instruction to the memory bank along with the page address and the P, T bits in the data input. On the next cycle the data input contains the U, V bits, and on the subsequent 64 cycles it contains the 64 words in the page. The page table entry for the page being paged-in must exist or the page-in will fail.

4 Instruction Memory

Each node contains a single 128K bank of SRAM instruction memory. As with data memory, there is an SRAM page table with one entry per physical page. However, this page table does not contain P, U or V bits, so its size is 128 x 128 = 16Kb = 2KB.

Instruction memory is read-only, so only control instructions and loads are supported. All memory operations complete in a single cycle. Page-outs are also not supported. Page-ins are similar to data page-ins, except that no P, T, U or V bits are paged in, so the page-in takes 64 cycles (as opposed to 66).

5 Network Interface

The network interface can be divided fairly neatly into two symmetrical halfs; an 'in' half and an 'out' half. Each half consists of a number of memory request buffers, a page buffer, and a translation cache. A single fork buffer is shared between these two halves (this simplifies local forks).

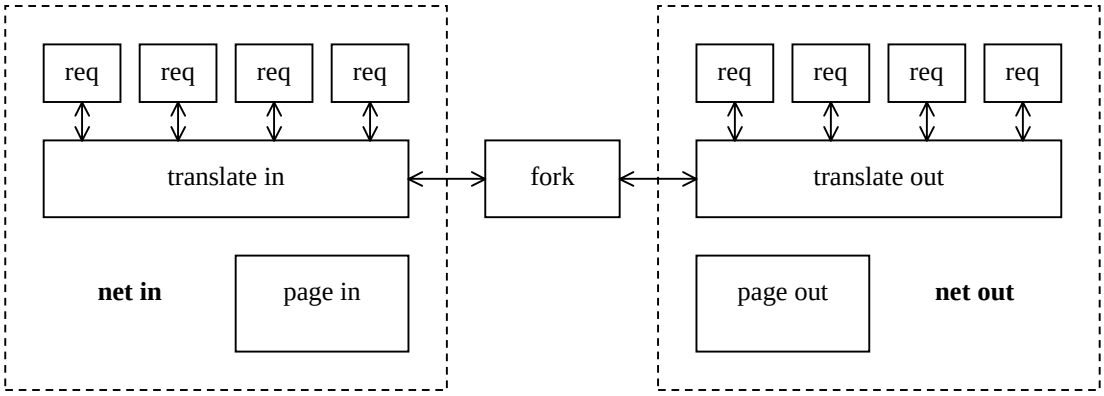

Figure 3: Network Interface

The network interface is responsible for translating pointers to sparse objects as they migrate across the node boundary. On each cycle, a single local→global translation and a single global→local translation can be performed. When a memory request or fork is fully translated, it is passed on to the controller ('in' requests) or the network ('out' requests). If a pointer causes a translation cache miss, an event is sent to the controller. When the processor supplies the missing translation, the translation caches are updated and any pending translations snoop the update directly.

Both requests and events may be rejected by the node controller; this is indicated by two 'accept' bits. Outgoing requests must be accepted by the network interface; it is the controller's responsibility to ensure that there are always enough request-out buffers to absorb all active requests with remote destinations.

On a given cycle, several requests, a fork and a page transfer could potentially all be ready to be passed on to the processor or network. Round-robin scheduling is used to break ties. When an attempt fails, the round robin pointer is advanced.

6 Controller

The controller is primarily a gigantic arbitrated crossbar which manages the flow of data between various producers and consumers (Table 1). All inputs to the controller must be valid at the start of the clock cycle. Inputs which may be rejected by the controller have a corresponding 'accept' output bit which is valid by the end of the clock cycle. All outputs are valid by the end of the clock cycle and must be accepted by the destination components.

Producers	Consumers
data memory	data memory
instruction memory	instruction memory
network interface	network interface
fork buffer	fork buffer
cload controller	processor data return path
processor	processor event input

Table 1: Producers and Consumers managed by the node controller

6.1 Internal State

The controller contains a small amount of internal state to manage memory requests, paging, context loading and forks. When a memory request is sent to data/instruction memory, the controller stores the return address so that

when the operation completes the reply can be sent to the correct location. The controller can manage one page-in and one page-out at a time; during a page transfer data is streamed directly between the memory bank and the network interface. When a *cload* instruction is received from the processor, a finite state machine converts it into a sequence of *load128* instructions. Finally, the controller is responsible for generating a fork event when the fork buffer is filled, and transferring the contents of the fork buffer to the processor or memory in response to an *fload* or *fstore* instruction.

6.2 Priority

The controller assigns priority to the various data producers in the following order:

1. Active page transfers
2. Instruction memory
3. Data memory (round-robin priority breaks ties)
4. Processor
5. cload controller
6. Network interface
7. Fork buffer

6.3 Rejecting Inputs

In addition to rejecting inputs when the requested consumer is unavailable, the controller will selectively reject inputs in order to avoid running out of space in the event queue or the network interface request-out buffers.
When the processor asserts the event high-water signal, the controller will reject memory requests and joins from the network. Forks are still accepted, because at most one fork event at a time can exist in the event queue (until it is serviced by context 0, the fork buffer will be unavailable). Page-ins are also accepted, since EV_PG_IN events are placed in a separate event queue (see Section 2.9.4).

The controller keeps count of the number of active memory requests with remote destinations to ensure that the network interface has enough free request-out buffers to sink these requests when they complete. Both processor and network memory requests with remote destinations are rejected if there are not enough free request-out buffers.

References

[Grossman00] J.P. Grossman, “A High Performance Coffee Maker”, Project Aries Technical Memo ARIES-TM-09, Artificial Intelligence Laboratory, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, November, 2000.