LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ACD Requirements

Computation Structures Group Memo 357
June, 1994

B.S. Ang
D. Chiou
J. C. Hoe
X.-W. Shen
Massachusetts Institute of Technology

J. Morris
University of Tasmania

S.K. Nandy
Indian Institute of Science, Bangalore

and
M.J. Beckerle
Motorola Cambridge Research Center

Limited Distribution

Distribution of this report is limited to these CSG members: B.S. Ang, Arvind, G.A. Boughton, A. Caro, D. Chiou, Kyoo-Chan Cho, D. Henry, J.C. Hoe, C.F. Joerg, J. Morris, Hanpei Koike, Rajat Moona, S.K. Nandy, Andy Shaw, X.-W. Shen

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 Defense under the Office of Naval Research contract N00014-92-J-1310.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

ACD Requirements

B.S. Ang, D. Chiou, J. C. Hoe, J. Morris, S.K. Nandy, X.-W. Shen and M.J. Beckerle

June, 1994

Abstract

*T-NG was designed as a message passing and distributed shared memory multiprocessor. A *T-NG system consists of a number of sites (each consisting of a number of processors) connected by a high speed routing network. The address capture device (ACD) is the key component of each site which provides the distributed shared memory capability. The ACD monitors addresses emitted by processors in its site and collaborates with one processor on that site - designated the network processor - to send messages to other sites to fetch and store data. The ACD also detects transactions needed to maintain coherent caches over the whole machine and requests the network processor to forward messages to other sites. This memo describes the design requirements for an ACD in a *T-NG system.

1 Introduction

This copy of this report was current on June 27, 1994 at 12:05.

1.1 *T-NG

The *T-NG system was designed as a message passing multiprocessor that also supports cache-coherent distributed shared memory in order to exploit temporal and spatial locality in memory reference patterns. The *T-NG concept is described in detail elsewhere[6, 7].

A *T-NG system consists of a number of sites connected by a switch fabric of Arctic routers[2]. A possible implementation of a site is depicted in Figure 1. This figure shows a site configuration containing four NES modules (each consisting of a data processor, network interface unit (NIU) and an Arctic router). It is possible to replace three of the NES modules in Figure 1 with a module containing two processors (each with a second level cache). Each processor in a site may carry out normal computation (*i.e.* be a data processor or **dP**). There are two groups of functions which are separate from the main computation stream. A processor, designated the Network Processor (**nP**), is responsible for

- formatting and transmission of messages when requested by the ACD.

A second group of functions are assigned to a processor designated as the Protocol Processor (**pP**): it is responsible for:

- processing of remote memory references sent to a site from other sites *and*
- maintaining directories of remotely cached lines to support a coherent cache scheme.

There is considerable flexibility in the assignment of these tasks to physical processors: one processor may perform the functions of both **nP** and **pP**, but other task allocations are possible.

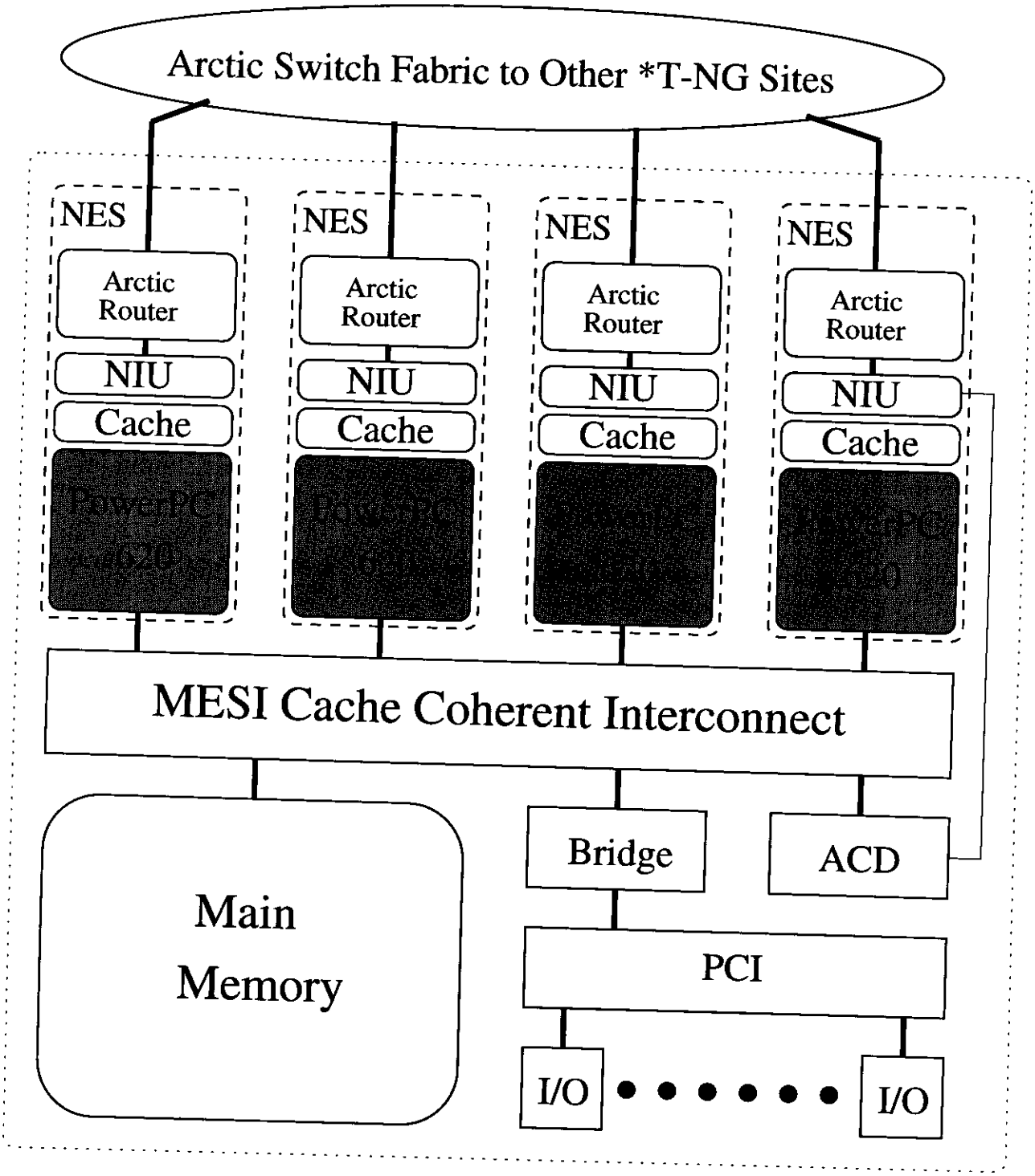


Figure 1: *T-NG architecture

The optimum configuration for a site will depend on the type of programs to be run on it: this report applies to any system configuration in which each site has at least one NES module whose processor can assume the role of the **nP** and **pP**. Various alternative architectures for Hermes, the communications unit for *T-NG are discussed in [1].

The Address Capture Device (ACD) is a key component of a *T-NG system as it provides the distributed shared memory capability. The ACD's function in a *T-NG system is the detection of bus transactions which refer to shared memory locations, the capture and storage of sufficient information to enable the **nP** to format and transmit messages to remote locations *and* also enable coherent caches to be maintained across all sites.

This report is organized as follows: the remainder of this introduction contains a glossary of terms used. Section 2 sets out the requirements for a *T-NG system. Requirements for the ACD are derived from these system requirements. Section 3 describes the ACD operation, beginning with a discussion of memory addressing, following with descriptions of blocking and non-blocking memory operations and listing some design issues that need further information or study. Section 4 describes the functions of the ACD hardware. Outstanding design issues are identified in Section 5.

1.2 Nomenclature

The following terms are used in this report:

site	a collection of dual-processor modules, memory, I/O device interface (containing the ACD), and one or more NES modules
processor	a data processor in a site (in this context, processor may refer to the processor on a dual-processor or NES module)
dP	a synonym for processor
process	any one of the multiple Unix processes
ACD	Address Capture Device - a block of fast logic which is designed to capture an address transaction from the bus within the time constraints imposed by the PowerPC processor
SMU	Shared Memory Unit - term used in the original *T-NG papers for the collection of state machines and buffer logic which provides support for remote loads and stores by generating messages and the part of the cache coherence protocol which is not handled by the **nP** - included the ACD. In the architecture described in this report, all the functions of the SMU are handled by the **nP**, rendering the SMU unnecessary as a separate hardware unit.
nP	network processor: the processor in one of the NES modules which processes messages for the ACD (replaces **sP**)
pP	protocol processor: the processor in one of the NES modules which is responsible for maintaining the cache coherence protocol - usually the same processor as the **nP**, but the architecture permits distinct processors to carry out the two functions
NES	Network Endpoint System - a module (presumably contained on a single PCB) comprising a processor (**dP**, **nP** or **pP**), NIU and an Arctic router

L3 bus — The main processor-memory-I/O device bus[1]: a three-level bus architecture is assumed - an internal L1 bus supports the primary cache and a fast external L2 bus supports a co-processor or 2nd level cache, thus the main system bus is L3

TIF — Transactions-In-Flight - transactions which are currently propagating through the network or being processed by a **pP** at a remote site

2 System Requirements

This section summarizes the requirements for a *T-NG system. Some background constraints on the whole system are

1. The machine will consist of a number of PowerPC-based sites each consisting of four processor module slots. Each slot may contain a dual processor module or an NES module (at least one of the slots must contain an NES module), memory and an interface to I/O devices.
2. The operating system will be an essentially un-modified Unix.
3. The additional hardware to support message passing and shared memory will be as simple as possible commensurate with performance.

2.1 Programs

The system must be able to run C and FORTRAN programs such as those in the SPLASH benchmark suite. Thus it must provide global shared memory and a mechanism for passing messages.

It is intended that it will also be able to run programs compiled from implicitly parallel languages, *e.g.* pH, efficiently.

2.2 Shared Memory

Individual processes running on any site (or any processor within a site) will be able to create threads which will run on any processor in the machine. Those threads will be able to see a large virtual memory space which may be distributed over the physical memory of all the sites.

2.3 Threaded Code

Either hardware of software (or some combination thereof) will need to provide support for

1. split-phase loads from remote memory locations,
2. joins,
3. starting threads on remote processors *and*
4. suspending and resuming threads on the local processor.

[1]The term *bus* is used here and elsewhere to refer to a general interconnection matrix, which may or may not be a bus in the conventional sense.

Split-phase loads will have two flavours -

1. cached - a remote memory location is loaded into a register *via* the local processor's cache
2. uncached - a remote memory location is fetched into a local frame.

Cached loads will be needed when the compiler generates code to convert a local cache miss to a split-phase load. Uncached loads will be generated by pH and other compilers loading values to local frames and expecting values to be cached using the local frame address as the cache tag rather than the global address which will be used for cached loads.

2.4 Operating system

A Unix variant will run on the data processors in each site. This implies:

1. Processes may be interrupted at any time, *e.g.* when a time-slice ends.
2. The memory system may not block processes for unbounded times.

Thus the **nP** will need to keep track of outgoing blocking transactions and be able to complete them after it has assumed that so much time has elapsed that the network or receiving processor has had some error, *e.g.* a receiving processor has died. Some mechanism to alert the process receiving invalid data from an **nP**-ACD completed transaction will be needed.

If a protocol of 'heart-beat' messages were implemented, then processes would be able to determine precisely that sites or individual processors had failed and thus enhance reliability. However it alone would not allow for messages lost or corrupted by the network.

2.5 Reliability

Applications and operating system software are primarily responsible for handling network faults. The system must provide sufficient information to allow reliable protocols to be implemented in software. It may be assumed that the communications hardware is sufficiently reliable that responsibility for implementation of error management and recovery protocols can be left to software.

Normal messages (generated by user processes) must not be able to block high priority messages (generated by kernel software).

Memory management and protection throughout the system must be at least as strong as that provided on individual processors, so that user programs are unable to corrupt the operating system and operating system faults on one site do not affect the operating system on another site.

2.6 Performance

Global cache coherence will need to be maintained. Message generation will need to consume as few cycles in the data processors as possible. Message latency through the network will need to be minimized.

3 ACD operation

3.1 Memory Addressing

3.1.1 Memory addresses

The address space seen by any process will be divided into local and shared spaces. A process which is running on more than one site will have a distinct local address space on each site, but the shared space will be common to all threads of the process. (This separation allows the bus-snooping protocol to maintain cache coherence for the local address spaces and only requires the ACD to monitor shared addresses.)

3.1.2 Virtual Memory

All shared addresses pass through two memory mapping units - one on the requesting **dP** and one on the **nP** of the home site.

For a shared address, the first translation, which takes place in the requesting **dP**'s MMU, converts a shared virtual address into a *pseudo-physical address* (which - because the **dP** hardware believes it to be a physical address - is constrained to a maximum of m bits, where m is the number of bits in a physical address impressed on the bus). This pseudo-physical address is sent to the home site's **nP** which treats it as virtual address and transforms it to a physical address on the home site. If the page containing this address is paged out, the home site's page tables are updated: the requesting site's page tables are unaltered as they point to a location in the home site **pP**'s virtual address space[2].

This enables *T-NG to provide all the facilities normally expected of a virtual memory system

- page faults are handled transparently and
- the protection and sharing capabilities of individual MMUs are available globally.

With the dual translation, there is no need to maintain consistency of page tables, as only the home site page tables change on paging operations.

3.1.3 Pseudo-physical addresses

The pseudo-physical address will have the format:

Bits	r	p	q
Function	*selector*	*site_id*	*address*

The processor's MMUs will hide the mapping of *site_id*'s to actual processors from application software. One simple scheme would interpret a zero *site_id* as a reference to unshared memory on the current site, a *site_id* matching the site's id as a reference to shared memory on the current site and any other value as a reference to a remote site. Note that if this scheme is adopted, the ACD may be able to ignore references to shared memory on the current site: this will depend on the cache coherence protocol used and therefore would ideally be made a configurable option.

[2]Note that the pseudo-physical address (40 bits) is considerably smaller than a virtual address (52 bits): thus total shared memory space is constrained by the physical address space size.

Since there are m bits in a physical address impressed on the bus, then

$$m = p + q + r$$

It is assumed that p will be some small integer (say 6-8). The r bits of *selector* may include a bit designating *global/local* and for *kernel/user* address spaces. r may also be zero.

3.2 *T-NG architecture

The system in which the ACD resides is illustrated in Figure 1. Each processor shown has a direct connection via an NIU to every other similarly equipped processor in the network[3]. One of these processors will perform the functions of the **nP** - sending messages to implement blocking loads and stores and maintaining cache directories.

3.3 Principles of Operation

There are two major operations which the ACD must support:

Blocking Loads/Stores A blocking load or store is a reference to a memory location on a remote site which is satisfied while the processor waits.

Non-blocking Loads A non-blocking or *split-phase* load is a reference to a remote memory location which, if there is a miss in the local caches, the software in the requesting processor converts to a message requesting the data and then suspends the current thread of computation.

Remember that non-blocking loads come in two flavours: those which return values into registers and leave them cached at the global address and those that load data into local frames (*cf.* Section 2.3). We are concerned here with the former type as the latter type is handled entirely by the transmission of messages and doesn't need intervention from the ACD.

Note that due to the 620's ability to pipeline reads and writes, the notion of a 'blocking' load or store is somewhat more complex than it might be in a simpler processor. However, since the number of outstanding data transactions is limited, the processor must eventually block in the conventional sense until one of the 'blocking' transactions completes.

3.3.1 Basic Function

The ACD will monitor the address bus and associated control lines; when it detects references to shared locations, it will latch the address. If the address refers to a remote location which is not locally cached in a modified state[4], the ACD will complete the address bus transaction so as to free the address bus. It will also need to capture the signals which define the type of transaction and the **dP** from which it originates.

[3] It is possible to add processor modules to the system which contain a pair of processors and no NIU.
[4] The 620 cache will pushout a modified cache line, but not one marked exclusive or shared.

3.3.2 Blocking Loads/Stores

When a **dP** makes a reference to a memory location in a remote site, the sequence of operations will be (see also Figure 2):

1. The ACD will recognize the transaction by noting that p bits of the physical address appearing on the bus refer to a remote site.
2. The ACD will capture the address in a block of internal memory addressed by the *processor_id|transaction_id*.
3. The ACD will also capture and store the signals identifying the type, size, etc of the transaction.
4. If the transaction is writing to memory, the ACD will read up to a full cache sector from the data bus.
5. The ACD will signal the **nP** that there is a blocking transaction for it to process.
6. The **nP** will read the transaction data from the ACD's memory.
7. The **nP** will format a message and transmit it to the remote site.
8. The **pP** on the remote site will receive the message, read or store the data, format a reply message and despatch it to the originating **nP**.
9. The requesting **nP** receives the return message and places the data in the ACD's memory.
10. The **nP** alerts the ACD to complete the transaction.
11. The ACD completes the bus transaction.

When completing the bus transaction, the ACD must ensure that the state of the cache line in the requesting **dP** is:

shared if a read was issued *or*

exclusive if a write (appearing as RWITM on the bus) was issued.

Thus the ACD must assert the AResp lines appropriately.

Lost or corrupted messages

PowerPC processors cannot start to process interrupts while there are outstanding bus transactions. This means that a Unix process context switch cannot take place while blocking transactions are incomplete. Since the failure of the network should not lock up the whole system, if a message is lost, a mechanism must be provided to enable the requesting process to be swapped out. The kernel must then be advised that the process lost some data and the kernel should send a signal to the process to that effect. (It will be up to the process itself to determine how to handle the signal: the optimum procedure will depend on the individual process - some might simply abort, others back up to the last checkpoint, *etc.*) The **nP** will need to keep TIF records and advise the ACD to

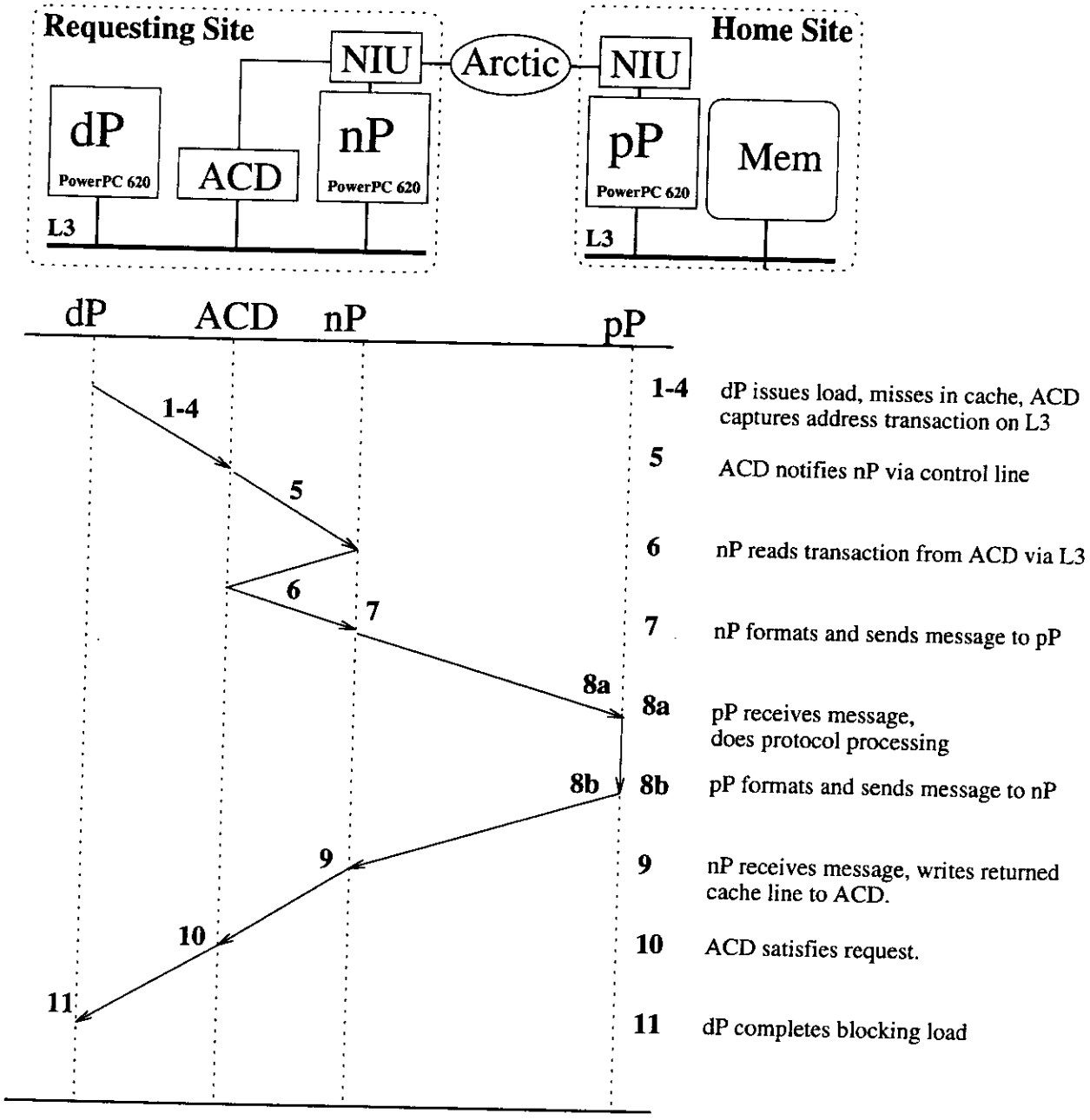


Figure 2: Steps in a blocking load

complete transactions for which no responses have been received after some interval. An optimum value for this interval will need to determined: it will depend on the size of the Arctic network and the amount of traffic. The Arctic routers themselves cannot guarantee to deliver a general (*i.e.* low priority) message in a finite time, although a reasonable upper bound to the delivery time can be derived readily enough.

There are a number of possible mechanisms for the **nP** to alert the Unix kernel that a process has received incorrect data: the optimum one will depend on the Unix implementation, but consideration needs to be given to allowing the ACD to interrupt any processor for this purpose.

3.3.3 Non-blocking ('Split-phase') Loads

Split-phase loads are designed to tolerate network latency by switching the processor to another computation thread after remote loads are initiated. The message sent to the remote site contains a *continuation* which is to be activated on the requesting site when the value from the remote site is available to the requesting thread.

To maximize cache utilization, the compiler will emit code which first attempts to load the shared datum from the local cache. If the software detects a special pattern (the miss pattern, **#miss**) returned from this attempt, it will assume that a cache miss occurred and switch to a sequence of instructions which forms a message requesting the same datum from the remote site. Once the message is despatched, the processor will switch to another thread so as not to idle while the remote access is being transmitted through the network.

This code relies for its efficiency on the compiler being able to select a data pattern which is highly unlikely (**#miss**) as a real data value. In general, this will be no problem, *e.g.* for floating point values a NaN pattern can be used. If the desired value is actually **#miss**, then it will be returned on the second read. Thus the code will become extremely inefficient, but remain correct, if **#miss** is a constant of the program! To avoid the pathological situation, it may be necessary to enable a process to set the miss pattern. The chosen value for **#miss** then becomes part of the process state and needs to be restored on a context switch.

In the following, we outline three schemes for handling blocking loads, in each case, we set out the code emitted by the compiler and consider the sequence of operations which are performed by the ACD, **nP** and **pP** for the following task:

> load register $\mathtt{r}_a$ from a global address, ga, in register $\mathtt{r}_{ga}$.

Register $\mathtt{r}_{miss}$ contains **#miss** and **msg**, **save_state**, **next** and **rest_state** are pseudo-operations for which the compiler emits appropriate sequences of instructions.

We explain why the naive Scheme A is not sufficient and discuss the conditions necessary for Scheme B. If the operation of **sync** and **tlbsync** allows Scheme B to produce a correct result, then Scheme C may be considered as a possible alternative which may give better performance under some operating conditions.

Scheme A

```
ld      r_a,r_ga
```

```
        cmp     crf_x,r_a,r_miss
        bne     crf_x,split
ok:     ...                 r_a loaded from [r_ga]
        ...
split:  dcbf    r_a         force miss pattern
                            from cache
        msg     sp_read,r_ga,sp_ret
                            split read msg
        save_state          Save state
        next                Jump to next runnable thread
sp_ret: rest_state
        ld      r_a,r_ga    This load must succeed
        b       ok          Continue thread
```

This code requires that the ACD is able to guarantee that the second read succeeds. Thus the value returned must be stored at the local site until the second read is issued. A first solution to this problem would have the **nP** store the returned cache line and its address in buffers in the site's main memory when the message returned from the remote site. When the second read arrives, the value is fetched from the buffer memory and supplied to the data bus. The line is then cleared from the local buffer memory. However it is possible - and highly probable - that another thread might also issue a read to *ga* in the interval between the return of the original value to the local ACD and re-activation of the original thread via the continuation. This would cause the ACD to assume that this read is the second one issued by the original thread, supply the value and remove it from its memory. When the second read from the original thread is finally issued, the ACD would assume it to be a first read and supply the miss pattern, leading to incorrect execution.

Scheme B

To avoid this problem, it is necessary to be able to set the ACD into two states:

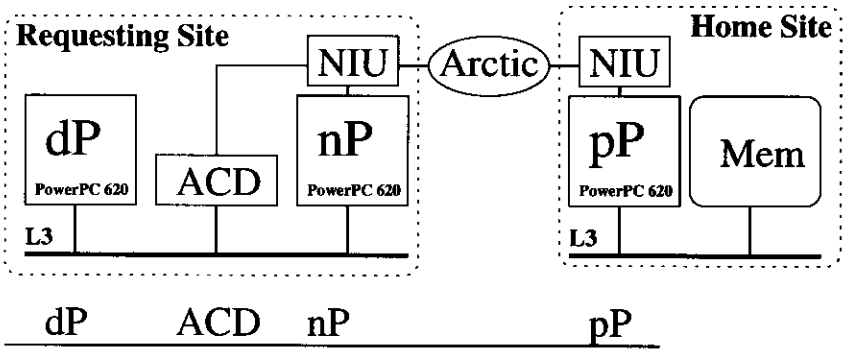
convertible Reads are convertible to split phase reads. the ACD and **nP** will attempt to satisfy the read from local memory (or L3 caches) and, if unsuccessful, will return **#miss**.

blocking Reads are blocking. The ACD and **nP** will attempt to read from the local memory and issue a blocking read to the remote cluster if unsuccessful. If the read is satisfied from the local memory, local memory will be cleared.

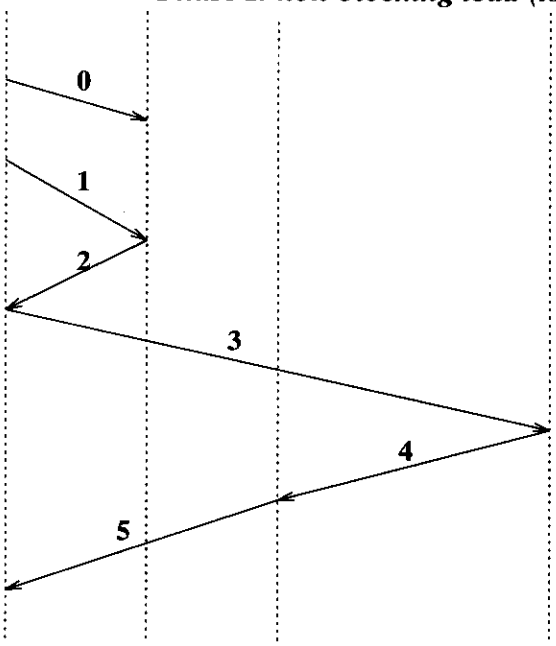
Thus the initial read is issued in **convertible** mode and the second in **blocking** mode.

Thus the compiler-emitted code needs to be: (r_{ACD} contains the address of the ACD configuration register, **#convert** is the command which when placed in this register sets the ACD into convertible mode and **#blocking** is the command which sets the ACD into blocking mode. The steps in this load are shown in Figure 3.

```
        st      r_ACD,#convert
                        ensure correct mode
        sync            force ACD configuration out
```



Phase I: non-blocking load (local miss->remote fetch)



0. dP syncs, writes ACD control to non-blocking;* syncs, performs load, syncs

1. dP issues load, misses in cache, address goes to L3 ACD captures address transaction

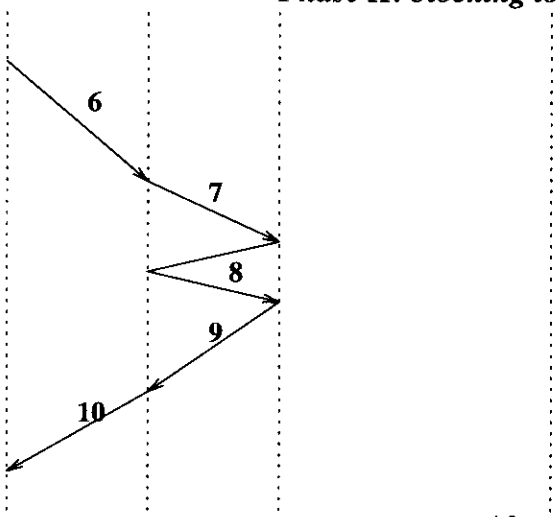
2. ACD returns miss pattern to dP

3. dP receives miss pattern, check, branches sends split-phase global caching load via its own NIU

4. pP receives message, does protocol processing returns cache line to nP

5. nP receives message, stores cache-line, writes continuation to dP queue

Phase II: blocking load (local hit)



6. dP reads continuation, executes
   dP issues load, misses in cache, address goes to L3
   ACD captures address transaction

7. ACD notifies nP via control line

8. nP reads transaction from ACD via L3

9. nP finds cache-line in software cache, writes cache-line to ACD

10. ACD returns cache-line to dP via L3

11. dP completes phase 2 of split-phased load

* It may be better to toggle between blocking and non-blocking, rather than setting it each time

Figure 3: Steps in a split-phase load

```
        ld      r_a,r_ga
        sync                    Note 1
        cmp     crf_x,r_a,r_miss
        bne     crf_x,split
ok:     ...                     r_a loaded from [r_ga]
        ...
split:  dcbf    r_a             flush miss pattern
        msg     sp_read,r_ga,sp_ret
                                split read msg
        save_state              Save state - Note 2
        next                    Jump to next runnable thread
sp_ret: rest_state
        st      r_ACD,#blocking
        sync                    Config must be written before ...
        ld      r_a,r_ga        this load is issued
        sync                    Note 1
        b       ok              Continue thread
```

Notes

1. Although reads are specified to have priority over writes in access to the bus, it is essential that this read completes before the ACD state can be changed by another store operation.
2. Because a context switch may occur between the **sync** and **ld** instructions, the state saved on context switch must include the ACD configuration register.
3. In order that the cache line is marked shared in the requesting processor's cache, the ACD must assert the appropriate lines when completing the second read.

Auto-reversion to convertible configuration

This scheme requires two instructions (one - **sync** - a potentially expensive one) to be inserted by the compiler ahead of *each* load from a shared address. If the ACD was to revert automatically to the **convertible** configuration after each load in the **blocking** configuration, then the cost of each load can be significantly reduced.

Note that we will need to distinguish between an **always blocking** configuration (needed for 'standard' blocking loads) and this **auto-revert blocking** configuration.

Scheme C

If we make the most pessimistic assumption (*cf.* Sections 3.4.1, 3.4.2) about the behaviour of the synchronizing instructions, Scheme B may also fail if the **nP** issues a synchronizing instruction after the second read has been issued by the **dP**, but before the ACD has been able to complete it, since this requires **nP** assistance. It appears that **sync** does not inhibit the issue of new bus transactions by the **nP** which are essential to the operation of Scheme B and thus does not preclude Scheme B. However, unless **tlbsync** operates in a very sophisticated manner (*cf.* Section 3.4.2), it will prevent new bus transactions being issued and thus make Scheme B unworkable.

In scheme B, the **nP** performs more of the overhead processing on behalf of the **dP**, but the **dP** is blocked while the **nP** transfers the requested line to the ACD. In addition, using Scheme B, the data may stay in the **nP** cache (and never be copied to main memory buffers as it can be invalidated after the second read), so Scheme B may result in lower overall bus occupancy and latency.

The following scheme does not rely on the ACD and **nP** cooperating to complete a transaction initiated by a **dP**. Depending on other demands on the system (particularly on the **nP**), overall system efficiency may improve if this scheme is used as an alternative to Scheme B.

Instead of holding the returned cache line in buffer space managed by the **nP**, the line is returned to the **dP** as part of the continuation. The **dP** reads the line from the continuation queue, *but must now arrange for it to appear in its cache at the right address and with the right state (shared).* It achieves this by writing the cache line into the ACD's buffer and setting the ACD into a 'supply next read from your buffer' mode. (Now both the ACD mode *and* the buffered cache line become part of the process state.)

In the following code example, r_{CQ} contains a pointer to the head of the continuation queue from which the continuation itself (IP:FP) has just been popped, r_{ACDbuf} points to the ACD buffer and **#bufload** is the ACD command which instructs it to supply the next shared memory load from its buffer.

```
        st      rACD,#convert
                        ensure correct mode
        sync            force ACD configuration out
        ld      ra,rga
        sync            cf. Scheme B note 1
        cmp     crfx,ra,rmiss
        bne     crfx,split
ok:     ...             ra loaded from [rga]
        ...
split:  dcbf    ra      force line from cache
        msg     sp_read,rga,sp_ret
                        split read msg
        save_state      Save state
        next            Jump to next runnable thread

; continuation is resumed here
sp_ret: rest_state
        lmw     r24,rCQ     load cache line
        stmw    rACDbuf,r24store in ACD buffer
        dcbf    rACDbuf     force line to write buffers
        st      rACD,#bufload
        sync            Config must be written before ...
        ld      ra,rga      this load is issued
        sync            cf. Scheme B note 1
        b       ok          Continue thread
```

Note that this code uses **lmw** and **stmw** for simplicity: other sequences of instructions may be needed or be more efficient.

This code sequence assumes that the ACD provides a configuration register and cache line buffer for each **dP** and that it uses the *processor_id* bits accompanying each transaction to address the appropriate buffer. If a process is interrupted and resumes on a different processor, then the configuration register and cache line buffer for the new processor are replaced.

The number of ACD buffers required can be reduced by providing a lock on a single ACD buffer. Acquisition of the lock could be combined with setting the ACD configuration register to reduce **dP** cycles required. Single-threading all processes through a single ACD buffer would affect performance, but the performance loss may not be significant. The lock is held for a very small number of instructions and the probability of a context switch (and consequent high probability of contention for the lock) in this small region may be acceptably low.

An alternative to locking ACD access is to save ACD state (including buffered data) on context switch and re-try ACD accesses from other processors while the ACD is in **bufload** state. The interval between setting the **bufload** state and clearing this state is sufficiently short (2 instructions in the example above) that re-trying the other processors is acceptable. On a context switch, the ACD state of the new process (generally this will *not* be **bufload**) is restored so that another processor waiting will be allowed to proceed. Providing a buffer per processor would eliminate the need for and potential performance penalty of re-trying.

3.3.4 L3 cache

In both styles of remote access, the **nP** handles the format and despatch of the message, so it becomes a straight-forward exercise to implement a software cache of arbitrary size for remote locations in the site's main memory.

If an L3 cache is present, it may be optimal to consult the L3 cache (by alerting the **nP** which is managing it) before completing a load with **#miss**: this issue is under study.

3.3.5 General Message Passing

dPs needing to send messages for synchronization, uncached blocking loads, *etc.* can do so efficiently via the NIU attached to their L2 cache bus. Thus any system which needed to run applications making extensive use of message passing capabilities would presumably be configured with four NES modules so that any processor can send messages directly.

The ACD is therefore not required to participate in application generated messages.

3.4 Design Issues

This section highlights several areas where operation of the 620 may create difficulties in achieving the desired functions. In some of these areas, the precise behaviour of the processor could not be ascertained from the available documentation.

3.4.1 sync behaviour

An initial examination of the 620 documentation led to some doubt as to the precise behaviour of the **sync** instruction. We have since been advised by Seye Ewedemi (Motorola, Austin) that

A sync seen by a snooper is not re-run by that snooper unless there are pending snoop-based pushbacks.

As the 'correct' behaviour of **sync** is vital for the avoidance of deadlock in a number of situations, the original question is documented here. *All operations in which the ACD and the* **nP** *must cooperate to complete a bus transaction are susceptible to this problem.*

As noted in Section 3.3.3, there is a potential for deadlock if any processor issues a **sync** instruction while an operation on which the ACD and **nP** need to cooperate is in progress. The following sequence of operations illustrates the problem:

> A **dP** issues a global blocking load which the ACD captures. The ACD signals the **nP** to complete the transaction. The **nP** must now read the information from the ACD, thus requiring a bus transaction. However, as part of some completely unrelated operation, the **nP** has executed a **sync** instruction right after the **dP** issued its blocking load. The **nP** has not issued instructions to load the ACD information at that time - in fact, it probably does not even know about the ACD request at the point it executed the **sync**.
>
> The **nP** may not issue any more general bus operations until the **sync** it executed completes, although it may be able to issue **pushout**'s and other responsive bus transactions[5]

Question: Does the **sync** have to wait until the blocking load completes? If it does, there is a deadlock, since the **nP** must complete the **sync** and issue more bus operations in order to complete the **dP**'s global blocking load. If the **sync** does not have to wait until the blocking load is complete, than there is no deadlock in this situation. We need to find out what the **sync** instruction needs to complete and what may occur while a **sync** instruction is outstanding. We also need to determine the behavior for instructions other than loads.

Although it now appears that the **sync** *instruction will not cause this problem, this point should be checked carefully!*

3.4.2 tlbsync behaviour

The **tlbsync** instruction has the potential to cause the problem discussed in the previous section for **sync**.

As currently implemented, **tlbsync** requires completion of all outstanding transactions on a site with a common shared set of page tables *before* new transactions can be initiated. This implies that if any processor, in the process of completing alterations to the page map tables as a consequence of a page fault, issues a **tlbsync**

- after a remote site blocking load has been issued and
- before the **nP** has been able to complete the outstanding transaction,

the system will deadlock. The **tlbsync** cannot determine whether bus transactions may have been caused by attempts to read page table entries which may be affected by previous **tlbie** operations and must be completed.

[5]The documentation available does not give sufficient detail in this area.

If **tlbsync** doesn't inhibit the issue of new transactions - simply requiring existing ones to complete - then deadlock may still occur if the **nP** issues a **tlbsync** after a remote blocking load has been issued and before the **nP** has been alerted to complete the load. This would require that the **nP** run only 'firmware' style code that excluded **tblsync**.

3.4.3 Critical Regions

Locks, semaphores and critical regions will be handled by passing messages to the **pP** on each site. Since the **pP** can effectively single-thread itself, then a variety of schemes for managing critical regions globally may be implemented in software. No support is required from the ACD.

Note that this assumes the ability to compile a program with a library of synchronization primitives appropriate for *T-NG. A program attempting to synchronize its threads with **sync**, (*i.e.* relying on its semantics being extended across sites) will not run on multiple sites as attempting to implement **sync** across sites will lead to deadlock situations.

3.4.4 Weak consistency

We can find no way to implement sequential consistency across sites when programs rely entirely on blocking loads and stores. Thus weaker consistency models will need to be adopted. From an efficiency standpoint, this is not a drawback.

The use of weak consistency and other relaxed cache coherency models is currently being explored.

3.4.5 Snoop Pushouts

Replacement of cache lines can cause data which must be written to memories on remote sites to appear on the bus independent of any program-generated transactions at any time. The ACD must capture these pushouts and alert the **nP** to process them.

Thus the ACD must provide at least one buffer to capture snoop-generated transactions. If the minimal single buffer is provided then further snoop-generated pushouts must be re-tried until buffer space is available.

3.4.6 Flushes required for cache coherence

Cache coherence protocols will generate flush operations to invalidate entries in processor caches.

These may be generated in a number of ways:

1. A **dP** may execute a **dcbf** instruction. If this refers to a shared location, it must be captured by the ACD and propagated to the L3 cache and other sites.
2. The **nP** may receive an invalidate command from a remote site. It may place the invalidate operation on the site's bus by issuing a **dcbf** instruction. In this case, the ACD may ignore the Write-w-kill bus transaction, but must must capture and pushouts generated by it.

A possible deadlock arises if the **nP** generates a Write-w-kill which is captured by the ACD for sevice by the **nP**. Thus either

Signal Name	Label	Bits	Monitored	Captured
Bus Operation Code	AType< 0 : 4 >	5	Yes	Yes
Size	ASizeData< 0 : 3 >	4	No	Yes
Burst Size	ASizeBurst	1	No	Yes
WIMAN codes	Address< 0 : 4 >	5	?	?
Processor ID	BusPID< 0 : 4 >	5	Index[1]	Yes
Transaction ID	TransID< 0 : 2 >	3	Index[1]	Yes
Address				
High bits	Address< $a : b$ >	p	Yes	Yes
Low bits	Address< $c : d$ >	$m - p$	No	Yes
Address Status	AStatIn< 0 : 1 >	2	Yes	No
Address Response	ARespIn< 0 : 2 >	3	Yes	Yes
Total			$p + 10$	$m + 21$

Notes

1. BusPID< 0 : 4 > |TransID< 0 : 2 > is used as an index into the ACD's transaction tables.

Table 1: Bus signals monitored or captured by the ACD

- the ACD does not cause the Write-w-kill to be re-tried until complete - making sequential consistency unachievable - *or*
- the ACD can recognize **nP**generated transactions and is thus able to ignore an **nP**-generated Write-w-kill.

4 ACD Hardware

4.1 Signals monitored

Figure 1 lists the signals which need to be monitored and captured by the ACD. *This table needs careful review.*

The ACD must be capable of completing the address transaction within tight time constraints imposed by the processor.

The ACD will also need to detect cache-coherency related transactions for shared addresses.

4.2 Configuration Register

A configuration register in the ACD can be written by a process running in a **dP** to signal the ACD that cache misses appearing on the L3 bus are to be converted to split-phase loads rather than processed as blocking loads. Since up to 7 processors could be issuing references to shared memory simultaneously, a configuration needs to be stored for each **dP**.

ACD state	Description
blocking	Read and store accesses are blocking
convertible	ACD responds with miss pattern
blocking-auto revert	ACD blocks on next load and then reverts to convertible
buffer-load	ACD responds to next load from its internal buffer

Table 2: ACD states required

4.3 ACD - nP control line

The ACD must be able to alert the **nP** that there is a transaction waiting for it to process. It could do this by:

- interrupting the **nP** using the single interrupt line *or*
- setting a bit in a memory mapped register (*e.g.* the ACD configuration register) which is polled by the **nP**.

Note that the requirement for handling lost or corrupted messages may require the ACD to be able to interrupt *all* the **dP**'s.

4.4 Responses to bus transactions

This section summarizes the responses that the ACD should make to the various bus transaction types.

4.5 ACD operations

For certain combinations of signals on the bus, the ACD will need to invoke a handler which will capture the signals that will be needed by the **nP** to complete the transaction. This section describes each handler in detail. The conditions under which each handler is invoked are set out in Table 3. Essentially there is one handler for every bus transaction type (as determined by the bus signals AType$< 0 : 4 >$) which needs to be recognized by the ACD, but in some cases, other signals on the bus (*e.g.* ARespIn) will determine which ACD handler to select.

ACD handlers have been given names with a _a suffix (*e.g.* **RWITM**_a) to distinguish them from the bus transactions of the same name which invoke them. Similarly messages which a handler causes to be sent to the home location by the **nP** are given names with a _h suffix (*e.g.* **DClaim**_h).

Cache coherence protocols

Different cache coherence protocols will require slightly different handling of each ACD request by the **nP**. The following descriptions indicate possible **nP** processing, but variations are possible.

Bus Operation	Program Operation	Cache State	ACD AResp Out	AResp In	ACD Action
read-cache (*or* read-burst)	LD DCBT LARX	Ca,I	S	Null S	**read_cache_a** See note 2.
				M	**read_transfer_a**
				Retry Rerun	Do nothing. A write-w-clean could follow.
RWITM	ST DCBTST STCX	Ca,Wb I	Null	Null	**RWITM_a** See note 2.
				M	No ACD action needed - data will be supplied by cache that intervened.
				Retry Rerun	Do nothing.
DClaim	ST DCBTST STCX	Ca,Wb S	Retry	Null	Not possible.
				Retry	**Dclaim_a**
	DCBZ	Ca,Wb IS		Rerun	Can this happen?
Write-w-kill	Deallocate DCBF	Ca,M	Null	Null	**write_w_kill_a**
				Retry Rerun	Do nothing.
Write-w-clean	DCBST	Ca,M	Null	Null	**write_w_clean_a**
				Retry Rerun	Do nothing.
Read-no-cache (also read-non-burst)	LD LARX	Noca	Null	Null	**read_non_burst_a**
				M	Should not happen.
				Retry Rerun	Do nothing.
Write-w-flush	ST DCBTST	Ca,Wt MESI			**write_w_flush_a**
	ST STCX	NoCa			

Table 3:

Questions

1. If a flush of the cache line appears on the bus, what happens to this outstanding operation?
2. Will every processor be able to see ARespIn?

Bus Operation	Program Operation	Cache State	ACD AResp Out	AResp In	ACD Action
sync	SYNC			Null	No action *cf.* Section 4.5.9
eieio	EIEIO				
larx-reserve	LARX	Ca,MES *and* L3 enabled			
Dkill	DCBI	Ca,MESI			
flush	DCBF	Ca,ESI *or* Noca			
clean	DCBST	Ca,SEI			
tlbsync	TLBSYNC				
tlbie	TLBIE				
ikill	ISYNC				
pio-load-immed					
pio-load-last					
pio-store-immed					
pio-store-last					

Table 4: Bus operations not supported globally

4.5.1 read_cache_a

ACD captures the address, completes the address part of the transaction with AStat=PosAck AResp=Shared and notifies the **nP**. The **nP** is alerted and sends a **read_cache_**h message to the home location. Eventually, a cache line worth of data is returned to the **nP** which writes it into the appropriate ACD buffer (addressed by the transaction tags). Writing the returned value causes the ACD to invoke the remainder of the **read_cache_**a handler to request the data bus and complete the data part of the transaction. See note 2.

4.5.2 read_transfer_a

A read from processor A has caused processor B to push out a modified copy. The ACD captures the address and data. The **nP** sends the data to the home with a **write_w_clean_**h message.

4.5.3 RWITM_a

This handler is invoked when AType=RWITM *and* ARespIn≠M. ACD captures address, completes address transaction with AStat=PosAck and notifies **nP**. The ACD's AResp will be PosAck. Completion of the data part of the transaction is essentially the same as for **read_cache_**a. See note 2.

4.5.4 DClaim_a

The DClaim bus operation with address X is used when a **dP** wants to write to a cache line that is currently in its cache in the S state. The purpose of this operation is to obtain ownership of cache line X so that the write operation can proceed. Since another process in a distributed environment may attempt to write to the same cache line at the same time, the DClaim operation should not complete until global ownership of the line has been obtained. This can be achieved by requiring the originating processor to re-try the DClaim operation until the home site surrenders ownership to the requesting site in the following manner:

The ACD first checks that it has not already initiated a message to the home in response to this transaction. If this is the first time, the ACD captures the address but asserts ARespOut=Retry to force retry until the **Dclaim**_h message can be sent to the home site by the **nP** and the acknowledgement is returned. ACD's transaction record will be marked so that retries continue to be retried without alerting the **nP** again until the acknowledgement arrives from the home. The **nP** then clears the retry flag in the ACD's transaction record and the ACD ceases retrying the DClaim bus transaction.

This method requires the requesting processor to retry the operation for an extended period, potentially wasting considerable bus cycles doing so. A more efficient alternative is to retry the DClaim when it first appears on the bus, but immediately perform a flush of cache line X after that. This would remove the cache line X from the **dP**. The **dP** will then have to issue a RWITM in order to perform the write. This alternative requires the ability to force a processor to abort a bus transaction and re-start it *after* having checked the cache line status (changed by the flush operation to invalid) again and thus changing the bus transaction issued from DClaim to RWITM.

It is not clear whether the 620 would permit this alternative.

4.5.5 write_w_flush_a

write_w_flush is an single word (non-burst) transaction issued when segments are marked WriteThrough or CacheInvalid. Other caches holding this line must be flused and the updated word written to memory. The ACD captures the address and data and responds with PosAck. The **nP** sends a **write_w_flush**_h message to the home location.

4.5.6 write_w_kill_a

The ACD captures the address and data and then notifies the **nP**. The **nP** will send a **write_w_kill**_h message to the home which will send an acknowledgement back.

4.5.7 write_w_clean_a

The processing for **write_w_clean**_a is identical to that for **write_w_kill**_a except that the message sent is **write_w_clean**_h which causes the home directory to note the retention of the clean copy on this site.

4.5.8 read_non_burst_a

The ACD captures the address and size and notifies the **nP**. The **nP** sends the **read_non_burst_h** This data word is not cached and consequently may require minimal processing by the cache coherence protocol. On receipt of the returned message, the **nP** will transfer the data to the ACD which will complete the data part of the transaction.

4.5.9 Synchronization transactions

As user programs can send synchronization messages, the following synchronization operations will not be needed beyond a site:

- **sync**
- **eieio**
- **tlbsync**

5 Outstanding Issues

5.1 Form of pseudo-physical address

The form of the pseudo-physical address needs to be resolved (*cf.* Section 3.1.3):

$$p = ?$$

i.e. how many bits will be devoted to the *site_id*?

$$r = 0 ?$$

Will the operating system make fixed divisions of the address space for specific purposes?

5.2 Operating system

When a process dies or is killed, outstanding messages which it generated must be flushed from the network - or at least allowed to disappear gracefully. They must not perturb other processes.

5.3 Cache coherence

Many issues related to maintenance of coherent caches have not yet been resolved.

Some points which require further work are:

- It is highly desirable to have a sizeable L3 cache for shared memory managed by the **nP**. This would enable effective sharing of global data between the **dP**s at a site. Without an L3 cache, a **dP** requesting a cache line which is currently in the cache in shared state of another **dP** on the same site will still need to send a message to the remote site. However, since the functions of the original SMU will now be provided by software in the **nP** or **pP**, this does not affect ACD design.

  However the operation of **tlbsync** may make it impossible to implement an L3 cache managed by the **nP**.

read_cache_a
read_transfer_a
RWITM_a
DClaim_a
write_w_flush_a
write_w_kill_a
write_w_clean_a
read_non_burst_a

Table 5: Transaction handlers

6 Conclusion

6.1 ACD capabilities

In the following, let

s	denote the maximum number of processors in a site,
t	denote the maximum number of outstanding transactions per processor and
$u = st$	denote the total number of outstanding transactions.

The 620 specification provides 5 bits for BusPID$< 0 : 4 >$, thus $s \leq 32$ However as currently envisaged a *T-NG site would have fewer processors, so: $s \leq 7$.

Three bits are provided for TransID$< 0 : 2 >$, thus $t \leq 8$.

6.1.1 Configuration

The ACD mode must be able to be set to the modes listed in Table 2.

The miss pattern *for each processor* should be able to be set.

Thus s registers must be provided for both mode and miss pattern.

6.1.2 Transaction Handlers

Section 4.5 describes the transaction handlers that are required for the ACD. They are listed in Table 5.

6.1.3 Status Register

The ACD needs to alert the **nP** when there are transactions for it to process.

One possible way to do this would be to provide a status register of u bits in which each bit is set if the corresponding entry in the TIF record table has been updated since the last read of the status register.

Thus whenever the ACD detects and captures a transaction, it sets the corresponding bit in the status register. Every time the **nP** reads the status register it is reset by the ACD itself. Each poll from the **nP** is able to read the flags for all the transactions that have been captured since the last poll. This method reduces the number of bus accesses required by the **nP**, but uses more internal cycles for shifting.

An alternative would be to provide a queue of inlet pointers in the ACD. Each inlet pointer addresses a handler in the **nP** which processes the next transaction waiting in the ACD. This method requires fewer **nP** cycles but more bus accesses and more complex ACD hardware.

6.1.4 Storage capacity

The ACD must be able to keep TIF records for the $u = st$ transactions which may be 'in flight' at any one time.

This implies buffer storage for a maximum of $u = 7 \times 8 = 56$ TIF records must be provided.

If only one blocking read is permitted at any one time (*cf.* Sections 3.3.2, 3.3.3), we only need to allow for pushouts. The 620 transaction buffers are partitioned into 4 read and 4 write buffers, so $t_{write} = 4$ and the number of buffers required is $s \times t_{write} + n_{readbuf}$ where $n_{readbuf} \geq 1$ is the number of buffers provided for blocking reads.

The contents of each record is shown in Table 6.

	Bits
Address	40
AType	5
Size	4
ASizeBurst	1
Data	512
Active	1

Table 6: ACD TIF record

These records will be indexed by $BusPID < 0:2 > |TransID < 0:2 >$ (assuming a maximum of 8 processors per site, so that $BusPID < 3:4 >$ may be ignored.)

If Scheme C for blocking loads is implemented with a lock for the ACD, the number of buffers in the ACD needed to support blocking loads can be constrained to the maximum number that resources will allow.

However the need to provide buffer capacity to support pushbacks can only be reduced by forcing pushback operations to retry with consequent performance penalties through occupation of the L3 bus. Thus there is some minimum number of buffers for pushbacks which will be needed to ensure optimal performance.

6.1.5 Interrupt capability

The ACD must be able to alert the **nP** when transactions requiring processing have been captured. This could be achieved with an interrupt line from the ACD to the processor designated as the **nP** or a line to the NIU of the **nP**.

When transactions have been completed with invalid data, there must be a mechanism by which the kernel can be notified so that it can send a signal to the process which received the invalid data.

6.2 Concluding remarks

This report has identified a number of significant problems in the implementation of blocking loads and stores using an architecture in which the **nP** and the ACD communicate via the L3 bus. The major problems arise from synchronization operations which require transactions to complete before they can complete themselves. If the **nP** needs to generate a new transaction on the L3 bus in order to enable a transaction captured by the ACD to complete, then deadlock becomes possible - either because the **nP** itself is waiting for the previous transaction to complete or because new transactions are not possible until previous ones have completed.

References

[1] B.S. Ang *et al*, *Hermes: Communicating* *T-NGs, MIT LCS CSG Memo 357, 1994.

[2] G.A. Boughton *et al*, *Draft Arctic User's Manual*, MIT LCS CSG Working Paper, 1994.

[3] Motorola Inc., *PowerPC 601: RISC Microprocessor User's Manual*, Motorola, 1993.

[4] IBM Microelectronics, Motorola Inc., *PowerPC 601: RISC Microprocessor User's Manual*, Rev 1, IBM/Motorola, 1993.

[5] IBM Corp., *PowerPC Architecture*, 1st ed, May 1993.

[6] R.S. Nikhil, G. M. Papadopoulos, Arvind, **T: A Multithreaded Massively Parallel Architecture*, MIT LCS CSG Memo 325-1, 1991.

[7] B.S. Ang, Arvind, D. Chiou, *StarT the Next Generation: Integrating Global Caches and Dataflow Architectures*, MIT LCS CSG Memo 354, 1994.

[8] S.K. Nandy, Ranjani Narayan, Horace Thompson, *In–Coherent - An Incessantly Coherent Cache Scheme for Shared Memory Multiprocessor Systems*, LCS CSG memo 356, in preparation.

[9] M.S. Allen, M.C.Becker, *Multiprocessing Aspects of the PowerPC*, IEEE CompCon, 1993.

[10] W.-D. Weber, *Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors*, Tech. Rep. CSL-TR-93-557, Stanford University, 1993.