Architectural Support for Block Transfers in a Shared-Memory Multiprocessor

Steven J.E. Wilton and Zvonko G. Vranesic
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada, M5S 1A4

Abstract

This paper examines how the performance of a shared-memory multiprocessor can be improved by including hardware support for block transfers. A system similar to the Hector multiprocessor developed at the University of Toronto is used as a base architecture. It is shown that such hardware support can improve the performance of initialization code by as much as 50%, but that the amount of improvement depends on the memory access behavior of the program and the way in which the operating system issues block transfer requests.
Keywords: *Shared-memory multiprocessor, block transfer support*

1 Introduction

It is well known that the design of a multiprocessor operating system is highly dependent on the architecture of the computer on which it is to be run [1]. The reverse is also true; the architecture of a multiprocessor should be greatly influenced by the requirements of the operating system. For example, it is quite common for an operating system for a non-uniform memory access time (NUMA) multiprocessor to transfer large blocks of data from one memory unit to another. Such an operating system will be much more efficient if the system architecture facilitates efficient block transfers.

In this paper, we will examine how one shared memory multiprocessor can be enhanced to support efficient block transfers. We are especially interested in blocks that are equal to the virtual page size. In the base multiprocessor, one processor must issue separate instruction(s) to transfer each word in a page. In the enhanced system, any processor can issue a single instruction which causes the multiprocessor to transfer a block of words from a remote memory to the processor's local memory. Remote-to-local transfers are supported, since they are common in many NUMA machines [2]. To examine the extent to which block transfer support affects performance, a simulator was developed to model the multiprocessor at the register transfer level.

2 Base multiprocessor

As an example multiprocessor, we will consider a machine that uses a ring-structured interconnection network. Such networks are found in the KSR-1 multiprocessor [3] and the experimental Hector machine [4]. In order to provide a small enough model for effective experimentation through simulation, we have chosen the single-ring structure shown in figure 1. The system consists of nodes connected by a unidirectional bit-parallel slotted ring. Each node has one processing module connected to it, comprising a processor, a cache, and a local memory.

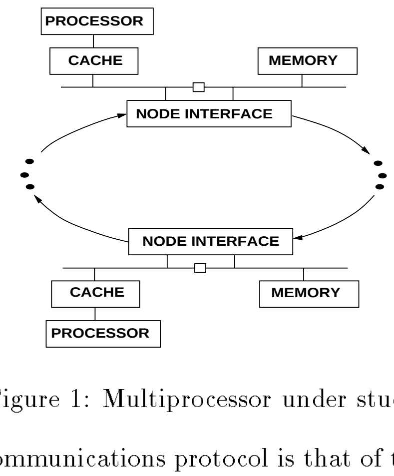


Figure 1: Multiprocessor under study

The communications protocol is that of the Hector multiprocessor. Every processor can access the memory on any node, but there is a disparity between the access time to the processor's local memory and the access time to another processor's memory. A processor never has two outstanding requests at the same time.

Information is transferred in packets. The width of the ring is large enough to hold either a full request message or 8 bytes of data in a single slot, as well as overhead such as the addresses of the destination and source nodes. There is one slot per node and packets are transferred from one node to the next in a single clock cycle.

2.1 Simple read requests

When accessing a remote memory, the processor must communicate with the memory through the network. Figure 2 is a simplified diagram of the data path on a single node.

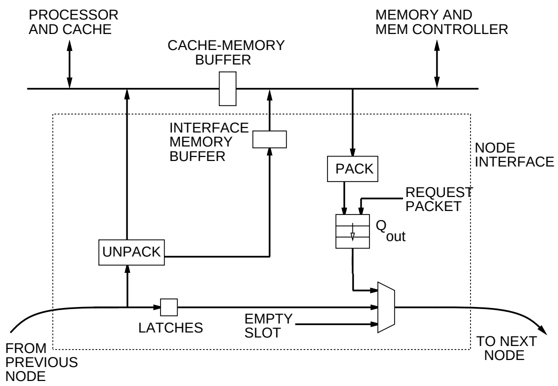


Figure 2: Single-node architecture

When the processor performs a read and the requested data is not in the cache or the local memory, the node interface constructs a request packet and stores it in the queue Q_{out} until an empty slot is detected on the ring. The first empty slot is filled with the request packet. When the remote node receives the request, it either accepts or rejects it. The request is rejected if the node interface is already busy satisfying another request. If this occurs, the request slot is marked with a negative acknowledgement, and the packet is returned to the sender. Upon receiving the negative acknowledgement, the requesting node will retry.

Once the request is accepted, it will pass through the node interface-memory buffer, and be sent to the memory controller. The memory controller then reads the data from its memory, and sends it to the node interface. The node interface packs the data into either one or two packets, and stores them in queue Q_{out}. When the requesting node interface receives the data, the data is sent to the cache.

To understand the purpose of the cache-memory buffer and the node interface-memory buffer, consider two processors A and B. If processor A performs a local memory access, while processor B requests data from node A, then the memory unit on node A might receive a request from the node interface (to satisfy B's request) at the same time that it receives a request from the cache (A's request). The buffers hold the two requests, and the memory control unit satisfies each in a round-robin fashion.

3 Block transfer support

We will now examine how block transfers can be performed in the multiprocessor. Remote-to-local block transfers are performed in the base system using a sequence of read and write commands. Each read command causes a single word to be transferred from the remote memory to one of the processor's registers and is followed by a write command which writes the received word into the processor's local memory. Each read that misses in the cache causes a read request to be sent through the network. The writes are all local writes; they do not cause any interprocessor traffic.

Now consider an "enhanced" multiprocessor in which the operating system (or the user program) can transfer a block from a remote memory to the local memory using the single instruction:

```
LONG_READ   remote memory base, local memory base,
            block size
```

Figure 3 shows the new node interface. A new queue (Q_{write}), large enough to contain the largest block of data that will be transferred, has been added. Also, queue Q_{out} has been made large enough to contain an entire block.

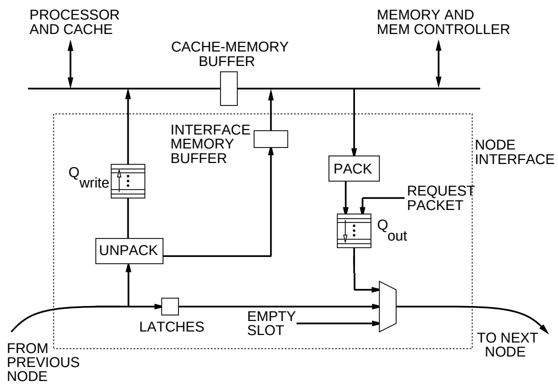


Figure 3: Enhanced node architecture

Consider what happens when the processor executes a block transfer instruction in the enhanced multiprocessor. First, the processor signals the local node interface that a block transfer is to occur, and transfers the source and destination address to the node interface. A request packet, called a *block read* request, is constructed and stored in Q_{out}. The first empty slot on the ring will be filled with the request packet.

When the remote node interface receives the request (and accepts it), it reads the entire block from memory. We will refer to this as the *read phase*. Upon reading data from its memory, the remote node interface constructs data return packets and stores them in Q_{out}. Eight bytes of data can be packed into each packet. When an empty slot appears on the ring, the top entry in Q_{out} is returned to the requesting node. The packing and transmission of packets is called the *transfer phase*. If the network is heavily utilized, it is possible that much (or all) of the block will be read from memory before it is sent; queue Q_{out} must therefore be large enough to contain an entire block.

When the requesting node interface receives the data packets, it writes the data directly to the local memory through the cache-memory buffer. This is the *write phase*. If the data arrives on the network faster that it can be stored in memory, queue Q_{write} will hold the overflow. Thus, Q_{write} should also be large enough to contain an entire block. Once the write phase has been completed, the node interface signals the processor that it can continue.

Note that the data being written during the write phase goes through the cache-memory buffer. All other requests, including read requests originating from another node, go through the interface-memory buffer. Consider a node in the write phase of its own block transfer. If an unrelated request from another node arrives, the new request can be accepted, since the path through the node interface-memory buffer is not being used by the block write. The round-robin scheduling of requests in the two memory buffers described earlier would result in the memory bandwidth being shared between the block write and the new request. Note that this is only possible because we have assumed that the processor will not issue a new request until the previous request has been satisfied. During a block write, we are guaranteed that the cache will not issue a new request and therefore will not use the cache-memory buffer.

It is intuitive that such a system would perform block transfers much faster than the base architecture; not only is less time wasted transferring individual request messages, but also if many processors are trying to access the same memory, this new "enhanced" architecture will only have to compete once to transfer a page, while the base architecture will have to compete for each word (or cache line).

4 Non-block transfer traffic

We now consider how this enhancement affects the access time of non-block transfer requests (those remote requests that have nothing to do with block transfers). The access time of these requests will increase for two reasons. First, LONG_READ instructions have an unfair advantage over normal read or write instructions when it comes to accessing a busy memory. The requests caused by both types of instructions fight to access a remote memory identically; that is, a remote node interface will reject an incoming request whenever it is busy, regardless of the type of incoming request. However, when a block read request is accepted by a node interface, the entire block will be read before any other requests destined to the same node could possibly be accepted. In the base multiprocessor, however, each read that is part of a block transfer must compete individually. Thus, the memory contention seen by *non*-block transfer requests will be smaller in the base multiprocessor than in the enhanced system.

Second, in a slotted ring, if many consecutive slots are used to transfer a block, the clustering of the full slots will tend to increase the average ring access time. Network issues are discussed in detail in [5].

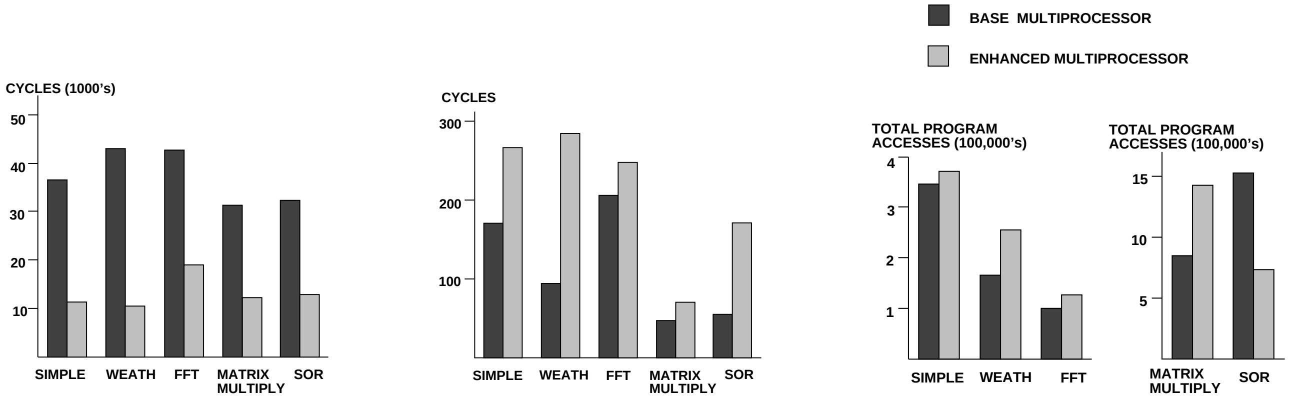
5 Evaluation

The effectiveness of the proposed architectural enhancements for block transfers was investigated through trace-driven simulation. Two types of workloads were considered: address traces and synthetic algorithms. The address traces used were SIMPLE, WEATHER, and FFT, and are discussed in [6]. The synthetic algorithms produce access patterns that would result from the execution of matrix multiply and successive over-relaxation (SOR) algorithms. To insert block transfers into the workloads, the page replication operations of a typical operating system were simulated as described in [2].

Simulations were run using both the base and enhanced architectures. In all simulations, a 16-node system with a page size of 4KB was simulated. Block transfers occur as a result of page replications; therefore, all block transfers move 4KB of data. Since page replications are most common in the initialization phases of programs, we concentrated on the beginning of the address trace simulations (the first five million cycles of SIMPLE and WEATHER, 1.5 million cycles of FFT, and two million cycles of the two synthetic algorithms).

Figure 4a shows the average time required to replicate a 4KB page on each system. A page replication on the enhanced system takes between $\frac{1}{4}$ and $\frac{1}{2}$ as long as it does on the base multiprocessor.

It is expected that the enhanced multiprocessor will suffer from a longer non-block transfer remote access time. Figure 4b shows that this is indeed the case. The vertical axis in this graph is the average remote access time for normal read and write requests. The amount that the enhancement slows down these requests depends on the block transfer behavior. Workloads which have long periods of low block transfer activity (such as FFT) do not show a marked increase in non-block transfer remote access time, while those which are constantly performing block transfers do show a marked increase (the WEATHER trace, for



a) Replication time b) Non-block transfer remote access time c) Program accesses

Figure 4: Simulation Results

example).

The above results show that in the enhanced multiprocessor, replications are satisfied much faster than they are in the base multiprocessor, while non-block transfer requests take longer. The extent to which these two factors affect the overall system performance is shown in figure 4c. The height of each bar represents the total number of *program accesses* satisfied in a fixed time (five million cycles for SIMPLE and WEATHER, 1.5 million cycles for FFT, and two million cycles for the two synthetic algorithms). A program access is defined as any memory reference made by the program. Many of these accesses will result in page faults, which might, in turn, cause the operating system to perform a replication. These accesses performed by the operating system during a replication are *not* program accesses. The number of program accesses is proportional to the proportion of the trace file or algorithm that has been completed.

As shown, the enhanced multiprocessor shows performance that is significantly better than the base multiprocessor in most cases. The improvement ranges from 20% in the FFT trace to almost 50% in the WEATHER trace. The only workload in which this system exhibits a poorer performance is the SOR algorithm, in which non-block transfer remote accesses are intermingled with block transfer requests.

6 Concluding remarks

Remote-to-local block transfers are both simple and worthwhile to support in hardware. By reducing the time required to perform such a transfer, the page replication and migration operations that are common in NUMA shared memory multiprocessors can be more effective in dynamically placing memory blocks close to the processors that are accessing them. We have shown how our example multiprocessor can be enhanced to efficiently support block transfers. Finally, we should note that block size is an important parameter which has a significant effect on the overall performance. This has been considered in [2].

Providing efficient block transfers is only the first step. More aggressive memory management policies should be developed to take advantage of the relatively inexpensive block transfers.

References

[1] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox, "NUMA policies and their relation to memory architecture," in *Proceedings, Architectural Support for Programming Languages and Operating Systems*, pp. 212–221, April 1991.

[2] S. J. Wilton and Z. G. Vranesic, "Architectural support for block transfers in a shared-memory multiprocessor," Tech. Rep. EECG-4-15-93, University of Toronto, 1993.

[3] Kendal Square Research, *KSR-1 Principles of Operations.* Waltham, Ma., 1991.

[4] Z. Vranesic, M. Stumm, D. Lewis, and R. White, "Hector: A hierarchically structured shared-memory multiprocessor," *Computer*, vol. 24, pp. 72–80, January 1991.

[5] S. J. Wilton, "Block transfers in a shared memory multiprocessor," Master's thesis, University of Toronto, August 1992.

[6] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, vol. 23, pp. 49–58, June 1990.