# Reducing PE/Memory Traffic in Multiprocessors by The Difference Coding of Memory Addresses*

David M. Koppelman
Louisiana State University
Baton Rouge, LA 70803
(504) 388-5482 koppel@gate.ee.lsu.edu

Abstract—**A method of reducing the volume of data flowing through the network in a shared memory parallel computer (multiprocessor) is described. The reduction is achieved by difference coding the memory addresses in messages sent between processing elements (PEs) and memories. In an implementation each PE would store the last address sent to each memory and vice versa. Messages that would normally contain an address instead contain the difference between the address associated with the current and most recent messages.**

**Trace driven simulation shows that only 70% or less of traffic volume (including data and overhead) is necessary, even in systems using coherent caches. The reduction in traffic could result in a lower cost or lower latency network. The cost of the hardware to achieve this is small, and the delay added is insignificant compared to network latency.**

## I. Introduction

Although useful, interconnection networks for parallel computers are expensive and their latency is large; their cost could dominate the cost of a sufficiently large system and their latency (time for a message to traverse the network) could dominate performance. Much current research is aimed at reducing the latency of networks, usually by reducing congestion (message traffic of sufficient density and distribution to cause an increase in latency), at the expense of increasing cost. (See [5,8,18,19] for an overview and [4,12,15] for examples of current research.) Instead of concentrating on the design of the network this paper describes a method of reducing the volume of data a shared memory parallel computer (multiprocessor) needs to send over the network. This reduction in data volume could lower the cost of the network required, or reduce the latency of the network. The hardware needed to realize this cost reduction is called DCM, for *difference coded memory address.*

DCM achieves cost reduction by reducing the number of bits needed to code a memory address sent between processors (PEs) and memories (MUs), thus reducing the volume of data the network must handle. (The memory addresses are part of the messages sent over the network in a multiprocessor.) Instead of sending memory addresses across the network, difference codes (the difference between the actual memory address and a predicted memory address) are sent. The original memory addresses are reconstructed at the MU using stored values of recent addresses and the difference code sent to the MU by the PE. The same process is used for messages sent from MUs to PEs.

There are several methods of implementing DCM, the methods differ on the pairs of messages used to compute difference codes. An implementation using the lowest cost method stores only one address per PE/MU pair. It is possible that some aspect of program or system behavior (perhaps related to address space mapping) would result in large difference code sizes for this implementation, in which case smaller difference codes might be achieved using a more elaborate DCM

* In *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 11, pp. 1156-1168, November 1994.

implementation. The details of DCM implementation and address space mapping are described below.

Trace driven simulations performed to test the low–cost implementation indicate a reduction in data volume to about 70% had complete addresses been sent. The traces were from four shared memory programs, each use 64 PEs and a 32 bit address space. The next generation of shared memory parallel computers will probably be built using 64 bit processors. Although it will be quite some time before programs use every addresses, programs could easily make sparse use of such an address space. The savings by difference coding memory addresses in such systems would be large.

The idea of difference coding memory addresses has been explored for serial computers in several forms. Early related work was done in areas such as indexed addressing and opcode coding. Indexed addressing is, in a sense, explicit difference coding. The compiler and programmer arrange for memory reference and jump target addresses to be close together. The fields in instructions holding indexed addressing offsets can be made small. The result is smaller and faster programs because memory addresses can be coded using fewer bits. See, for example, [9,22] for studies of address reference behavior aimed at improved addressing and compilation techniques.

Other investigations have specifically looked at reducing the volume of address data transmitted from processor to memory in serial systems. A study by Farrens investigated the reduction achieved by sending only low order bits of an address to memory (the processor and memory each cache the high order bits). Trace driven simulations were performed, the results indicated that 99% of addresses could be coded with only 16 bits (for a 32 bit address space)[7]. A similar study was carried out by Pleszkun, *et al* [17], also for serial systems.

The work reported here is different from the above studies in several respects. The earlier simulations were of serial systems, so that several specific issues relevant to parallel systems, which are used here, were not considered. There are also differences in how coding is implemented here. Farrens and Pleszkun simply send the low order bits, whereas the scheme described here sends a difference. Sending the low order bits is appropriate in a serial system because the added time for subtraction would be a significant fraction of the memory access time. In a parallel system with a network between PEs and MUs the additional clock cycle for subtraction would be a small fraction of the network latency. No latency at all would be added if the subtraction were done in parallel with other message preparation activities. Furthermore, the reduced message size might more than compensate for any subtraction latency. Finally, Farrens' base registers, which hold the high order bits, cover contiguous areas of the address space. A more general scheme of mapping the analog of base registers to address space, which results in much smaller codes, is used here. This scheme has the advantage of exploiting the structure of shared memory parallel programs.

The remainder of this paper is as follows. In the next section difference coding and other background material will be covered. In Section III [p. 4] the DCM hardware will be described. In Section IV [p. 8] the simulation methodology and results are presented. In Section V [p. 14] address space mapping is discussed: the relationship between address space mapping and DCM performance is discussed followed by several methods of determining address space mapping. Conclusions appear in Section VI [p. 18].

## II. Background

In shared memory parallel computer designs PEs share a common address space; this address space is divided between a number of MUs. A network is used to provide communication between PEs and MUs; the network might be a two dimensional mesh, a high dimension direct network

(such as a hypercube), or a multistage interconnection network (MIN) [5,18,19]. Regardless of the network used, cost and performance are important issues. Performance is usually determined by the latency of the network.

Much current interconnection network research is aimed at reducing congestion. There are many proposed solutions to the congestion problem. Most congestion reduction schemes allow the network to handle a greater volume of data without being congested, for example, combining messages, providing multiple paths, or improved routing [16,18,19]. The approach taken here is different: the design of the network is unchanged, but the volume of data is reduced. The lower volume of data can lead to a reduction in congestion and latency or allow a lower cost network to be substituted.

*A. Difference Coding*

The reduction in traffic volume is achieved by difference coding memory addresses. Difference coding has long been used to compress data, usually to make efficient use of expensive storage or communication media [10]. Suppose data to be compressed is represented as a sequence of integers. The sequence of integers is converted into a compressed sequence of integers using a *predictor,* which estimates the current integer in a sequence given some number of most recent integers in the sequence. The compressed sequence consists of the differences, called *difference codes,* between the actual integers and the predicted integers. If the predictions are close to the actual integers then the differences making up the compressed sequence will consists of small integers, which take less space when properly coded [10].

The predictors used in this paper are known as first and second order linear predictors. *$N$th order linear predictors* calculate their prediction by taking a weighted sum of the $N$ most recent integers and adding an offset (requiring $N$ multiplications and $N$ additions). A *first order linear predictor* only examines the most recent integer and performs two operations (weighting and adding an offset) [10]. If no weighting is done then such a predictor only has to store the most recent integer and perform subtraction. A *second order linear predictor* multiplies the two most recent integers by a weight, adds the products, then adds an offset to the sum.

The difference codes (using a first order linear predictor without weighting) of a list of integers $x_0, x_1, \ldots$ is a list of integers $y_0, y_1, \ldots$ where $y_i = x_i - x_{i-1}$, $i \geq 1$ and $y_0 = x_0$. The average value of the elements of $y$ will be small when the values of $x$ change slowly. Compression is achieved if the number of bits needed to code $y$ is less than the number of bits needed to code $x$.

Suppose the elements of $x$ are integers from 0 to $B-1$, so that $x$ can be encoded in $b = \lceil \log_2 B \rceil$ bit words. If the greatest absolute difference between two consecutive $x$'s is $D$ then $y$ could be encoded as $d = \lceil \log_2 D \rceil + 1$ bit words (one bit is added for negative integers), for a savings of $b - d$ bits per word.

A better solution is to have a variety of word sizes for $y$. Each word is associated with a field giving the word size. For example, when $b = 16$ and $d = 17$, $x$ can range from 0 to 65,535, and can be coded with 16 bits. Because $d = 17$ there must be at least two word sizes for $y$ if any compression is to be achieved. If it is known that a vast majority of $y$'s will be less than 8, there could be two word sizes, 5 bits and 17 bits. Since most words will be of the 5 bit size the average word size will be closer to 5 than 17.

In DCM, the $x's$ are memory addresses and the $y's$ are the difference codes to be sent across the network. First and second order linear predictors are used.

*B. System Model*

The parallel system model used to assess DCM is simple, it consists of PEs connected to the inputs of a network; the outputs of the network are connected to MUs. The PEs, sharing a common

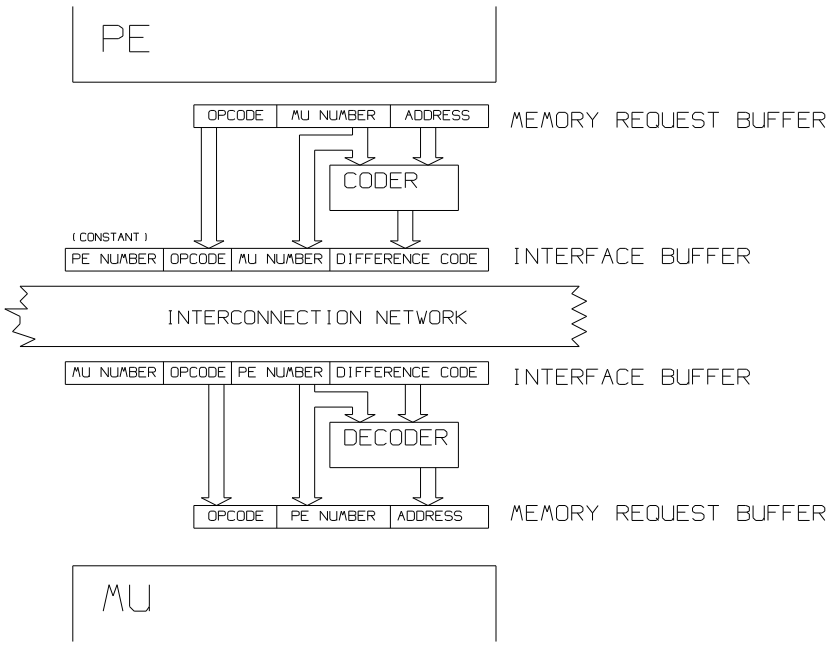


Fig. 1. Interface Between PE and Network.

address space, each consist of a CPU, and perhaps some cache memory [3,5,21]. The PEs access data from the MUs through the interconnection network, for brevity called the network; to access data the PEs issue a *request*. Requests can be memory read, write, test & set, and so on. (In this paper *request* also refers to the information needed to make a request including the MU number, address, data (if necessary), etc.) A request might be satisfied by the PEs cache; otherwise, and in the case the cache needs to access an MU, the request is sent to the appropriate MU through the network, in which case the request is converted to a message. The MU processes the request and returns any result via a message sent through the network. (Note that the network is two way.) If caches are used for shared data then additional messages are sent to maintain coherence. (See Section IV [p. 8] for more on coherent caches.)

A message sent through the network is divided into *packets*, the packet size being the width of the datapath. To enter a request into the network a PE writes the request into an *interface buffer*. From there it is divided into packets and sent into the network. At the output of the network packets are reassembled into requests.

## III. Structure

There are several variations of DCM. Refer to Figure 1 for the following discussion. All of the DCM schemes convert a memory address to a difference code by subtracting a prediction of the memory address from the actual memory address; this operation takes place in the *coder*. The difference (along with other information) is sent to the MU via the network; the MU reconstructs the original memory address using its own prediction (which is identical to the PEs prediction) and an adder. On the PE side, the coder is between the PE and the interface buffer, where it converts the address to a difference code. On the MU side the *decoder* is between the interface buffer and the MU.

Each PE generates memory requests which might consist of an opcode, an MU number, a memory address, and, in the case of writes, data. (Data is not shown in Figure 1.) MUs generate

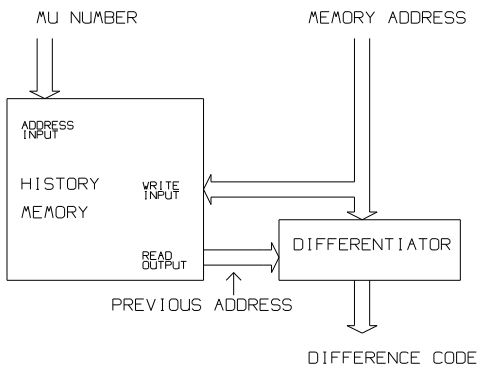


Fig. 2. The Unilocus Coder.

messages containing data and coherence information. For the following discussion it is assumed that the requests are presented in bit parallel fashion.

The MU number is used by the predictor: predictions of addresses bound for a particular MU are based only on the memory addresses seen by that MU. (An MU does not see messages to other MUs, it can only take into account messages it itself has received when making predictions.) The MU number and address enter the coder, which computes the difference code. This difference code is then written into a field in the network interface buffer. The network interface buffer also has fields for the opcode, the MU number and the source PE number.

The number of address bits leaving the coder is the same as the number of address bits entering the coder. Compression is realized when the network interface converts the request in the buffer into packets: only as many packets as necessary to hold the difference code are sent.

On the memory side of the network the messages are reassembled. The difference code along with the source PE number enter the decoder, which converts the difference code back into a memory address. The memory address, along with the opcode and source PE are reassembled into a request for servicing by the memory. The various schemes differ only in the details of the coder and decoder.

*A. Unilocus Method*

The simplest DCM implementation is called the *unilocus* method. The locus referred to is the area (small range of addresses) in memory that the PE frequently accesses; the unilocus method works best when consecutive requests seen by an MU are frequently to the same locus.

In the unilocus method of generating difference codes the prediction is based on the most recent address sent between a PE and MU. Therefore the most recent request address for each PE/MU pair must be stored. The coder contains a high speed memory for storing request addresses, called a *history memory* and a differentiator. (See Figure 2.) The differentiator performs subtraction in the basic coder.

The memory address input entering the coder is connected to one input of the differentiator and to the history memory's write input (assuming the history memory has separate connections for reading and writing data). The MU number is connected to the history memory's address input. The most recent request address is retrieved from the history memory; the differentiator computes the difference code, which is connected to the output of the coder. The current request address is written into the history memory. The decoder is similar; the only difference is that the differentiator is replaced by an integrator and the output of the integrator is connected to the history memory write inputs.

For example, suppose that in some system, PE 4 issues the following requests: (7 1000), (5 100), (5 106), (7 1500), where the first number of each pair is the destination PE number and the

second number is the address. Assume that before the first request the history memory contains only zero entries (as it might when the system is reset). The differentiator performs subtraction.

When the first pair enters the coder, the history memory will retrieve the entry for 7, which will be 0. The differentiator will subtract 0, from the current address, 1000; the difference, 1000, exits the coder. At about the same time the current address, 1000, is written into the history memory at location 7. The output of the coder enters the difference code field in the network interface buffer. Assume that the rest of the information needed for the memory request, including the opcode, MU number, and source PE, are written into the buffer.

The request is divided into packets and sent to the MU by the network interface. The packets are identified so that the MUs network interface can reassemble the difference code without ambiguity. The number of packets sent depends upon the difference code and the number of bits reserved in the message. In the example, the difference code is 1000, which takes 11 bits to code (including a sign bit). If the first packet of a message had four bits reserved for a difference code and subsequent packets has eight bits request could be sent in two packets.

At MU 7, the packets arrive. The network interface will assemble the three packets into a request. After the request is assembled the difference code, 1000, and the source PE, 4, enter the decoder. The source PE number is used as an address in the history memory; because MU 7 has not yet received a request from PE 4, a 0 is retrieved. The 0 is added to the difference code; the sum, 1000, is the reconstructed address. The reconstructed address is written into the history memory at location 4 and also exits the decoder. The memory request is assembled using the reconstructed address and the other information from the network interface buffer.

The second request to be issued by PE 4 is (5 100). The procedure is the same as that for the (7 1000) request. When the third request is issued by PE 4, (5 106), the difference coding scheme will realize a reduction: the difference code is 6, requiring only 4 bits to code, and only one packet. The history memory in the coder retrieves the most recent address sent to MU 5, which is 100, and performs the subtraction yielding the difference code of 6.

The differentiator could perform a variety of functions, the one to be considered is subtraction. Let $x_n$ denote the current address and $x_{n-1}$ and $x_{n-2}$ denote the two most recent addresses. Subtraction could be the simple difference, $x_n - x_{n-1}$. If the address sequence slowly increases then the difference codes will be small positive numbers. Depending on system details, such as cache block size, a small offset, $o$, might be included, yielding the difference code $x_n - x_{n-1} - o$. A good value of $o$ might be found through analysis of address reference sequences; based upon the simulations described below $o$ is taken as zero. A third approach is to predict using the last difference (which would require storing two addresses), that is, use $x_n - 2x_{n-1} + x_{n-2}$ as the difference code. The most effective method, based on simulations, uses a first order predictor.

DCM is economically plausible if the savings in the datapath or latency more than offset the cost of the additional hardware. If there are $M$ PEs and $N$ MUs then each PE would need to store $N$ addresses and each MU would need to store $M$ addresses. The amount of storage needed per MU and PE for last addresses is small compared to the cache and memory sizes for most shared memory systems in use or contemplated.

The memory for storage of addresses might be too large in very large systems, for example those with greater than 10,000 PEs. To reduce the storage needed, a scalable system (a system that could be made arbitrarily large) could cache only the addresses from those PEs or MUs which have recently communicated. A cache miss at the PE would mean that the entire request address would have to be sent to the MU; a cache miss at the MU would mean the MU would have to send for the complete address from the PE—this should be an infrequent occurrence.

*B. Multilocus Methods*

The efficacy of the difference coded memory address scheme rests on the assumption that the difference between consecutive memory access addresses are frequently small. It is possible for a program to exhibit a large degree of spatial locality and yet confound the scheme because consecutive memory accesses are to distant locations while alternate accesses are to nearby locations. The multilocus methods can realize a large degree of compression despite such program behavior.

Unlike the unilocus coder, a multilocus coder stores several request addresses for each PE/MU pair. The request address used by the predictor can be based on some function of the request address or the one which yields the smallest difference code. The difference code, along with the locus identity is sent to the MU. The MU uses this information along with the contents of its history memory to reconstruct the address and to update the history memory.

The scheme in which the predictor chooses a locus based on the minimum code size will be called *minimum multilocus*. The scheme in which a locus is chosen based on the request address will be called *indexed multilocus*. In the minimum multilocus scheme the history memory is initialized so that loci are in some predetermined positions. (E.g., evenly spaced through memory.) When a message is to be sent the predictor finds the difference codes for all loci, the smallest one is used in the message. Because all loci must be tried, the cost or performance of this method will be a factor when the number of loci is large.

The indexed multilocus scheme avoids this by choosing a locus based on the request address, not the code size produced. The locus to be chosen can be determined by some subset of address bits, or some other mapping. The mapping would be chosen so that code sizes are small. Unlike address space mapping (see the Section IV [next page]), the mapping in the indexed multilocus scheme can be frequently changed, the only loss of performance is due to the time needed to invalidate and re-warm the history memories.

The multilocus schemes are similar to the address size reduction techniques used by Pleszkun *et al* and Farrens & Park. However, neither of these methods were meant for parallel programs and rather than sending a difference code, they send the low order bits of the address, along with an index specifying the location of the high order bits. Pleszkun *et al* specify several mechanisms for predicting how the loci can change; an index to the correct prediction is sent in place of an address.

The cost of these schemes, as well as the minimum multilocus scheme, is high because of the need for some type of associative memory or other special hardware. The added cost of the indexed multilocus scheme over the unilocus scheme is due mostly to the additional history memory (which should be relatively small).

The multilocus schemes may not be necessary because the advantage of considering differences within only one locus can be had at the cost of a unilocus coder by proper mapping of addresses to memories, as will be described further below.

*C. Hardware Cost Analysis*

The goal of DCM is to reduce the cost of a shared memory parallel system by reducing the volume of network traffic. If this goal is to be met, it must be determined whether DCM in fact makes a significant reduction in the volume of traffic without increasing the cost of the system or reducing its performance. The cost could be estimated in many ways, to many degrees of realism; the approach taken here is to tally the total number of each type of small circuit (such as adders and memory bits). In this form a reasonable first order cost comparison can be made.

The system in which DCM will be used will be said to have $M$ PEs, $N$ MUs, $b_M$ bits per message, and a $b_A$ bit address space. In the unilocus scheme the coder consists of a history memory and a differentiator. The history memory stores $Nb_A$ bits. The cost of each type of

differentiator will vary; let $k_{\text{diff}}$ be the cost of a bit slice (analogous to a binary full adder) of the differentiator. Then the differentiator will cost $k_{\text{diff}} b_A$. The cost of the decoder is the same, except $N$ is replaced with $M$.

The total contribution to the cost of the system of the unilocus scheme is $2MNb_A$ bits of memory, and $(M+N)b_A k_{\text{diff}}$ of differentiator logic, plus control logic. The highest order term is the one for history memory, since this term is small compared to other memory in the system, the cost should not be a major problem.

Suppose the reduction in data volume is "used" to proportionately reduce the width of the data path in an omega (or similar) multistage interconnection network. These MINs are frequently considered for use in shared memory machines because of their simple control and low cost. An $N$–input/$N$–output omega network consists of $\log_2 N$ stages each containing $N/2$ $2 \times 2$ switching devices [13,19]. Let $R$ be the volume of data sent through the network in a system using DCM divided by the volume of data in a system not using DCM. Values for $R$ obtained in the simulations range from 0.85 to lower than 0.70. The cost savings in switching hardware (including the crossbars and queues in the MIN) would be $O((1-R)N \log N)$. There would be a further savings in wiring costs, which could be as high as $O((1-R)N^2/\log^2 N)$ [11]. Since a network is a large part of the cost of a computer, sometimes implemented with exotic technology, the actual cost savings would be large.

The cost of the minimum multilocus scheme is, of course, higher. Because it is of limited practicality and because the indexed multilocus scheme is an effective substitute the cost will not be analyzed here. The cost history memory for the indexed multilocus scheme is $2lMNb_A$ bits, where $l$ is the number of addresses stored for each PE/MU pair. This cost would be reduced if the history memory were organized as a cache. The cost of the hardware to map an address to a locus could be as low as zero (if bits of the address are used to determine the mapping).

The advantage of the multilocus schemes, that small difference codes can be achieved despite having consecutive addresses being very different, is perhaps outweighed by the cost of the hardware to implement it. It is possible to realize the advantages of the multilocus scheme with no additional hardware at all by carefully dividing the memory address space to the MUs.

The multilocus hardware explicitly matches close memory addresses. This same effect could be achieved if there were only few loci per PE/MU pair. By carefully dividing the address space among the MUs (interleaving) the number of loci per MU/PE pair could be minimized. This reduction results in small difference codes being sent even while only one address per PE/MU pair is stored.

The simulation study described below tested this idea by varying the address interleaving. It was found that the average difference code size, using the unilocus scheme, varied greatly with interleaving, thus indicating that difference code size could be minimized without the elaborate hardware. The minimum difference code size obtained was less than eight bits for most simulations. There would be little additional benefit in reducing this further using more elaborate hardware, so that the hardware more complex than unilocus is unnecessary.

## IV. Simulations

Trace driven simulations were carried out to determine the efficacy of the technique. A 64 PE multiprocessor was simulated using address traces from four benchmark programs. The simulations were carried out using a variety of cache coherence schemes, block and set sizes, and interleavings. Simulation results were ultimately used to select a machine configuration that would maximize the reduction in the volume of data using difference coding of memory addresses.

*A. Methodology*

The machine simulated consists of 64 PEs sharing a common address space. This address space is divided between 64 MUs. The PEs, which have local caches, are connected to the MUs through an unspecified network; network latency was not modeled.

Two way set associative caches with random replacement are simulated. (Caches with associativity greater than two and with a better replacement policy [such as LRU] were not simulated because they would yield only marginally better results and are rarely implemented.) Block sizes from 4 to 64 bytes are used in the simulations, the cache size in all experiments is $2^{16}$ bytes.

In all the simulated systems, the number of addresses per block, $B$, and sets per cache, $S$, are a power of two. Let $B = 2^b$ and $S = 2^s$. A memory address is mapped into a cache the following way: the $b$ least significant bits determine the location within a block, the next $s$ less significant bits determine the set, the remaining bits are the addresses' tag.

Six consecutive bits of an address determine to which of the 64 MUs the address is mapped (interleaving). The least significant bit is called the *interleaving bit;* the index of the interleaving bit is called the interleaving. The index of the least significant bit is defined as zero. The indices of the six bits used is called the *interleaving range*. For example, if the least significant bits are being used for interleaving then the interleaving is 0 and the interleaving range is 0-5. Any six consecutive bits could be used; the interleaving bit used was varied in the simulation study; it had a strong effect on the difference code size.

TABLE 1
LENGTH OF MESSAGES IN SIMULATED SYSTEM.

| Message Type | Length | Length |
|---|---|---|
| Req | 6 | A+O |
| Req Resp | 2 | O |
| BlockWa | 22 | A+O+B |
| BlockWoA | 18 | O+B |
| WordWa | 10 | W+O+A |
| WordWoA | 6 | W+O |

Memory requests issued by a PE may result in zero, one, or more messages being sent over the network. The number of messages and their length depend upon cache and coherence scheme used. The types of messages and their length are summarized in Table 1. The lengths are given as functions of block size (B), overhead (O), word size (W), and address size (A), as well as actual values using a block size of 16 bytes, an overhead of 2 bytes, a word size of 2 bytes, and an address size of 4 bytes.

Several cache coherence schemes were simulated. The simplest scheme is no cache at all. When a PE issues a memory read, a *request* message consisting of a 2 byte opcode and a 4 byte address is sent to the appropriate MU. The MU sends back a wordWoA (word without address) message consisting of a 4 byte data word along with a 2 byte opcode. (The opcode can consist of any miscellaneous information needed for the transactions.)

The simplest *cache* scheme is to cache only private data. In such a scheme, a cache miss due to a private address results in a *request* message to an MU; the MU returns a blockWoA (block without address) message consisting of a block of data, and a 2 byte opcode. When a dirty cache block is replaced the appropriate MU is sent a blockWa (block with address) message consisting of a block of data, a 2 byte opcode, and an address.

With a cache coherence scheme shared data is cached while insuring that all cached copies of a memory location are identical. The methods to be described here are those described by Chaiken in [2]. Non-identical copies of the same memory location are avoided by insuring that only one copy of a memory location that is to be written to is cached. To write to shared data, a PE must get a *exclusive* copy of the data from that location by *invalidating* (in effect erasing) all other cached copies. After obtaining an exclusive copy the PE can write to it without generating network traffic. If another PE reads that location, then the exclusive copy will revert to a *clean* copy, which will be copied into the reading PEs cache.

The coordination of invalidations and exclusive copies is handled at the MU. In a directory based cache coherence scheme the MU keeps track of which PEs have cached copies of each block. In a *full map* directory, each block has a directory and each directory has one bit for each PE. The bit is set if there is a cached copy at the PE. In a *limited* directory the directory has a fixed number of entries (much less than the number of PEs). An entry is blank or contains the PE number where a cached copy of the block is held. If no entries are blank, and a PE reads a copy of the block, then one of the PEs in the block's directory has its copy invalidated. The state information maintained for a block also encodes when and where an exclusive copy exists.

In the simulation, full map and limited directories with one to six entries are used. Invalidates are sent from MUs to PEs using the Req (*request*) message (see Table 1), the PE acknowledges using a Req Resp (request response) message, consisting of a 2 byte opcode. A read to an exclusive copy results in four messages: a Req message from the reading PE to the MU, a Req message from the MU to the PE with the exclusive copy, a blockWoA (block without address) message from the PE to the MU, and a blockWoA message from the MU to the reading PE. More information on cache coherence can be found in [2].

To determine the effectiveness of difference coding the simulator collected statistics on the number of bits that would be needed to code a memory address, and on the size of the messages sent.

The number of bits needed to code an address was computed using two prediction schemes; first and second order. This information was collected for 27 interleaving ranges: $5+i$ through $i$ for $0 \leq i < 27$. The number of references to each memory unit under each of the interleaving schemes was also counted, to verify that all MUs are being used more or less evenly. Information about the average size of messages was also collected.

The simulator uses the above information to compute a maximum savings in data volume: the average message length using only the number of bits necessary to code an address divided by the average message length using 32 bits for the address.

Simulations were performed on four parallel traces, collected from the programs FFT, weather, simple, and speech, these programs are discussed in [3]. In all but the program speech, private references are identified, and are used as the basis for the private cache scheme.

The traces contain untimestamped memory references. The simulator processes these references in order, with the state of the memory system updated before the next reference is read from the trace. The simulator maintains the state of all caches and cache directories, as well as the information needed for difference coding. Statistics were collected on the state of the caches (to monitor warmth, and insure efficient and correct operation), as well as on details of network traffic and difference coding.

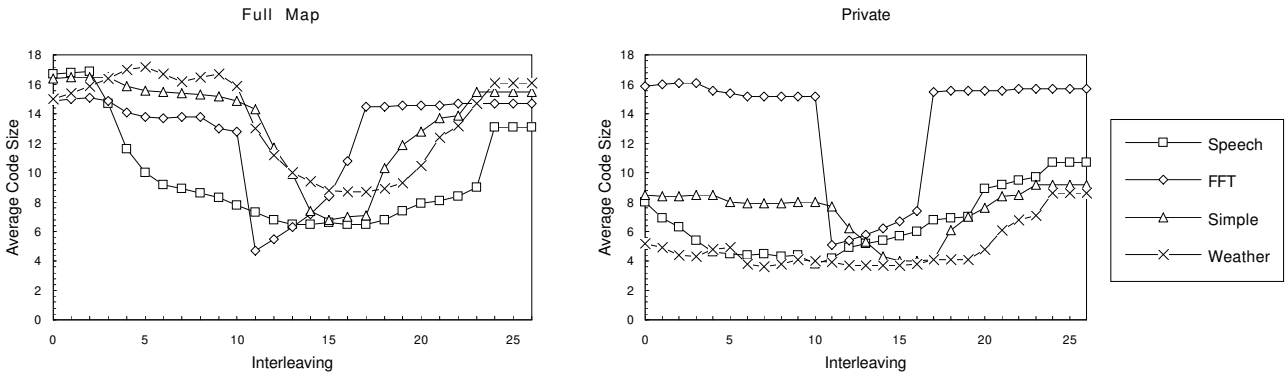


Fig. 3. Code Size vs. Interleaving.

*B. Simulation Results*

TABLE 2
SIMULATION RESULTS

| Benchmark | Coherence | Bits Needed | Reduction | % Address | Msg. Size |
|---|---|---|---|---|---|
| FFT | Priv | 5.1 | .70 | 36 | 7.2 |
| Weather | F.M. | 7.2* | .72 | 36 | 6.0 |
| Simple | Priv | 4.0 | .70 | 34 | 6.2 |
| Speech | F.M. | 6.5 | .73 | 34 | 6.0 |

Simulations performed on the four traces show reductions in data volume from 73% to 70% of the original volume for realistic cache configurations, and even greater reduction for suboptimal configurations. The results from some representative simulations are tabulated in Table 2, additional data appears below. The simulations summarized in the table were all for systems using $2^{13}$ sets of four byte blocks. The *bits needed* column shows the average number of bits to code an address, the *reduction* column shows the total volume of data sent over the network using DCM divided by the total volume of data without DCM, the *% address* column shows the percentage of network data consisting of addresses (without DCM), and the *msg. size* column shows the average message size (in bytes). All but one of the traces showed greatest reduction with a first order predictor. The entry marked with an asterisk had smaller difference codes using a second order predictor.

In general, the simulations show that the reduction in volume is greatest when block sizes are smallest, when full map cache coherence is used, and when the approximately center bits of the memory address are used for interleaving. These factors will be discussed in turn.

In Figure 3 the average difference code size of an address is plotted against interleaving. The data in Figure 3 are for systems with $2^{13}$ sets and a block size of four. Each benchmark was simulated using private and full map coherence. As can be clearly seen, interleaving has a strong effect on difference code size, with the middle bits resulting in the smallest codes. This effect is more pronounced when full map coherence is used, demonstrating that caches interfere with difference coding. This effect of caching on code size is discussed further below.

In an actual system data is sent across the network in messages consisting of fixed sized packets. The reduction in traffic volume in Table 2 may not be the actual reduction obtained because in any message the last packet may have unused bits, resulting in a volume of traffic which is different than the data volume. Message packets can have unused bits with or without DCM, so that the

results above are not biased one way or the other. Nevertheless, an analysis will be performed to determine the average number of bits needed to code addresses taking into account packet size and unused bits.

The number of bits needed for the address depends upon the number of bits in the difference code, the number of bits needed to specify the size of the difference code, the number of unused bits available in the data/opcode portion of the message, and the way these data are coded. In the analysis two different codings will be compared, using packets of 8 and 16 bits. A varying number of unused bits will be used.

Two methods of specifying the size of the difference code information will be used, *exact* and *chained*. In the exact method there is a field, called the *size* field, which specifies the number of packets which are used to hold the difference code. In the chained method one bit in each packet, called the *more* bit, is set if there is difference code packet following the current one. The exact method takes fewer bits if the difference codes are large, the chained method is better if difference codes are small. If there is at least one unused bit in the data/opcode portion of the message, then a field, called the *zero* field will be used to specify a difference code of zero. The remaining bits are used to specify the difference code itself, with one bit for the sign and the remainder for the magnitude.

The number of bits needed to code a difference code magnitude of size $c$ using the exact method is

$$A_u(c) = \begin{cases} \left\lceil \frac{c+2+\left\lceil \frac{b_A}{p} \right\rceil - u}{p} \right\rceil p, & \text{if } u > 0; \\ \left\lceil \frac{c+1+\left\lceil \frac{b_A}{p} \right\rceil}{p} \right\rceil p, & \text{if } u = 0; \end{cases}$$

where $u$ is the number of unused bits and $p$ is the packet size. For the chained method the number of bits needed is

$$A_u(c) = \begin{cases} \left\lceil \frac{c+2-u}{p-1} \right\rceil p, & \text{if } u > 0; \\ \left\lceil \frac{c+1}{p-1} \right\rceil p, & \text{if } u = 0. \end{cases}$$

Let $\overline{A}_u$ denote the average number of bits needed to code an address where $u$ unused bits are available (taken over all requests in a trace using some method). Then the reduction in data volume, taking into account fixed size packets is $\frac{b_M - b_A + u + \overline{A}_u}{b_M + u}$. Note that as $u$ increases, $\overline{A}_u$ decreases, mitigating the effect of the unused bits.

Values for $\overline{A}_u$ were found through simulation using packets of sizes 8 and 16 bits and values of $u$ of 0, 1, and $p/2$. Both methods of specifying the size of difference codes were used, the chained method always resulted in a smaller $\overline{A}_u$. The values of $\overline{A}_u$ and data volume reductions using the chained method obtained for simulations of full-map caches are listed in the table below. Note that the reductions obtained are comparable to those obtained without considering division into packets. Note also that there is a significant reduction in going from zero unused bits to one unused bit, but a much smaller additional reduction when a greater number of bits are unused.

TABLE 3
REDUCTION IN CODE SIZES.

| Benchmark | $p = 16$ | | | $p = 8$ | | |
|---|---|---|---|---|---|---|
| | $u = 0$ | $u = 1$ | $u = 8$ | $u = 0$ | $u = 1$ | $u = 4$ |
| FFT | 16.0/.83 | 8.7/.75 | 6.8/.75 | 11.5/.78 | 7.7/.74 | 7.1/.74 |
| Weather | 16.6/.83 | 13.9/.80 | 12.6/.80 | 15.0/.81 | 13.6/.80 | 13.2/.80 |
| Speech | 16.4/.83 | 16.0/.80 | 5.6/.73 | 11.3/.77 | 11.1/.77 | 10.2/.77 |
| Simple | 16.2/.83 | 13.0/.79 | 6.7/.75 | 12.3/.78 | 10.7/.77 | 10.3/.77 |

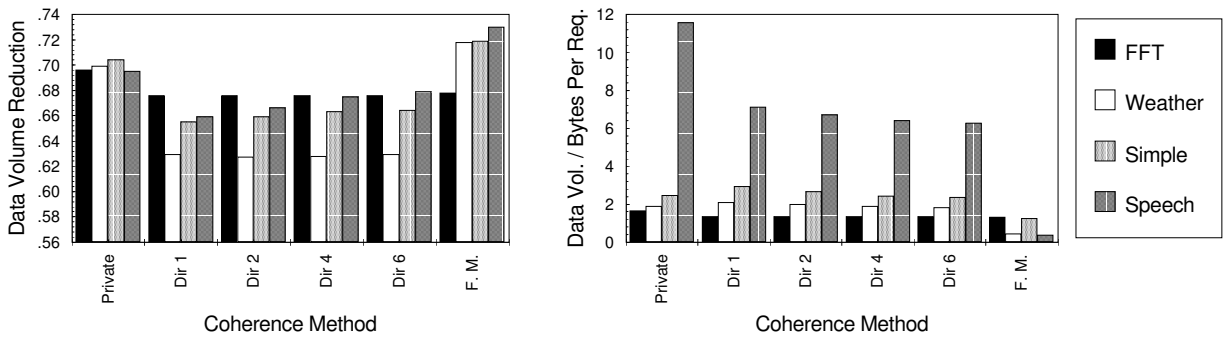


Fig. 4. Cache Coherence vs. Data Volume.

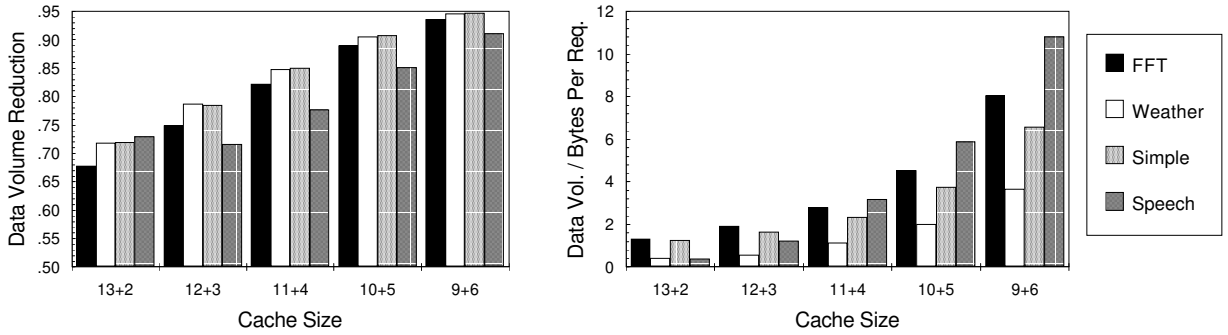


Fig. 5. Block Size vs. Data Volume.

*C. Interleaving Effects*

Because the code size varies with interleaving, it would be advantageous to adjust the interleaving for maximum effect. It may be that in many systems the interleaving of the memory address space to the MUs is fixed, or cannot be easily changed. As can be seen from Figure 3 there is only a small penalty for using interleaving bits near the optimal value. A system using difference coding of memory addresses would either need adjustable interleaving or its compilers would have to be written to take advantage of the hardware. See Section V [next page] for more details.

More important than the average difference code size is the reduction in data volume. In Figure 4 data volume and reduction in data volume are plotted against cache coherence method. The systems simulated have two–way $2^{13}$ set caches with a block size of two. As can be seen from the graph, reduction in data volume is about .7 (data volume with DCM is 70% of the data volume without DCM) for all the coherence schemes and benchmarks. Cache hit ratios also varied over a small range. Only absolute data volume varied with benchmark and coherence. The data volume in the simulated FFT benchmark varied the least, perhaps because of well organized communication. The data volume for the weather and simple benchmarks varied much more, suggesting that widely shared data generate a large number of coherence messages.

Regardless of which of the tested coherence methods are used, DCM achieved a significant reduction in data volume. Although full map cache directories worked quite well, they are expensive, so that in actual systems private coherence may be used.

Finally, the effect of block size on data volume was tested. Simulations were performed on a system with a full map cache. The block sizes were varied from 4 to 64 bytes while proportionally reducing the number of sets, thus keeping the cache size at a constant $2^{16}$ bytes. Results are plotted in Figure 5; as can be seen DCM is less effective with increasing block size. But, by examining the

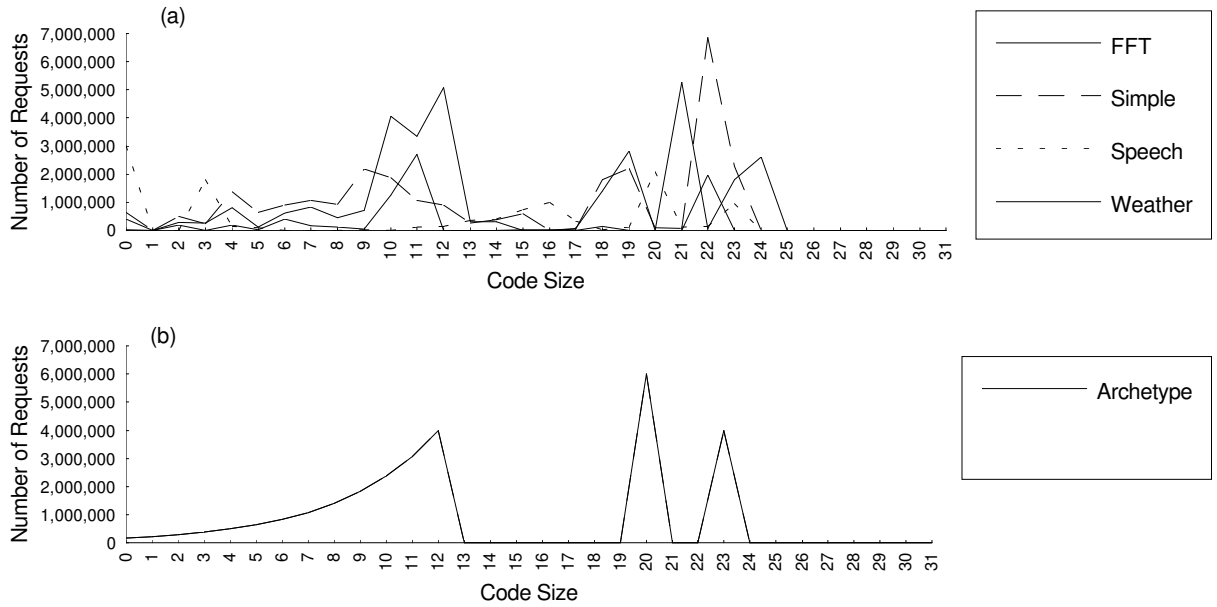


Fig. 6. Distribution of Code Sizes For One Memory.

cache data volumes in the plot, it can be seen that there is no advantage to large block sizes in these simulations.

The advantage of a large block size is a higher hit ratio, as large blocks capture spatial locality. In parallel systems block sizes are smaller than in serial systems for several reasons. First, the interconnection network is an expensive resource, so that any data moved across it should be useful. Large blocks will have more data which is never read. Second, large blocks can contain data which a second processor needs to write. This will increase the number of invalidation messages, contesting the network and slowing down the system [6,20]. The dependence of DCM on small block sizes then does not conflict with the block size needs of other parts of the system.

## V. Interleaving

Since the effectiveness of DCM is strongly dependent on interleaving a method of finding an interleaving favorable to DCM without compromising other aspects of system performance needs to be found. After considering the effects of interleaving on DCM and other parts of the system four methods of determining interleaving will be described: using a fixed interleaving, having the operating system specify an interleaving, having the compiler specify an interleaving, and determining the interleaving by observing a sample run.

### *A. Effect of Interleaving on Code Size*

To understand how the interleaving affects difference code size the simulator was designed to find, in addition to other data, the distribution of difference code sizes. Examination of this data revealed that despite the fact that the traces differed in program type, (and in the case of speech, parallel programming method), all exhibited some common behavior.

Figure 6(*a*) shows the distribution of code sizes for a simulated system using a single memory (uncached) for the four traces. Note that the distribution of each of the traces has roughly the following shape: an exponential increase in height (for code sizes 0-11), followed by a drop to almost

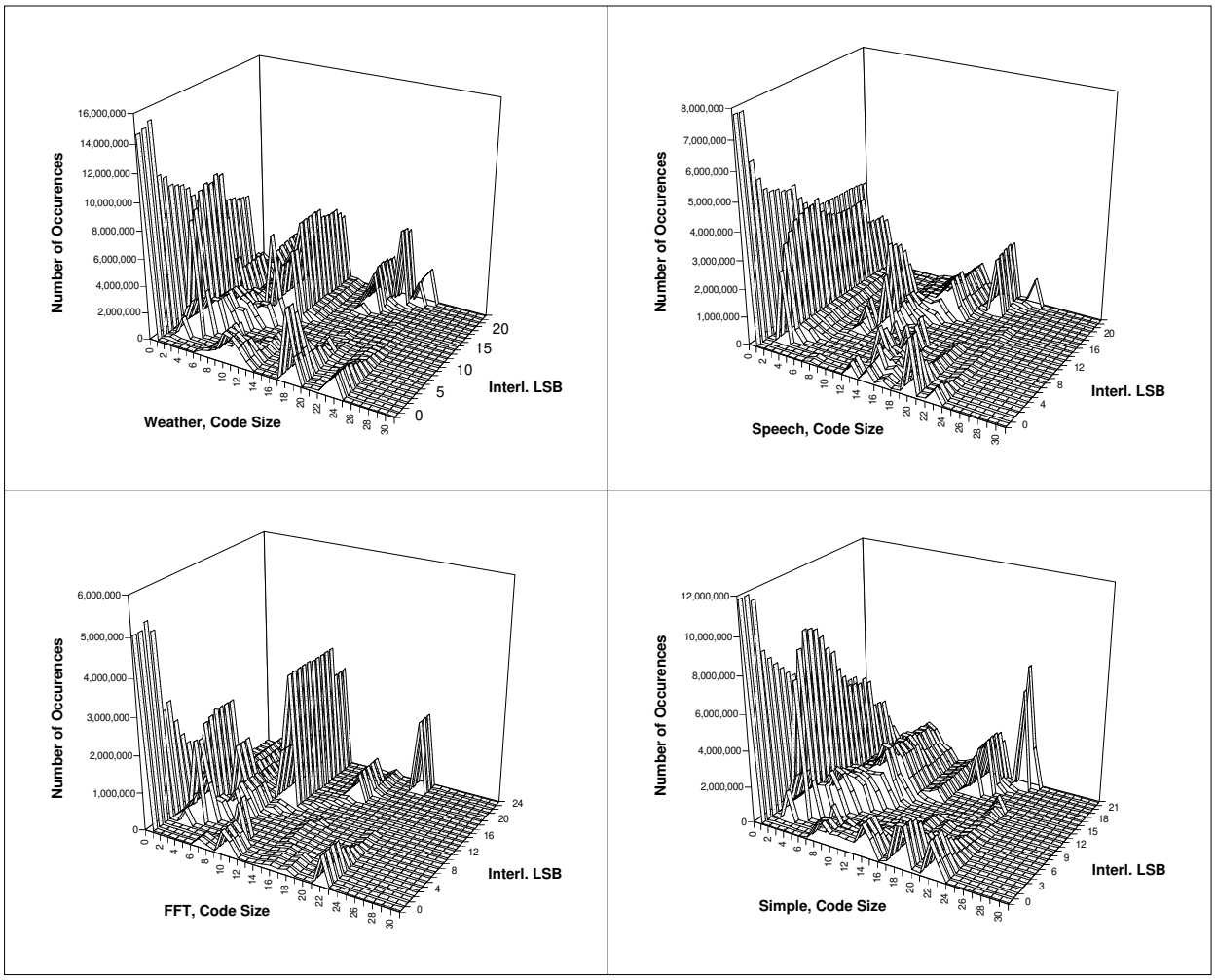


Fig. 7. Code Size Distribution vs. Interleaving Without Caches.

zero, followed by one or more spikes (between code sizes 17 and 25). This archetypical distribution is plotted in Figure 6($b$).

Figure 7 shows the distribution of difference code sizes for the four traces for a variety of interleavings, on a system without a cache and using 61 memories. A prime number of MUs was used to minimize aliasing effects, as has been done by others [1]. The effect of aliasing on overall performance was minor but certain features of the 3D plots are hidden when the number of memories is a power of two. The axis marked "interl. LSB" shows the least significant address bit used for memory number. The other axes indicate code size and number of occurrences (of the code size). Each ribbon represents a simulation for a particular interleaving; the height shows the number of messages which generate that particular difference code size.

The plots for the four traces have common features. Each has a valley (flat stripe) running diagonally along the floor. The valley is caused directly by the interleaving: any pair of addresses which would generate a difference code in the interleaving range reside on different memories, so no such difference code can be generated. Each plot also shows a concentration of difference code sizes in a few ranges. To reduce average difference code size interleaving would be set to cover one of the larger ranges. As can be seen, this does not greatly alter the shape of other parts of the distribution.

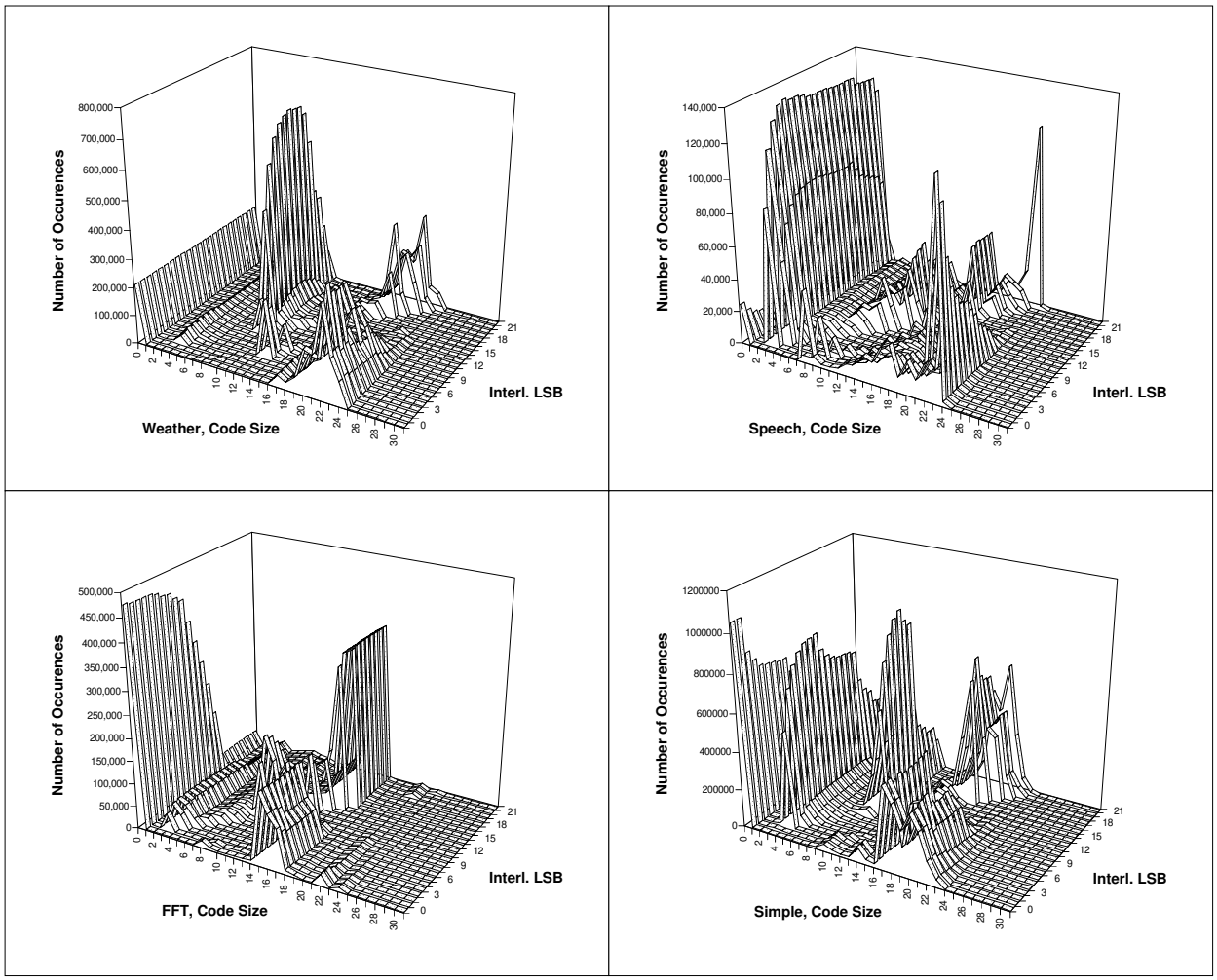


Fig. 8. Code Size Distribution vs. Interleaving With Caches.

Consider the number of requests associated with code size zero in the FFT and weather simulations. As interleaving increases, the number of these requests increases, reaches a peak, and then decrease. That can be explained informally as follows. A code size of zero corresponds to two consecutive requests to the same memory addresses, a large number of such code sizes would require that some working set memory addresses be alone on memories. When interleaving is zero, addresses are spread most evenly among memories, resulting in a large number of addresses alone in a memory. As interleaving increases, nearby frequently accessed memory locations are concentrated in some memories. Thus there are fewer working set addresses alone on memories, and fewer code sizes of zero. The initial small increase is probably due to data alignment.

In Figure 8 similar plots are shown for systems using caches. The principle difference is of course the large reduction in the number of messages. Another important difference is the greater uniformity in distribution as the interleaving is changed. The uniformity is probably due to the caches. Because of the caches, most of the requests the DCM hardware processes are for shared data, which would miss the cache or generate coherence messages. The shared data might be accessed in an orderly fashion, reflecting the underlying algorithm. Requests that hit the cache would include code, iteration variables, important local data structures, etc. Each type of request might have an orderly pattern of addresses, but when they are interspersed the resulting sequences of addresses are more chaotic.

*B. Detection of Optimal Interleaving*

To detect the minimum code size interleaving a variety of techniques of varying practicality can be used. The techniques differ in the system level the information they use for determining interleaving resides. The levels are hardware (specifically cache characteristics), compiler, program, and the lowest level, program input data. It's easier to make use of the information at the higher levels, so that it would be better if this information were sufficient. This seems to be the case for the four traces tested.

At the hardware level information about cache structure and other hardware attributes is used to determine interleaving. For this to work there would have to be an interleaving which would work well for most compilers and programs. This interleaving might be determined by a property of running code which varies little from program to program or compiler to compiler (so that the compiler and program can be ignored). Alternatively, it is possible that cache characteristics determine interleaving.

For example, consider systems using set associative caches (such as the one simulated) where the interleaving is done using a subset of the tag bits. A cache replacement would result in requests for which only the tag part of the addresses differ, these would be directed to different MUs. Since the most significant bits of an address are usually the tag bits a large difference code is avoided. (As an added benefit, the traffic is split between two MUs.) To find the best system level interleaving benchmark programs would be run and the interleaving found would always be used on that system.

It is possible that the compiler (but not the program) is most important in determining interleaving. This might be due to the distribution of code, static data, parameter stack, the heap, and other memory items. The pattern of distribution would vary from compiler to compiler but might be similar enough from program to program using a single compiler. In such a case the interleaving would be determined only by the compiler and could be found once for each compiler through analysis of benchmark programs.

It is likely that the program itself might contain the information to determine the best interleaving (for almost any input data). The compiler would find the interleaving by analyzing the program, perhaps using the information it collects when compiling the program. The analysis the compiler does could be quite detailed, or it might be as simple as computing a function of instruction frequencies. It would not be practical for the compiler to determine interleaving if it had to perform a detailed analysis because the difficulties of determining just what analysis is necessary, writing the code to perform the analysis, and the added time to compile the programs would probably not offset the performance gain. On the other hand a simple analysis, as described below, might be easily implemented so that interleaving can be determined with little overhead.

A simple compiler analysis would base the interleaving information on aggregate properties of a program. For example, rather than considering individual addresses instructions might be classified based on the areas of memory they access. Interleaving would be based on finding frequencies of instruction class pairs (e.g., an instruction which accesses a static shared variable followed by one that performs a pointer access), perhaps with weighting based on expected execution frequencies. Furthermore, library functions could be analyzed in advance so exact information on their effect would be available. The amount of time to analyze a program in this way would depend upon the exact method used. The fastest methods should add little to compile time.

It is possible that for most programs the best interleaving can only be found using the program's input data. (Regardless of whether typical programs have this property, it would not be difficult to write one.) This level is the most difficult to handle because the program would have to be monitored over some portion of its execution. After the interleaving was determined, the memory space would have to be reshuffled, which would be time consuming. Such data-dependent interleaving

adjustment could be made to work if a program was to be run many times with similar data; the interleaving could be found for one run and used on subsequent runs.

If interleaving changes with phases of execution then memory would have to be shuffled after each phase, this may be impractical because of the time needed to shuffle memory. If phase changes could be detected (perhaps by the use of special instructions) then interleaving could be dynamically changed based on a sample run, but to be practical the phases would have to be very long compared to the time to shuffle memory.

The optimal interleaving could be determined from a sample run using hardware, perhaps the same hardware that implements DCM. During the sample run address space would be mapped to MUs in some standard way. Difference coding would not be used, rather the DCM hardware would compute difference codes for all possible interleavings, keeping track of the average code sizes.

An economical way to do this would be to carefully partition the address space into subsets, one subset per interleaving. The DCM hardware would compute the difference code for an address using the appropriate interleaving. The address space would be carefully divided so that the average difference code size for a partition would be close to the average difference code size for the entire address space using that interleaving.

Each of these methods of extracting interleaving information needs to be tested further, starting at the system level. Testing at the system, compiler, and data level would be done by running many simulations to accumulate empirical evidence. Finding a workable scheme for extracting information at the program level would be far more involved since an exact method of finding the interleaving would first have to be worked out. For the traces simulated system information is sufficient, if this is the case elaborate techniques need not be employed.

## VI. Conclusion

In this paper a method for reducing the information flow in a shared memory parallel computer was described. The method, called DCM, takes advantage of spatial locality in memory address references. This is done by sending only the difference between consecutive addresses, thus reducing the number of bits needed to code the memory address. The memory unit receiving the difference reconstructs the address based upon a most recent address which it has stored.

The technique of difference coding memory addresses can be used on almost any single-address-space system regardless of cache coherence protocol, prefetching, or other message generating technique used. All that is necessary is that memory addresses are sent between an identifiable sender to an identifiable receiver.

The reduction in data volume takes advantage of the spatial and temporal locality of memory addresses sent between processor/memory pairs. This locality is affected by several system factors, such as address space mapping, data distribution, cache structure, and the parallel algorithm being executed. Although these factors might reduce some locality, communication between tasks cannot be avoided and it seems unlikely that this communication would occur in an unpredictable manner. Therefore regardless of these factors there will probably always be locality for DCM to exploit.

A simulation study was carried out, which showed data volume could be reduced to 70% or less of its original size. The simulations were run over several benchmark programs and using several cache coherence techniques; the reductions were observed over all the programs and coherence techniques. The reduction in data volume could result in a lower cost or higher performance system. The simulations were done for programs accessing a 32 bit address space; the reductions for 64 bit address spaces would be higher.

Because of its importance to DCM and effect on performance the area of interleaving merits further investigation. First, the highest level at which interleaving information can be found should be determined. This would require simulations of a wide variety of programs compiled by different compilers. Program analyzers and specially written compilers might also be developed and tested. Second, the effect of minimum–code–size interleavings on congestion, latency, and other aspects of system performance needs to be tested to verify that system performance would not be degraded. This would refine the cost and performance estimates for DCM.

There are several variations of DCM that might be explored further. In one variation the bits mapping addresses to memory modules are not contiguous. For example these bits could be interspersed with the bits used to determine the cache set number. This would insure that cache replacement requests are split between two MUs.

The multilocus schemes promise smaller code sizes without the need for a special address space mapping. By organizing the history memories as a cache, one could potentially use many loci between each PE/MU pair without using a large amount of history memory. Both ideas should be investigated to determine their effectiveness.

Better codings for memory addresses might be found. For the traces analyzed here (with one exception), a first order linear predictor was sufficient. It might be worthwhile to investigate the use of nonlinear predictors. For example, the loci chosen in a multilocus scheme might be based on the most recent few loci, similar to the methods used for branch prediction [14].

Finally, DCM should be tested on a variety of systems with a variety of programs. The simulations done here used only virtual addresses. More detailed analyses might be done using real addresses. Many new multiprocessor architectures are being proposed. The work here tested DCM using what might be a typical architecture, this work could be extended by testing the idea on other architectures, particularly those which are likely to be used for future multiprocessor designs.

References


[1] P. Budnik and D. Kuck, "The organization and use of parallel memories," *IEEE Transactions on Computers*, vol. 20, pp. 1566–1569, December 1971.

[2] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory–based cache coherence in large–scale multiprocessors," Massachusetts Institute of Technology, VLSI Memo, 1989, 1989.

[3] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory based cache coherence in large–scale multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 49–59, June 1990.

[4] W. J. Dally, "Express cubes: improving the performance of k–ary n–cube interconnection networks," *IEEE Transactions on Computers*, vol. 40, no. 9, pp. 1016–1023, September 1991.

[5] R. Duncan, "A survey of parallel computer architectures," *IEEE Computer*, vol. 23, no. 2, pp. 5–16, February 1990.

[6] S. J. Eggers and T. E. Jeremiassen, "Eliminating false sharing," in *Proceedings of the International Conference on Parallel Processing*, August 1991, vol. I, pp. 377–381.

[7] M. Farrens and A. Park, "Dynamic base register caching: a technique for reducing address bus width," *ACM Computer Architecture News*, vol. 19, no. 3, pp. 128–137, May 1991.

[8] T.-y. Feng, "A survey of interconnection networks," *IEEE Computer*, pp. 12–27, December 1981.

[9] D. Hammerstrom and E. Davidson, "Information content of CPU memory referencing behavior," in *Proceedings of the International Symposium on Computer Architecture*, March 1977, pp. 184–192.

[10] N. S. Jayant and P. Noll, "Digital coding of waveforms," Englewood Cliffs, New Jersey: Prentice–Hall, 1984.

[11] D. Kleitman, F. T. Leighton, M. Lepley, and G. L. Miller, "New layouts for the shuffle exchange graph," in *Proceedings of the ACM Symposium on the Theory of Computing*, 1981, pp. 278–292.

[12] V. P. Kumar and S. M. Reddy, "Augmented shuffle–exchange multistage interconnection networks," *IEEE Computer*, vol. 20, no. 6, pp. 30–41, June 1987.

[13] D. H. Lawrie, "Access and alignment in an array processor," *IEEE Transactions on Computers*, vol. 24, no. 12, pp. 1145–1155, December 1975.

[14] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, vol. 17, no. 1, pp. 6–22, January 1984.

[15] J. Y. Ngai and C. L. Seitz, "A framework for adaptive routing in multicomputer networks," *ACM Computer Architecture News*, vol. 19, no. 1, pp. 6–14, March 1991.

[16] G. F. Pfister and V. A. Norton, "'Hot spot' contention and combining in multistage interconnection networks," *IEEE Transactions on Computers*, vol. 34, no. 10, pp. 943–948, October 1985.

[17] A. R. Pleszkun, B. R. Rau, and E. S. Davidson, "An address prediction mechanism for reducing processor–memory address bandwidth," in *Proceedings of the IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Analysis*, 1981, pp. 141–148.

[18] C. L. Seitz, "Concurrent VLSI architectures," *IEEE Transactions on Computers*, vol. 33, no. 12, pp. 1247–1265, December 1984.

[19] H. J. Siegel, W. G. Nation, C. P. Kruskal, and L. M. Napolitano, jr., "Using the multistage cube topology in parallel supercomputers," *Proceedings of the IEEE*, vol. 77, no. 12, pp. 1932–1953, 1989.

[20] P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *IEEE Computer*, vol. 23, no. 6, pp. 12–24, June 1990.

[21] H. S. Stone, "High performance computer architecture," Reading, Massachusetts: Addison–Wesley, 1990.

[22] W. T. Wilner, "Burroughs B1700 memory utilization," in *Proceedings of the AFIPS*, 1972, FJCC, vol. 41, pp. 579–586.