1

THE PERFORMANCE IMPACT OF ADDRESS RELATION CACHING

Peter A. Dinda
David R. O'Hallaron

School of Computer Science,
Carnegie Mellon University,
Pittsburgh, PA 15213

ABSTRACT

An important portion of end–to–end latency in data transfer is spent in address computation, determining a relation between sender and receiver addresses. In deposit model communication, this computation happens only on the sender and some of its results are embedded in the message. Conventionally, address computation takes place on–line, as the message is assembled. If the amount of address computation is significant, and the communication is repeated, it may make sense to remove address computation from the critical path by caching its results. However, assembling a message using the cache uses additional memory bandwidth.

We present a fine grain analytic model for simple address relation caching in deposit model communication. The model predicts how many times a communication must be repeated in order for the average end–to–end latency of an implementation which caches to break even with that of an implementation which doesn't cache. The model also predicts speedup and those regimes where a caching implementation never breaks even. The model shows that the effectiveness of caching depends on CPU speed, memory bandwidth and the complexity of the address computation. We verify the model on the iWarp and the Paragon and find that, for both machines, caching can improve performance even when address computation is quite simple (one instruction per data word on the iWarp and 16 instructions per data word on the Paragon.)

To show the practical benefit of address relation caching, we examine the performance of an HPF distributed array communication library that can be configured to use caching. In some cases, caching can double the performance of the library. Finally, we discuss other benefits to caching and several open issues.

This research was sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152, in part by the National Science Foundation under Grant ASC-9318163, and in part by a grant from the Intel Corporation.
The authors' email addresses are: pdinda@cs.cmu.edu and droh@cs.cmu.edu.

1 INTRODUCTION

A basic communication operation for applications running on distributed memory machines is a *point-to-point communication*, where the *sender* transfers data, in the form of a *message*, from various locations in its memory to various locations in the memory of the *receiver*. Because each communication represents overhead to the application, it is important to perform this operation as quickly as possible. This task is complicated by the fact that applications do not always keep the data they need to communicate in convenient contiguous memory locations.

In general, point-to-point communication with conventional message passing systems like PVM [13], NX [6], and MPI [14] involves three basic steps on the sender: (1) *address computation*, which determines the local addresses of the data items to be transferred, (2) *message assembly*, which copies the data items into a contiguous buffer, and (3) *message transfer*, which copies the message from the buffer to the wire. The receiver performs a symmetric set of steps. The performance measure of interest is *end–to–end latency*, which is the time to perform all three steps on both the sender and receiver.

A different and typically more efficient approach, known as *deposit model message passing* [10, 12], combines the address computation steps on the sender and the receiver into a single step on the sender. The result of the address computation step on the sender is an *address relation* that associates addresses on the sender with addresses on the receiver. The sender performs the same steps as above, but passes the receiver addresses along with the data. The receiver uses these addresses to scatter the data to the correct memory locations. Other properties of the deposit model make it possible for sophisticated implementations to assemble and disassemble to and from the wire, possibly via specialized hardware. In this paper, we concentrate on a simple implementation that stages data in memory at both the sender and receiver.

In any implementation of deposit model message passing, and in conventional message passing systems, address computation is in the critical path of data transfer. Address relation caching decouples address computation and data transfer by doing address computation off-line, storing the address relation in memory, and doing message assembly using the stored address relation. A primary motivation is to amortize the address computation cost of communications that are repeated (for example, an HPF array assignment statement in a loop.) A similar approach is used by the CHAOS system [7] to improve the performance of irregular collective communications.

To determine the effectiveness of address relation caching, we develop a simple analytic model for predicting the performance tradeoffs between cached and uncached point–to–point communications for a simple representation of the address relation. The model shows that there are regimes where caching is always ineffective, as well as regimes where caching becomes effective after the communication has been repeated a sufficient number of times. For these latter regimes, the model predicts the number of times the communication must be repeated in order for caching to break even, as well as speedup. Speedup is bounded only by the complexity of the address relation computation.

We validate the simple model on both the Intel Paragon and Intel iWarp systems. The model's predictions closely match our measurements. Because of iWarp's higher ratio of memory bandwidth to instruction issue rate, caching becomes effective with far simpler address computations than on the Paragon. However, in both cases, even if only a small number of instructions are executed per tuple of the address relation, caching is effective.

To demonstrate the practical benefit of caching, we compare the cached and uncached performance of an HPF [2] communication library on the Paragon. We find that caching becomes more effective as the distribution of the data becomes more irregular, and thus the address relations become harder to compute.

We conclude with discussion about other issues and benefits of caching.

2 DEPOSIT MODEL COMMUNICATION

In a conventional communication system, such as PVM [13], NX [6], or MPI [14], the sender computes a binary relation S between local addresses and addresses offset into the *message* that is transported between it and the receiver. If $(s, m) \in S$ then the data item at local address s is copied into the message at offset m. The receiver computes a binary relation R between addresses offset into the message and local addresses. If $(m, r) \in R$, then the data item at offset m in the message is copied to local address r. No actual address information is contained in the message.

In deposit model communication, the sender computes a binary relation SR between local addresses at the sender and local addresses at the receiver – if $(s, r) \in SR$ then the data item at address s will be transported to address r at the receiver. We refer to SR as the *address relation*. The sender uses the address relation to *assemble* a message that conceptually consists of 2-tuples (r, d) where data item d (read from address s) is to be written to, or deposited at address r on the receiver. The receiver runs a simple *deposit engine* which receives the message and *disassembles* the message, writing each d to its corresponding local address r. If the message actually contains 2-tuples, it is in the *address-data pair* format. Often, the format of the message can be optimized so that the overhead of including addresses in the message is very low. One such format is referred to as *address-data block*. In this format, the message consists of 3-tuples (r, l, D), where D consists of l data items that are deposited at consecutive addresses starting at r. Obviously, other optimizations are also possible, depending on the properties of the address relation.

Strictly speaking, including addresses in the message is a variant of deposit called *direct deposit*. Other properties of the deposit model make it possible for sophisticated implementations to eliminate in memory staging of the message, possibly by using specialized hardware. We concentrate on a simple implementation that stages the message in memory and uses a simple address-data pair message format. Further details of deposit model communication can be found in [12].

3 ANALYTIC MODELS

This section presents models for the average end–to–end latency of a point-to-point communication using uncached and cached implementations of deposit model message passing. We consider the simplest (and least efficient) message format and cache structure and derive expressions for the average latency for both uncached and cached communication. We compare these expressions to provide some insight into the effect of caching.

3.1 Algorithm properties

Clearly, the time spent computing the address relation depends on the algorithm and its implementation. We model computing the address relation as instruction execution limited and as delivering the address relation in an on-line manner. We assume that all computation is done in registers, so we concentrate on the parameter n_g, the number of instructions executed per element of the relation.

3.2 Machine properties

Since computing the address relation is instruction execution limited, we must know r_i, the number of instructions the machine can execute per second. We also need to know r_c, the communication bandwidth between the two nodes. The memory system is parameterized by four values. The first two values, r_{rc} and r_{wc}, are the read and write bandwidths for unit–stride accesses, respectively. The other two values, r_{rr} and r_{wr}, are the read and write bandwidths for random memory accesses.

3.3 Uncached communication

The time for an uncached point-to-point communication has four components: t_g, the time to compute the address relation; t_{au}, the message assembly time; t_c, the message communication time; and t_d, the message disassembly time at the receiver. The total time for the communication is

$$t_{uncached} = t_g + t_{au} + t_c + t_d$$

In this simple model, we assume that the message format is address-data pair.

The time to compute the address relation is largely determined by the complexity of algorithm used, its implementation, and the instruction execution rate of the machine. Thus for an n word message, the time required is

$$t_g = \frac{n n_g}{r_i}$$

The code that computes the address relation presents tuples (s, r) in registers in an on-line manner. For each of these tuples, we read the data word at address s, write r to the message buffer, then write the data word to the message buffer. Note that the writes to the message buffer are to consecutive addresses, so their performance can be captured by r_{wc}, the contiguous write bandwidth of the machine. On the other hand, the sender side addresses of the tuples may not be contiguous, so we capture their performance with r_{rr}, the random read bandwidth of the machine. Furthermore, we capture the loop overhead as a number of instructions, n_{au}. Thus the time to assemble the message is

$$t_{au} = n\left(\frac{1}{r_{rr}} + \frac{2}{r_{wc}} + \frac{n_{au}}{r_i}\right)$$

Note that this expression does not take into account cache thrashing between the reads and writes. An implementation will probably optimize access patterns by reading several data items at a time. This will bring it close to the above expression.

After the message has been assembled, we use the communication hardware to transport it to the receiver. The message communication time is

$$t_c = \frac{2n}{r_c}$$

Note that r_c is really a function of $2n$, but we assume that n is sufficiently large that we can use a fixed value for r_c. Further, this value is the same without and with caching and becomes irrelevant in comparing them.

On the receiving node, each word of the message is read consecutively and each address-data pair results in one random write to memory. Additionally, n_d loop overhead instructions are executed for a total disassembly time of

$$t_d = n\left(\frac{2}{r_{rc}} + \frac{1}{r_{wr}} + \frac{n_d}{r_i}\right)$$

Note that this expression does not take into account cache thrashing between the reads and writes. Again, an implementation will probably improve performance by writing several data items at a time, bringing it close to the expression.

Substituting into the expression for $t_{uncached}$, we see that an n word communication between two nodes requires time

$$t_{uncached} = n\left(\frac{n_g + n_{au} + n_d}{r_i} + \frac{1}{r_{rr}} + \frac{2}{r_{wc}} + \frac{2}{r_c} + \frac{2}{r_{rc}} + \frac{1}{r_{wr}}\right) \tag{1.1}$$

3.4 Cached communication

The average time for an cached point-to-point communication has five components: t_g, the time to compute the address relation; t_o, the time to store the address relation in the cache, t_{ac}, the message assembly time from the cached address relation; t_c, the message communication time; and t_d, the message disassembly time at the receiver. The average time for the communication is

$$t_{cached} = \frac{t_g + t_o}{n_i} + t_{ac} + t_c + t_d$$

Note that the first two components are amortized over the number of times the communication is performed, n_i. t_g, t_c, and t_d are the same between the uncached and cached schemes and were determined in the preceding section. We continue the assumption that the message format is address-data pair. Further, we choose the simplest possible cache data structure: an array of 2-tuples that represent the relation.

For the simple model, as each tuple of the address relation is generated, we write it to an array. Thus storing the address relation in the cache involves two contiguous writes for each in-register tuple produced by the address relation computation. In addition, we execute n_o loop overhead instructions, so

$$t_o = n\left(\frac{2}{r_{wc}} + \frac{n_o}{r_i}\right)$$

When assembling the message from the cached address relation, for each tuple in the message, we first perform two consecutive reads to read a tuple of the cached address relation. Then, we perform a random read to read the data word. Finally, we perform two consecutive writes to write the data word and the receiver address into the message buffer. We also execute n_{ac} loop overhead instructions, thus the time to assemble a message with n data words is

$$t_{ac} = n\left(\frac{2}{r_{rc}} + \frac{1}{r_{rr}} + \frac{2}{r_{wc}} + \frac{n_{ac}}{r_i}\right)$$

Note that this expression does not take into account cache thrashing between the reads and writes, but, again, a good implementation will perform several data word reads at a time and thus approach this performance.

Substituting into the expression for t_{cached}, we see that an n word communication between two nodes requires time

$$\begin{aligned} t_{cached} &= n\left(\frac{(n_g + n_o)/r_i + 2/r_{wc}}{n_i} + \frac{n_{ac} + n_d}{r_i} + \frac{2}{r_{rc}}\right. \\ &\quad \left. + \frac{1}{r_{rr}} + \frac{2}{r_{wc}} + \frac{2}{r_c} + \frac{2}{r_{rc}} + \frac{1}{r_{wr}}\right) \end{aligned} \tag{1.2}$$

3.5 Effects of caching

Equations 1.1 and 1.2 provide insight into the following questions: For a point–to–point communication that occurs multiple times at run–time, can a cached implementation ever outperform an uncached implementation? If so, how many times must the communication be repeated before the cached implementation wins? For a large number of iterations, what is the speedup in message assembly time of the cached implementation over the uncached implementation?

Caching can only be effective if we can assemble the message faster using the cache than without using the cache, i.e.,

$$t_{ac} < t_g + t_{au}$$

or

$$n_g > \frac{2r_i}{r_{rc}} + n_{ac} - n_{au} \tag{1.3}$$

The right hand side of equation 1.3 is a machine–specific threshold below which caching is ineffective. Notice that it is largely determined by the ratio of instruction issue rate and memory read bandwidth.

If 1.3 is true, then we can also predict how many times cached address relation must be used in order to fully amortize the cost of caching the relation. This break even point is found by setting $t_{cached} = t_{uncached}$ and solving for n_i. In doing this we find that the break even point for caching is

$$n_i = \left\lceil \frac{r_{rc}}{r_{wc}} \left(\frac{(n_g + n_o) r_{wc} + 2r_i}{(n_g + n_{au} - n_{ac}) r_{rc} - 2r_i} \right) \right\rceil \tag{1.4}$$

It is important to point out that as n_g moves away from the threshold, the break even point decreases sharply because the n_g terms in the fraction begin to dominate.

Finally, the speedup in message assembly time for the cached implementation over the the uncached implementation is

$$\frac{t_g + t_{au}}{t_{ac}} = \frac{r_{rc} r_{wc} r_{rr} (n_g + n_{au}) + r_i r_{rc} (r_{wc} + 2r_{rr})}{r_{rc} r_{wc} r_{rr} n_{ac} + r_i r_{rc} (r_{wc} + 2r_{rr})} \tag{1.5}$$

The speedup is linear in n_g, so is bounded only by the complexity of the address computation.

4 MODEL VALIDATION

In this section we validate the models from Section 3 for point–to–point communications on an iWarp system and on a Paragon. To test the predictive power of the models, we use (1.4) to predict the number of iterations necessary for the average cached message assembly time to break even with the uncached message assembly time as a function of n_g, and we use (1.5) to predict the speedup of cached over uncached message assembly, also as a function of n_g. In each case, the address relation that is generated is the identity relation, and n_g, the number of instructions to compute the relation, is varied by inserting nops into the code stream. The predictions are compared to actual measured running times on the hardware.

n_g	Act	Pred-Stat	Pred-Dyn
1	13	INF	13
2	7	12	7
3	5	7	5
4	4	5	4
5	4	4	4
6	3	4	3
7	3	3	3
8	3	3	3

(a)

n_g	Act	Pred-Stat	Pred-Dyn
1	1.05	1.00	1.07
2	1.12	1.09	1.15
3	1.22	1.16	1.23
4	1.28	1.25	1.32
5	1.35	1.33	1.39
6	1.43	1.41	1.47
7	1.49	1.49	1.54
8	1.59	1.59	1.61

(b)

Figure 1 Model validation on iWarp: (a) Iterations required to break even (Eqn. 1.4), (b) Speedup in message assembly (Eqn. 1.5)

4.1 iWarp

On the iWarp [1] system, $r_{rc} = r_{rr} = r_{wc} = r_{wr} = 10^7$ words/second, and a node can execute $r_i = 2 \times 10^7$ instructions/second. The memory system parameters were derived directly from the iWarp specification. Specifically, the iWarp clock speed is 20 MHz and all memory operations complete in two cycles. There is no cache and we assume no dual dispatch of loads and stores.

The implementation parameters (n_{au}, n_o, and n_{ac}) can be estimated *statically* by simply counting the number of instructions in each loop, or *dynamically* by accounting for bypass stalls that arise in the loops. The static estimates for iWarp are $n_{au} = 5$, $n_o = 6$, and $n_{ac} = 2$. The dynamic estimates are $n_{au} = 7$, $n_o = 8$, and $n_{ac} = 3$.

Figure 1(a) shows the measured and predicted number of iterations needed on iWarp for cached assembly to break even with uncached assembly for different n_g. The predictions made using the dynamic measurements are exact. The predictions made using the static measurements are pessimistic for small n_g, but quickly converge to the actual numbers as n_g increases.

Figure 1(b) compares the actual and predicted speedup of cached over uncached message assembly on iWarp. As in Figure 1(a), the dynamic predictions are quite close for all n_g and the static predictions are pessimistic for small n_g. However, it is important to note that even with static measurements, the predictions only become inaccurate when n_g approaches the minimum number of instructions where the cached implementation can outperform the uncached implementation. Another important point is that the purpose of Figure 1(b) is to help validate the models when n_g is small, and not to highlight impressive speedups. In fact, the speedups grow arbitrarily large as n_g increases (see Equation 1.5.)

4.2 Paragon

A Paragon node contains a 50 MHz pipelined, multiple issue RISC processor [5, 4] . We estimated the instruction issue rate by noting that we do not use the floating

n_g	Act	Pred-Stat
14	NONE	NONE
15	NONE	NONE
16	19	NONE
17	11	41
18	9	20
19	7	14
20	6	11
21	5	9
22	5	7
23	4	7
24	4	6
25	4	5

(a)

n_g	Act	Pred-Stat
14	NONE	NONE
15	NONE	NONE
16	1.03	NONE
17	1.06	1.01
18	1.09	1.02
19	1.12	1.03
20	1.15	1.04
21	1.17	1.05
22	1.20	1.06
23	1.22	1.07
24	1.24	1.08
25	1.26	1.09

(b)

Figure 2 Model validation on Paragon: (a) Iterations to break even (Eqn. 1.4), (b) Speedup in message assembly (Eqn. 1.5)

point pipeline. Therefore we can issue at most $r_i = 5 \times 10^7$ instructions/second. We directly measured the memory parameters and found $r_{rc} = 9 \times 10^6$ words/second, $r_{wc} = 8.2 \times 10^6$ words/second, and $r_{rr} = r_{wr} = 0.9 \times 10^6$ words/second. Only a static measurement (instruction counting) of the implementation properties was performed, which found $n_{au} = 6$, $n_o = 7$, and $n_{ac} = 11$.

Figure 2(a) shows the measured and predicted number of iterations needed for cached assembly to break even with uncached assembly for different n_g. The predictions are pessimistic, but quickly converge to the actual values with increasing n_g. Notice that n_g must be much larger to permit break even on the Paragon than on the iWarp. This is not surprising given that the Paragon's instruction issue rate is much higher than its memory bandwidth, while the iWarp's issue rate is well matched to its memory bandwidth.

Figure 2(b) compares the actual and predicted speedup of cached message assembly over uncached message assembly on the Paragon. Notice that speedup does not increase as quickly with n_g as on the iWarp (Figure 1(b).) This is the result of the lower memory bandwidth relative to instruction issue rate on the Paragon. The predictions are not as accurate as for the iWarp, reflecting the fact the Paragon's memory system is more complex. However, the predictions are pessimistic and converge to the correct values.

5 CACHING IN HPF

This section investigates the impact of caching on the performance of array assignments in task parallel HPF programs. In a task parallel array assignment statement, the source and destination arrays are distributed over disjoint sets of processors. The test cases presented are written in Fx [3] (a dialect of HPF that integrates task and data parallelism) and compiled and run on an Intel Paragon. Communication is performed

Figure 3 Experimental setup

with an enhanced version of the Fx run–time library that supports cached and uncached message assembly. The appropriate library function is invoked on each source node, with the distributions of the source and destination arrays passed in as arguments. Address computation is performed using the algorithm in [9]. The library can either cache the results of this computation, or use them to directly assemble a message.

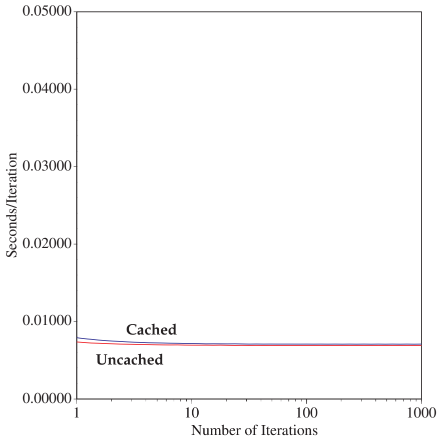
We measured the performance of the library, with and without caching, for a variety of test cases. As shown in Figure 3, each test case consists of two data parallel tasks that repeatedly assign a 512×512 *source array* S, distributed over 16 nodes in the *source task*, to a *destination array* D, distributed over a disjoint set of 16 nodes in the *destination task*. Each test case is characterized by the distributions of the source and target arrays.

The results indicate that the effectiveness of caching depends on the source and destination distributions, and thus on the complexity of address computation. In some cases caching is worse, in other cases it is marginally better, and in other cases it is significantly better. Three of the Paragon test cases (Figure 4) illustrate this point quite clearly:

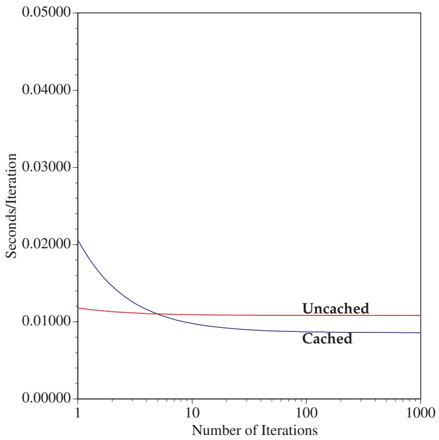
Case 1: S(:,BLOCK) $\rightarrow$ D(:,BLOCK). In this case, the distributions of the source and destination arrays are identical. The address computation for a block–to–block assignment of this sort is a common case that is highly optimized in the library. Because address computation is very inexpensive, caching is not helpful in this case. In fact, although it is hard to see in the figure, even after 1000 iterations the average end–to–end latency for the cached version is marginally higher than that of the uncached version. For this case, cached message assembly is slightly slower than uncached message assembly.

Case 2: S(BLOCK,:) $\rightarrow$ D(:,BLOCK). This example assigns a source array that is distributed by rows, to a destination array that is distributed by columns. Because of the additional computation involved (compared to Case 1), as well as the small number of address-address blocks necessary to represent the address relation, caching proves to be a marginal win. Ultimately, it improves average latency by about 20%.

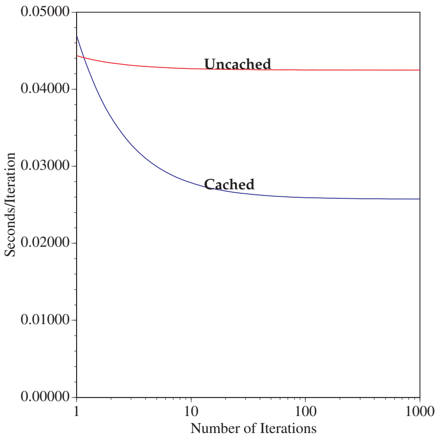
Case 3: S(:,CYCLIC(5)) $\rightarrow$ D(:,CYCLIC(20)). This example assigns a source array with a small block size of 5 to a destination array with a block size of 20. The library does not have an optimization for this case, and, in fact, the inner loop of the address computation only generates one tuple each time it is entered. Despite this, there is a fair amount of contiguity in the address relation, so there are few address–address



S(:,BLOCK) → D(:,BLOCK)
Case 1



S(BLOCK,:) → D(:,BLOCK)
Case 2



S(:,CYCLIC(5)) → D(:,CYCLIC(20))
Case 3

Figure 4 Effectiveness of caching for HPF on Paragon. Each graph plots the average end–to–end latency versus the number of iterations for cached and uncached message assembly.

blocks in the cache. As a result, the communication is ultimately a factor of two faster if message assembly is performed from the cache.

In general, as distributions become more irregular, address computation becomes more time expensive because it becomes increasingly difficult for the the communication library to provide special–case optimizations. In these cases, such as case 3 of Figure 4, caching can have a significant impact.

6 DISCUSSION

In this section, we comment on other issues in address relation caching. We begin by noting the benefits of decoupling address computation and message assembly in the communication system. Next, we discuss where to make the decision to cache an address computation, pointing out that compilers play a key role. Continuing, we comment on cache data structures and replacement policy. Finally, we point out that the cache provides an opportunity for optimizations.

6.1 Decoupling address computation and message assembly

By taking address computation out of the critical path of data transfer, address relation caching decouples address computation and message assembly. One advantage of this is that it makes it easier to achieve reasonable communication performance without a large number of special–case optimizations in the compiler and run–time library. Instead of optimizing each address computation algorithm for each architecture, only message assembly from cache needs to be optimized for each architecture in order for all data transfers to benefit.

Address relations are an ideal interface to the communication system because they are implementation neutral. This permits communication systems to be highly specialized for the available hardware while still presenting the same interface to the programmer or compiler. However, to best utilize the hardware, it may be necessary to change the representation of the address relation before using it. For example, the relation may be transformed into a DMA gather map on the sender to make best use of a network adapter such as the one described in [8]. Because transforming the relation may be expensive, it is desirable for the communication system to cache the (transformed) relation instead of the application.

6.2 Deciding to cache

Our analytical model and HPF results show that caching can sometimes be ineffective and that the number of times a cached address relation must be reused to break even depends on the complexity of address generation. Given this, where should the decision to cache or not be made? One option is to make this decision in the compiler. This is almost certainly the right place if the distributions and machine characteristics and the number of times the communication is performed are known at compile–time. If this is not the case, the decision can be made at run–time, where at least the distributions and machine characteristics are known. However, the number of repetitions may be data dependent and unknown until after the communication has been performed. To deal with this case, the library could keep track of the number of times a particular communication is performed uncached, and switch to a cached approach after some number of repetitions.

6.3 Cache data structures

There is a continuum of cache data structures that ranges from individually storing each tuple of the address relation in memory to simply performing the original address computation. The cache data structure should be chosen to minimize the time spent accessing the cache given memory constraints. The structure should be implementation dependent in order to make best use of available hardware.

6.4 Replacement policy

The cache replacement policy should make use of compile–time knowledge. The HPF library described above uses a strictly direct–mapped policy at run–time, but the cache entry tags are selected by the compiler, which knows the run–time policy. This means the compiler can assure that useful address relations remain in the cache. What to do when there is insufficient memory to hold even one address relation in cache is unclear.

6.5 Cache optimizations

The cache is an intermediate layer between address computation and message assembly. As such, it presents opportunities for optimization. For example, since the entire address relation is known to the cache, it could optimize it for memory locality on the sender, receiver, or both. As shown in [11], the choice is architecture–dependent. If the distributions, machine characteristics, and repetition count are known at compile–time, it is reasonable to perform address generation at this point, "pre-cache" the results, and always assemble from cache when running the program.

7 CONCLUSIONS

Address relation caching at the communication system level is an effective strategy for facilitating high bandwidth, low latency data transfer. Because address computation is removed from the critical path, its cost can not determine communication performance. Our results show that address relation caching is effective even when the cost of address computation is very low, even on the order of a few instructions. Further, the overhead of caching can be generally be quickly amortized if the communication is repeated. Although our analysis was in the context of deposit model communication, address relation caching is not limited this model.

ACKNOWLEDGEMENTS

We would like to thank Thomas Stricker for illuminating the finer points of deposit model communication, James Stichnoth for endless patience in explaining the details of performing HPF array assignment statements, and Ed Segall for helping to debug the HPF library.

REFERENCES

[1] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88*, pages 330–339, November 1988.

[2] High Performance Fortran Forum. High Performance Fortran language specification version 1.0 draft, January 1993.

[3] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology*, 2(3):16–26, 1994.

[4] *i860 Microprocessor Family Programmer's Reference Manual*. Intel Corporation, 1992.

[5] Intel Corp. *Paragon X/PS Product Overview*, March 1991.

[6] P. Pierce and G. Regnier. The paragon implementation of the nx message passing interface. In *Proc. Scalable High Performance Computing Conference*, pages 184–190, Knoxville, TN, May 1994. IEEE Computer Society Press.

[7] J. Saltz, , S. Petiton, H. Berryman, and A. Rifkin. Performance effects of irregular communication patterns on massively parallel multiprocessors. *Journal of Parallel and Distributed Computing*, 13:202–212, 1991.

[8] P. Steenkiste, B. Zill, H. Kung, S. Schlick, J. Hughes, B. Kowalski, and J. Mullaney. A host interface architecture for high-speed networks. In *Proceedings of the 4th IFIP Conference on High Performance Networks*, pages A3 1–16, Liege, Belgium, December 1992. IFIP, Elsevier.

[9] J. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[10] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 21(1):150–159, April 1994.

[11] T. Stricker and T. Gross. Optimizing memory system performance for communication in parallel computers. In *Proc. 22nd Intl. Symp. on Computer Architecture*, Portofino, Italy, June 1995. ACM/IEEE. to appear.

[12] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proc. Intl. Conf. on Supercomputing*, page accepted, Barcelona, July 1995. ACM.

[13] V. S. Sunderam. PVM : A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[14] D. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, April 1994.