The Impact of Address Relation Caching on the Performance of Deposit Model Message Passing *(extended abstract)*

Peter A. Dinda and David R. O'Hallaron
Carnegie Mellon University

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 [7], NX [2], and MPI [8] involves three basic steps on the sender: (1) *address generation*, 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* [5, 6], combines the address generation steps on the sender and the receiver into a single step on the sender. The result of the address generation step on the sender is an *address relation* that associates addresses on the sender with addresses on the receiver. The receiver's addresses are passed along with the data, and the receiver uses these addresses to deposit the data from the wire directly into memory.

If an address relation is reused (for example inside a loop), then there is an opportunity to improve performance by caching the relation. A similar approach is used by the CHAOS system [3] to improve the performance of irregular collective communications. In this paper, we evaluate the effectiveness of caching address relations in deposit model message passing systems. We develop a simple analytic model for predicting the performance tradeoffs between cached and uncached point–to–point communications, validate the model on both the Intel Paragon and Intel iWarp systems, and evaluate the effectiveness of caching for HPF programs running on Paragon. The main result is that caching becomes more effective as the distribution of the data becomes more irregular, and thus thus the address relations become harder to compute.

2 Models for point–to–point communication

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.[1]

2.1 A simple model for 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 a sequence of address-data pairs, where each data word in the message is paired with its target address on the receiver. The final expression for the uncached average end-to-end latency for a communication of n data words is

$$\begin{aligned} t_{uncached} \quad &= \quad n \left(\frac{n_g + n_{au} + n_d}{r_i} + \frac{1}{r_{rr}} + \frac{2}{r_{wc}} \right. \\ & \qquad \left. + \frac{2}{r_c} + \frac{2}{r_{rc}} + \frac{1}{r_{wr}} \right) \end{aligned} \tag{1}$$

where n_g is the number of instructions executed in address generation per data word, n_{au} is the number of loop overhead instructions per data word in message assembly, and n_d is the number of loop overhead instructions per data word in message disassembly. The other parameters are machine properties: r_{rc} and r_{wc} are the stride-1 memory read and write bandwidths of the machine in words/second, r_{rr} and r_{wr} are the read and write bandwidths of the machine given a random access pattern in words/second, r_i is the instruction execution rate of the machine in instructions/second, and r_c is the bandwidth of a communication link in words/second.

2.2 A simple model for cached communication

The time for an cached point-to-point communication has five components: t_g, the time to compute the address relation; t_o, the

[1] The derivations are omitted in the abstract but will appear in the final paper.

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 n_i, the number of times the communication is performed, and that t_g, t_c, and t_d are the same between the uncached and cached schemes. We continue the assumption that the message format is address-data pairs. Further, we choose the simplest possible cache data structure – an array of 2–tuples that represent the relation. The final expression for the cached average end–to–end latency for a communication of n data words after n_i iterations is

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

where n_g is the number of instructions executed in address generation per data word, n_o is the number of instructions executed in inserting a tuple of the address relation into the cache, n_{ac} is the number of loop overhead instructions per data word in message assembly using the cache, and n_d is the number of loop overhead instructions per data word in message disassembly. The other parameters are the machine properties described in Section 2.1.

2.3 Evaluating the effectiveness of caching

Equations 1 and 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 faster than without using the cache, i.e.,

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

or

$$r_{rc} > \frac{2r_i}{n_g + n_{au} - n_{ac}} \quad (3)$$

If 3 is true, then the number of times the cached information must be used in order to outperform the uncached implementation can be determined by setting $t_{cached} = t_{uncached}$ and solving for n_i. In doing this we find that the break even point for caching is

$$n_i = \left\lceil \frac{r_{rc}}{r_{wc}} \left(\frac{(n_g + n_o)r_{wc} + 2r_i}{(n_g + n_{au} - n_{ac})r_{rc} - 2r_i} \right) . \right\rceil \quad (4)$$

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

$$\frac{t_g + t_{au}}{t_{ac}} = \frac{r_{rc}r_{wc}r_{rr}(n_g + n_{au}) + r_i r_{rc}(r_{wc} + 2r_{rr})}{r_{rc}r_{wc}r_{rr}n_{ac} + r_i r_{rc}(r_{wc} + 2r_{rr})} \quad (5)$$

3 Model validation

In this section we validate the models from Section 2 for point–to–point communications on an iWarp system.[2] To test the predictive power of the models, we use (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 (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.

On the iWarp 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 algorithm and 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 shows the measured and predicted number of iterations needed 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.

n_g	Actual	Pred. (Static)	Pred. (Dynamic)
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

Figure 1: Iterations required for cached message assembly to break even with uncached message assembly on iWarp (Eqn. 4)

Figure 2 compares the actual and predicted speedup of cached over uncached message assembly. As in Figure 1, 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 2 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 5.)

[2] In the full paper we also give results for the Paragon and provide a more detailed description of how we estimate the various parameters.

n_g	Actual	Pred. (Static)	Pred. (Dynamic)
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

Figure 2: Speedup of cached message assembly over uncached message assembly on iWarp (5)

4 Impact of caching in task parallel HPF programs

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 [1] (a dialect of HPF that integrates task and data parallelism) and compiled and run on an Intel Paragon. Communication is performed 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 generation is performed using the algorithm in [4]. 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

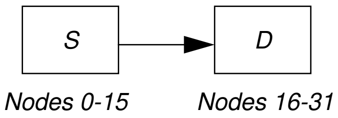

Figure 3: Experimental setup

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 generation. 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 illustrate this point quite clearly:

Case 1: S(:,BLOCK) $\longrightarrow$ D(:,BLOCK). In this case, the distributions of the source and destination arrays are identical. The address generation for a block–to–block assignment of this sort is a common case that is highly optimized in the library. Because address generation is very inexpensive, caching is not helpful in this case (see Figure 4.) In fact, although it cannot be easily seen in the figure, the average end–to–end latency for the cached version after a large number of iterations is actually 20% higher than that of the uncached version, which is the unamortized cost of cached message assembly.

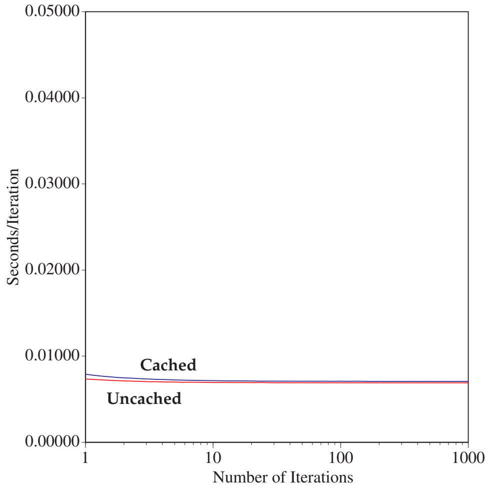

Figure 4: **Case 1:** S(:,BLOCK) $\longrightarrow$ D(:,BLOCK). Caching is ineffective on Paragon

Case 2: S(BLOCK,:) $\longrightarrow$ 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%, as shown in Figure 5.

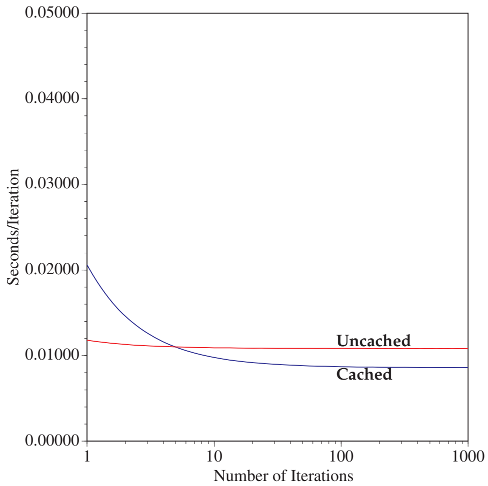

Figure 5: **Case 2:** S(BLOCK,:) $\longrightarrow$ D(:,BLOCK). Caching is marginally effective on Paragon.

Case 3: S(:,CYCLIC(5)) $\longrightarrow$ 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 blocks in the cache. As a result, the communication is ultimately a factor of two faster if message assembly is performed from the cache. This is shown in Figure 6.

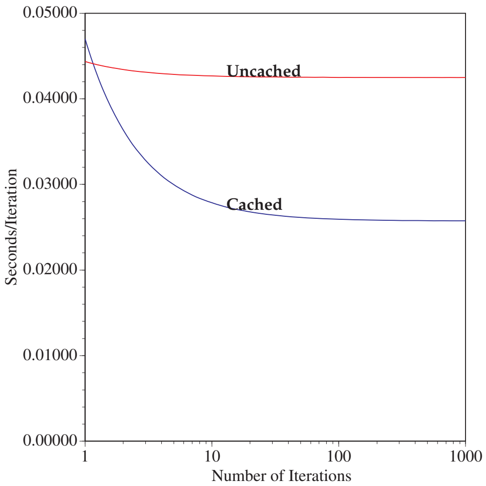

Figure 6: **Case 3:** S(:,CYCLIC(5)) $\rightarrow$ D(:,CYCLIC(20). Caching improves latency by a factor of 2 on Paragon.

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, as we see in Figure 6, caching can have a significant impact.

5 Discussion and conclusions

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.

The algorithm and distributions used during address generation are one representation of the address relation. Caching permits us to explore other representations, some of which may be more efficient to assemble from. If the distributions, machine characteristics, and repetition count are known at compile–time, it is reasonable to perform address generation at this point, "precache" the results, and always assemble from cache when running the program.

Caching provides a path for achieving reasonable communication performance without a large number of special–case optimizations in the compiler and run–time library. Keeping these modules simple could help shorten development time, simplify testing, reduce the number of bugs, simplify maintenance, and in general cut costs.

A final point is that address relation caching is not limited to deposit model message passing. In general communication, two address relations are computed (and could be cached) — on the sender there is a relation between sender data addresses and offsets into the message buffer while on the receiver the relation computed is from offsets into the message buffer and receiver data addresses.

References

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

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

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

[4] Stichnoth, J. Efficient compilation of array statements for private memory multicomputers. Tech. Rep. CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, Feb. 1993.

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

[6] Stricker, T., Stichnoth, J., O'Hallaron, D., Hinrichs, S., and Gross, T. Decoupling communication services for compiled parallel programs. Tech. Rep. CMU-CS-94-139, Carnegie Mellon University, School of Computer Science, 1994.

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

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