Communication and Computation", In *Proc. of the 19th International Symposium on Computer Architecture*, May 1992.

[Foster95] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[Gokhale95] M. Gokhale, B. Holmes, and K. Iobst, "Processing In Memory: the Terasys Massively Parallel PIM Array," *IEEE Computer*, April 1995, pp. 23-31.

[Hall96] M. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S. Liao, E. Bugnion and M.S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer* 29(12), Dec., 1996, pp. 84-89.

[Herlihy91] M. Herlihy, "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems* 13(1):124-129, 1991.

[IBMMot94] IBM Microelectronics and Motorola Inc., "The PowerPC Microprocessor Family: The Programming Environments," IBM Microelectronics Document MPRPPCFPE-01, Motorola Document MPCFPE/AD (9/94).

[Knowles91] S. Knowles, "Arithmetic Processor Design for the T9000 Transputer," Proc. SPIE, vol. 1566, 1991, pp 230-243.

[Kogge94] P.M. Kogge. "The EXECUBE Approach to Massively Parallel Processing," 1994 Int. Conf. on Parallel Processing, Chicago, IL, August, 1994.

[Kogge96] Kogge, Peter M., S. C. Bass, J. B. Brockman, D. Z. Chen, E, H. Sha, "Pursuing a Petaflop: Point designs for 100TF Computers Using PIM Technologies," 6th Symp. on Frontiers of Massively Parallel Computation, Annapolis, MD, Oct. 25-31, 1996.

[Kogge98] P.Kogge, J.B. Brockman, V. Freeh, "Processing-In-Memory Based Systems: Performance Evaluation Considerations", Workshop on Performance Analysis and its Impact on Design, held in conjunction with ISCA '98, May 1998.

[Mowry92] T. Mowry and M. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", In *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1992.

[Oskin98] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. "Active Pages: A Model of Computation for Intelligent Memory". In *Proc. of the 25th International Symposium on Computer Architecture (ISCA)*, June, 1998.

[Palmer86] Palmer, J. F., "The nCube Family of Parallel Supercomputers," Proc. IEEE Int. Conf. on Computer Design, 1986, p. 107.

[Patterson97] D. Patterson et al., "A Case for Intelligent DRAM: IRAM," IEEE Micro , April 1997.

[Rinard97] M. Rinard and P. Diniz, "Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers," *ACM Transactions on Programming Languages and Systems* 19(6), Nov. 1997, pp. 942-991.

[Rixner98] S. Rixner, W. Dally, U. Kapasi, B. Kailany, A. Lopez-Lagunas, P.R. Mattson and J.D. Owens. "A Bandwidth-Efficient Architecture for Media Processing," in Micro 31, 1998.

[Saulsbury95] A. Saulsbury, T. Wilkinson, J. Carter and A. Landin, "An Argument for Simple COMA", In *Proc. of the Symposium on High-Performance Computer Architecture*, 1995.

[Saulsbury96] A. Sauslbury, F. Pong and A. Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration, Proc. of the International Symposium on Computer Architecture, May, 1996.

[Shimizu96] Shimizu et al., "A Multimedia 32b RISC Microprocessor with 16Mb DRAM," In *International Solid State Circuit Conference*, Feb. 1996, pp. 216-17.

[Steele97] C. S. Steele, et al, "A Bus-Efficient Low-Latency Network Interface for the PDSS Multicomputer", *Proc. of the International Symposium on High-Performance Distributed Computing*, August 1997, pp. 213-22.

[Sunaga96] T. Sunaga, P.M. Kogge, et al, "A Processor In Memory Chip for Massively Parallel Embedded Applications," IEEE J. of Solid State Circuits, Oct. 1996, pp. 1556-1559.

[Wolfe89] M. J. Wolfe, "More Iteration Space Tiling", In *Proc. of the IEEE Supercomputing Conference*, Nov. 1989.

data set size.

8.0 Conclusions and Future Work

This paper has described the DIVA system, an architecture incorporating PIM devices as smart memories to one or more external host processors. Other distinguishing features of DIVA include its PIM-to-PIM interconnect and explicit support for in-memory operations on irregular data structures. In this paper, we presented system-level requirements for in-memory acceleration of irregular applications. We presented three case studies, sparse conjugate gradient, natural join and an OO7 database query, to demonstrate how irregular applications can be mapped to the DIVA architecture. High-level simulation results show a speedup for all three applications, resulting from increased processor-memory bandwidth, much more effective use of cache on the host processor, lower latency accesses and parallelism.

Future descriptions of the DIVA project will include details of the PIM VLSI device, architecture studies using a high-fidelity system simulator based on RSIM, the DIVA compiler and run-time systems, and further application studies.

Acknowledgments

The authors wish to thank Thomas Sterling for his early contributions to this project. The DIVA project is sponsored by DARPA contract F30602-98-2-0180.

References

[Anderson93] J. Anderson and M. Lam, “Global Optimizations for Parallelism and Locality on Scalable Parallel Machines“, In Proc. of the ACM Conference on Programing Language Design and Implementation, (PLDI’93), ACM Press, New York, June 1993.

[Birnbaum99] M. Birnbaum,and H. Sachs, “How VSIA Answers the SOC Dilemma,” IEEE Computer, June, 1999, pp. 42-50.

[Brockman99] J. Brockman, P. Kogge, V. Freeh, S. Kuntz, T. Sterling. “Microservers: A New Memory Semantics for Massively Parallel Computing”, In *Proc. of the ACM International Conference on Supercomputing*, June, 1999, pp. 454-463.

[Bik97] A.J.C. Bik and H.A.G. Wijshoff, “Simple Qualitative Experiments with a Sparse Compiler”. In *Proc.s of the 9th International Workshop on Languages and Compilers for Parallel Computing*, 1996. Appeared in Lecture Notes in Computer Science, No. 1239, pages 466--480, 1997.

[Blume96] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, “Parallel Programming with Polaris, *IEEE Computer* 29(12), Dec. 1996, pp. 78-83.

[Burger96] D. Burger, J. Goodman and A. Kagi. “Memory Bandwidth Limitations of Future Microprocessors,” In *Proc. of the 23rd International Symposium on Computer Architecture (ISCA)*, May, 1996.

[Burger97] Doug Burger, Stefanos Kaxiras, and James R. Goodman, “DataScalar Architectures”, In *Proc. of the 19th International Symposium on Computer Architecture (ISCA)*, June, 1997.

[Carslile95] M. Carlisle, A. Roges and L. Hendren, “Early Experiences with Olden”, In Proc. of the ACM Conference on Principles and Practice of Parallel Processin

[Carter99] J.B. Carter et. al, “Impulse: Building a Smarter Memory Controller”, Fifth Int’l Symposium on High Performance Computer Architecture, pp. 70-79, January 1999.

[Cmelik94] R. Cmelik and D. Keppel. "Shade: A Fast Instruction Set Simulator for Execution Profiling", In Proc. of the ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems, June, 1994.

[Dally92] Dally, W. J., et al, “The Message Driven Processor: A Multicomputer Processing Node with Efficient Mechanism,” IEEE Micro, April 1992, pp. 23-38.

[Draper96] J. Draper, “The Red Rover Algorithm for Deadlock-Free Routing on Bidirectional Rings”, In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, August 1996, pp. 345-54.

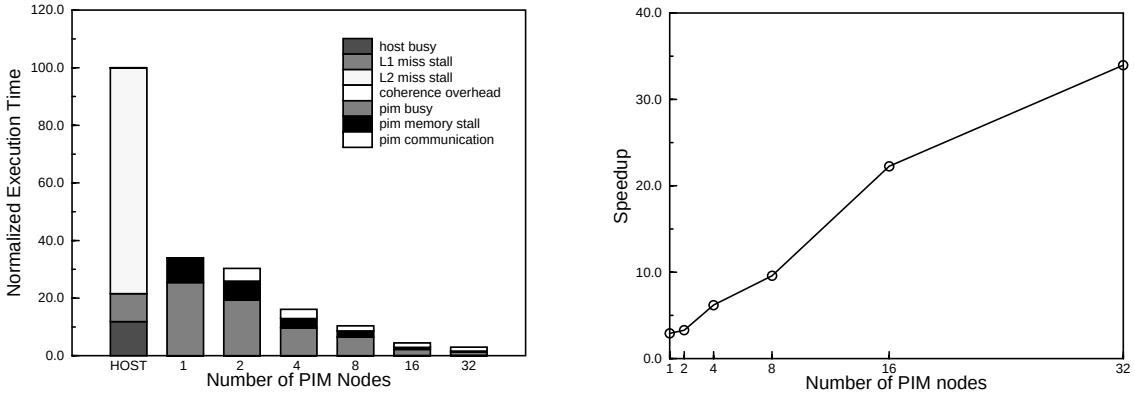
[vonEicken92] T. von Eicken, D. Culler, S. C. Goldstein,and K. Schauser, “Active Messages: a Mechanism for Integrated

defined hierarchically in terms of other base or complex assemblies or base assemblies. Base assembly components are defined in terms of composite parts which in turn consist of more than one library atomic part. Each of these objects have specific attributes such as a unique identifier, creation date and other type-specific fields.

This database application was originally developed at the University of Wisconsin to study the performance of various database management systems [007]. We have ported this application to a C++ stand-alone program by implementing the dictionary and relations abstraction using hash-tables and linked lists in a total of 9,000 lines of C++ code. Our performance evaluation concentrates on a specific database query, query #6. Query #6 finds all assemblies (base or complex) B that reference (directly or transitively) a composite part with a more recent build date than B's build date. This query is implemented using set operations over the database relations and extensively uses the iteration abstraction from C++ to access successive objects in a given relation.

Besides the overall organization of the database objects in a graph data structure, the database schema also relies heavily on singly-linked and hash-table pointer-based data structures for indexing of the object in each category (documents, manual, base assemblies, etc.). The primary access pattern over the indexing structure traverses a singly-linked list or a hash-table, searching for a particular subset of objects matching a given predicate. In addition, the application also traverses the overall graph structure of the objects in the database. Such traversals perform poorly on conventional systems because they exhibit almost no temporal reuse of memory accesses, and there is little spatial locality due to the way the pointer-based data structures are created.

To take advantage of the PIM architecture, we perform two key transformations on the original application. The first transformation takes advantage of the fact that the computation accesses a set of objects; the order in which the elements of the set are accessed by the application is irrelevant, so these accesses can be performed in parallel. The second transformation restructures the code so that the PIM nodes traverse the linked data structure that represents the relations in the schema and selects the set of objects the computation needs to access. Each PIM selects a subset of the objects in the relation from its local memory only. The host then gathers the partial results and constructs a larger set. The host is responsible for any updates to the storage. Figure 9 shows the execution time breakdown and speedups for 007.



(a) Execution breakdown

(b) Speedup

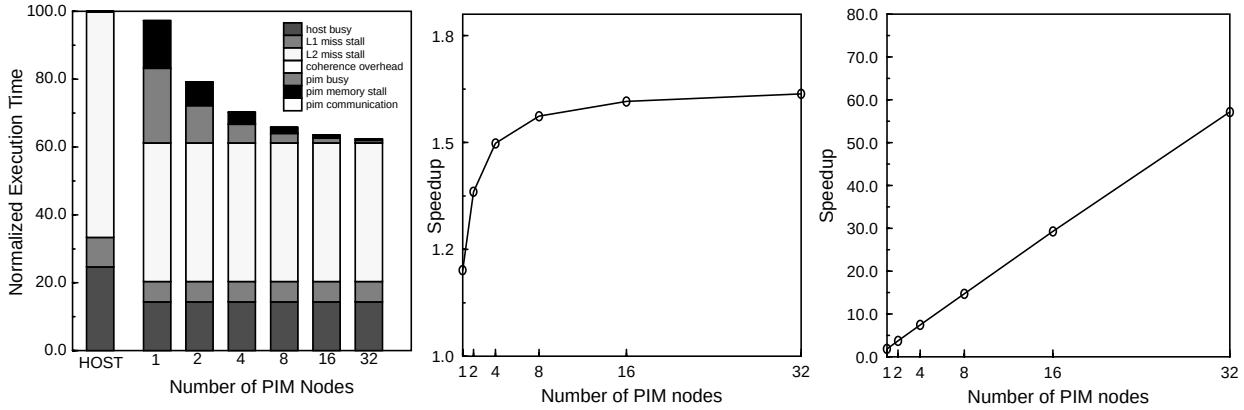
Figure 9: Execution Breakdown and Speedups for 007.

The results show an impressive superlinear speedup. As the execution breakdown reveals, this result is due to the severe performance impact of the L2 miss stalls (almost 80% of the sequential computation for this query) for the case where only the host executes the computation. When the computation is partitioned across the PIM nodes, each PIM fetches data from its local memory and communicates very infrequently. The overhead of coherence is also negligible for all runs, as the query does not update the objects in the database but rather collects overall statistics. As a result, the performance scales well up to 16. For 32 PIM nodes, speedup, while still impressive, trails off a little due to the relative frequency of communication compared to computation for this

cuted on the PIMs, much fewer accesses to array Y are brought into the host, and the miss rates on L1 and L2 cache were reduced respectively to 10% and 7%. As Figure 7(a) shows, this contributes to a significant reduction of the application time waiting for results from memory. Figure 7(b) shows the overall application speedups for different numbers of PIM nodes as compared to the entire application executing on the host. At 16 PIMs, the application speedup is more than 8 over the original sequential execution time. While this application scales very well for up to 16 PIM nodes, the problem size we use is too small relative to the overhead of the reduction computation to scale much beyond 32 PIMs.

7.2 Hash-Based Natural Join

The Natural Join is a fundamental operation in relational database systems. It consists of generating all possible combinations of tuples for two relations R and S with a common attribute A. In the implementation used in these experiments, the algorithm builds a hash table for each of the relations R and S indexed by the attribute A. Then, for each hashed value in the table, the algorithm joins all tuples of the two relations that have a common value for the attribute A.



(a) Execution breakdown (b) Overall Speedup (c) Speedup for Join phase

Figure 8: Execution Breakdown and Speedups for Natural Join.

The strategy to map this application to DIVA is to distribute the hash table along contiguous blocks of the table entries. Each PIM node has a set of consecutive entries of the hash table and the hash-table collision lists corresponding to each of the table entries it owns. Once the host processor has constructed the distributed hash table, the natural join operation proceeds by having each PIM node computing a local natural join operation. At the end, the host simply scans the partial hash tables local to each PIM node to read the results.

Figure 8 shows performance results when the first phase, constructing the local hash tables, is performed by the host, and the second phase, the join of local hash tables, executes in the PIMs. The speedups for the join phase of the computation are superlinear, as shown in Figure 8(c). These superlinear speedups result from the combined effects of the smaller memory latencies at the PIMs, as compared to the cache miss latencies suffered by the host, and the parallelism obtained by distributing the computation across PIMs. Due to Amdahl's Law, the overall speedup is limited, as the first phase, which accounts for about half the baseline execution time, is executed sequentially on the host. Even more speedup is possible from two sources, both of which we are exploring: (1) building portions of the local hash table in parallel on the PIMs and merging the results; and, (2) performing in parallel on the ASAP unit the comparison of a key from the R-tuple with that of several S-tuples with the same hash value.

7.3 Object-Oriented Database Benchmark (007)

The 007 application implements a representative object-oriented database for CAD applications. The database schema defines several one-to-one and one-to-many relationships among database objects. These objects consist of documents, manuals and base or complex assembly components. Each complex assembly component is

Figure 5(b) and Figure 5(c) present the corresponding code for the DIVA architecture, which makes use of the parcel and synchronization primitives to orchestrate the computation. Figure 6 illustrates graphically the data mapping for the various arrays in this computation for a system with 4 PIM nodes.

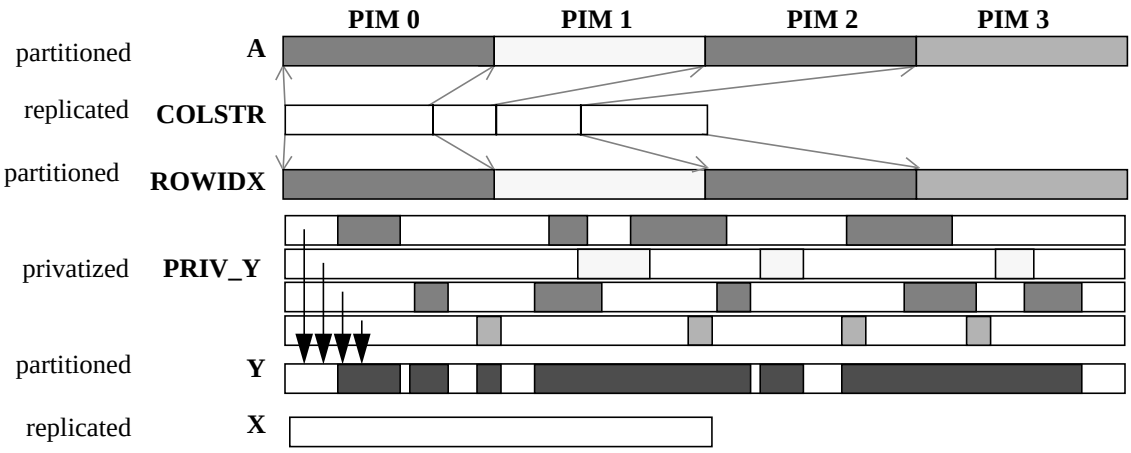
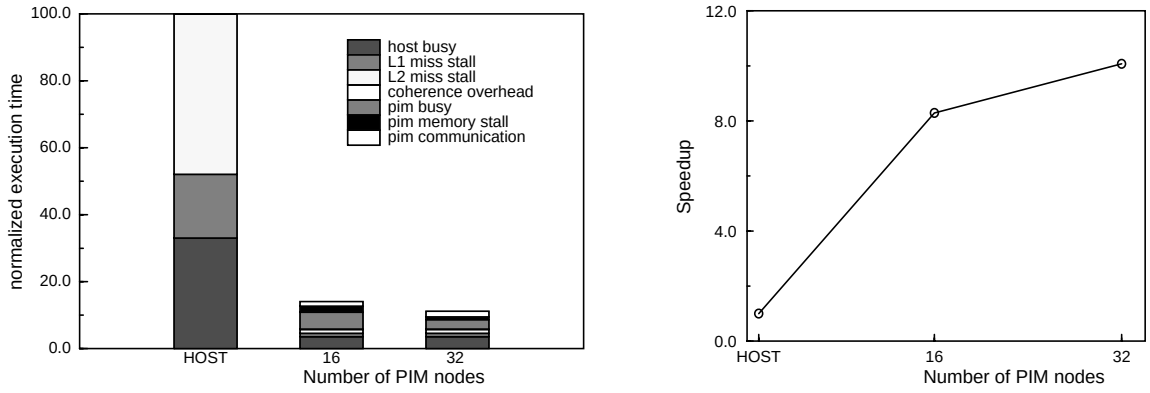


Figure 6: Data Mapping on DIVA for CG.

Figure 7 illustrates simulation results for this application. We separate original sequential execution into several components in Figure 7(a). The host busy category accounts for the time spent executing instructions. The L1 and L2 miss stall categories represent time spent waiting for memory accesses to be satisfied from either the L2 cache or main memory. In the version of the program that executes the matrix-vector product on the PIMs, we show time spent in the host and on average in one PIM, and we include additional categories (the host is idle during PIM execution, so this is an accurate reflection of overall execution time). The coherency overhead refers to time spent by the host flushing cache lines prior to execution on the PIMs. Note that additonal coherency overhead is charged as L1 and L2 cache misses in the host when PIMs are used; by flushing data from the cache prior to PIM computations, extra cache misses in the host may occur in later host computation. This cache miss effect due to flushing is not significant in the programs presented here because the irregular accesses in the PIM computations were polluting the host cache when executed on the host. Additional categories show time spent in the PIMs, including PIM-to-PIM communication overhead and time spent in local memory stalls.



(a) Execution Breakdown

(b) Speedup

Figure 7: Execution Breakdown and Speedups for CG.

As the results in Figure 7(a) indicate, the original application suffers significantly from poor cache locality with overall L1 and L2 cache miss rates of 15% and 20%, respectively. When the matrix-vector product is exe-

present results on three applications evaluated with this simulation methodology.

7.1 NAS Sparse Conjugate Gradient (CG)

CG implements a linear system solver using a conjugate gradient iterative method. Its main data structures are three very large arrays of floating-point double-precision values. The main computation consists of a sparse matrix-vector product (see Figure 5(a)) and accounts for about 80% of the total sequential execution time. The computation is structured as a single loop performing commutative and associative updates to array Y indexed by the values in the ROWIDX array. The sparseness of the computation is derived from the indirection of the accesses to the Y array whereas both arrays A and X are accessed using simple loop indexing functions.

To effectively map this computation to DIVA, we parallelize the execution of the sparse matrix-vector product by exploiting the commutativity and associativity of the addition operations. In this version, each PIM node has a local copy of array Y (named PRIV_Y), and performs its updates on its own private copy; after all PIMs complete their local computation, the local results are merged using a parallel reduction algorithm. The parallel reduction algorithm ensures that there is no network contention during the communication phase. However, since each PIM node has to communicate its copy of array Y to other nodes, the total amount of communication increases with the number of PIMs, as well as the number of steps of the parallel reduction. This example makes use of a lightweight run-time system and the parcel communication mechanism to generate and manage concurrency. The basic code generation strategy is for the compiler to split the computation between the host and the PIM nodes and to initiate the computation on the PIMs by sending parcel, using the *SendParcel* primitive. PIM nodes are activated by the receipt of a given parcel and proceed to execute the code associated with it. This code might in turn generate other concurrent computation on the same or on other PIM nodes. The host can enforce termination of a given computation using an explicit barrier synchronization construct (*Barrier*) or implicitly through memory. Also included in this run-time system is a *Flush* primitive that allows the compiler to maintain the consistency of the data between the host caches and the PIM nodes.

(a) Original Loop Nest.

```
DO J = 1, N
  DO K = COLSTR[J], COLSTR[J+1]-1
        Y[ROWIDX[K]] = Y[ROWIDX[K]] + A[K] * X[J]
```

(b) DIVA Host Program.

```
Flush(Y);
PartitionSize = Sizeof (ROWIDX) / NumPimNodes;
for (i=0; i<NUM_PIMNODES; i++) {
   Send_parcel (ROWIDX[I*PartitionSize], LoopBody, PartitionSize,
              A[I*PartitionSize], PRIV_COLSTR[0,I],PRIV_X[0,I],Y);
}
Barrier();
```

(c) Code for PIM node command *LoopBody*.

```
BarrierEnter();
for (j=1; j<=N; j++) {
   Lower = Max(PRIV_COLSTR[J], PIMID*PartitionSize);
   Upper = Min(PRIV_COLSTR[J+1]-1, (PIMID+1)*PartitionSize-1);
   for (i=Lower; i<=Upper; i++) {
        K1 = K - PIMID*PartitionSize;
        PRIV_Y[ROWIDX[K1]] = PRIV_Y[ROWIDX[K1]] + (A[K1] * PRIV_X[J])
  }
}
ParallelReduction(Y,PRIV_Y,PIMID,NUM_PIMNODES);
BarrierRelease();
```

Figure 5: CG Matrix-vector product and its mapping to DIVA.

using Shade [Cmelik94]. Shade executes application programs and generates traces under the control of a user-supplied trace analyzer. We simulate parallel execution in our experiments by recording the simulated time at the beginning of a parallel section and setting the parallel execution time at the end of the concurrent execution to be the maximum value of the simulated time by each of the participating PIM nodes. The Shade-based simulator does not directly model the PiRC interconnection. To account for network latency and congestion, we generate traces of time-stamped network requests for each application, and use these traces as inputs to a network simulator [Draper96]. The throughput and contention derived by the network simulator are then used as parameters to the Shade simulator.

The PIM chips modeled in these experiments are much simpler than what was presented in Section 2. There is a single node per chip, and we only consider applications that use standard scalar integer and floating-point processing on the PIM nodes (*i.e.*, no ASAP instructions). These simplifications reduce the contention for on-chip resources, and allow us to get meaningful early results from the simple Shade-based simulation strategy. We anticipate that the multiple processing nodes per PIM chip and the ASAP functional units planned for the actual DIVA implementation will yield much better on-chip computation rates, albeit with additional costs due to contention for internal memory banks and PiRC channels.

In our simulations, each PIM node consists of a PIM processor, a 2M-byte memory bank, a host interface and a PiRC network interface. Since processor technology is optimized for speed and DRAM technology is optimized for density and yield, the PIM processing logic is expected to be slower than the host processor logic. Based on projections, we assume that the PIM processor cycle is twice the host processor cycle. The PIM node memory bank is organized as 8192 2K-bit memory rows, and the DRAM interface provides a 256-bit sub row per memory access. We assume the first access to a 2K-bit row (random-mode access) takes 2 PIM cycles, and each subsequent access to the same row (page-mode access) takes 1 PIM cycle. These parameters are based on current memory speeds [Kogge98].

The host has separate instruction and data on-chip caches, and a unified off-chip second level cache. We model a parcel issue as a sequence of writes to specific memory addresses, the last of which triggers the delivery of the parcel. Coherence between the caches and memory is enforced by software (e.g., the compiler), using an instruction to flush data from the cache. At a flush instruction, the simulator invalidates the cache line and, if the line is modified, writes it back to memory. We summarize the simulation parameters in Table 1. We now

	Cache Parameter	**Instruction L1**	**Data L1**	**Data L2**
Host Caches	***size***	32 K bytes	32 K bytes	1 M bytes
	associativity	2	2	2
	line size	64 bytes	32 bytes	32 bytes
	replacement	LRU	LRU	LRU
	write policy	write back	write back	write back
	latency (hit)	1 cycle	1 cycle	10 cycles
	latency (miss)	10 cycles	10 cycles	100 cycles
PIM Node	***processor cycle***	2 cycles[a]		
	memory size	2 M bytes		
	memory row size	256 bits		
	memory latency	1 cycle*(page mode), 4 cycles* (random mode)		
PiRC Network	***channel width***	32 bits		
	network cycle	4 cycles*		

a. Host processor cycles

Table 1: Simulation Parameters used in Application Studies.

need to know its exact location. Coherence of data shared across PIM chips is not guaranteed by the hardware and must be managed by either the compiler or programmer, similar to what is required in the Cray T3E. Also as with distributed-shared-memory multiprocessors, locality of data accesses is very important to good performance.

Because of these similarities to a distributed-shared-memory multiprocessor, most parallelizing and locality-management compiler techniques and parallel programming paradigms can be leveraged for DIVA. Applicable compilation techniques include automatic parallelization for both regular [Blume96] [Hall96] and irregular applications [Rinard97], and data and computation co-location [Anderson93]. Explicitly parallel programming languages that permit some programmer control of locality are also applicable, such as High Performance Fortran and its extensions for irregular applications, Olden [Carlisle95], and CC++[Foster95]. As discussed in Section 4.1, the parcel mechanism is really a refinement, tailored to the DIVA architecture, of active messages, which were developed for message-passing multiprocessor systems [vonEicken92].

While there are many similarities between programming for DIVA and parallel programming, there are several important differences. One additional requirement is keeping the host cache coherent with the PIM memories. As discussed in Section 5.4, this is accomplished with explicit flushing, immediately prior to sending a parcel from the host, of objects in the host cache that may be touched by the PIM computation. In keeping with the above stated goal of making correct programs easy to develop, the required flushing can be optionally automated by the compiler through analysis of the object and arguments associated with the parcel. Further, DIVA applications can exploit fine-grain parallelism using the ASAP functional unit for operations on aggregate data objects, which demands a combination of compiler technology and a user development environment for exploiting complex ASAP-oriented computations (*e.g.*, string matching). Other high-level operations such as memory management can be optimized for the PIMs to improve the locality of pointer-based computations. As an example, when building a tree data structure in parallel, each PIM can locally allocate a subtree, with the host sequentially connecting the subtrees in the upper level of the trees. Locality for each subtree is then ensured.

An important component of the DIVA project is a large software effort to develop application programmer libraries, and compiler and run-time system support. The DIVA compiler, either automatically or in response to programmer specification, partitions computation and data across host and PIMs. This partitioning requires that it must generate code that interfaces with the operating system to control data placement on the PIMs, generate code to load application-specific PIM code onto the memories, and also generate parcels in the appropriate places in the code to initiate PIM computation, communicate and synchronize. This high-level code must then pass through separate backend compilers: one for the host, for which we can use an existing native backend compiler; and one for the PIMs, which requires a DIVA PIM-specific backend that generates standard RISC as well as ASAP instructions. There are also separate run-time systems for the host and PIMs. The host run-time system performs similar functions to a standard architecture-independent parallel run-time library (e.g., Pthreads), managing threads and synchronization. The PIM run-time system is a small, DIVA-specific system, primarily for parcel processing.

As part of the software development efforts, we are currently retargeting the Stanford SUIF compiler system to DIVA, allowing us to take advantage of its wealth of compiler analyses for distributed-shared-memory machines. In addition, we are developing an extensible approach to support compiler and programmer generation of ASAP instructions that are seamlessly integrated into the PIM backend. Since DIVA is targeting irregular computations, we are also investigating a memory management library for dynamic generation and reorganization of irregular data structures.

7.0 Case Studies

To derive preliminary performance estimates for complete applications, we developed a simulator for the major system components of DIVA. The simulated architecture consists of a host processor, and a number of PIMs interconnected via a PiRC ring network. We simulate computations executing on the host and PIMs

tent model of memory access must be chosen and maintained. Conventional NUMA and COMA models are suboptimal for irregular, data-intensive applications. Specifically, in a NUMA or COMA model, a reference to remote data by a local node causes the remote data to be automatically moved or copied to the local node, where it is made available under the same virtual address as the remote version. In general, the overhead of supporting this model becomes excessive for irregular applications, where there is by definition great potential for false sharing, and little temporal locality.

The philosophy in the DIVA system is therefore to move the computation to the data, rather than move the data to the computation. At any one time, the data at a virtual address is located on exactly one PIM node, and there are no cached copies on other PIM nodes. Global pages can be moved from one node to another for load balancing purposes, but this is a heavyweight operation that should be used infrequently and explicitly managed by the operating system. Consistency of the distributed address translation table must be maintained, but since this changes relatively rarely, software coherence methods are adequate.

During normal operation, therefore, data coherence issues do not arise *between* PIMs, and there is no need for a sophisticated, hardware-supported coherence mechanism. The movement of code is a much simpler problem, since code is read-only, and can be replicated easily. Moreover, the only references to code that get passed in parcels are indirect references that index into a method table, so the translation mechanism for code references is built into the application. The result is a memory model that can be supported by fairly simple hardware in the PIM nodes, independent of the host CPU details.

The remaining coherence issue, namely between the PIM system and the host, is the most difficult. Individual cache lines may be cached by the host processor(s). The simplest solution, adopted in this prototype, is to always explicitly flush PIM-accessible data, or keep it uncached. A more transparent approach is for each PIM to track ownership of individual cache lines, and request writebacks from the CPU caches as necessary. The hardware for this on each PIM is not excessive and scales well, so this is a suitable long-term solution. However, broader issues suggest it is premature to implement in our prototype. As stated at the beginning of this section, our goal is a memory model that is independent of which processor is used as a host; the mechanism for requesting writebacks is processor specific, and usually involves the requestor driving the address bus. In a large system with many potential requestors, this introduces significant arbitration, electrical drive, and portability problems. In the long term, it would be better to develop a standard (probably network-based) memory-to-processor channel for this activity, which would find other uses in smart memory systems.

Although the explicit flushing is a burden, either to the programmer or compiler, it is not expected to degrade performance significantly. In practice, even with automated hardware, the user would probably obtain higher performance in some applications by manually flushing cache lines anyhow, to minimize the number of write-back requests.

6.0 Developing Applications in DIVA

The success of a new architecture is highly dependent on the ease in which software can be developed for it. It should be straightforward to develop correct programs, even if it is somewhat more difficult to effectively exploit the performance-enhancing features of the architecture. DIVA offers a smooth migration path for developing applications. First, the applications programmer can begin with a standard sequential program, which will run correctly with no modification by using the PIMs as standard memory. Then, either the compiler or programmer can exploit the PIMs as smart memory in portions of the application where this is deemed profitable, gradually migrating the original sequential application to make full use of the DIVA architecture.

To the applications programmer or compiler, the abstract DIVA architecture appears very similar to a distributed-shared-memory multiprocessor. The host can serve as a master to coordinate activities on the PIMs. Each node on a PIM processor acts as a worker processor waiting for work, and possibly initiating work on other PIMs through the parcel mechanism. The memory associated with a PIM node can be thought of as its local memory. The PIM node can access a datum on other memory chips through a global address space without

supported page size of the host CPU. Also, each PIM node should ideally be able to hold in a fast TLB the translation information for all active global pages resident on it, to avoid the frequent TLB misses that would occur on an irregular application. This therefore suggests that global memory pages should be large; in the prototype, one simplifying option under consideration is a single very large global page (per application) on each node.

Parcel Buffers: The final step in invoking the PIMs is to request the host OS to allocate and map one or more virtual parcel buffers, for use in communicating parcels with the PIM system. Parcels are then sent to individual nodes to start the PIM computation. Finally, when the computation is complete, one of the PIM methods communicates this to the host, typically by setting a flag in the global memory, and the host picks up the results from the global memory.

The overall memory structure for a typical DIVA application is shown in Figure 4. In the far left column is the

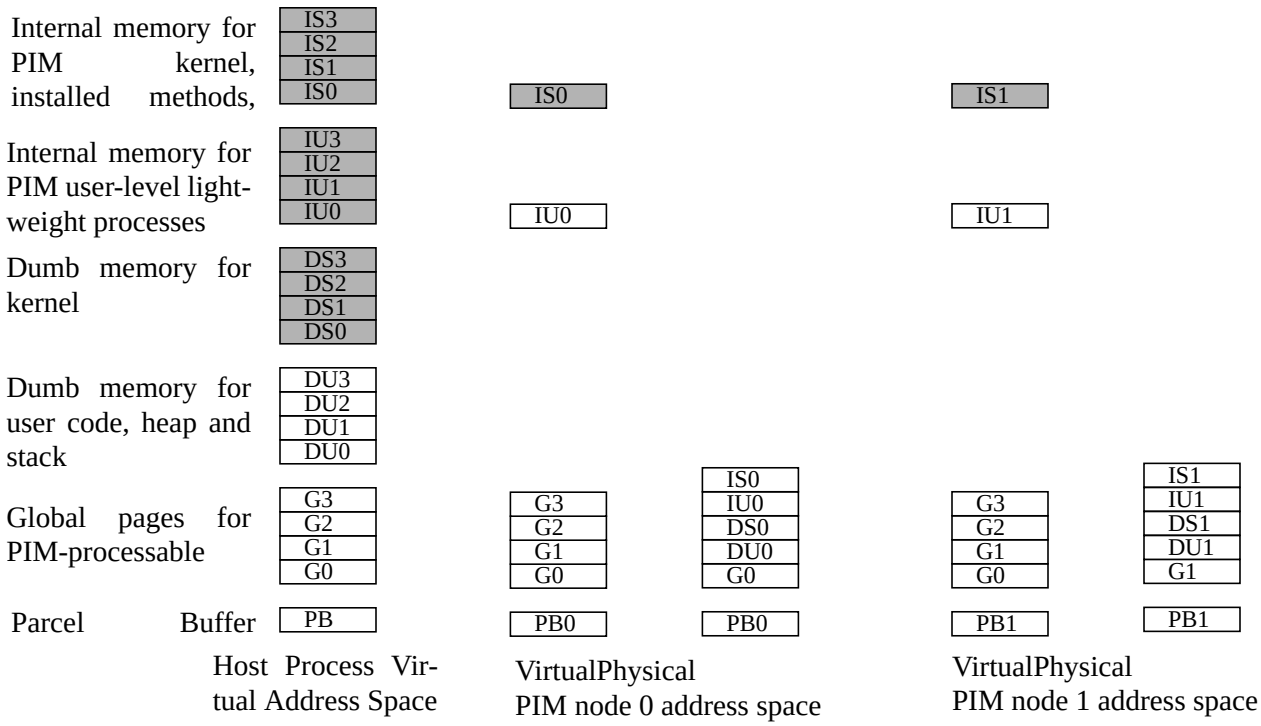


Figure 4: Decomposition of a host process address space across multiple PIM nodes. Shaded regions have supervisor-level protection.

virtual address space of a typical host application process, where each rectangle represents a page or segment from one of the memory regions. Shaded pages are accessible only while in supervisor mode. The label indicates whether the page is used for global data (G), dumb user or system pages (DU or DS) or internal user or system pages (IU or IS), as well as the PIM node (0-3) where the page currently resides. The second column shows the subset of pages visible to a method executing on PIM node 0. The third column shows the subset of pages actually resident on node 0. The last two columns show the same information for node 1. Note that global pages are visible from all nodes, while internal pages are visible only on their local nodes, where they appear at a common virtual address.

5.4 Coherence Management

In any system with distributed processing, the distributed information needs to be kept coherent, and a consis-

during application start-up, following an explicit system call, or transparently when the first PIM operation is attempted; some combination of these initialization steps can be profitably supported. For instance, a basic PIM kernel could be installed on each node at system boot time, as could code for any widely useful PIM methods. Application load time is a good time to install application-specific method code that is used frequently; individual methods from a large system library could be loaded dynamically on demand during application execution.

User-level code on the host never accesses this internal memory during normal operation. To the host, the internal pages appear within the supervisor region, like the kernel and its associated data structures. Moreover, the host only needs to access them under exceptional conditions, *e.g.*, application loads, service requests, and errors. Access from the host is thus guaranteed to be infrequent, through trusted code with access to translation tables. On the other hand, access to internal regions by the PIM processor needs to be highly efficient and well protected, since it is used for everything from local OS code and data to execution stacks and working memory for the many light-weight user-level methods launched in response to parcels during normal operation.

One can exploit these asymmetric requirements by adopting a memory-management approach for the internal memory that is very convenient for the PIM processor, but perhaps quite unrelated to the memory-management hardware on the host. A particularly useful scheme, planned for the prototype, is to give each lightweight local context on a PIM processor eight variable-sized segments or pages of internal memory, each defined by virtual and physical base addresses, size and access permissions. By convention, these are assigned to the following:

1. Supervisor-level kernel code (shared by all contexts on the node)
2. Supervisor-level kernel data and stack (shared by all contexts on the node)
3. User-level code (shared by all contexts in the same application)
4. User-level data (shared by all contexts in the same application)
5. User stack (unique to each context)
6. Miscellaneous (possibly unique to each context)
7. Supervisor-level parcel buffer device (shared by all contexts on the node)
8. User-level parcel buffer device (shared by all contexts in the same application).

Translation of internal virtual addresses can be made extremely fast and efficient by adopting some simple conventions, e.g., high bits of all the page virtual starting addresses are the same, the next three bits specify the page number, and the size is a power of two. Then, the TLB simplifies to a look-up table, the translation information for a lightweight context fits into 256 bits, and can be switched in one clock cycle. Since PIM nodes do not access each other's internal memory, the same virtual address range can be used for internal memory on every node, making PIM contexts relocatable from one node to another.

Global Memory: The next step in setting up to use the PIM features is to allocate DRAM on each PIM for use as smart "global" storage. This can be done at run time by a series of system calls such as **`mem_alloc(pim_node, virtual_address, size)`**, which allocates a region of memory of **`size`** bytes on **`pim_node`** and maps it at **`virtual_address`**, an unmapped virtual address range within the application address space. Unlike the dumb memory, whose mapping is visible only to the host process, or the internal memory, whose mapping is visible only to the associated PIM node and the host OS, the global memory is visible to the host process and to all PIM nodes involved in the application. Although only the host and the local node where the data resides can access an element of global memory directly (i.e., by read and write instructions), pointers to global objects are meaningful to all nodes and to the host, and can be communicated freely within parcels. Once global memory has been allocated, the host process can set up any initialized data by writing to it. In practice, global memory will make up the majority of memory in a data-intensive application using the PIM features. It is important that access to this memory be efficient from both the PIM and the host, and this therefore presents the greatest implementation challenge. Address translation must be compatible with both host CPU and PIM hardware. The page size must therefore be equal to or a multiple of the hardware-

communicate a parcel, a process reads or writes fields in the buffer, then performs a final read on a status field to pass the parcel to the PIM chip internals. In the rare case of corrupted accesses, a failure status is returned, and the application can retry.

5.2 Address Translation

Parcels, application code and data contain virtual addresses. For PIM processors to interpret these, they must have access to translation information, or there must be some fixed relationship between virtual and physical addresses. The latter option is simpler to implement, but was determined to be too restrictive. Each PIM thus contains translation hardware, and tables managed by the host. Any virtual page can reside on any PIM. However, the hardware is simplified by the characteristics of the system. For instance, for performance, a PIM needs to be able to rapidly determine if an address is local to its own memory bank, and find the physical address if it is. However, if the address is not local and communication is required, the additional cost of the non-local translation is negligible.

Each PIM therefore maintains translations for those virtual pages currently residing on it, plus part of a global, distributed table (similar to a home node concept as presented in [Saulsbury95]). Non-local translations are obtained by querying the distributed table, or, equivalently, submitting the virtual address in a parcel, for forwarding to the PIM where it resides. Advantages of this approach are that the translation tables on each PIM scale well; every address can be accessed in at most two parcel transmissions, and the application can optionally maintain location hints and use them to reduce this to a single parcel transmission in performance-critical cases.

5.3 PIM Memory Organization

The DRAM in the PIM subsystem is the primary storage for the DIVA system, and can be treated physically as a uniform, undifferentiated RAM. However, during operation the system uses the memory in three distinct ways, making it helpful to organize the memory on each PIM node logically into three regions according to whether it is used primarily by the host processor, primarily by the PIM processor, or significantly by both. These regions may be either physically contiguous or interspersed, and memory allocation within these regions can either be initiated by explicit system calls in the application, or undertaken at load time for all applications by the loader or start-up code. A flexible combination of static and dynamic allocation is usually most convenient for the user, but for this discussion assume explicit system calls are used.

An advantage of making this distinction is that different, optimized memory-management hardware can be used on each of the regions. As modern processor architectures demonstrate[IBMMot94], there is no conceptual problem with having multiple translation mechanisms in place, as long as they provide consistent virtual-to-physical mappings and access permissions.

Dumb Memory: Initially, the application is a normal (say Unix) process on the host. The various regions of its virtual address space (typically the user code, heap and stack and one or more kernel segments) are mapped as usual to some set of pages in DRAM, with some possibly paged out to disk. If the system memory contains both ordinary DRAM and PIM DRAM, these normal pages can be mapped into the ordinary DRAM, since they are never directly accessed by PIM processors. If all memory is PIM memory, the system can simply note that these pages are only accessed by the host, and that they need not appear in PIM-processor translation tables. A major use of dumb memory will be application code for the host CPU, which is meaningless to the PIM processors; also, many host processes will never require PIM services at all, and will remain in this configuration.

Internal Memory: If an application elects to use the PIM processing, the first step is to allocate and initialize a region of memory on each node to be used by that node for its local processing needs. These include: a small run-time kernel for parcel management, synchronization and exception handling; code for the application-level methods supported by the PIM; and storage for executing PIM programs such as buffers and stacks.

In practice, efficiency dictates whether this initialization step occurs at host boot time, application load time,

- *command:* an integer encoding the action to be performed, which may refer to a compiled function stored on the PIM.
- *arguments:* (other than *object*), specified as virtual addresses.

An obvious requirement on parcels is small size, to prevent overloading the host-to-memory interface and PIM-to-PIM interconnect. In DIVA, we expect a single packet to consist of a header and 256 bits of payload. A parcel requiring more bits must be sent in multiple packets. A related requirement is that processing parcels must be efficient (see 3.2.1).

In addition, protection must be provided on *arguments*, *pid* and *command* fields; the protection on memory accesses cannot rely on standard host mechanisms as the parcels pass virtual rather than physical addresses to the memory. Also, the order of parcel processing must preserve sequential semantics, but parcel execution should be overlapped to exploit parallelism. To accomplish these goals, we employ optional sequence numbers on parcels when a specific ordering of processing is required.

4.2 Host-Memory Interface

In the initial DIVA prototype, an underlying assumption is that DIVA PIM devices can also serve as conventional memory, so that they can be used as smart-memory coprocessors in a standard system. For this reason, the PIM VLSI device is being designed with a host interface consistent with the standard memory interface typical of commercial memories. This enables PIMs to be packaged in the form of DIMM modules with provisions for top-plane interconnections to support the PIM-to-PIM communication fabric. However, unlike commercial memories, computation activities give rise to new problems: how to communicate internal exceptions and possible memory busy conditions to the host system. These issues are being addressed as part of the larger system architecture.

5.0 Memory Model

Systems with smart memory resemble both uniprocessors (or small SMPs) with large memory, and large, heterogeneous multiprocessors. The semantics are made precise by the DIVA memory model, developed from the following list of requirements:

- a simple virtual machine for both programmers and compiler writers;
- application-level visibility and control of data placement;
- high overall performance;
- scalability to many PIM chips, larger PIM chips, and multiprocessor hosts;
- compatibility with conventional memory models and memory interfaces;
- support for virtual memory (i.e., paging to/from disk); and,
- a host-independent PIM chip architecture.

These requirements look ahead toward future uses of PIM chips, augmenting all sorts of systems and used to accelerate all sorts of applications, both at the small and large scale.

5.1 Parcel Buffers

For high performance, applications must communicate with PIM chips without invoking the host operating system. A conventional memory interface supports this naturally, but cannot generally guarantee atomicity or ordering when caching and write buffers exist. Each PIM chip therefore has a second intelligent interface, the Parcel Buffer, which is mapped into each process as a (roughly) parcel-sized piece of SRAM. The host OS ensures each process uses a different physical address for the multiply-mapped buffer, so the interface can identify the source of each transaction. Hardware in the interface transparently manages ownership of the buffer using a wait-free protocol [Herlihy91] that can be implemented simply at the application level without supervisor state interactions; this interface hardware ensures that access patterns are grammatically correct. To

in the range of 32 to 64 chips, depending on how many PIM chips can be packed onto a DIMM module. The PIM-to-PIM interconnect must then be amenable to the dense packing requirement of DIMM modules. Obviously, low latency and high bandwidth are also desirable properties of this interconnect. Furthermore, this network must be scalable to allow the addition or removal of modules from the system. This combination of requirements favors a one-dimensional network. Although higher-dimension networks offer lower network diameters, they are not easily scalable in all dimensions, especially in a densely packaged system. Also, the dense packing achievable with one-dimensional networks allows more data signals per channel. Hence, the slightly larger distances (in hops) of message traversals in a 32- or 64-hop one-dimensional network are compensated by shorter messages (in flits). Furthermore, router cycle times are faster in one-dimensional network routers because of simpler switching decisions.

The PIM interconnect requirements closely resemble those of interconnect in embedded scalable systems. We therefore use the interconnection network of one such system, the Package-Driven Scalable System (PDSS) [Steele97], as a model for designing the DIVA PIM interconnect. The DIVA PIM interconnect is then a point-to-point bidirectional ring using wormhole routing and the Red Rover routing algorithm [Draper96] to effect deadlock-free routing. It routes fixed-sized packets and uses source routing to achieve low latency. The interconnect is implemented by PIM Routing Co-processor (PiRC) devices - one per PIM chip.

Later generations of DIVA systems are envisioned to contain hundreds and even thousands of PIM chips. Clearly, the advantages of a flat ring topology do not extend to systems of this size. A more complex network scheme will be needed. One possibility is another level of interconnect for connecting host/PIM clusters. To provide adequate aggregate bandwidth, this higher-level interconnect will have to employ channels with greater bandwidth than those of the PIM chips. The details of these channels are beyond the scope of this paper.

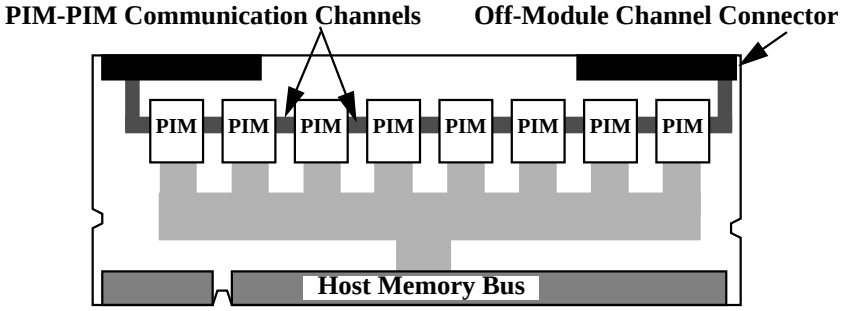


Figure 3: PIM DIMM Module Organization.

4.0 Required Mechanisms

We now present a collection of key mechanisms in DIVA.

4.1 Parcels

A *parcel* is the general mechanism for coordinating computation in memory, communicating data and performing synchronization across components of the DIVA system, a refinement of the parcel concept described previously [Brockman99]. Similar to an active message [vonEicken92], a parcel incorporates data and an encoded operation to apply to the data; a parcel is directed to a memory object, not a process or processor. A parcel has the following four fields:

- *pid:* indicates which process issued the parcel.
- *object:* the virtual address of the primary object the parcel will modify or access, used for routing the parcel.

able system with an unlimited number of chips, not just single chip solutions.

The DIVA architecture and the material presented in this paper is distinguished from these previous approaches in several ways: (1) unlike most of these other approaches, we consider an architecture where smart memory is optionally used to improve performance of a standard host processor; (2) we develop a system that can support in-memory manipulation of both regular and irregular data structures; and, (3) we consider the requirements imposed on the system architecture and system software for mapping application execution between host and memory.

3.0 Overview of DIVA System Architecture

In Figure 1, we show a small set of PIMs connected to a single external host through a host-memory interface; through this interface the host processor performs standard reads and writes, augmented as discussed in Section 3.3. The PIM chips communicate through separate PIM-to-PIM channels to bypass the system bus with additional memory traffic from parcels used to spawn computation, gather results, synchronize activity, or simply access non-local data. The separate interconnect is provided because PIM-to-PIM communication requires greater bandwidth than can be achieved with a conventional memory bus.

3.1 PIM VLSI Component

A PIM is a VLSI memory device augmented with general and special-purpose computing hardware. A PIM may consist of multiple *nodes*, each of which are comprised of a few megabytes of memory and a node processor. The inset in Figure 1 shows a PIM with four nodes. The nodes on a chip share resources for communication with the rest of the system. As a result each chip contains a single PIM Routing Co-processor (PiRC) and a host interface. We anticipate that DIVA PIMs, like many other PIM chips, will be split roughly 60% memory and 40% logic (reflecting the importance of memory density).

Within a single node, shown in Figure 2, the processing logic consists of a standard scalar microprocesor including a floating-point unit and a special DIVA functional unit called an *At-the-Sense-Amps Processor (ASAP)*. The key idea behind the ASAP is to perform *wide* operations on aggregate objects stored within a row of the local memory array. Rather than selecting a 32-bit object from the row as is done with conventional scalar processing, the ASAP unit operates on up to 256 bits in a single processor cycle. This fine-grain parallelism offers additional opportunity for exploiting the increased processor-memory bandwidth available in a PIM. The ASAP unit can be used to perform bit-level operations such as simple pattern matching, or higher-order computations such as searches, limited pointer chasing, and associative and commutative reduction operations. Details on a related wide-word unit are discussed elsewhere [Brockman99].

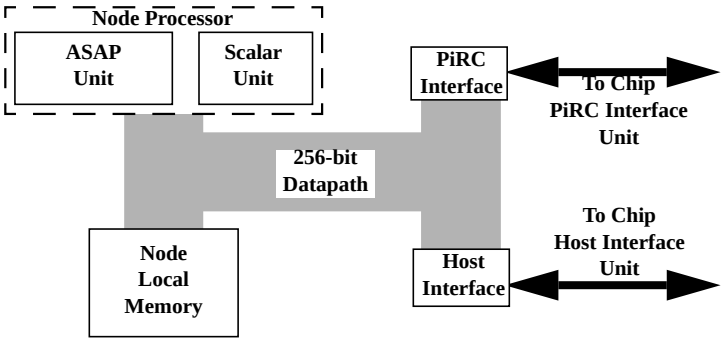


Figure 2: Processor-In-Memory Node Organization.

3.2 PIM Interconnection

We anticipate PIM chips to be physically grouped as conventional memory chips, mounted on DIMM modules, as shown in Figure 3. Bounded by host bus loading constraints, the number of PIM chips in a hosted cluster is

leveraging existing approaches from parallel programming. Section 7 presents three case studies of irregular computations from scientific and database computations; we present system-level simulation results to demonstrate the potential of PIM-based systems at achieving improved performance on these applications.

2.0 Background and Related Work

The concept of mixing memory and logic closer than in a CPU-Memory dichotomy is an old one. The DAPP, STARAN, CM-2, and GAPP all used many relatively small data flows positioned very close to memory arrays to implement very large SIMD machines (all with multiple data flows per chip). At least one such chip, the TERASYS [Gokhale95], was fabricated in relatively large volumes, and targeted as the main memory for one of the later Cray machines. This grew into more or less single chip systems which contained a CPU, some memory, and I/O with machines like the INMOS Transputer [Knowles91], the nCUBE [Palmer86], the J-machine [Dally92], and the SHARC (www.analogdevices.com). While these latter chips could scale to large arrays, their system architecture was a relatively conventional MPP of some form. The first DRAM-based multiple node PIM chip was EXECUBE, fabricated in 1992 and supporting a 3D binary hypercube MIMD/SIMD MPP on a single chip [Kogge94][Sunaga96]. A more recent chip is the Mitsubishi M32 R/D, where more than 2 MB of memory is tightly tied into the on-chip CPU's cache [Shimizu96].

What stopped all these designs from becoming mainstream architectures is very simple - *memory density.* Early PIM-like devices used SRAM for memory, and even with relatively primitive MOS technology, it was quite easy to put more processing power on a single chip than the on-chip data storage could feed. A rule of thumb for scientific computing is that one byte of storage for each FLOP provides a good system balance. Taking any of the previously discussed machines and computing the ratio of on-chip memory to performance (using whatever metric of performance the chip was designed for - usually not even floating point), the ratios are uniformly 0.0001 or worse. Even the EXECUBE chip had a storage to performance ratio of only 0.01. The chips were uniformly memory starved, requiring designs which included ports to off-chip memory.

This began to change around 1997, when DRAM chips with densities greater than 32 Mbits began to appear. At this density, a reasonable ratio of storage to processing can be achieved; for example, an entire video frame buffer can fit in one chip, along with logic to perform processing on it. With current CMOS projections, in a few years a single memory chip will contain more than enough memory capacity for a conventional PC. The realization that complete systems can now be placed on a single chip has led virtually every major semiconductor manufacturer to offer some form of an embedded DRAM macro that can be coupled with other predefined logic macros. At least one industrial organization has sprung up to help set standards to enable such systems [Birnbaum99].

While the technology has finally developed to the point of reasonable systems, architectures which take distinct advantage of the new capabilities have only recently come under serious study. In addition to the Mitsubishi M32 R/D, the IRAM is another system-on-a-chip embedded DRAM device with vector processing logic, designed for streaming computations [Patterson97]. Other approaches use PIM devices as the only processors in a multiprocessor architecture: a cache-coherent distributed-shared-memory system [Saulsbury96], and a large-scale distributed-memory system [Kogge96]. The Active Pages project, which is the most closely related to DIVA, associates configurable logic with each memory page to accelerate performance of an external host [Oskin98].

There are also several other architecture approaches, not based on PIM technology, designed to improve processor-memory bandwidth [Carter99][Burger97][Rixner98]. Impulse augments the memory system to perform application-specified scatter/gather operations on irregular data in the memory controller, so that contiguous data is brought into the cache [Carter99]. Imagine is a system-on-a-chip streaming architecture designed for media applications, which uses a stream programming model [Rixner98]. The DataScalar architecture is a multiprocessor system where each processor asynchronously executes the same code and broadcasts any local data to the other processors [Burger97]. DIVA is distinguished from these approaches as it supports a wide variety of parallel programming models; DIVA PIMs, with the appropriate interconnect, can be used in a scal-

An obvious class of applications well-suited to PIM technology is *regular* --- dense-matrix computations on large amounts of data that are “embarrassingly parallel,” such as image processing. While good candidates for DIVA, such applications also perform well on conventional systems. In this domain, locality-exploiting architecture features (such as long cache lines and vector units) and compiler optimizations (such as tiling [Wolfe89]), and techniques for hiding latency (such as prefetching [Mowry92]) are effective because such applications exhibit significant data reuse, and compilers are able to predict their memory access requirements.

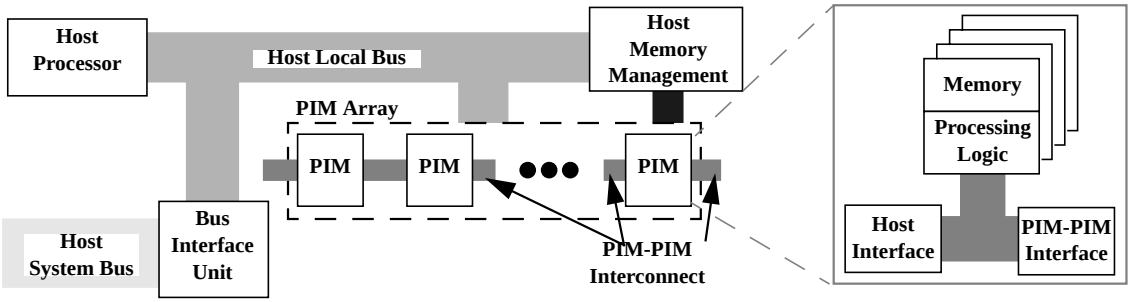


Figure 1: DIVA System Organization.

This paper argues the effectiveness of DIVA for a completely different class of applications: *irregular*, sparse-matrix and pointer-based computations with high processor-memory bandwidth requirements (*e.g.*, sparse conjugate gradient and database applications). Such applications perform poorly on conventional architectures because their control and data accesses cannot be statically predicted, and they do not make effective use of cache. As a result, their execution is dominated by waiting for memory accesses [Carter99]. DIVA can accelerate the performance of such applications by eliminating much of the memory traffic --- simple operations and dereferencing can be done *in situ* rather than laboriously moving data around the system. In addition to the reduction in memory latency for each access, there is potential for coarse-grain parallelism across multiple PIM chips. Performance improvements also result from secondary effects such as reduced host cache and TLB pollution because irregular accesses no longer need be brought into the host processor cache.

While several PIM-based architectures have been proposed in recent years, the DIVA project differs from other efforts in several ways. There are two distinct advantages to using PIMs as smart-memory coprocessors to one or more external hosts: (1) DIVA permits augmenting conventional systems in general-purpose computing environments; and, (2) applications can be gradually migrated from sequential versions that use DIVA PIMs as “dumb” memory toward fully exploiting smart-memory capabilities and parallel in-memory execution. At the same time, this co-processor model imposes fundamentally new requirements on the system software and interfaces. Supporting in-memory pointer accesses requires a new memory model, including a mechanism for address translation within memory. We also rely on the *parcel*, a mechanism for communicating computation to memory, either from a host or a PIM processor. DIVA also requires the host-to-memory interface be augmented because memory must now be able to communicate with the processor for synchronization, exceptions, to warn of high-latency events, etc.

The primary contributions of thiis paper are as follows:

- the first description of the DIVA architecture.
- the first presentation of system requirements for in-memory processing of irregular data structures.
- a detailed description of how to map applications to a PIM-based architecture, with two case studies from important irregular computations.

The remainder of the paper is organized into five main sections and a conclusion. The next section discusses background and previous work. Section 3 presents the system architecture, particularly the PIM-to-PIM interconnect. Section 4 discusses the requirements imposed on the system software and interfaces. Section 5 presents the DIVA memory model. In Section 6, we describe how a user application can be developed for DIVA,

Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture

Mary Hall, Peter Kogge*, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman*, Apoorv Srivastava, William Athas, Vincent Freeh*, Jaewook Shin, Joonseok Park

USC Information Sciences Institute
Marina del Rey, CA 90292

* University of Notre Dame
Notre Dame, IN 46556

Abstract

Processing-in-memory (PIM) chips that integrate processor logic into memory devices offer a new opportunity for bridging the growing gap between processor and memory speeds, especially for applications with high memory-bandwidth requirements. The Data-IntensiVe Architecture (DIVA) system combines PIM memories with one or more external host processors and a PIM-to-PIM interconnect. DIVA increases memory bandwidth through two mechanisms: (1) performing selected computation in memory, reducing the quantity of data transferred across the processor-memory interface; and (2) providing communication mechanisms called *parcels* for moving both data and computation throughout memory, further bypassing the processor-memory bus. DIVA uniquely supports acceleration of important *irregular applications*, including sparse-matrix and pointer-based computations. In this paper, we focus on several aspects of DIVA designed to effectively support such computations at very high performance levels: (1) the memory model and parcel definitions; (2) the PIM-to-PIM interconnect; and, (3) requirements for the processor-to-memory interface. We demonstrate the potential of PIM-based architectures in accelerating the performance of three irregular computations, sparse conjugate gradient, a natural-join database operation and an object-oriented database query.

1.0 Introduction

The increasing gap between processor and memory speeds is a well-known problem in computer architecture, with peak processor performance increasing at a rate of 60% per year while memory access times improve at merely 7%. To mask memory latency in current high-end computers now demands up to 25 times the number of overlapped operations required of supercomputers 30 years ago. Further, techniques designed to hide memory latency, such as multithreading and prefetching, actually increase the memory bandwidth requirements [Burger96]. Recent VLSI technology trends offer a promising solution to bridging the processor-memory gap: integrating processor logic and memory in a processing-in-memory (PIM) chip. Because PIM internal processors can be directly connected to the memory banks, the memory bandwidth is dramatically increased (up to 2 orders of magnitude, tens or even hundreds of gigabits aggregate bandwidth on a chip). Latency to on-chip logic is also reduced, down to as little as one-fourth that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

The Data-IntensiVe Architecture (DIVA) project is developing a system, from VLSI design through system architecture, systems software, compilers and applications, to take advantage of this technology for applications of growing importance to the high-performance computing community. DIVA combines PIM memory chips with one or more external host processors and a PIM-to-PIM interconnect (see Figure 1). Within a single PIM chip, we observe dramatic improvements in and bandwidth and significant reductions in memory latency. But a more important effect, and a distinguishing feature of DIVA, is the coupling of increased opportunity for concurrency with *aggregate* processor-memory bandwidth increases. Multiple memory chips can work in parallel on independent data, and perform PIM-to-PIM communication without going through the processor-memory bus.