Journal of Systems Architecture 46 (2000) 973-990, Elsevier Science B.V.

Distributed Vector Architectures

Stefanos Kaxiras
Bell Laboratories, Lucent Technologies
kaxiras@research.bell-labs.com

Abstract—Integrating processors and main memory is a promising approach to increase system performance. Such integration provides very high memory bandwidth that can be exploited efficiently by vector operations. However, traditional vector applications would easily overflow the limited memory of a single integrated node. To accommodate such workloads, we propose the DIstributed Vector Architecture (DIVA), that uses multiple vector-capable processor/memory nodes in a distributed shared-memory configuration, while maintaining the simple vector programming model. The advantages of our approach are twofold: (i) we dynamically parallelize the execution of vector instructions across the nodes, (ii) we reduce external traffic, by mapping vector computation—rather than data—across the nodes. We propose run-time mechanisms to assign elements of the architectural vector registers on nodes, using the data layout across the nodes as a blueprint. We describe DIVA implementations with a traditional request-response memory model and a data-push model. Using traces of vector supercomputer programs, we demonstrate that DIVA generates considerably less external traffic compared to single or multiple-node alternatives that are based solely on caching or paging. With timing simulations we show that a DIVA system with 2 to 8 nodes is up to three times faster than a single node using its local memory as a large cache and can even outperform a hypothetical system where the application fits in local memory.

Keywords: *vector processing, processor-memory integration*

1 Introduction

While microprocessors follow an explosive growth in performance, DRAM-based memory systems fall behind creating the memory wall [3,18]. Integration of main memory on the same module (MCM) or even on the same die (IRAM) with the processor promises a high-performance yet inexpensive memory system [12,13,15]. Through the elimination of the pin interface, such integration is expected to deliver:

- A substantial increase in memory bandwidth (hundredfold increase over the current workstation memory bandwidth) due to the vastly improved ability to interconnect the processing core to multiple DRAM row buffers [13].
- A reduction of the memory access latency (tenfold decrease over current workstation memory latency) following the elimination of crossing chip boundaries [13].

An inexpensive, high-performance memory system allows the implementation of a vector supercomputer on a chip [8,9,13]. Vector units are able to exploit very high memory bandwidth because of their very efficient instruction issue and because they can be implemented with deep and multiple pipelines. Lee argues that a vector processor can potentially provide significant performance improvements over an aggressive out-of-order superscalar, while at the same time consume significantly less die area [10].

Vector processing is a well-understood technology, backed by mature compiler support. It applies well to many scientific codes which have significant computation requirements. To the users, the appeal of vector processing is in its

simple programming model: programs are written in a serial programming language (such as FORTRAN) and a vectorizing compiler is responsible to extract the application parallelism and expose it to the hardware. A limited form of vector processing is also popular in microprocessors (with enhancements such as MMX [17]).

Vector processing has major performance impact on large scientific applications (e.g., weather prediction, crash-test simulations, physics simulations). The question that this work addresses is how to run such large application in a system that is built of small highly integrated processor/memory nodes. For such applications the limited and non-expandable memory of a single IRAM[1] or a single highly integrated processor/memory node is *insufficient to hold the dataset*. Inevitably, large applications require external access beyond the local memory of the node, diminishing the benefits of processor-memory integration. Paging to, or caching external memory leads to excessive traffic—thrashing—if the working set of the application does not fit in the device (see Section 4). All the applications we examined have large working sets and exhibit poor locality.

Since the memory of a single integrated processor/memory node, often, would be insufficient to hold such large applications, we propose a Distributed Vector Architecture (DIVA) that utilizes the combined memory and processing power of multiple nodes. We use the memory of the nodes to hold the dataset of the application and the aggregate computation power of the nodes to execute the application. Every node executes the same instructions but we partition the *computation* of the vector instructions across the nodes. Different parts of the vector computation are assigned dynamically on nodes, in an attempt to map the computation where the majority of the data is. This *computation mapping* leads to a considerable reduction of the inter-node (external) traffic compared to other approaches that are based on caching. Since external traffic is the most important limitation of current and future processors [3], DIVA is a promising approach for high-performance computing.

DIVA is similar to the SGI/CRAY SV-1 architecture [5]. Both were developed in CRAY at about the same time. In contrast to the SV-1 architecture where the partitioning of the vector computation is the responsibility of the compiler, DIVA employs dynamic techniques to distribute the computation across the nodes. Since we have no access to a SV-1 compiler we were unable to directly compare the two approaches. As of yet, no technical papers exist to describe SV-1 performance.

DIVA is also closely related to the DataScalar architecture. The DataScalar architecture (developed by Burger, Goodman and the author) [2], is an architecture that uses multiple nodes to execute *serial* programs that do not fit in the memory of one of the nodes. Like DIVA, DataScalar is based on the ESP execution model of the Massive Memory Machine [6]. In this model, all nodes execute all the instructions of an application. Each node maintains its own register set and performs (redundantly) all computations. A node accessing local data (owning node), broadcasts them to other nodes. The DataScalar architecture extends this paradigm to work with caches (that dramatically reduce the number of broadcasts) and out-of-order execution that allows nodes to run asynchronously. *Result communication* in

[1] Throughout this paper we will use the term IRAM to refer to highly integrated processor/memory nodes.

DataScalar allows code to be executed only by a subset of the nodes [2]. Using result communication, a DataScalar system (based on superscalar nodes) can emulate a DIVA system at the cost of performance.

We evaluate DIVA using simulation and we show that for our target work-load, the NAS benchmarks [1], DIVA actually generates less external traffic than other centralized (single-node) or distributed (multi-node) approaches. Using timing simulation, and simple vector-capable nodes, we show speedups for 2 to 8 DIVA nodes to be in the range of 0.72 to 2.98 over a single node using its memory as a large cache. DIVA also shows speedups in the range of 0.33 to 1.86 over the hypothetical case where the application would fit entirely in the memory of a single node. Our results also show that: (i) given multiple memory banks and high internal bandwidth, memory latency is not critical parameter for performance because of the ability of the vector units to tolerate it, (ii) pushing data to destination is beneficial only when we have a large number of nodes or a very slow node interconnect.

In the rest of this paper we describe the principles of the architecture and we present two possible implementations (Section 2). In Section 3, we briefly discuss compile-time data optimization and its relation to DIVA. In Section 4 we present an evaluation of DIVA where we show that the external traffic of DIVA is low and often much less than the traffic of other approaches based on caching. With timing simulations we estimate DIVA's performance and its relation to memory and network latency. Finally, we conclude in Section 5.

2 DIVA

In Figure 1 we show a DIstributed Vector Architecture (DIVA) based on a collection of highly integrated nodes with vector capabilities. The nodes are connected with a network in a distributed shared-memory configuration. A large vector application occupies the memory of multiple nodes, all of which cooperate on the execution of individual vector instructions. Each node executes all scalar instructions redundantly, maintaining its own scalar register set, similarly to the Massive Memory Machine [6] or to the DataScalar architecture [2]. Each node also sees the exact same vector instructions whose execution is dynamically parallelized. The application references *architectural* vector registers (Figure 1) that represent the aggregate of multiple *physical* vector registers, one from each node. Nodes operate only on their own physical registers (i.e., only on the subset of the architectural elements that is mapped in their physical registers). Since vectorizing compilers guarantee the independence of the computations within a vector instruction, nodes work in parallel on their own part of the vector instruction, providing a speed up proportional to the number of nodes involved.

The actual part of the vector instruction that is executed by a node depends on the mapping of the architectural vector elements to the physical vector elements. In this work, we propose a dynamic program-controlled mapping of architectural elements to physical elements. In Figure 2, we show such an *element mapping*. The architectural element 0 is assigned to node 1, element 1 to node 0, etc. We apply this mapping when element 0 corresponds to data present in node 1's memory (e.g., data m0), element 1 corresponds to data in node 0's memory (m1), and so on. Loading this architectural vector register requires no external traffic. However, we cannot map each architectural vector register independently, but we have to map multiple architectural vector registers that take part in the same computation in

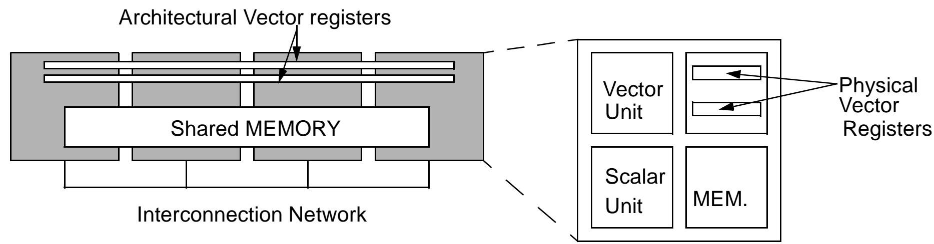

FIGURE 1. DIVA **system comprised of four nodes. Each node contains some part of the system memory along with a processor and a vector unit. In this example each node has two physical vector registers. An application running on all four nodes (occupying all their memory) refers to two architectural vector registers. Each architectural register is comprised of four physical registers (one per node).**

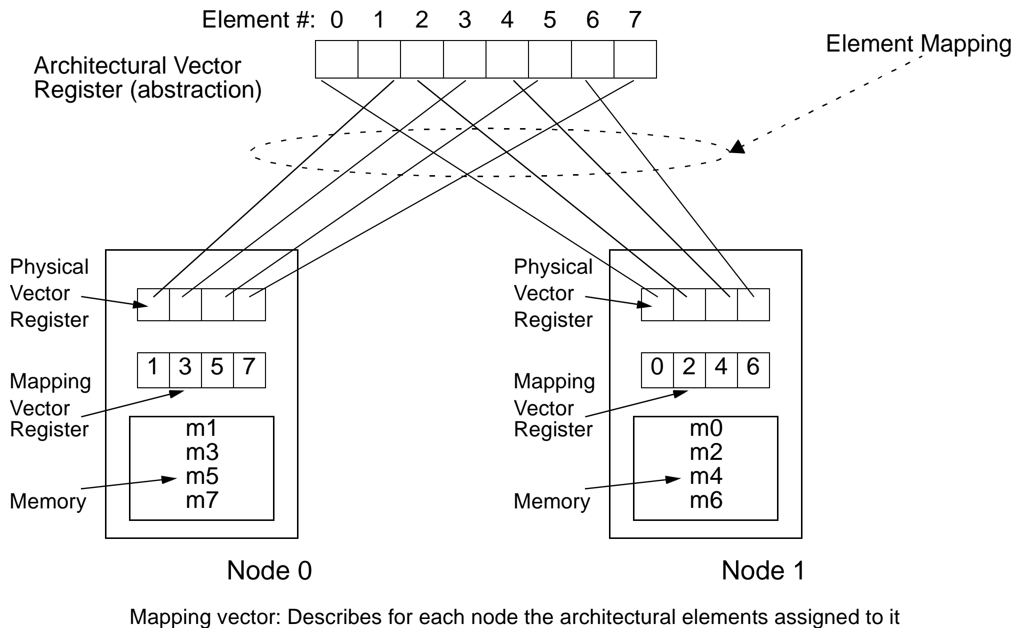

FIGURE 2. An element mapping of architectural to physical vector elements

exactly the same way so that their corresponding elements will align properly in the nodes. Unfortunately, the best mapping for loading one architectural vector register is not always compatible with the best mapping for loading another register with a different memory vector. Using a common mapping for all the registers involved in a computation inevitably results in external traffic.

In this work, we propose mechanisms to generate mappings exploiting the fact that every node observes the exact same vector instructions. Using these mechanisms, all nodes independently generate a consistent element mapping and use it to partition the work among them.

2.1 DIVA Implementations

In this section we describe two DIVA implementations that differ on how nodes access vector elements. In the first implementation (Request/Response DIVA) each node has a local view of the element mapping. A node executes its

own part of a vector load (store) and requests, if necessary, non-local data from other nodes. In the second implementation (Push-Mode DIVA) each node has a global view of the element mapping, thus is aware of what other nodes want from its local memory. A node inspects an entire vector load, accesses all the elements in its local memory and sends them to the appropriate destinations.

In both implementations the application dataset is block-interleaved across the nodes. Small caches in the nodes can be added to dynamically replicate part of the dataset. Since we parallelize vector instructions across a distributed system, synchronization is needed to guarantee a correct ordering of memory operations. Conservatively, we synchronize the end of vector store instructions with fast memory fences. The compiler, however, can eliminate most of this synchronization and enforce it only when needed. Similar synchronization issues exist in the CRAY supercomputers such as the CRAY T-90 or the CRAY X-MP which have a non-deterministic memory system. The CRAY instruction set already supports the necessary memory synchronization instructions (e.g. fences, etc.).

2.1.1 Request/Response DIVA

In the basic implementation of DIVA we parallelize vector instructions using two mechanisms: (i) a novel vector instruction, called SETMV, which creates element mappings on the fly and (ii) special vector registers, called *mapping vector registers,* which enforce the element mappings generated by the SETMV instruction.

FORTRAN CODE

```
      DO 100 I=1,16
      C(I)=A(I)+B(2*I)
100   CONTINUE
```

COMPUTATION SLICE

```
SETMV BASE=A, STRIDE=1, MV0
VLOAD V0, BASE=A, STRIDE=1, MV0 (VL=16)
VLOAD V1, BASE=B, STRIDE=2, MV0 (VL=16)
VADD V0, V0, V1 /* V0=V0+V1 */
VSTORE V0, BASE=C, STRIDE=1, MV0 (VL=16)
```

FIGURE 3. A SETMV instruction creates a mapping (stored in the mapping vector MV0) at the beginning of a computation slice. Memory instructions of the computation slice adhere to the mapping vector MV0.

SETMV instructions—All architectural vector registers that participate in the same computation must be mapped to the physical vector registers in exactly the same way. A SETMV instruction precedes every computation slice (Figure 3) and creates an element mapping for all the registers of the computation slice.

SETMV uses as a blueprint for the element mapping the layout of a vector in memory. Typically, the memory vector is referenced in the computation slice. In the example in Figure 3, the SETMV instruction uses the memory vector accessed by the first vector load. A node executing the SETMV instruction generates the addresses of all the elements of the memory vector and uses a run-time *location* test to determine which of these addresses are local. The location test is straightforward with any regular data distribution and the “home” node of a memory element can be deter-

mined by all other nodes. Architectural elements are assigned to the home node of the corresponding memory element.

The semantics of the SETMV instruction also handle, in a distributed fashion, cases where too many local elements in one node would overflow its physical vector registers. Each node keeps track of the number of elements assigned to other nodes using a number of counters. A node is full when the corresponding counter exceeds the size of the physical vector registers. Responsibility for elements that the full node cannot handle passes to the first non-full node according to a pre-specified order (e.g., based on the node identifier). This continues until all architectural vector elements are assigned to some node. This simple distributed algorithm is independent of the relative speed of the nodes and guarantees that no assignment conflicts will occur.

The cost of executing the SETMV instruction can be hidden by *chaining* the appropriate vector load instruction off of it. In the example of Figure 3, the SETMV uses the memory vector accessed in the first vector load. Chaining these two instructions allows the appropriate subset of the addresses generated by the SETMV to be used by the vector load.

***Mapping vectors*—**A *mapping vector* represents the element mapping generated by a SETMV[2]. Similarly to the architectural vector registers, a mapping vector is distributed in *physical* mapping vector registers across the nodes (Figure 2). A node executing a SETMV instruction sets its physical mapping vector register to point to its assigned architectural elements. In the example of Figure 2, the mapping vector register in node 0 points to the architectural elements 2, 6, 10, and 14.

Within a node, the physical mapping vector register constrains the vector load (store) instructions to the subset of the architectural elements assigned to the node. Vector loads (stores) use the mapping vector as an index vector register. Other instructions simply operate on the contents of the vector registers without concern for the mapping vectors.

***Caches*—**Caching in a Request/Response DIVA is straightforward since each node is responsible to access the elements it needs. A small part of the node memory can be used to cache external data. Caches replicate part of the application dataset and increase locality. However, caching introduces complexity in the design of DIVA. Tag storage and access must be taken into account and a cache-coherence protocol is required to keep the distributed caches coherent. The benefit of caching in DIVA systems, namely the reduction of external traffic, may not outweigh the implementation complexity or the potential run-time overhead. In fact, in Section 4 we show that DIVA does not need caches to exceed the performance of other alternatives, assuming a fast node interconnect. Only when external communication becomes prohibitively expensive we need to consider caching to maximize locality.

[2] Since each computation slice needs its own mapping vector, we can use a small number of mapping vectors to accommodate multiple independent computation slices interleaved in the instruction stream.

2.1.2 Push-mode DIVA

Push-mode DIVA discards the request/response model of the previous implementation in favor of a model where data are pushed to their destination. In this respect, the nodes become *intelligent memory* since they can determine when other nodes need their data and promptly service them. The basic mechanism to generate element mappings is the same SETMV instruction as with the previous implementation. Because of this, the amount of raw data transferred externally in Push-mode and in Request/Response DIVA is the same.

Again, a SETMV instruction at the beginning of each computational slice uses a memory vector to assign the architectural elements in the nodes. In addition to the distributed view of the mapping that is offered by the mapping vector registers, we also keep a global view of the mapping in *reverse-mapping vectors*. The SETMV instruction sets these registers according to the location of the elements. In Figure 4, the reverse mapping vectors describe the node where each element is assigned (e.g., element 0 in node 1, element 1 in node 0, element 2 in node 1, etc.). Note that the reverse mapping vectors are identical in every node in contrast to the mapping vectors that are always different.

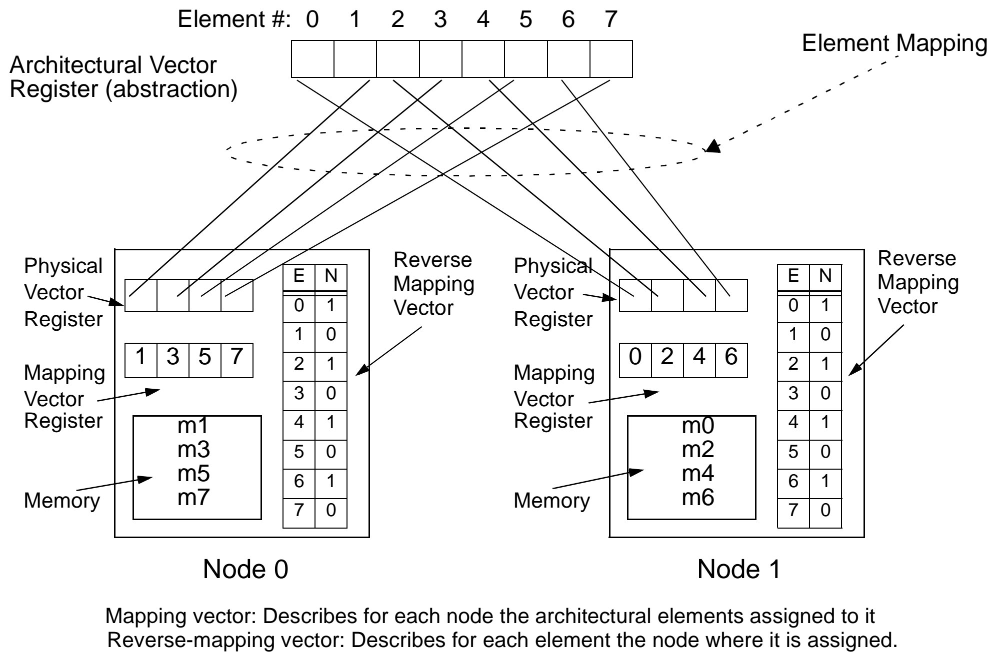

FIGURE 4. Push-mode DIVA using Reverse Mapping Vectors

In the Push-mode DIVA the execution of a vector load instruction differs from that of the request/response model: a node instead of just loading the elements described in its mapping vector register, generates all the addresses of the vector load but accesses only the elements that are local to it. For each accessed element the node consults its reverse-mapping vector register and directs the data to the appropriate destination (either a local physical vector register or a remote node). Vector store instructions execute as before: each node consults its mapping vector register and stores elements described therein.

The potential benefits of Push-mode are threefold: (i) external traffic can be reduced since we eliminate requests[3], (ii) data can be transferred to their destination faster since we break the request/response cycle, (iii) the utilization of the on-chip high bandwidth may be increased since the local processor accesses all its local data in a burst.

However, Push-Mode DIVA also has drawbacks, namely the need for each node to generate all the addresses of the vector load (store) instructions, and the cost of implementing the reverse-mapping vectors. In the context of processor/memory integration, where address generation is much cheaper than going off chip, and within the range of nodes and vector lengths we examined, there are cases where the benefits of Push-mode outweigh its drawbacks (see Section 4). To avoid the scalability problems of fixed-length reverse-mapping vector registers, we can store the parameters of the corresponding SETMV and generate the reverse-mapping vectors on demand (without storing them), in parallel with the address generation of the vector load instructions.

***Caches*—**Caching of external data in a Push-Mode DIVA is complicated because: (i) the notion of local data now includes cached data that can be loaded on local registers but not forwarded to other nodes, (ii) each node needs information about what local data are cached remotely to avoid superfluous transfers. In contrast to the request/response DIVA where caching can be implemented with just a snoopy bus protocol [7], push-mode DIVA requires an elaborate directory-based cache coherence protocol [4] that can track the cached copies. For completeness we describe such a protocol in Appendix I.

3 DIVA and Data Optimization

In a DIVA system, we distribute memory vectors across the nodes to maximize the available parallelism. Additionally, we want to align memory vectors accessed in the same computation slice, to minimize remote traffic. Data optimization techniques (i.e., compile-time data placement to minimize communication) can be applied in DIVA to achieve memory vector distribution and alignment. In general, data placement is a difficult and application dependent problem, although techniques have been developed for, and used in SIMD computers. A multinode system without computation mapping (e.g., SV-1) relies on data placement techniques to reduce external traffic. In contrast, DIVA represents a novel dynamic approach to this problem. The important implication of DIVA is that it gives us a new degree of freedom in data optimization: it is no longer critical to have an optimal data placement at compile time since we can dynamically map the computation of vector instructions.

In our evaluation we have used a simple block-interleaving data distribution scheme. We block-interleave memory in a DIVA system by selecting which bits of the address are used as the node address bits. The guiding heuristic for selecting a block size for an application is that it should effectively distribute across the nodes the application's dominant kind of memory vectors (determined by the application's dominant stride and dominant vector length). Although this heuristic proved useful, it did not always produce the best interleaving. Block-interleaving also leads to

[3] The data pushed to nodes are augmented with additional information such as destination vector register and element number tags. In the evaluation we count this as "overhead traffic."

simple locality tests used in the Push-mode DIVA.

As for aligning memory vectors, we have experimented with simple source code transformations. These transformations affected solely the allocation of data structures (not the execution of the programs) and involved changing some array dimensions to powers-of-two. The original codes were optimized for CRAY supercomputers and their data structure allocation reflected optimizations to reduce memory bank conflicts (e.g., prime number array dimensions). In contrast, memory vectors that start at multiples of power-of-two addresses are much more likely to align in nodes than memory vectors that start at arbitrary addresses. We have changed the allocation in three of the six benchmarks we use and we have achieved a reduction of external traffic ranging from 10% to 77%. Clearly, memory vector distribution and alignement can have significant performance implications in DIVA systems.

For the evaluations in the following section we assume a segmented memory space. Applications fit entirely in one segment. Virtual to physical address translation involves adding an offset to the virtual address. For each application the operating system sets the run-time interleaving. With more elaborate hardware it is possible to have simultaneously multiple interleavings for the same application. In this way, we can distribute effectively different data structures of the application (e.g., we can select interleavings to distribute either the rows or the columns of matrices that are multiplied together).

4 Evaluation

DIVA's performance comes from: (i) parallel execution of vector instructions across the nodes, (ii) low external traffic. External traffic is a critical measure, since if it were excessive it would invalidate the parallelism advantage. Furthermore, external traffic is a measure *independent* of any timing assumptions—it is a measure inherent in the applications and the architecture. The raw data traffic (i.e., excluding request traffic and other overhead traffic) for the Request/Response DIVA and the Push-mode DIVA is the same[4]. In this respect, our general conclusions concerning traffic apply to both DIVA implementations. The differences in their performance are apparent with timing simulations.

In the rest of this section, after a short description of our experimentation set-up, we present the effects of computation mapping on external traffic and we discuss two methods to further reduce DIVA's external traffic: compiler optimizations and caching. We wrap our traffic evaluation by showing that DIVA traffic compares favorably to single or multinode alternatives based on caching. We then proceed with performance evaluation using timing simulations. We also discuss how varying memory and network latency affect DIVA's performance. Finally, we conclude the evaluation section discussing scalability.

4.1 Methodology

For all experiments we used traces of 6 NAS benchmarks [1]. The BT, FT, LU, MG, and SP benchmarks represent

[4] However, caches may affect traffic differently in the two models.

floating point scientific codes and the IS benchmark represents integer vector codes. The traces were collected on a CRAY C-90 supercomputer with 128-element vector registers. Unfortunately, this methodology imposes a fixed architectural length of 128 elements regardless of the number of DIVA nodes. In an actual DIVA system the architectural vector length would be a function of the number of nodes.

Memory interleaving was partially guided by the heuristic mentioned in Section 3. Here, we report results for the best interleavings we have found by experimentation. Data distribution by interleaving is a simplistic method and significant improvements are possible in this area. Alignment of memory vectors (discussed in Section 3) was affected by simple changes in the allocation of data structures in three of the six benchmarks (BT, LU, and SP).

Nodes have a high performance memory system comprised of multiple memory banks and a high performance memory interconnect. These design decisions reflect opportunities of processor/memory integration: nodes in this evaluation are modeled after vector IRAM. We assume that the on-chip memory interconnect does not experience any contention, reflecting the substantially improved ability to interconnect read/write ports to multiple banks on a single integrated device. We do, however, model contention on the memory banks (bank conflicts). We assumed 128 memory banks (4096 bits wide). This design is between the 32 banks chosen in [15] and the 512 banks proposed in [14]. We chose this number to match the architectural vector length and the number of outstanding memory operations. To cover a wide spectrum of processor/memory integration implementations we explored five different on-chip access latencies: 2, 4, 8, 16 and 32 processor cycles. Assuming, either 1GHz or 500MHz clock rates we cover access latencies studied in both [2] and [15]. In this section we present results for an access latency of 8 cycles.

For the node interconnect we used a split-transaction bus whose clock rate is a fraction (1/2, 1/4, 1/8, 1/16 or 1/32) of the processor clock. In this section, we present results for a fast, low overhead bus, which takes 4 processor cycles to transfer 128 bits.

For the cache simulations we assumed direct mapped caches with 8 word blocks. A word in CRAY supercomputers is 64 bits. The single node system also has the same high performance internal organization as the DIVA nodes with 128 cache banks (4096 or 8192 bits wide — see Table 1). An important factor that affects performance in our models is the contention in the memory or cache banks. We minimized this contention by interleaving blocks internally in nodes. For DIVA, the block sizes used to interleave memory across the nodes, are also appropriate for internal interleaving: blocks are distributed round-robin across the nodes and internally (in every node) across the memory banks. Table 1 shows the block sizes we used. In DIVA, the product of the block size and the number of nodes is constant and it is the block size we use in the single node case. The size of the cache blocks (8 words) is a fraction of the interleaving block size. This internal interleaving prevents memory or cache banks from becoming hot spots and produces

consistent behavior regardless of the number of banks in the systems.

		DIVA		
	Single-node	**2 nodes**	**4 nodes**	**8 nodes**
BT	128	64	32	16
FT	16	8	4	2
IS	64	32	16	8
LU	16	8	4	2
MG	16	8	4	2
SP	16	8	4	2

Table 1: Block-interleaving size (in words). The interleaving size for BT requires cache banks 8129 bits wide.

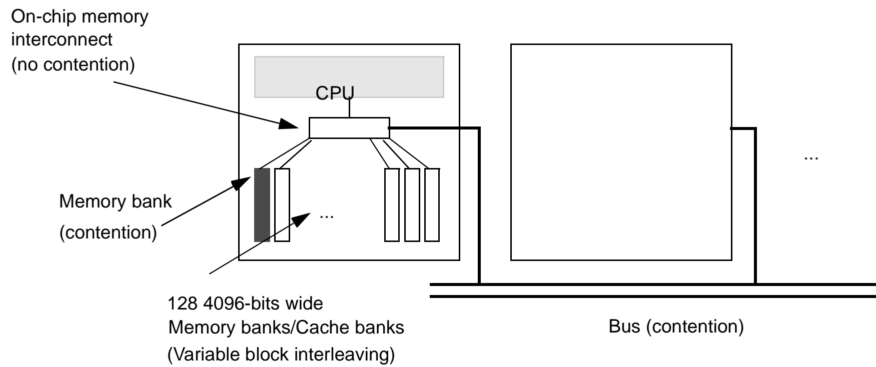

FIGURE 5. Node memory organization

4.2 External Traffic

Computation mapping is key in reducing DIVA's external traffic. Without it the external traffic of DIVA for the default data layout is excessive, invalidating the parallelism advantage.

To study the benefits of computation mapping we compare external traffic of a multinode DIVA system with and without computation (element) mapping. We examined systems with 2, 4, and 8 nodes. We scale the memory capacity of each node so the application occupies the memory of all nodes (e.g., if an application runs in 4 nodes, the memory capacity of the node is 1/4 of the application size). In Figure 6 we express external traffic as a percentage of the total traffic required by the application to move data between the memory and the vector registers. The external traffic without element mapping is approximately 50% of the total traffic for 2 nodes, 75% for 4 nodes and 87.5% for 4 nodes (Figure 6). These percentages represent the case where we have a random data distribution and on average each node finds 50%, 25% or 12.5% of its operands locally in the 2, 4, and 8 node cases respectively.

Computation mapping reduces external traffic from 41% to 88% (shown in Figure 7) over the previous case. The reductions are smaller for the larger number of nodes. However, the remaining external traffic is still considerable. This is characteristic of vector applications that move massive amounts of data and require very high performance memory systems.

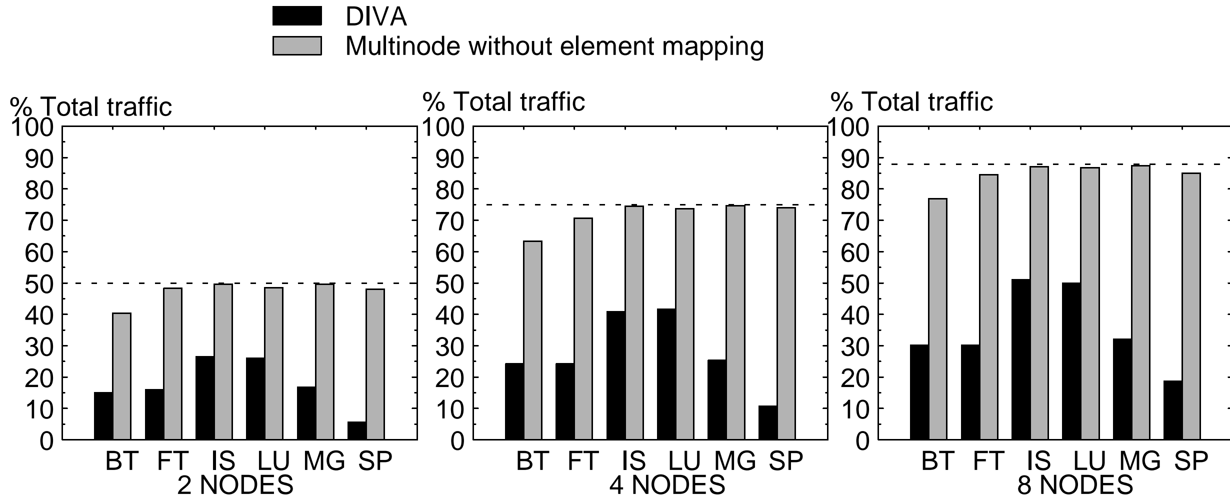

FIGURE 6. External Traffic as a percent of the of total application traffic

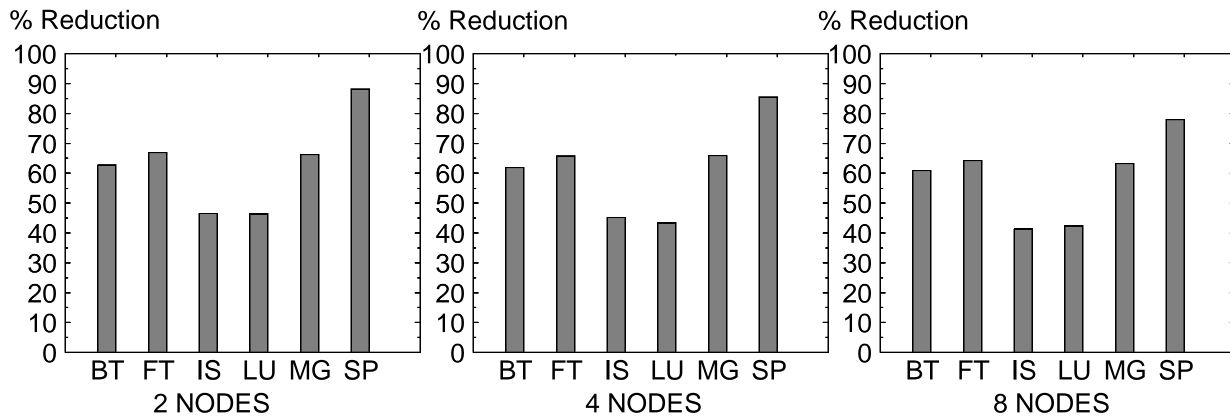

FIGURE 7. Reduction of external traffic attributed to computation mapping as a percent of the external traffic of a multinode system without computation mapping

4.3 Compiler Optimizations

In this section we discuss compiler involvement to reduce external traffic. The compiler can contribute to more effective element mappings by intelligently selecting which memory vectors are used as arguments in the SETMV instructions. The default case in our evaluation is that the compiler does not have enough information to select the memory vector of a computation slice to be used as the blueprint for the element mapping. In this case, the first memory vector accessed in the computation slice is used by the SETMV instruction (we refer to this case as “first choice” selection). If, however, the compiler does have enough information it can possibly select a memory vector leading to a better element mapping for the whole computation slice (we refer to this case as “best choice” selection). We believe that an actual compiler would be able to make some intelligent choices for some computation slices but presumably not for all. The resulting compiled program will contain a mix of SETMV instructions based on the “best choice” selection and SETMV instructions based on the “first choice” selection.

“Best choice” selection is based on the following simple heuristic, but more elaborate methods are possible. For each vector load/store in the computation slice we generate all its addresses and compute the owner node for all its elements according to the run-time memory interleaving. We then compare the owner nodes of the elements accessed in each vector load/store to the owner nodes of the elements of all the other vector loads/stores and we select the one with the most matches.

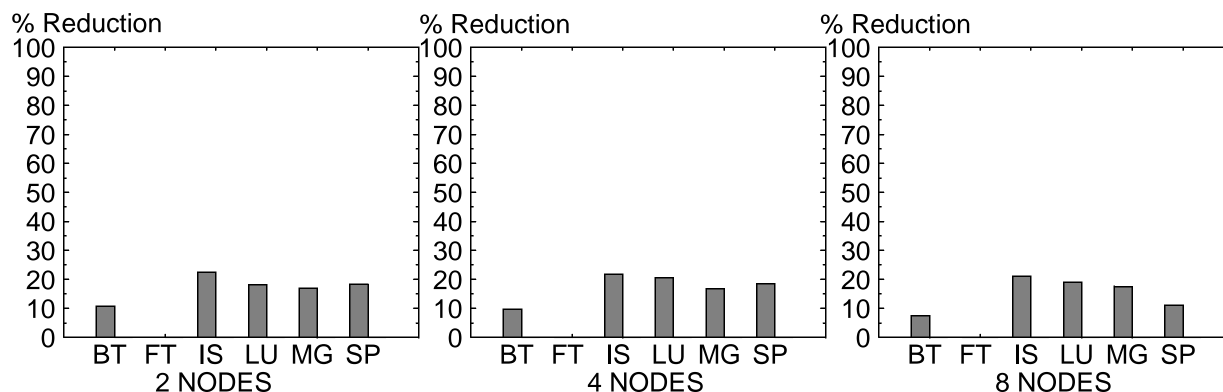

FIGURE 8. Reduction of DIVA external traffic when selecting the “best” memory vector of a computation slice as argument for the SETMV instruction. Traffic reduction is given as a percent of the external traffic of DIVA system when the first memory vector of a computation slice is used to create the mapping.

To study the effects of the “best choice” and “first choice” selection, we used a preprocessor that transforms the CRAY traces to DIVA traces. The preprocessor plays the role of the DIVA compiler and inserts the appropriate SETMV instructions in the trace. Since all the necessary information is present in the traces, the preprocessor can either do “first choice” or the approximate “best choice” selection of SETMV arguments. Selecting the “best” memory vector, reduces external traffic from 10% to 23% over “first choice” selection (Figure 8). In the case of the FT benchmark the “first choice” is also the “best choice” selection. These results indicate that compiler support for selecting SETMV arguments, is desirable but not critical: “first choice” does reasonably well.

4.4 Caching

In this section we show the effect of caching in DIVA and in the multinode system without element mapping. Caches can reduce DIVA traffic by replicating data, therefore providing a form of dynamic data placement optimization. Although computation mapping reduces traffic considerably, caches can still provide additional traffic reductions, substantial in some cases.

For our evaluation, we assume that a part of the node memory is devoted to caching external data. In all experiments we chose the cache size to be 1/16 of the node memory. We optimized the cache for traffic: it is 2-way set associative[5] write-back cache, with 1-word cache blocks

[5] We explored higher associativities and we found that they do not provide significant differences. In fact, in some cases higher associativities with LRU replacement are worse.

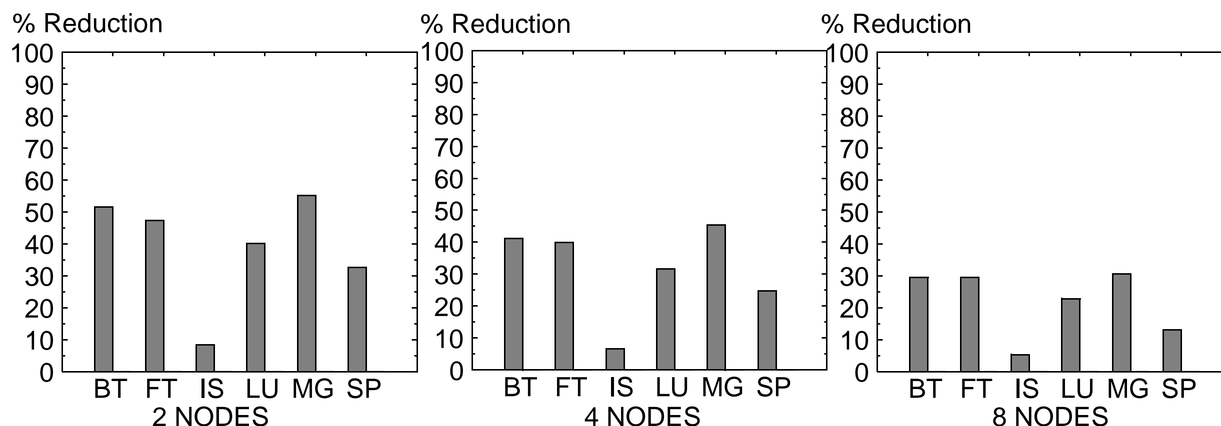

FIGURE 9. Reduction in external traffic for DIVA with small caches. Traffic reduction is given as a percentage of the external traffic of DIVA without caches.

Figure 9 shows that the reduction of external traffic for DIVA systems ranges from 5% to 55%. Whether caches are desirable in DIVA systems depends on their effect on performance and on the implementation complexity they bring in the design. In turn, their effect on performance depends on the performance of the node interconnect: with a very slow interconnect, caching is unavoidable. We have also determined that caches help considerably the system without element mapping. This system is inherently traffic-intensive and caches provide a decrease of the external traffic ranging from 21% to 76%: without the benefit of element mapping caching is much more important than in DIVA.

4.5 Traffic Comparisons for Five Systems

In this section we compare, in terms of traffic, five alternatives based on integrated processor/memory nodes (Figure 10):

1. A multinode DIVA system. The application is block-interleaved across the nodes. Each node contains a small cache for external data as in Section 4.4. Source code transformations to affect the alignment of memory vectors were applied only in this case.
2. A multinode shared-memory system without element mapping. This would be similar to an SV-1 system but without the benefit of an appropriate compiler. Again, each node holds a part of the application dataset and includes a small cache for external data (again as in Section 4.4).
3. A single-node system with a statically allocated part of the application[6] on the node memory and a cache for external data (with the same configuration as in cases 1 and 2).
4. A single-node system that uses its node memory as a large cache. The size of the cache is equal to the DRAM memory capacity of the node. The cache is 2-way set associative, with 4-word cache blocks (to reduce tag overhead), sectored with 1-word sub-blocks to reduce traffic [11], and write-back. We have determined that associativity does not play any significant role in such large caches. We used the cheetah cache simulator [16] to search

[6] We select a contiguous part of the dataset that minimizes traffic.

associativities in the range of 1 to 32. Effects in cache behavior were minimal. In fact, higher associativity with LRU replacement may even hurt performance.

5. A single-node system with demand-paging to external memory. 4KByte pages are swapped between node memory and external memory.

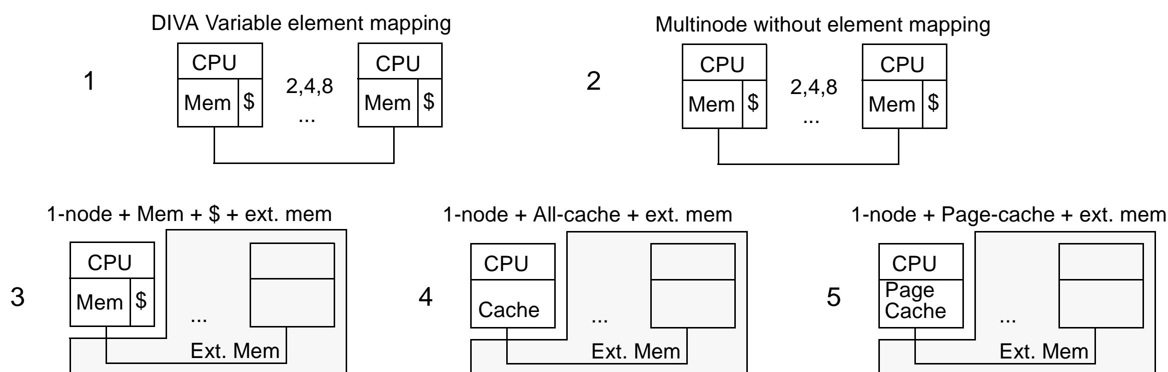

FIGURE 10. Five alternatives based on processor/memory nodes. 2 multinode systems (with and without element mapping) compare against three single node systems that use all their node memory as some form of cache.

We scale the memory of the other four alternatives in the same way as with DIVA nodes so we always compare equal memory capacity nodes. In Figure 11, we show the external traffic of the five systems, (again as a percentage of the total traffic required by the application to move data between the memory and the vector registers). Although we use caches to reduce DIVA traffic, we are unable to estimate the success of the compiler in selecting the “best” arguments for SETMV. Therefore, we conservatively assume that, lacking compile-time information, the compiler blindly selects the first accessed memory vector of each computation slice as the SETMV argument.

Four of the benchmarks (BT, IS, LU, and SP) thrash when they run in demand-paging on nodes with a memory capacity 1/4 or 1/8 of the application size. For the specific applications we use in our experiments we believe that paging without any provision for variable size blocks or other optimizations is not a good choice. In most cases, DIVA exhibits less traffic than any of the other four alternatives. On average, for all the programs and all the node configurations, DIVA produces 70% less traffic than the multi-node system, 34% less than the single-node with static memory allocation, 10% less than the single-node all-cache system, and 87% less than the single-node with demand-paging.

DIVA *without* any cache (not shown in Figure 11) is also very competitive, yielding 29% less traffic than a multinode with caches, 10% less than the static-allocation single-node with cache, 21% *more* traffic than the all-cache single-node, and 82% less traffic than the demand-paging single node. Our results indicate that external traffic is, many a times, less of a bottleneck for DIVA than it is for the other systems. Furthermore, it is important to note that DIVA traffic (as well as the traffic in the multinode system without element mapping) is divided over many nodes, while traffic in the single-node systems concentrates on a single node.

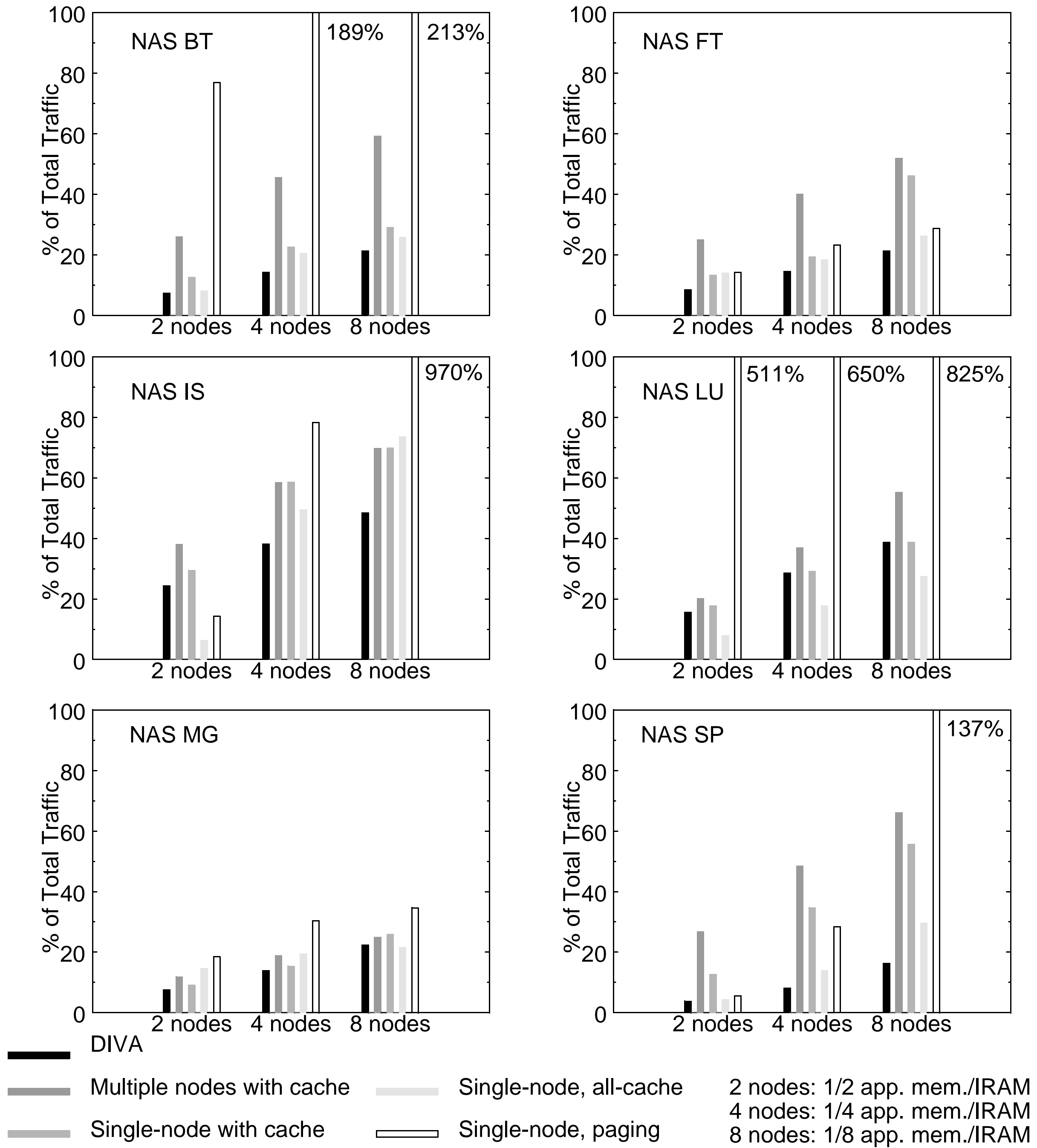

FIGURE 11. Traffic comparisons of five systems for 6 of the NAS benchmarks

4.6 Performance Evaluation

The positive outcome of our traffic evaluation suggested that DIVA, indeed, has a performance advantage over other approaches. In this section we quantify this advantage using timing simulations. For the simulations, we used a simple model of vector units that does not implement chaining of vector instructions. In this respect, DIVA speedups presented here may be inflated. However, we believe that chaining, although it will have a bigger impact on the

performance of the single node systems, has second-order effects on performance. This is because in our models, computation, which is affected the most from chaining, is considerably less expensive than memory operations.

We compare DIVA *without caches* to the single node alternative that uses all its memory as a large cache (case 4 in the previous section).

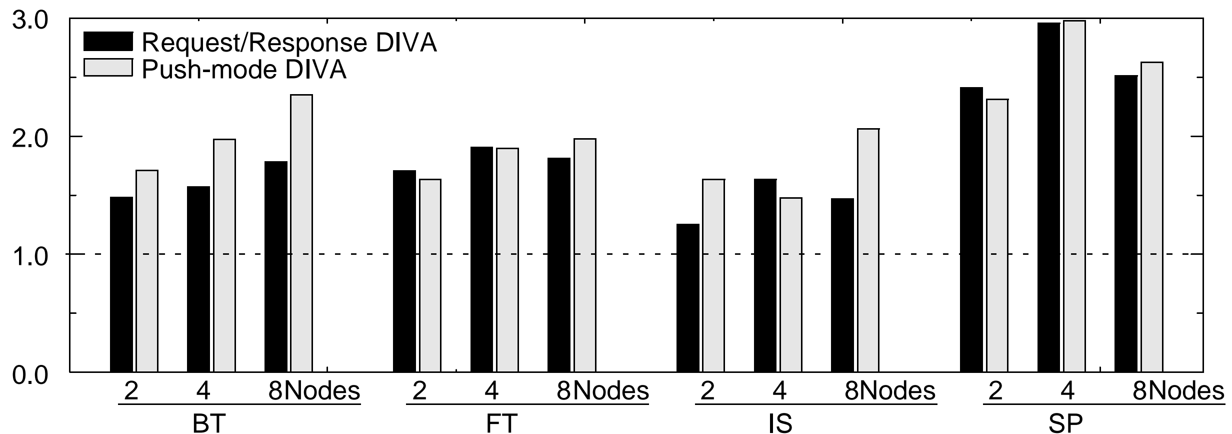

FIGURE 12. Speedup of DIVA systems (without caches) over the case of a single node using all its memory as a huge cache.

In Figure 12 we present speedups of the Request/Response and Push-mode DIVA over the single node alternative (the memory capacity of the nodes is scaled in relation to the number of nodes used in DIVA). In general, both DIVA implementations perform very well surpassing the performance of the single node considerably with a speedup of up to 2.98 (SP, 4 nodes) in three out of four benchmarks. In the case of the IS benchmark Request/response DIVA performs worse, while Push-mode DIVA provides a small speedup. It is interesting to note the correlation with external traffic: for the IS benchmark DIVA even *with* caches exhibits considerable traffic (Figure 11). The relative performance of the Request/Response DIVA and the Push-mode DIVA follows two patterns: (i) in the case of BT and IS, Push-mode consistently outperforms Request/Response, (ii) for FT and SP, Push-mode starts slower in small systems and improves over the Request/Response in systems with more nodes.

Comparing DIVA and the single-node all-cache case with an hypothetical system where the whole application would fit in the memory of a single node also leads to interesting conclusions: because of inter-node parallelism DIVA can outperform the hypothetical case (Figure 13) with a speedup of up to 1.86 (SP, 4 nodes). Of course, the all-cache single node always performs worse than the hypothetical case.

4.7 Sensitivity Analysis

In this section we explore the effect of memory and network latency on the performance of the DIVA systems and the single node all-cache system. Figure 14 shows how the execution time changes when we vary memory latency from 2 to 32 processor cycles. Remarkably, the change in performance is minimal especially for the DIVA systems. Execution time does not change much because of the high-performance memory system inside the nodes. Memory accesses

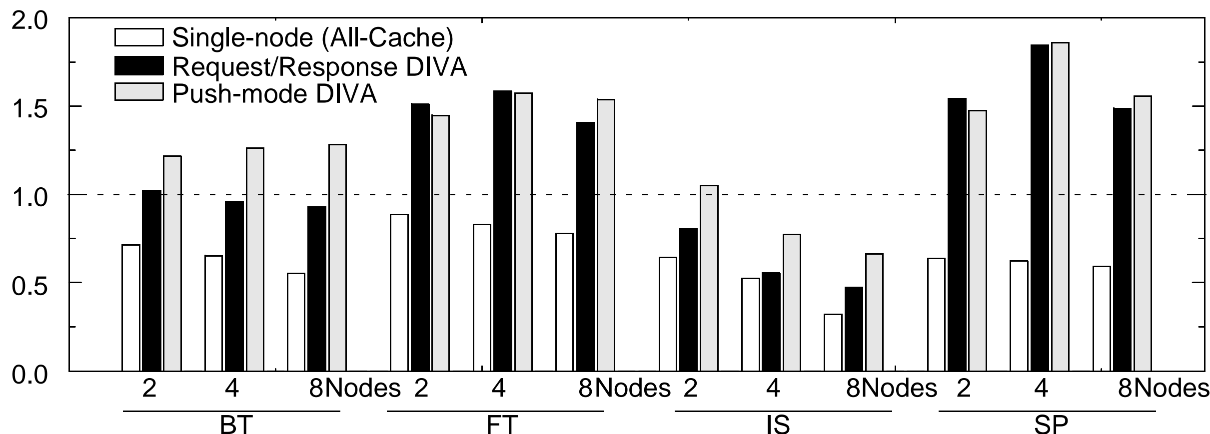

FIGURE 13. Relative performance of the DIVA systems and the single-node all-cache system normalized to the performance of the hypothetical case when applications would fit entirely in a single node.

are serialized only on bank conflicts; otherwise they are pipelined and vector loads see only the latency of one access plus a number of cycles equal to the vector length. Furthermore, internal interleaving provides good bank behavior and few conflicts.

Figure 15 shows the change in execution time varying the bus latency to transfer 128 bits from 2 to 32 processor cycles. Bus latency has a much more pronounced effect on performance since the bus serializes external traffic. The effect on performance also depends on the amount of external traffic. The cases that exhibit the most external traffic are hurt the most from increased network latency. Finally, Push-mode DIVA is affected less than the other two systems from increased bus latency. This makes it an appealing choice with a slow network.

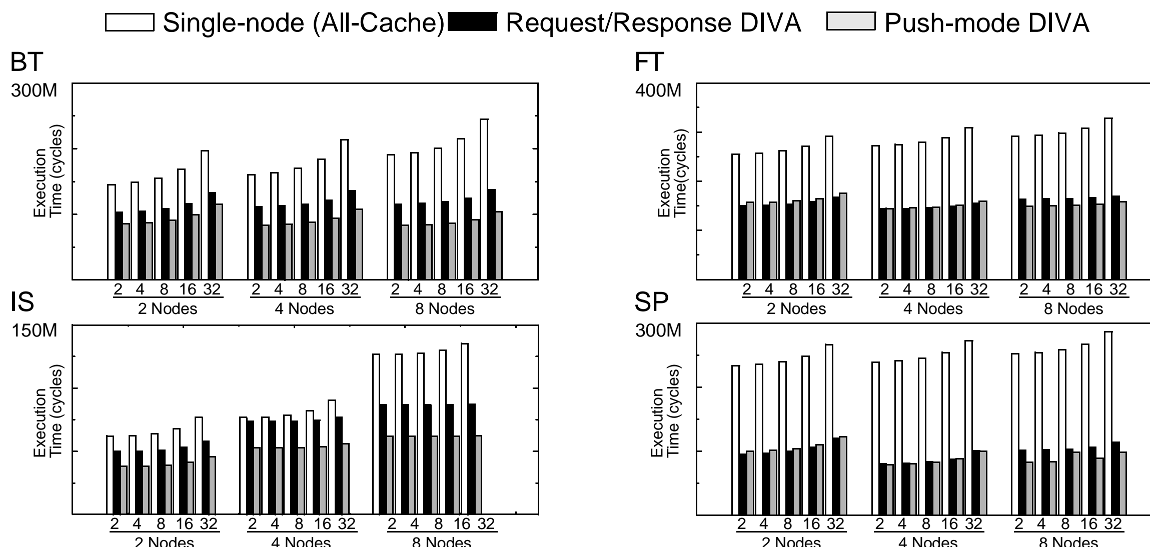

FIGURE 14. Performance of the Request/Response DIVA, Push-mode DIVA and single-node varying memory latency from 2 to 32 processor cycles.

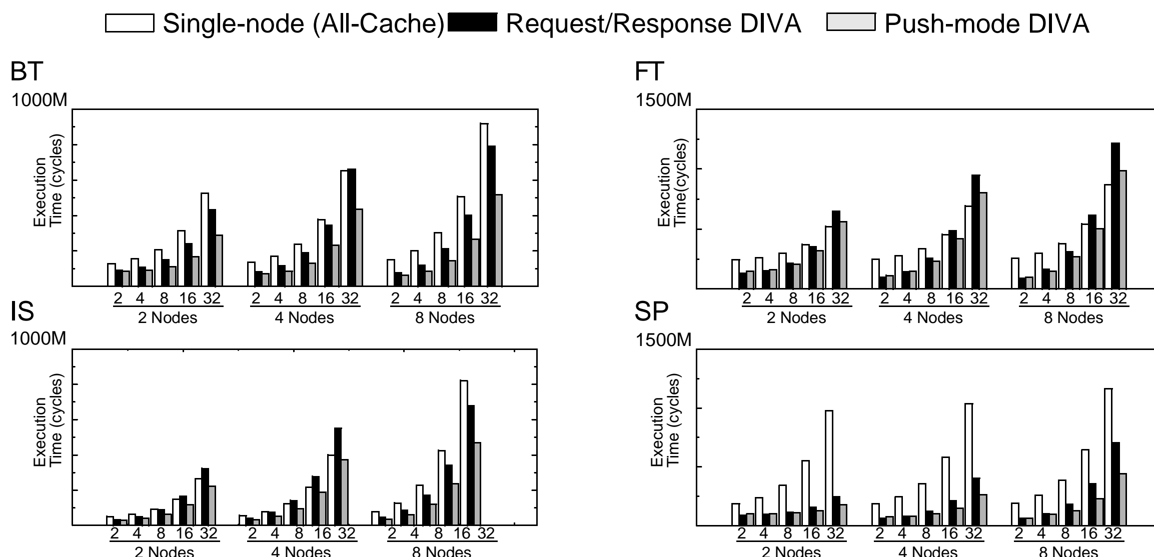

FIGURE 15. Performance of the Request/Response DIVA, Push-mode DIVA and single-node varying network latency from 2 to 32 processor cycles

4.8 Scalability

We conclude the evaluation section with a discussion of scalability. Although DIVA scales from 2 to 4 nodes and occasionally to 8 nodes showing increased speedups, our objective here is not to run the same (fixed size) application on more and more nodes in hope that we will get more speedup, but rather to use the minimum number of nodes able to hold the application dataset. In other words the question here is ***not*** "how an *X* node DIVA compares to a *2X* node DIVA?" since one cannot run the same applications as the other. Rather, the question is how an X node DIVA compares to alternative systems. We have shown that DIVA outperforms other alternatives. The fact that all systems—DIVA based or cache based—eventually perform worse as the memory capacity of the nodes becomes a smaller fraction of the application size, is the price we have to pay to run large applications. We have shown that this price is smaller for DIVA.

5 Conclusions

In this work we propose an architecture based on multiple IRAM nodes that can execute efficiently large-scale, scientific vector applications. We parallelize and dynamically map the computation of vector instructions across the nodes. The reduced traffic of DIVA, coupled with inter-node parallelism, and the very high expected performance of highly integrated processor/memory nodes (IRAM being a specific example) can potentially yield significant performance advantages for scientific computation. The implications of this are numerous including desktop supercomputers and cost-effective large-scale shared-memory machines with vector capabilities. However, significant work needs to be done to explore techniques for data optimization (data placement and memory vector alignment), and explore the capabilities of the compiler. These are promising areas for additional performance increases.

We present an extensive analysis of the external traffic of DIVA systems and we show that: (i) computation mapping

provides significant reductions in external traffic, (ii) compiler optimizations and caching can reduce DIVA's traffic further but they are not critical to its performance, (iii) DIVA generates less traffic than either single-node or multinode systems. Our timing simulations show that the two implementations of DIVA we propose: (i) execute up to 2.98 times faster than the best single-node all-cache system, (ii) can even outperform by a factor of 1.86 the hypothetical case where applications would fit in a single node.

6 References

[1] David H. Bailey et al., "The NAS Parallel Benchmark: Summary and Preliminary Results." *IEEE Supercomputing '91*, pp 158-165. November 1991.

[2] Doug Burger, Stefanos Kaxiras, and James R. Goodman, "DataScalar Architectures," In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[3] Doug Burger, James R. Goodman, and Alain Kägi, "Memory Bandwidth Limitations of Future Microprocessors." In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[4] Lucien M. Censier and Paul Feautrier, "A New Solution to Coherence Problems in Multicache Systems." *IEEE Transactions on Computers*, Vol. 27, No. 12, pp. 1112-1118, December 1978.

[5] Greg Faanes, "A CMOS Vector Processor with a Custom Streaming Cache" In *Proceedings of Hot Chips 10*, August 1998.

[6] Hector Garcia-Molina, Richard J. Lipton, and Jacobo Valdes, "A Massive Memory Machine." *IEEE Transactions on Computers*, C-33(5):391–399, May 1984.

[7] James R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", In *Proceedings of the 10th International Symposium on Computer Architecture*, June 1983.

[8] Stefanos Kaxiras, Rabin Sugumar, Jim Schwarzmeier, "Distributed Vector Architecture: Beyond a Single Vector-IRAM," *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, Denver, Colorado, June 1, 1997. (http://iram.cs.berkeley.edu/isca97-workshop)

[9] Stefanos Kaxiras and Rabin Sugumar, "Distributed Vector Architecture: Fine Grain Parallelism with Efficient Communication," *University of Wisconsin-Madison, Dept. of Computer Sciences, TR-1339*, February 1997.

[10] Corina Lee, presentation in *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, Denver, Colorado, June 1, 1997. (http://iram.cs.berkeley.edu/isca97-workshop)

[11] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The cache. *IBM Systems Journal*, 7(1):15-21, 1968.

[12] David Patterson, Tom Anderson, and Kathy Yelick, "The Case for IRAM." In *Proceedings of HOT Chips 8*, August 1996.

[13] David Patterson et al., "The case for Intelligent RAM," *IEEE Micro*, Vol 17, No. 2, March/April 1997

[14] Stylianos Perissakis et al. "Scaling Processors to 1 Billion Transistors and Beyond: IRAM", *IEEE Computer Special Issue: Future Microprocessors - How to use a Billion Transistors*, September 1997.

[15] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration." In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[16] Rabin A. Sugumar and Santosh G. Abraham "Set-Associative Cache Simulation Using Generalized Binomial Trees," *ACM Transactions on Computer Systems*, Vol. 13, No. 1, February 1995.

[17] Uri Weiser, "Intel MMX Technology-An Overview," in the *Proceedings of the Hot Chips 8*, August 1996.

[18] William A. Wulf, Sally A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News,* Vol 23, No. 1, March 1995.

7 APPENDIX I: Caching in Push-Mode DIVA

This section describes an implementation of caching in Push-Mode DIVA. The ideas described here are also applicable to the DataScalar architecture replacing its *cache correspondence* scheme. This discussion serves to demonstrate that caching in Push-Mode *can* be done rather than it *must* be done. Since caching in the Push-mode incurs non-trivial complexity, its applicability is constrained to extreme cases (e.g., cases with very many nodes or a very slow node interconnect).

Interaction with caches in Push-mode DIVA is complicated because: (i) the notion of local data now includes cached data that can be loaded on local registers but not forwarded to other nodes, (ii) each node needs information about what local data are cached remotely to avoid superfluous transfers. Here, we describe a directory-based scheme for Push-mode caching.

We assume that each node has a cache for external data. When a node executes a vector load instruction data are either in the node's local memory or present in the cache for external data. However, only data in the node's memory can be pushed to other nodes. Each node keeps a directory for its local memory data blocks. The directory performs the same basic function as the directories of DSM systems, i.e., it keeps track of cache copies that are given to other nodes [4]. When a data block is written an invalidation or an update protocol can be used to keep the distributed caches coherent.

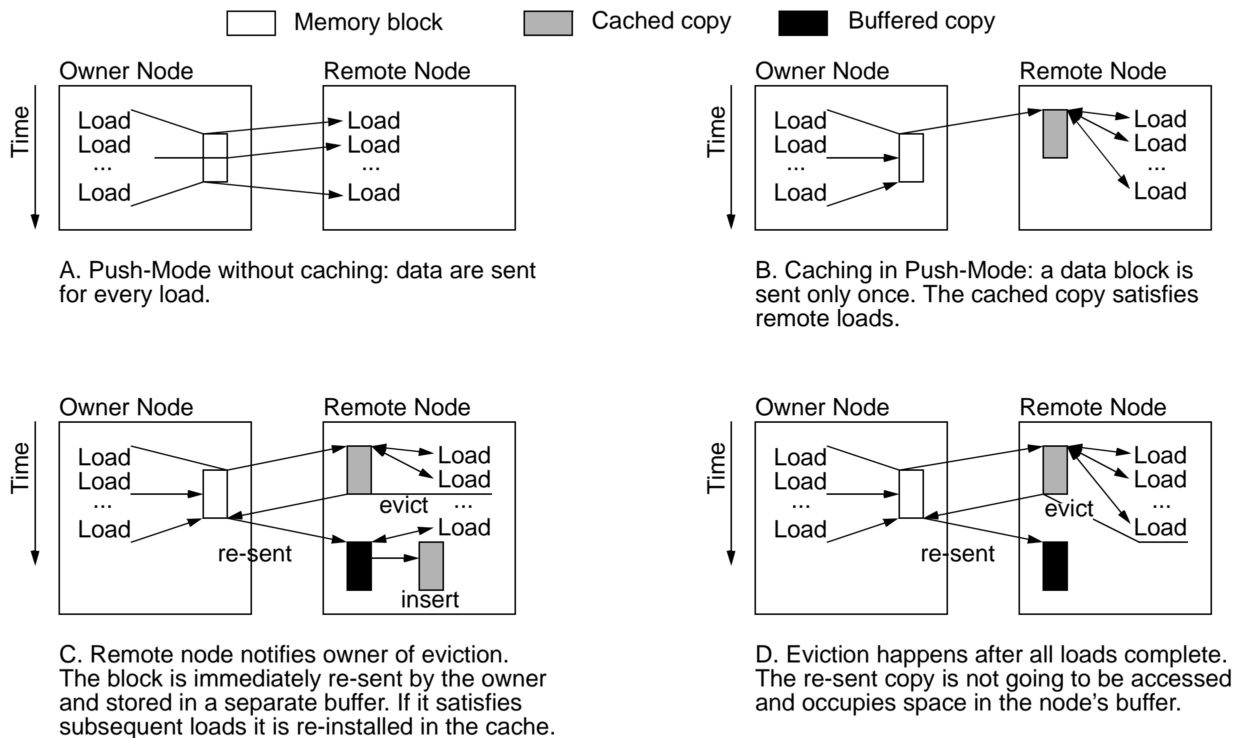

FIGURE 16. Caching in Push-Mode DIVA

In our directory scheme cached copies cannot be discarded without a notification of the home node. This is necessary

because, if the home node is always responsible to send the data, it must know at all times whether remote nodes have them or not. On the other hand, a remote node cannot make a decision whether to wait for data or to request them, since it does not know what the home node might do.

The operation of the directory is as follows: Whenever the home node pushes data to a remote node, it updates the directory entry for the data block and records the fact that a remote node has a copy. Subsequent accesses to a data block, do not push data to nodes that already have a copy but update the directory entry to record the fact that one or more load accesses happened after data were pushed to remote node(s) (Figure 16 B). The nodes that have cached data are responsible to notify the home node whenever they evict a block from their cache. When the eviction notice arrives at the home node there are two possibilities:

1. The home node did not access the data block after the copy has been given to the evicting node; in this case there home node just erases the evicting node from its directory entry (subsequent accesses to the data block will send a new copy).
2. The home node accessed the data block after the copy was given to the evicting node. In this case a new copy is send immediately to the evicting node since the home node conservatively assumes that the evicting node has performed fewer loads and it will need the data again. The new copy is kept in a separate buffer in the evicting node until it is consumed by a load and reinserted in the cache or it is known that it is not needed anymore (Figure 16 C and D).

In the second case (Figure 16 D), the problem with the senders' conservative assumption (that the evicting nodes will need the data again) results in situations where data are re-sent but never consumed. Since, in reality, we do not have infinite buffers, extraneous copies will eventually cause overflows. To solve this problem, we need synchronization points where we know that the same number of loads have been performed in both the home node and the remote node. Ordinary synchronization points that guard the end of vector store instructions in the application can serve this purpose. However, program synchronization points can be arbitrarily far apart, thus making the size of the buffer arbitrarily large. To maintain reasonable size buffers the nodes perform garbage collection on their buffers. The garbage collection is based on the observation that a copy can be discarded at the point when the node completes the same instruction as the home node did when it sent the copy. To implement this scheme, data sent in response to an eviction are tagged with the number of completed instructions in the owner node. The remote node keeps the data in its buffer until it completes instructions beyond the data tag. Since nodes do not usually run arbitrarily ahead of each other (many of their operations are interrelated) this scheme allows reasonable sized buffers with very low overflow probability. As a last resort, the uncommon case of an overflow corresponds to a system-wide synchronization, where the overflowing node, clears its buffer, stops all nodes that run ahead and reverts to the request/response model to catch-up.