Distributed Vector Architecture: Beyond a Single Vector-IRAM

Stefanos Kaxiras†, Rabin Sugumar‡, James Schwarzmeier‡

† Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St.
Madison WI, 53705
kaxiras@cs.wisc.edu

‡ CRAY Research/Silicon Graphics
900 Lowater Rd.
Chippewa Falls, WI 54729
{rabin, jads}@cray.com

Abstract—The integration of memory on the same die as the processor (IRAM) has the potential to offer unprecedented bandwidth that can be exploited efficiently by vector processors. However, real-world scientific vector applications with their very large memory requirements and their poor locality, would easily overflow any single IRAM device. In this environment, traditional approaches such as caching or paging generate considerable traffic, diminishing the performance advantage of processor-memory integration. To exploit the full potential of IRAM in the realm of large-scale scientific computing, we propose a DIstributed Vector Architecture (DIVA), that uses multiple vector-capable IRAM nodes in a distributed shared-memory configuration. The advantages of our approach are twofold: (i) we speed up the execution of the vector instructions by parallelizing them across the nodes, (ii) we reduce external traffic, by bringing computation to data rather than data to computation. We dynamically map the computation of individual vector instructions on nodes to coincide, to the extent possible, with the corresponding data in memory. As an implementation, we propose a mechanism to assign at run-time elements of the architectural vector registers on nodes, using the layout of data in memory as a blueprint. Using traces of vector supercomputer programs we demonstrate that DIVA often generates considerably less external traffic compared to single or multiple-node alternatives that are based solely on caching or paging. Considerable performance gains are then possible because of DIVA's inter-node parallelism.

1 Introduction

While microprocessors follow an explosive growth in performance, DRAM-based memory systems fall behind creating the infamous memory wall [5,9]. Integration of main memory on the same die with a microprocessor (IRAM) promises a high-performance yet inexpensive memory system [1,2,7]. Through the elimination of the pin interface, IRAM 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.
- A reduction of the memory access latency (tenfold decrease over current workstation memory latency) following the elimination of crossing chip boundaries.

An inexpensive, high-performance memory system, coupled with the need for a processing core that can translate bandwidth into performance, is a compelling reason for implementing a vector supercomputer on a chip [10,2]. Vector units have demonstrated an excellent ability to exploit high bandwidth. This is because of their very efficient instruction issue and because they can be implemented with deep pipelines. Their execution can also be parallelized using multiple "pipes," i.e., different pipelines operating concurrently. Furthermore, vector units represent a well-understood technology, with relatively simple implementations, that is backed by mature compiler support.

Although the marriage of vector units and IRAM at first seems idyllic, considering the application domain of scientific vector applications, where vector units traditionally have thrived, and the limited, non-expandable nature of the on-chip memory, some signs of disagreement arise. Scientific vector applications are memory intensive —codes such as weather prediction, crash-test simulations, or physics simulations run with huge data sets— and they would overflow any single device with a limited and non-expandable memory[1]. Such applica-

[1] There are, however, some applications such as multimedia and cryptography that can be vectorized and have moderate memory requirements [2]. These applications could fit in a single IRAM. However, in this work we concentrate exclusively on large scientific applications.

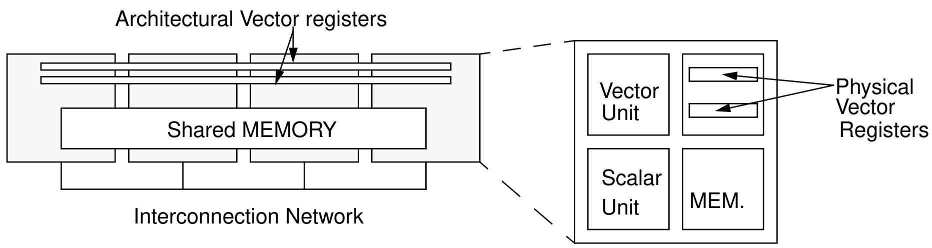

FIGURE 1. DIVA **system comprised of four nodes. Each node contains some part of the system memory along with a processor and a vector unit. In this example each node has two physical vector registers. An application running on all four nodes (occupying all their memory) refers to two architectural vector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

tions inevitably require external access. The problem is magnified by the increase of the relative cost of external accesses, since processor-memory integration makes on-chip accesses much faster. Providing an expensive external memory system to speed up external accesses would invalidate the cost-performance advantage of IRAM. Paging to external memory also leads to excessive traffic (thrashing), if the working set of the application does not fit in the device (see Section 3). The applications in question have poor locality and, frequently, their working sets represent a significant part of their full dataset. Using the on-chip memory as a huge cache for external memory could help alleviate the cost of external accesses, but still, the caching behavior of the target applications generates considerable external traffic (see Section 3).

In this paper, we attack the problem of running large vector applications by employing multiple Vector IRAM nodes. Not only do we distribute the dataset of the application over the memory of the nodes, but we also distribute the computation of the vector instructions across the nodes (i.e., we parallelize individual vector instructions). A simplistic approach is to statically assign the computation of specific vector elements on specific nodes. This would generate excessive traffic, since data would have to be shipped to the arbitrary node where the computation is taking place. Instead, we continuously re-assign element computation on nodes, attempting to put it where the data are. Of course, data movement is still necessary, since operating on two elements residing in different nodes requires at least one data movement to bring them together. Despite the distributed nature of our approach, we show that for our target work-load, the NAS benchmarks [3], it actually has less external traffic than other centralized (single-node) or distributed (multiple-node) approaches. In the rest of this paper we describe the principles of the architecture and we present one possible implementation (Section 2). In Section 3 we present an evaluation of this implementation. Finally, we conclude in Section 5.

2 DIVA

In Figure 1 we show a DIstributed Vector Architecture (DIVA) based on a collection of IRAM nodes with vector capabilities. The nodes are connected together with an interconnection 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. 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 happen to be present in their physical registers). Since vectorizing compilers guarantee the independence of the computations within a vector instruction, nodes work concurrently on their own part of the vector instruction, providing a speed up proportional to their number.

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. Since our goal is to bring the computation to data rather than data to the computation, we propose a dynamic, program controlled mapping of architectural elements to physical elements. By mapping architectural elements to coincide in the same nodes with the corresponding data in memory, we effectively map the computation of the vector instructions to reduce traffic.

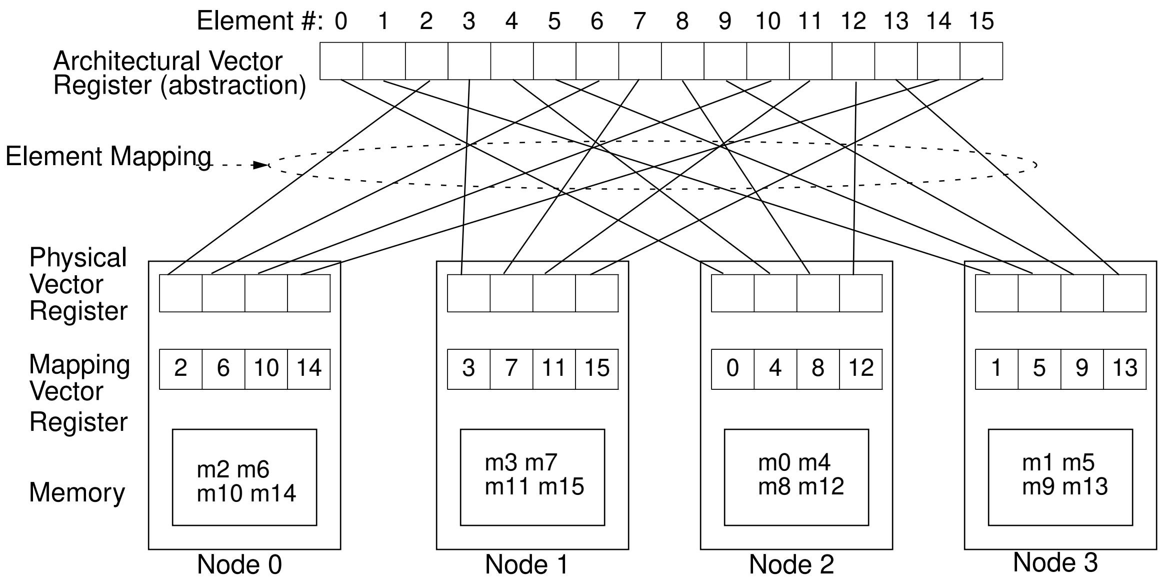

FIGURE 2. An element mapping of architectural to physical vector elements.

In Figure 2, we show a mapping of the elements of an architectural vector register to the elements of the physical vector registers. In this figure, the architectural element 0 is assigned to node 2, element 1 to node 3, etc.... We apply this mapping when element 0 is loaded with data (m0) present in node 2's memory, element 1 loaded with data (m1) from node 3's memory, and so on. Loading this architectural vector register would thus require no external traffic. Unfortunately, we have to map multiple architectural vector registers that take part in the same computation in exactly the same way, otherwise their corresponding elements will not align properly in the nodes. The best mapping for loading an architectural vector register with a memory vector, is not always compatible with the best mapping for loading another register with a different memory vector. If these two architectural vector registers are to be part of the same computation (e.g., added or multiplied together), we have to select one of the mappings and forgo the other (or even forgo both and use a third mapping). Data placement techniques can help align memory vectors used in the same computations and thus reduce mapping conflicts.

Mapping the computation of vector instructions among the nodes is possible because in DIVA the architectural elements do not have a predefined fixed mapping. Mappings can be generated statically at compile-time if sufficient information is available, or at run-time, in which case appropriate mechanisms are needed. In the next section we describe a DIVA implementation based on mechanisms to dynamically generate element mappings. As we show in Section 3, this implementation performs well with minimal, or even without any compile-time support.

2.1 An example DIVA implementation

The emphasis for the DIVA implementation we discuss in this paper, is on distributing vector computation dynamically in hardware. A simplistic distribution of the application dataset is provided by block-interleaving the memory across the nodes. Each node executes scalar instruction redundantly, maintaining its own scalar register set, similarly to the Massive Memory Machine [6] or the DataScalar architecture [4]. Alternatively a SIMD configuration is possible, where only one master node executes scalar instructions and broadcasts vector instructions to slave nodes so they can executed them in parallel. The parallelization of vector instructions is based on two mechanisms: (i) a vector instruction, called SETMV that generates element mappings on the fly, and (ii), special vector registers, called *mapping vector registers,* that enforce the element mappings generated by the SETMV.

SETMV instruction and Mapping Vector

As we have mentioned previously, architectural vector registers that participate in the same computation must be mapped on physical registers in exactly the same way. Before we load or initialize any register of a com-

```
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.

putation slice (a group of related instructions such as those in Figure 3, that load some registers, compute on them and store results), we must have an element mapping. This is the job of the compiler if we rely on static, compile-time techniques or a run-time mechanism such as the SETMV instruction we describe here. A SETMV instruction, therefore, precedes every computation slice (Figure 3).

SETMV creates an element mapping using as a blueprint the layout of a memory vector referenced in the computation slice. In the example in Figure 3, the SETMV instruction uses the memory vector accessed in the first vector load. SETMV is the only vector instruction that runs in is entirety in every node. A node executing this instruction generates the addresses of all the elements of the memory vector and uses a run-time locality test to determine which of these addresses are local to it. Architectural elements are assigned to the node where the corresponding memory element is local.

The semantics of the SETMV instruction also handle, in a distributed fashion, cases where an imbalance in the number of elements local to a node would overflow the physical vector registers with more architectural elements than they can handle. Each node keeps track of the number of elements assigned to other nodes using a counter per node. When a counter exceeds the size of the physical vector registers (counter overflow) the corresponding node is full. Responsibility for the extra elements in an overflowing node 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 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 same memory vector accessed in the first vector load. Chaining these two instructions allows the addresses generated by the SETMV, that correspond to locally assigned elements, to be used by the vector load. The cost of generating extraneous addresses (of non-locally assigned elements) can be largely hidden by the relatively longer memory accesses of the vector load.

A *mapping vector* represents the element mapping generated by a SETMV. Similarly to the architectural vector registers, a mapping vector is distributed in *mapping vector registers* across the nodes (Figure 2). The execution of the SETMV instruction in a node results in setting a mapping vector register to contain the architectural element numbers assigned to the node (in the example of Figure 2, the mapping vector register in node 0 contains the architectural element numbers 2, 6, 10, and 14). We use a small number of mapping vectors to accommodate multiple independent computation slices that are interleaved in the instruction stream.

In each node, a mapping vector register tailors the behavior of the corresponding vector load (store) instructions specifically to the subset of the architectural elements assigned to the node. To execute a vector load (store), a node consults its appropriate mapping vector register and loads (stores) only the elements described therein. Other computations (e.g., addition, multiplication, etc.) are not affected by the mapping vectors and proceed at full speed as soon as the physical vector registers are loaded[2].

Compiler involvement

We have examined the compiler involvement in selecting the memory vector that SETMV instructions use as a blueprint to generate element mappings. The default case in our evaluation is that the compiler does not have enough information to select intelligently which of the memory vectors of a computation slice should 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).

[2] Full-empty bits in the physical elements can be used to signal completion of memory accesses.

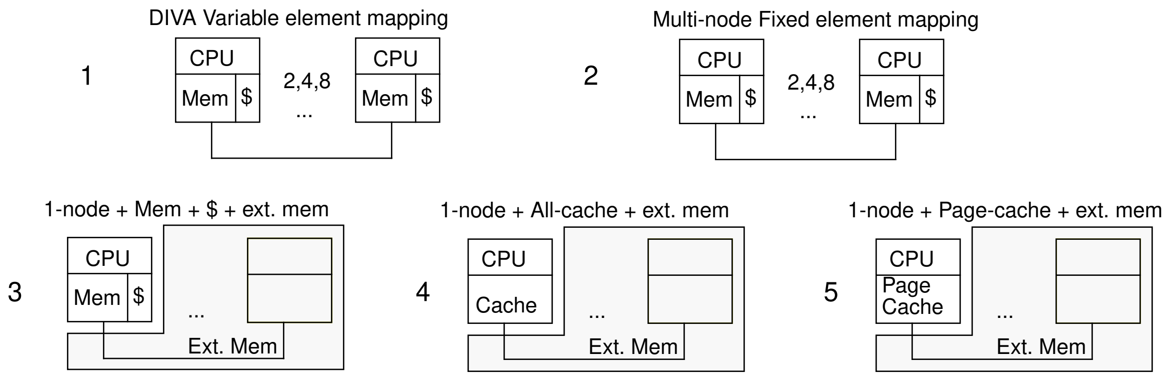

FIGURE 4. Comparison Systems

"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 home node for all its elements according to the run-time memory interleaving. We then compare the home nodes of each vector load/store to the home nodes of all the other vector loads/stores and we select the one with the most matches.

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.

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, it is a difficult and application dependent problem, although techniques have been developed for, and used in SIMD computers. In DIVA, the ability to map the computation of vector instructions gives us a new degree of freedom that can simplify data optimization. This is an area for further study.

As of yet, we have only studied 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.

As for aligning memory vectors, we have only scratched the surface of the problem, experimenting with limited, simple source code transformations. These transformations affected solely the allocation of data structures and involved changing some array dimensions to powers-of-two. 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.

For the evaluations in the following section we assume a segmented memory space. Applications fit 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 the 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). However, in the evaluation of Section 3, we report results for a single interleaving per application.

3 Evaluation

DIVA's performance is based on two factors: (i) the parallel execution of vector instructions across the nodes, (ii) the low external traffic from controlling the mapping of the vector computation to the nodes. As a preliminary evaluation, we set out to examine how the external traffic of a DIVA system compares to other alternatives. External traffic is a critical measure, since if it were excessive it would invalidate the parallelism advantage of DIVA. At this stage, we are only concerned with the characteristics of the memory organization that affect external accesses and not with the details of the internal memory hierarchy that affect the performance of the vector units. We used trace driven simulation to evaluate the DIVA implementation. Traces for 6 vectorized NAS benchmarks (BT, FT, IS, LU, MG, SP)[3], that represent scientific codes, were collected on a CRAY supercomputer with 128-element vector registers. Memory interleaving was partially guided by the heuristic mentioned in Section 2. Here, we report results for the best interleavings we have found. Still, data distribution by interleaving is a crude method and significant improvements are possible in this area.

We compare, in terms of traffic, five alternatives based on IRAM nodes (Figure 4):

1. A multi-node DIVA system. The application is interleaved across the nodes. Each node contains a small cache for external data. This cache is 1/16 of the DRAM memory capacity of the node and provides on average a 27% decrease of the external traffic, over DIVA without any cache. The cache is optimized for traffic and it is 2-way set-associative, 1-word block, write-back, write-validate. Source code transformations to affect the alignment of memory vectors were applied only in this case.
2. A multi-node shared-memory system without variable mapping of the architectural vector registers. Again, each node holds a part of the application dataset and includes a small cache for external data (with the same specifications as in the previous case). This system is inherently traffic-intensive and caching proves quite effective, providing a 40% decrease of the external traffic over the same system without caches.
3. A single-node system with a statically allocated part of the application on the on-chip memory and an cache for external data. The cache is 1/16 of the DRAM memory capacity of the node and has the same specifications as in the first case.
4. A single-node system that uses its on-chip memory as a huge cache. The size of the cache is equal to the DRAM memory capacity of the node. The cache is 2-way, 4-word block (to reduce tag overhead), sectored with 1-word sub-blocks to reduce traffic [8], write-back, and write-validate. Although this system may not be realistic (because of tag storage overhead and tag access overhead), it represents a “lower bound” for cache-based systems.
5. A single-node system with demand-paging to external memory. 4K pages are swapped in and out of the on-chip memory. This system represents another point the spectrum of cache-based systems with fairly large, fixed-size cache blocks (pages) and full associativity. Possibly, a practical implementation of a cache-based system would fall between cases 4 and 5.

We examined DIVA 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 IRAM node is 1/4 of the application size). Since a node cache is always 1/16 of the node memory, the total cache capacity for the multi-node systems is always 1/16 of the application size. We scale the memory and the cache of the other four alternatives in the same way so we always compare systems built with identical (equal memory and cache capacity) IRAMs.

For the DIVA system, alignment of memory vectors, was affected by simplistic changes in the allocation of data structures in three of the six benchmarks (BT, LU, and SP). These changes yield an average reduction in external traffic for 2,4, and 8 nodes of 60% for BT, 24% for LU and 66% for SP. For the results we present here, we assumed that the compiler, lacking compile-time information, blindly selects the first accessed memory vector of each computation slice to use in the SETMV instruction (“first choise” selection). We have found, however, that using “best choise” selection of memory vectors, reduces external traffic by up to 22% (15% on average). DIVA traffic with an actual compiler should fall between the “first” and “best” case.

Table 1 shows the external traffic of the five systems, as a percentage of the total traffic required by the application to move data between the memory and the vector registers. The same results are plotted in Figure 5.

Four of the benchmarks (BT, IS, LU, and SP) thrash when they run in demand-paging IRAMs with a memory capacity 1/4 or 1/8 of the application size. 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 5) is also very competitive, yielding 29% less traffic than a multi-node 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.

	Nodes	System: DIVA	Multi+$	1+M+$	1+All $	1+Page
BT	2	7.28	25.92	12.47	7.98	76.91
	4	14.21	45.52	22.55	20.35	189.80
	8	21.23	59.13	29.01	25.65	213.15
FT	2	8.40	24.94	13.19	13.89	14.20
	4	14.52	40.03	19.27	18.27	23.21
	8	21.30	51.82	46.08	26.11	28.76
IS	2	24.27	37.90	29.32	6.26	14.28
	4	38.10	58.53	58.55	49.51	78.34
	8	48.39	69.76	69.87	73.58	970.89
LU	2	15.60	20.04	17.63	7.87	511.23
	4	28.52	36.84	29.13	17.59	650.88
	8	38.62	55.19	38.72	27.47	825.10
MG	2	7.53	11.81	9.14	14.51	18.47
	4	13.84	18.72	15.38	19.34	30.27
	8	22.28	24.84	25.86	21.45	34.64
SP	2	3.82	26.71	12.65	4.22	5.63
	4	8.09	48.44	34.59	13.78	28.41
	8	16.22	66.00	55.53	29.48	137.36

Table 1: External traffic (% of total) of the five systems for the 6 NAS benchmarks

3.1 Scalability issues

We expect DIVA to scale to a small number of nodes (possibly up to 16). The results in Table 1 (also in Figure 5) show that DIVA's average traffic for the 6 NAS benchmarks increases from 11% for two nodes, to 19% for four nodes, to 28% for eight nodes. This traffic increase, coupled with the decrease in the amount of work per node, limit the obtainable speed-up over other systems.

These results are for a fixed architectural length of 128 elements regardless of the number of nodes. This is a limitation of our trace-driven approach since we used traces for 128-element vectors for all node configurations. In an actual DIVA system the architectural vector length would be a function of the number of nodes. This can make a difference for the applications that can use larger vectors. However, for some applications the vector length may not scale with respect to the dataset size. For example, in some of the NAS benchmarks the dataset size is a function of all the dimensions of the program's multi-dimensional arrays while the vector length is a function of only one dimension.

For the DIVA implementation presented in this paper, a factor that limits its scalability to arbitrary number of nodes is the SETMV instruction. First, SETMV is not parallelized and its execution for very large vectors eventually limits speedup. Second, SETMV semantics require a number of counters (one for every node in the system) which in turn implies a pre-defined upper limit in the number of nodes. However, we believe that SETMV is not a bottleneck for the vector lengths of ordinary applications and for the range of nodes where DIVA is applicable. Furthermore, the cost of SETMV is amortized over a computation slice. Other approaches to create element mappings are possible, instead of the SETMV instruction. For example, in each node special instructions can modify the base address and stride of vector loads/stores to access only local elements [11]. However, such an approach would introduce overhead for every vector load/store instruction which may not be a desirable trade-off for small number of nodes and small to medium vector lengths.

4 Related work

Researchers of the IRAM group at U.C. Berkeley have proposed vector units for IRAM and are exploring various implementation issues of Vector IRAM chips [2]. The emphasis of this work is on applications that fit in the memory of an IRAM node.

The DataScalar architecture (developed by Burger, Goodman and one of the authors) [4], is an architecture that uses multiple nodes to execute serial programs that do not fit in the memory of one of the nodes. 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 DataScalar allows code to be executed only by a subset of the nodes [4]. Results generated by this code are broadcast as required so they are visible to the rest of the nodes. Locality tests determine where code is executed. Using result communication, a DataScalar system (based on superscalar nodes) can emulate a DIVA system. Loops that would correspond to a series of vector instructions in DIVA systems can be executed in parallel using result communication. Loop iterations start with a locality test that replaces the corresponding element locality test of the SETMV instruction. The locality test assigns each iteration to a specific node in the same way SETMV assigns architectural elements to nodes.

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 can potentially yield significant performance advantages facilitating the acceptance of IRAM for scientific computation. The implications of this are numerous including desktop supercomputers and cost-effective large-scale shared-memory machines with vector capabilities.

Significant work needs to be done to further evaluate DIVA, explore techniques for data optimization (data placement and memory vector alignment), and explore the capabilities of the compiler. We are currently in the process of developing timing simulations of DIVA systems.

At the same time we are examining alternative DIVA implementations based on different schemes for element mapping. To further reduce external traffic, we pursue a promising direction working on a push-mode DIVA where we eliminate requests by having each node discover what the other nodes need from its local memory and pushing the data to the appropriate destinations.

6 References

[1] David Patterson, Tom Anderson, and Kathy Yelick, “The Case for IRAM.” In *Proceedings of HOT Chips 8*, Stanford, California, August 1996.

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

[3] David H. Bailey et al., “The NAS parallel benchmark: Summary and Preliminary Results.” IEEE Supercomputing ‘91, pp 158-165. Nov., 1991.

[4] Doug Burger, Stefanos Kaxiras, and James R. Goodman, “DataScalar Architectures,” To appear in the 24th ISCA, June, 1997.

[5] Doug Burger, James R. Goodman, and Alain Kägi, “Memory Bandwidth Limitations of Future Microprocessors.” In *Proceedings of the 23rd ISCA*, pages 79–90, May 1996.

[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] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk, “Missing the Memory Wall: The Case for Processor/Memory Integration.” In *Proceedings of the 23rd ISCA*, May 1996.

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

[9] W.A. Wulf, S.A. McKee, “Hitting the Memory Wall: Implications of the Obvious,” *Computer Architecture News* Vol 23, No. 1, March 1995.

[10] 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

[11] Krste Asanovic, personal communication, March 1997

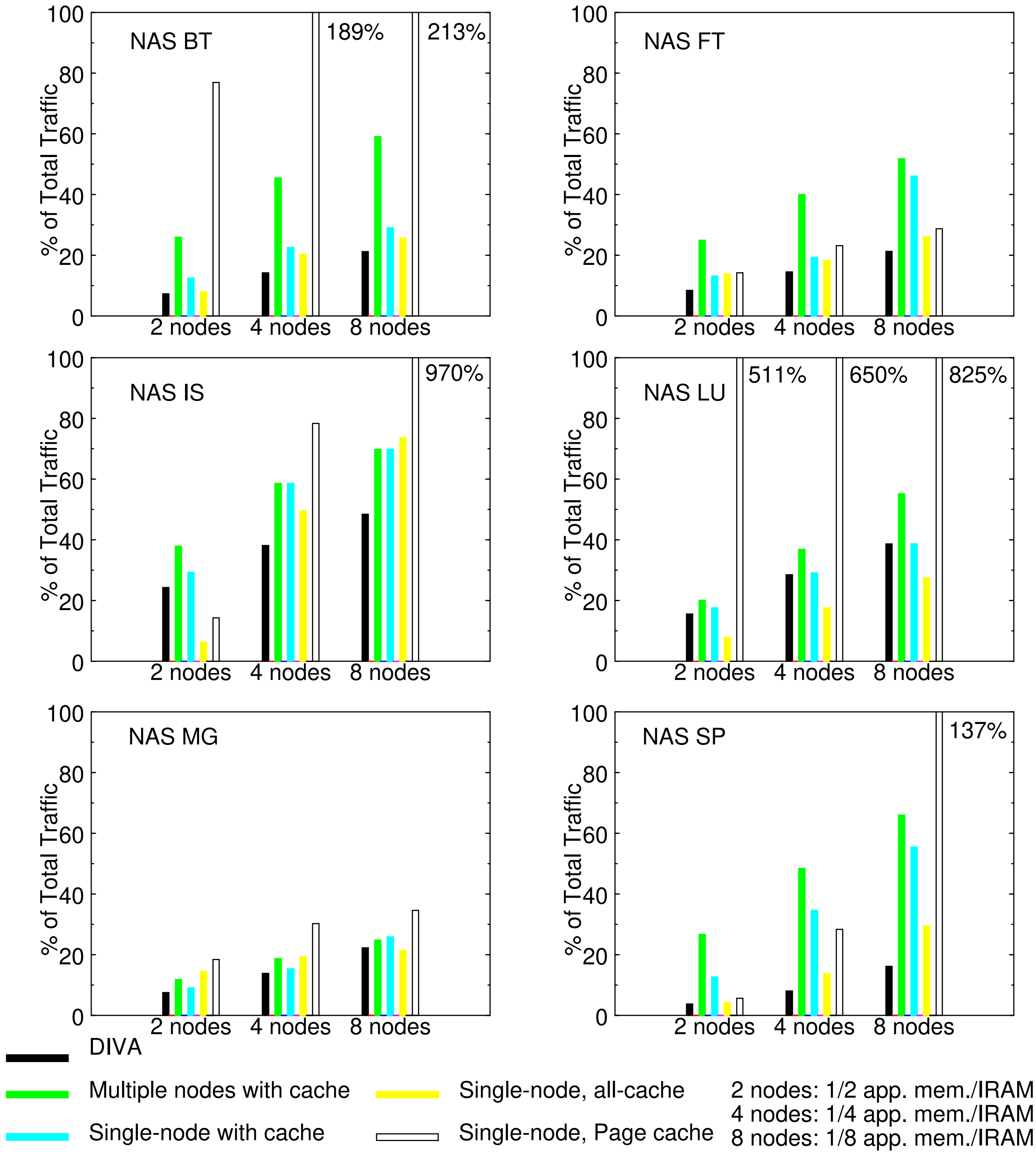

FIGURE 5. Traffic comparisons