UW CS TR-1339 Feb. 1997

Distributed Vector Architecture: Fine Grain Parallelism with Efficient Communication

Stefanos Kaxiras†, Rabin Sugumar‡

† Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St.
Madison, WI 53706
kaxiras@cs.wisc.edu

‡ CRAY Research
900 Lowater Rd
Chippewa Falls, WI 54729
rabin@cray.com

February 3, 1997

Abstract—As processing power continues to increase while memory access latency and bandwidth become serious bottlenecks, processors and DRAM memory will be packaged increasingly tighter together, possibly on a single chip. This integration would introduce orders of magnitude superior bandwidth/latency to local memory than to remote memory. In this situation, an on-chip vector unit is advantageous since it can make efficient use of such high internal bandwidth. However, real-life vector applications, which have enormous memory requirements, would not fit in the non-expandable memory of a single integrated device and their performance would be primarily determined by the amount of remote traffic they require. We propose a solution for running large vector applications on multiple, vector-capable, tightly integrated processor-memory nodes. Vector processors of individual nodes cooperate together to work as a single larger vector processor, while the vector application occupies the memory of all nodes. The physical vector registers of the nodes combine together to form larger architectural registers. Vector operations on the architectural registers are distributed among the nodes, each of which operates on its assigned elements. One of the novel contributions of our work is a variable, program defined, mapping of elements of the architectural vector registers to elements of the physical vector registers. This capability considerably reduces remote traffic for loading and storing physical vector registers to and from the distributed memory of the system. We introduce the notion of mapping vectors *to specify this variable mapping. We present heuristics for selecting traffic efficient mapping vectors and selecting appropriate memory interleavings. Simulation results show that the DIstributed*

*Vector Architecture (*DIVA*) we propose has the potential to result in lower remote traffic than other approaches.*

1 Introduction

1.1 Technology Trends

The ever-increasing gap between processing power and memory speed points to a future of higher integration of processors and DRAM memory [7,8,5]. This integration, which currently exists at the level of the printed circuit board (PCB), may be achieved by initially putting processors and memory on multi-chip modules (MCM) and eventually on the same chip. Such tightly coupled systems will offer two advantages: first, a substantial increase in the available bandwidth between the processor and its memory and second, a reduction of the memory access latency. The bandwidth advantage follows from the vastly improved ability to interconnect the processor with its memory banks and the latency advantage follows from the elimination of the overhead of crossing chip boundaries.

The improved memory bandwidth, coupled with the improvement of access latency, is an excellent opportunity to implement of on-chip vector units [7]. Vector units can exploit significant memory bandwidth because of their efficient issue and their ability to have deep pipelines. However, providing ample external bandwidth is expensive. This is evident in the design of CRAY vector supercomputers such as the C-90 and the T-90 that employ Static RAM (SRAM) and elaborate interconnection networks to achieve very high performance from their memory systems. With the integration of vector units and memory on the same device we can build systems with the potential for significantly lower cost-performance than traditional supercomputers. In this paper, we are not examining the details of building a such device but rather how to connect such devices together to form larger systems.

1.2 Distributed Vector Processing

The importance of vector processing in the high-performance scientific arena is evident from the successful career of the vector supercomputer. Vector processing is a good fit for many real-life problems. Its serial programming model is also popular among engineers and scientists since the burden of extracting the application parallelism (and hence performance) falls to the vectorizing compiler. This proven model, now in use for two decades, is supported by significant vectorizing compiler technology and accounts for a very important section of scientific computation.

However, 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. These applications would require external memory access. Furthermore, processor-memory integration will increase the relative cost of external accesses by making on-chip accesses much faster. Providing a very expensive external memory system to speed up external accesses, would invalidate the cost-performance advantage of such devices. Cache memory on the integrated device could help alleviate the cost of external accesses, but for a large class of vector applications caches are not quite as effective as in other application domains. In this work, we propose a way to address the problem of external accesses.

We propose an architecture based on a collection of highly integrated processor-memory devices with vector capabilities (Figure 1). We use the term *node* to refer to such devices. A number of such nodes are connected together with some kind of interconnection network —bus, mesh, ring, etc...). We focus on single-threaded vector applications (or to a single thread of a parallel application that is not amenable to further high level parallelization) whose memory requirements exceed the memory capacity of any of the nodes. The aggregate memory of the devices, however, satisfies the memory requirements of the application. No other additional memory is present in the system beyond that of the nodes.

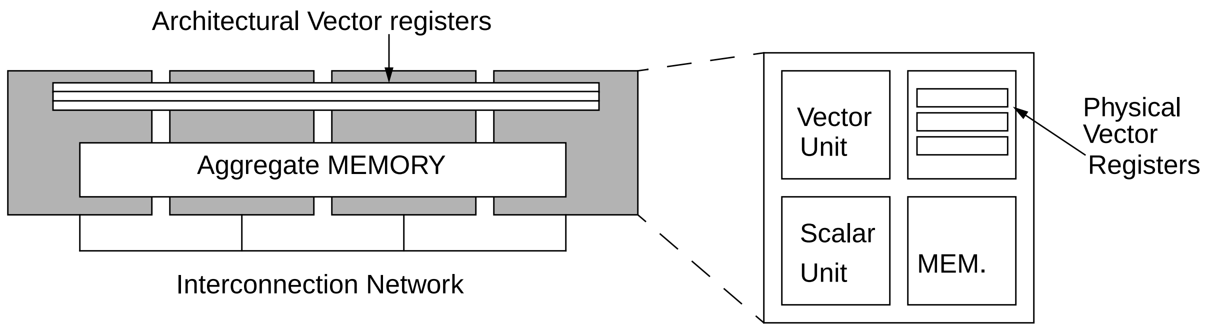


FIGURE 1. DIVA **system comprised of four nodes. Each node contains some part of the system memory along with a processor and a vector unit. In this example each node has three physical vector registers. An application running on all four nodes (occupying all their memory) refers to architectural vector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

The straightforward solution to execute an application that does not fit in one node would be to execute it on a processor of one node, but use memory on other nodes to hold its dataset. However, we propose to use the vector capability of all the devices to work simultaneously on the application. This provides two advantages: first, the aggregate vector power of the nodes can speed up vector instructions; second, external communication can be reduced by loading and storing vector elements locally on the nodes. In this model, the vector application references architectural vector registers that are formed by combining together the physical vector registers of the nodes (Figure 1). The length of the architectural vector registers depends on the number of nodes used by the application and the length of the physical vector registers in these nodes.

One of the main contributions of our work is a method of assigning elements of the architectural vector registers to the elements of the physical vector registers. In essence, we distribute the elements of the architectural vector registers around the nodes to increase the locality of vector loads and stores. *Mapping vectors* define the correspondence of architectural elements to physical elements. They are set at any instant by the application to reduce external communication. By applying heuristics to select mapping vectors, as well as heuristics to interleave the memory, we can achieve locality for vector loads and stores that leads to less remote communication than other approaches based on caches (simulation results are discussed in Section 3).

1.3 Paper Overview

In the next section we present a detailed description of DIVA and its execution model, focusing on mapping vectors. In the rest of Section 2 we discuss heuristics for mapping vector selection and memory interleaving selection. In Section 3 we describe our simulation methodology and present simulation results. We compare, in terms of remote traffic, DIVA to COMA [13,14] and to CC-NUMA models, where the application executes on one processor and remote memory is accessed as required. Section 4 concludes the paper with a summary and future work.

2 DIVA

2.1 Architecture and Execution Model

We conceive DIVA as an evolution of the scalable shared memory systems available today, e.g., the T3E [2,3], the Origin 2000 [11], or the Convex Exemplar [12]. These are clustered designs where a small number of processors are tightly coupled to a local memory, and these clusters are connected together through an interconnection network. For simplicity, in this work we are only looking at single processor nodes although similar ideas are applicable to clustered systems.

A DIVA processor has scalar and vector execution capability. The basic structure of the processor is similar to vector processors of the CRAY PVP machines [1] or the Convex vector machines. Briefly, the processor has a scalar execution unit which performs scalar computations and flow control. Vector instructions are sent to the vector execution unit which is a register based unit. Vector registers are loaded from and stored to memory through explicit instructions. All computation instructions work on the vector registers. Each DIVA node can work independently on an application[1]; in this mode DIVA nodes operate like traditional vector processors. Multiple DIVA nodes also have the capability to work together on a single vector application whose dataset is distributed among the nodes' memories.

In this cooperative mode of operation, all DIVA nodes execute all scalar instructions of the application. Each processor maintains its own scalar register set and performs (redundantly) all scalar computations. When a processor accesses scalar data that reside in its local memory (owning processor) it broadcasts them to other nodes. When a processor tries to access remote scalar data, it will receive them from the broadcasting processor. This scheme was proposed by Garcia-Molinas et al. [6] as a way to build a system with massive memory from a cluster of VAX-11 computers. Burger, Kaxiras and Goodman introduced the DataScalar architecture that extended this paradigm to work with caches (that reduce the number of broadcasts dramatically) and out-of-order execution that allows nodes to run asynchronously, fetching their local data and broadcasting them for the benefit of the rest of the nodes [4].

DIVA nodes however, cooperate on the execution of vector instructions, each executing a different part of the instruction in parallel with other nodes. This partition of work is possible, because vector instructions refer to architectural vector registers while nodes operate only on their physical vector registers. In Figure 2, four *physical* vector registers in four DIVA nodes combine to form a 16-element *architectural* vector register. Instructions that refer to this architectural register exe-

[1] This in fact may be the preferred mode of operation when the application fits in a single node.

cute with a potential four-fold speed-up.

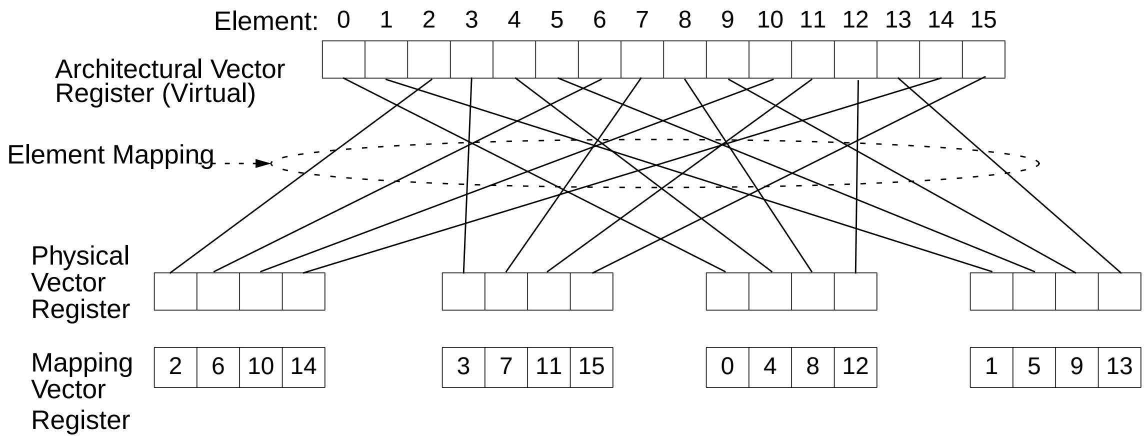


FIGURE 2. Physical vector registers combine to form architectural vector registers

A mapping vector describes the assignment of architectural elements to physical elements and it is distributed in mapping vector registers in the nodes (Figure 2). Ideally, we want to assign architectural elements in nodes where the corresponding memory data are located, thus reducing the number of external accesses needed to load or store these elements.

Prior to using the physical vector registers that comprise an architectural register, a mapping vector must be created and the mapping vector registers must be set accordingly. We introduce the SETMV instruction as the mechanism to create a mapping vector (other mechanisms are possible and we will briefly discuss some in Section 2.2). To illustrate the use of mapping vectors, consider the following simple loop and the sequence of vector instructions it is compiled to.

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

```
SETMV 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)
```

The SETMV instruction defines a mapping of architectural elements to physical elements. This mapping must be the same for the physical vector registers that combine to form the architectural register V0 and for the physical vector registers that combine to form V1. This is because, we are adding V0 and V1 together which requires exact alignment of V0 and V1 elements in the corresponding physical registers. By specifying that both v0 and v1 are loaded using the mapping vector MV0, we provide the necessary guarantee of aligned elements for the vector add instruction.

To put it all together, we will describe the execution of the example code in a four node DIVA system shown in Figure 3. For each node we show two physical vector registers (PhV0 and PhV1) of length 4. Four PhV0 (PhV1) combine to form the architectural vector register V0 (V1) of length 16. We also show one mapping vector (MV0) and 16 words of memory for each node. Memory is word-interleaved, i.e., consecutive words map on adjacent nodes (the address of each memory location is also shown in this figure). In the general case, memory is block-interleaved specifically for each application to provide a good lay-out of memory vectors (see Section 2.3). The shaded areas in memory represent the memory vectors A and B which are referenced in the example code: vector A starts at address 6 and it is accessed with a stride of 1 (light shade), while vector B starts at address 30 and it is accessed with a stride of 2 (dark shade).

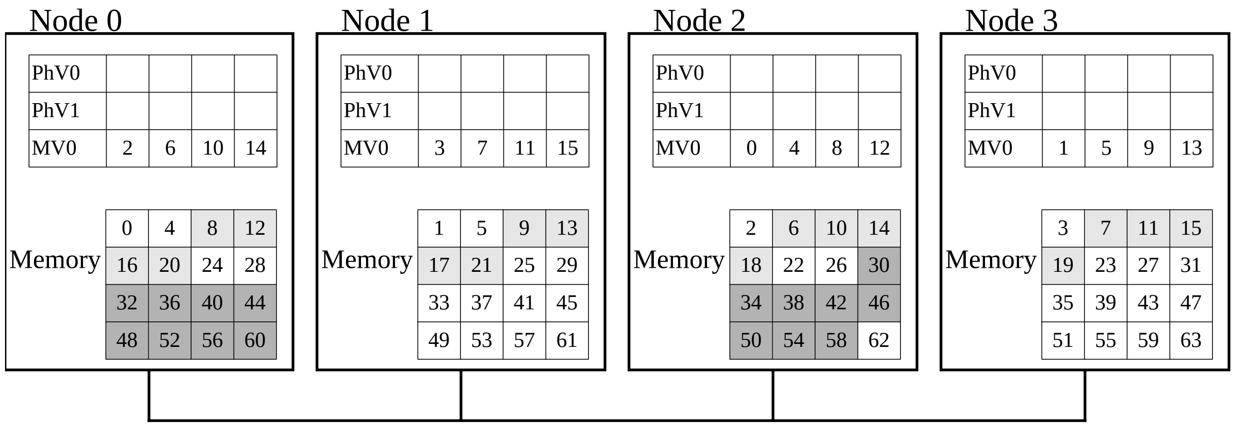


FIGURE 3. Four node DIVA system. For each node, two physical registers (PhV0 and PhV1), one mapping vector (MV0) and 16 words of memory are shown. The lay-out of vectors A and B (referenced in the example code) is also depicted as shaded areas in memory.

Figure 3 shows the state of the system after the execution of the SETMV instruction: the mapping vectors are set in every node. When the nodes encounter the first vector load instruction they each load their physical vectors with the elements described in their mapping vectors. Elements of a physical vector are loaded according to the following formula: $PhV0[i] \leftarrow (BaseAddress + Stride \times MV0[i])$, where i=0,1,2,3 for Figure 3. In the general case, the number of valid entries in the mapping vector registers controls the length of the vector operations in each node. Application gather and scatter instructions are executed in the usual manner. For example, the following formula is used for a gather instruction: $PhVx[i] \leftarrow (BaseAddress + PhVindex[i])$. In this case, a hidden indirection is that PhVindex (the index register) already follows a mapping vector which is inherited by PhVX.

In Figure 3, the mapping vector is set to exactly mirror the lay-out of vector A in memory: element 0 of V1 is assigned to node 2 where the staring address 6 of vector A is located, element 1 to node 3 where address 7 is located, etc.... This results in no external communication for the first load — all elements are loaded from local memory. However, the second load has to follow the same mapping vector (otherwise the elements of V1 would not align with the elements of V0). Vector B maps only on nodes 0 and 2. The particular element assignment of MV0 leads to 12 remote accesses (e.g., element 2 of V1 is assigned to node 0, while address 34 is located on node 2). Only four local accesses (all in node 2) take place. Figure 4 shows the contents of the physical vector registers after the two loads complete. The shaded elements of V1 are the ones that required remote accesses. To summarize, in this example, a mapping vector was set to mirror the lay-out of vector A in memory. This led to 12 remote accesses (all from loading vector B) out of a total of 32 accesses for the two loads.

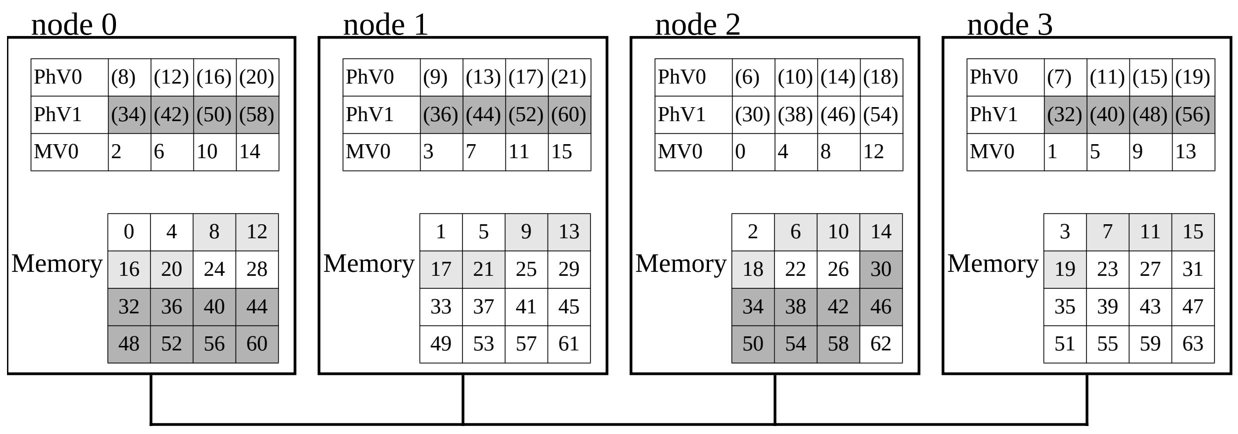


FIGURE 4. State of the DIVA system after executing the two loads of the example code. V0 elements are loaded locally, because mapping vector MV0 is set to mirror vector A in memory. 12 elements of V1, however, require remote communication according to the same mapping vector.

2.2 Mapping vector selection

A mapping vector must be defined for every distinct computation slice, i.e., a group of related vector instructions that load some vector registers, compute on them and store the results. Once a mapping vector is used to load or initialize a vector register, the rest of the registers in the computational slice must use the same mapping vector for their elements to align properly in the physical registers.

To accommodate unrelated computation slices interleaved in the instruction stream, as in the example of Figure 5, more than one mapping vector may be needed. Two different mapping vectors (MV0 and MV1) are needed in this example, since it is likely that each computation would perform better with its own way to assign the location of the architectural vector elements. CRAY compilers rarely interleave more than two independent computation slices; so in practice, we could do with as few as two mapping vectors[2].

Example code

```
                                 -------------------------------------------------------------
                                              SLICE 1               |          SLICE 2
                                 -------------------------------------------------------------
     DO 100 I=1,16               SETMV MV0, BASE=A,STRIDE=1         |
     C(I)=A(I)+B(2*I)            VLOAD  V0,BASE=A,STRIDE=1,MV0      |
     F(I)=D(I)+E(I)                                                 | SETMV MV1 BASE=D,STRIDE=1
100  CONTINUE                                                       | VLOAD  V3,BASE=D,STRIDE=1,MV1
                                 VLOAD  V1,BASE=B,STRIDE=2,MV0      |
                                 VADD   V0, V0+V1                   |
                                                                    | VLOAD  V4,BASE=E,STRIDE=1,MV1
                                                                    | VADD   V3, V3+V4
                                                                    | VSTORE V3,BASE=F,STRIDE=1,MV1
                                 VSTORE V0, BASE=C,STRIDE=1,MV0     |
```

FIGURE 5. Sample code with two independent computation slices that are interleaved in the instruction stream.

The goal in selecting a mapping vector for a computation slice is to minimize the overall communication of the slice's memory operations. This can be done either in compile-time or at run-time. At compile-time, the compiler computes a mapping vector and stores it as static data along with the binary of the application. When the binary is executed, the mapping vector is simply loaded from memory. This provides the compiler with great flexibility in computing mapping vectors that minimize the overall communication of a slice. Unfortunately, this approach requires considerable information to be available at compile-time. The base addresses and strides of the memory operations, as well as the run-time memory interleaving, must be known to compute mapping vectors. This information may not be available since base address and stride arguments (kept in scalar registers) are frequently unknown at compile-time. Because generating mapping vectors at compile time is not trivial and requires considerable compiler involvement, we are not going to expand on it in this paper. It is an area of research that we are investigating further.

In this paper we examine the case where the mapping vectors are constructed at run-time by a spe-

[2] The useful range for mapping vectors is from 1 up to the number of architectural vector registers.

cial SETMV instruction[3]. This instruction takes three arguments: a mapping vector identifier, a base address and a stride (its syntax in our pseudo-assembly is "SETMV MV0,base=A,stride=N" or "SETMV MV0,S1,S2" where S1 and S2 are scalar registers). It creates a mapping vector that mirrors a memory vector whose lay-out is defined by the base address and the stride. Each node is responsible to decide which elements of the architectural register are be assigned to it. This is done in a distributed fashion: each node generates all the addresses of the memory vector and decides which of them, according to the memory interleaving used at that instant, are local. Each node assigns elements corresponding to local addresses to its physical register elements.

SETMV semantics are straightforward when everything fits nicely (the number of local addresses in a node is equal to its physical vector length). There are cases, however, where some nodes have more elements in their local memory than what they can fit in their physical vector registers (an *element overflow* condition), while other nodes do not have enough local elements to fill their registers. For instance, in the example of the previous section, if we used the lay-out of vector B as the basis for setting the mapping vector (i.e., SETMV MV0,base=30,stride=2) nodes 0 and 2 would each try to assign 8 elements in their 4-element physical vectors. The SETMV instruction semantics are aware of such cases and redistribute elements when this happens. This is again done in a distributed fashion, without any communication. Since every node goes through all the addresses of the SETMV instruction, they can keep count of elements assigned in all nodes. Every node implements a number of counters (one for each node in the system). The counters' size is equal to the length of the physical vector registers. A counter overflow indicates that a 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 SETMV instruction limits the mapping vector to mirror a memory vector described only by a base address and a stride. Arbitrary mapping vectors can be generated with an indexed version of

[3] Alternatively, instead of a specialized instruction (SETMV), the compiler could put in code that computes traffic efficient mapping vectors at run-time. It is likely that saving a few remote transfers would pay for the additional time spent in computing the mapping vectors. In this work, we describe only SETMV; general code could emulate this instruction or even provide better functionality.

the SETMV, where with the help of an index register, we can describe any irregular memory vector. We have not examined yet the possibilities of this more general form of SETMV, which presents similar challenges as with the case of generating mapping vectors at compile-time. Instead, we have limited our investigation to the simpler form of SETMV.

The compiler inserts a SETMV instruction at the beginning of every computation slice and has to choose its base address and stride arguments. These arguments (whether they are literals or scalar register identifiers) are copied from one of the load or store instructions of the corresponding computation slice. In other words, we choose a mapping vector for a computation slice, to mirror a memory vector referenced in that slice. According to the available information, the compiler can make choices of varying optimality for selecting these arguments:

- If the compiler has no information about the run-time interleaving or the base addresses and strides of the loads and stores of a slice, it blindly copies the arguments of the first load (or store) it encounters in the slice. We call this *first choice* selection.
- If the compiler does have information on base address, strides and run-time interleaving, it can select the arguments of a load or store that lead to less overall traffic for the whole slice. We call this *best choice* selection.

Best choice selection is based on the following simple heuristic, but more elaborate methods are possible. For each memory operation in the 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 memory operation to the home nodes of all the other memory operations and we select the one with the most matches.

For a typical vector program the compiler will be able to make an intelligent choice for some of computation slices, but not for others. 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.

In a DIVA program, vector loads and stores must designate a mapping vector. This can be implemented by using an extra mapping vector identifier field in the instructions. Alternatively, one of the mapping vectors can be implicitly active. In this case, a new instruction is needed to activate a mapping vector. Arithmetic or logic vector instructions do not have to designate a mapping vector, since they operate on vector registers already loaded or initialized according to a specific mapping vector.

2.3 Memory Interleaving

Ideally, in a DIVA system we want to control data placement so we can distribute and align the memory vectors of an application. We want to distribute memory vectors across the nodes to take advantage of the system's multiple vector units and our ability to distribute the architectural vector registers. Additionally, we want to align memory vectors accessed in the same computational slice, to minimize remote traffic.

To achieve distribution and alignment of memory vectors we can do of the following: i) use the compiler to allocate arrays and other data structures appropriately (or even in custom ways); ii) use directives in the source code to specify particular allocation policies for data alignment and distribution; iii) interleave memory. For this study we restrict our attention to distributing memory vectors across the DIVA nodes by simply interleaving memory. Remote traffic in a DIVA system is, therefore, a function of both memory interleaving and mapping vector selection. Without any other provision for custom data placement, simply interleaving memory leads to acceptable distribution of memory vectors, but it does not offer any help in preventing misalignment of related vectors. In the evaluation section we examine programs compiled for vector supercomputers, without any provision for DIVA-specific memory allocation. This defaults to running legacy code on DIVA systems.

We interleave memory in a DIVA system by choosing which bits of an address are the node address bits. By using the low order bits of an address, we interleave words in the DIVA nodes. Shifting the node address bits toward the high order bits of an address results in interleaving larger and larger blocks (e.g., if we shift the node address bits 4 places toward the high order bits, we interleave blocks of 16 words among the DIVA nodes).

For the evaluations in the following section we assume a segmented memory space. Entire applications fit in one segment. Virtual to physical address translation involves just adding an offset to the virtual address. For each application the operating system sets the run-time interleaving. It is possible to have simultaneously multiple interleavings for the same application. This serves to distribute different data structures in memory (e.g., the two factors of a matrix multiplication can be interleaved differently so their memory vectors are distributed in the same manner). However, in the evaluation of Section 3, we report results for a single interleaving per application. We discuss heuristics for selecting such an interleaving in the next section.

3 Preliminary Evaluation

3.1 Metric

We focus our evaluation on the external traffic generated by DIVA. We consider this a critical measure since the divergence between internal and external bandwidth will only increase with progressively higher integration of processors and memory. The major bottleneck of any system based on highly integrated processor-memory nodes will be external accesses [5].

3.2 Methodology

We use trace-driven simulation to evaluate DIVA and compare it with other systems. We collected instruction traces for 6 of the NAS kernels and 6 of the NAS applications [9] running on CRAY C-90 vector supercomputers. Since these instruction traces lack the SETMV instructions, we used a preprocessor to take the place of the DIVA compiler.

The preprocessor discovers computation slices in the traces, assigns mapping vectors to them, inserts the appropriate SETMV instructions at the beginning of each slice, and adds mapping vector arguments to the appropriate instructions. The preprocessor works in two different modes to simulate the case where the compiler has enough information to make an intelligent choice for the SETMV arguments and the case where it does not. In the first case, the preprocessor uses the heuristic described in Section 2.2 to select the best memory operation of a slice (best choice selection). In the second case, the preprocessor selects the arguments of the first memory operation of a slice and uses them for the corresponding SETMV (first choice selection). A realistic compiler would produce a mix of first and best selections.

We examine DIVA systems with 2, 4, and 8 nodes. Since we use traces from 128 element vector supercomputers the architectural vector registers in DIVA are fixed to 128 elements. The length of the nodes' physical vector registers is 64, 32, and 16 for the 2, 4, and 8 node configurations respectively. In reality, the architectural length of DIVA varies with the number of nodes: adding more nodes allows for larger vectors and more parallelism.

We also examine DIVA systems with and without caches. The first case is NUMA DIVA without any remote-data caches. Each node has a vector processor and local memory and there is no remote-data caching in the system[4]. The second case is CC-NUMA DIVA, a NUMA DIVA with the addition of small caches in the nodes. These are remote-data caches that are kept coherent with an invalidation protocol. We assume that the remote-data caches are implemented in the same technology as local memory and are always 1/16th of the memory size. They are optimized to reduce traffic: they are write-back, write-validate, with a line size of 1 word (8 Bytes). Because of the relatively large size of the caches, we examined them as direct-mapped. We intent to expand our evaluation to set-associative caches.

3.3 Baseline systems

We compare DIVA (NUMA DIVA and CC-NUMA DIVA) to two baseline systems that rely only on caches to reduce external communication. For both of these systems, we assume that only one node executes the program but it references memory in other nodes. The exact same traces are used for both DIVA (which parallelizes individual vector instructions across the nodes) and the baseline systems. Remote traffic results when the node executing the application tries to access data located on another node. In the first baseline system (STANDARD CC-NUMA) each node has a part of the global memory and a remote-data cache (1/16th of the local memory). This cache is also optimized in the same way as the caches we use in the DIVA systems: it is write-back, write-validate, with a line size of 1 word. As of yet, we have examined direct-mapped caches but we plan to expand our evaluation to set associative caches.

In the second baseline system (COMA) the whole memory of a node is used as a huge cache. This represents the case of a Cache Only Memory Architecture (COMA) [13] system. Since these caches are the size of the local memory, tag overhead becomes a serious issue. We implement these caches as write-back, write-validate, 4-word line sectored caches [15]. DIVA can also be implemented as a COMA, and we plan to expand our evaluation to include COMA DIVA.

[4] We are not concerned with caching of local data inside a node since our main concern is remote traffic. Caching for local data and the various ways it can be implemented is beyond the scope of this paper.

We implicitly assume that a single node in the baseline systems has equivalent vector capabilities as with a multiple node DIVA system. In other words, we assume that the length of the physical vectors in a baseline node is equal to the architectural vector length of a DIVA and that vector instructions (besides memory operations) execute with similar speed (thus giving no advantage to DIVA's parallelization of individual vector instructions). In reality, either DIVA can employ more vector processing power on a single application than the baseline systems or, for the same processing power, DIVA nodes can be less expensive than the baseline nodes.

3.4 Memory and Cache Size Scaling

In all systems we examine (NUMA DIVA, CC-NUMA DIVA, STD. CC-NUMA and COMA) we scale the memory and cache sizes for each application. Memory on a node is always assumed to be *1/n* of the application size, where *n* is the number of nodes required by the application. The small cache-coherent caches are always fixed to 1/16th of a node's memory capacity, while the COMA caches are equal to a node's memory capacity.

The real world situation is one where memory and cache sizes are fixed, while application size varies. We do the reverse here: we keep application size fixed and scale memory and cache size to fit the application in whatever number of nodes we chose. We do this to simplify the simulation process. To keep things simple, we also round the application size up to the closest power-of-two and we assume a perfect fit in memory. That is, we assume that, if the application runs on two nodes, each node has a memory capacity of one-half the size of the application; if it runs on four nodes, each node has a capacity of one-fourth of the application's size, and so on.

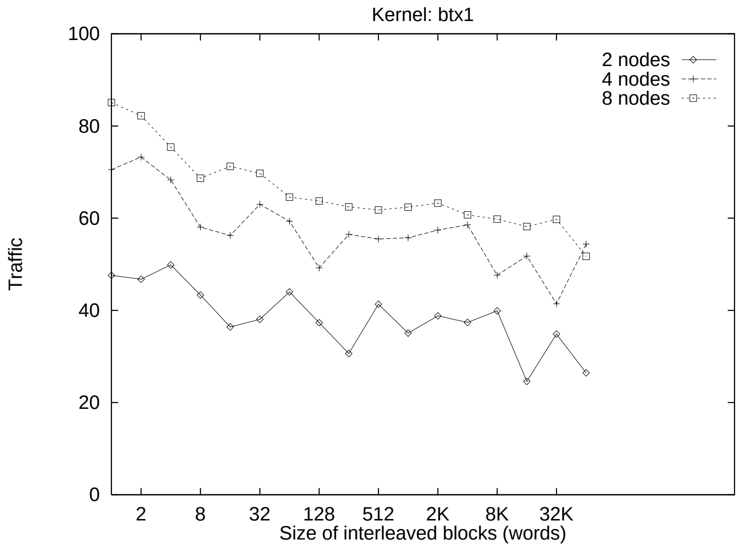
3.5 Kernels

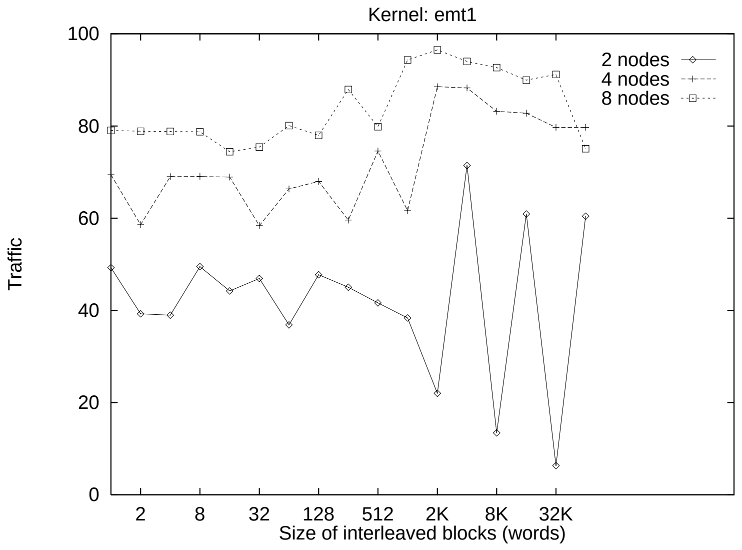
We use the set of six small kernels to illustrate the behavior of a DIVA system since they are too small to be used for traffic comparisons. The kernels are from the suite of NAS kernels and are the following[5]: BTRIX (BTX1), EMIT (EMT1), FFT (FT11), GMTRY (GMY1), MXM (MXM1), VPENTA (VPT1). They represent important computation segments from large scientific applications.

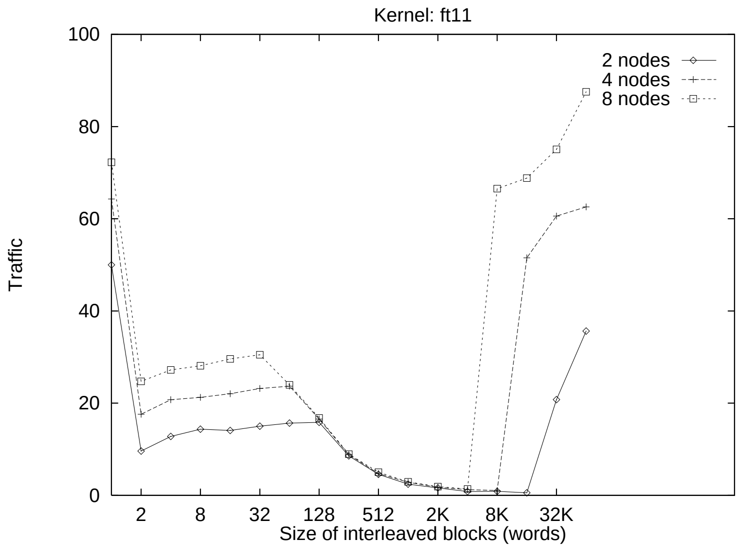
We present the behavior of the kernels on a NUMA DIVA system. We use the first choice selection for the SETMV instructions. In Figure 6 we present six graphs, one for each kernel. To show the

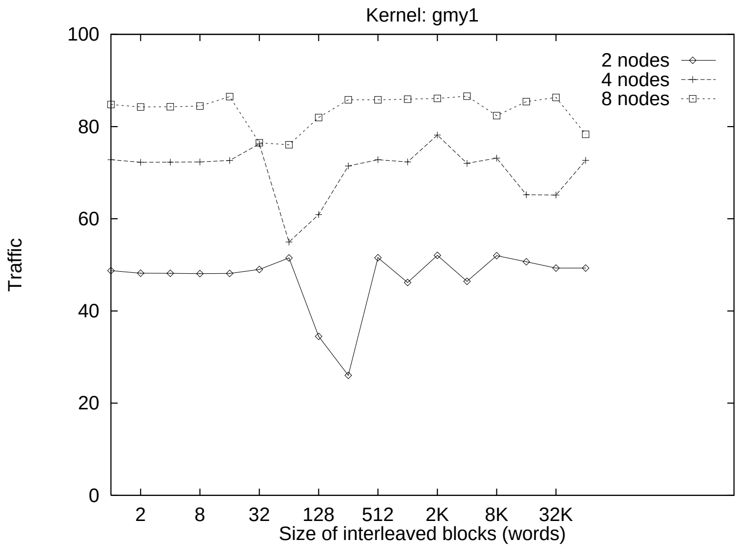
[5] We use the short-hand name in the graphs in Figure 6 and Figure 10.

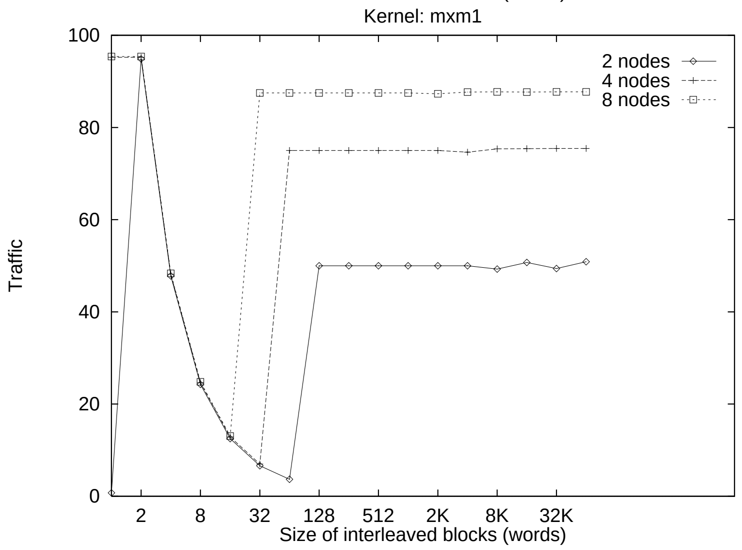
effect of memory interleaving on remote traffic, we use 17 different memory interleavings ranging from 1 word to 64Kwords (the horizontal axis represents interleaving size in words). The vertical axis represents traffic as a percentage of the total data traffic between the vector registers and memory required by the application. We do not consider request traffic. In each of the graphs we superimpose results for 2, 4 and 8 DIVA nodes. The traffic for 2 nodes is the lower curve, the traffic for the 4 nodes the middle, and traffic for 8 nodes the top curve.











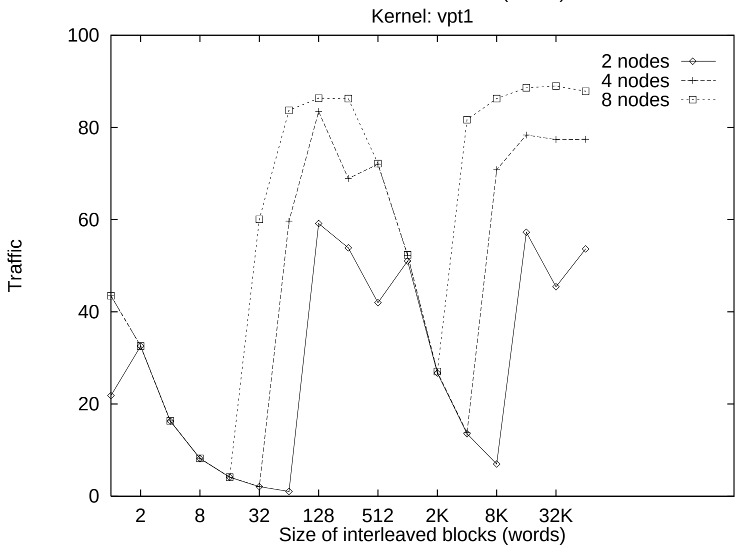


FIGURE 6. Kernel Traffic

For many of the kernels, there is a correlation between DIVA traffic for a specific interleaving and the predominant stride and vector length. In Table 1we present the predominant strides and vector lengths for these kernels.

Kernel	**Dominant Strides**	**Dominant Vector length**
BTRIX	825, 1	28
EMIT	1	64
FFT	258	128
GMTRY	2	99, 1 to 98, 100
MXM	1	128
VPENTA	129	128

Table 1: Kernel statistics

For FFT, MXM, and VPENTA the interleavings where DIVA has the lowest traffic correspond to the interleavings that would distribute memory vectors with the dominant stride and vector length evenly across all nodes (Figure 6). For a memory vector of stride S and length L, two interleavings I that distribute it on N nodes are given by the following formulae: $I = \frac{S}{N} \cdot L$ and $I = \frac{S}{N}$.

FFT and VPENTA have low traffic at the points approximately described by both the above equations while MXM at the points described by the first equation (Figure 6). Broadly speaking, the explanation of this result is that the first equation produces interleavings that distribute memory vectors in contiguous groups of elements among the nodes while the second distributes consecutive elements among the nodes. It then becomes a matter of how well different vectors used in the same computation align in the nodes. It is more likely for two vectors to align in the same node, if they are distributed in contiguous groups, than to align when their consecutive elements are interleaved in the nodes. To illustrate this situation, consider the following example code that produces a common reference pattern in vector applications:

```
      DO 100 I=1,16
        A[I] = A[I+1] * 3.14159
100   CONTINUE
```

Lets assume for simplicity that we are going to load `A[I]` and `A[I+1]` with two independent load instructions without any optimization at the register level; so vector `A` is accessed twice in the same loop with an offset of 1. If we distribute this vector in memory according to the first formula (Figure 7) we will have a misalignment in the two sets of accesses. If the first set (`A[I]`) executes with no remote traffic then the second (`A[I+1]`) will cost us 4 remote accesses out of a total of 16 (remember that the accesses depicted in Figure 7 must correspond to aligned physical elements).

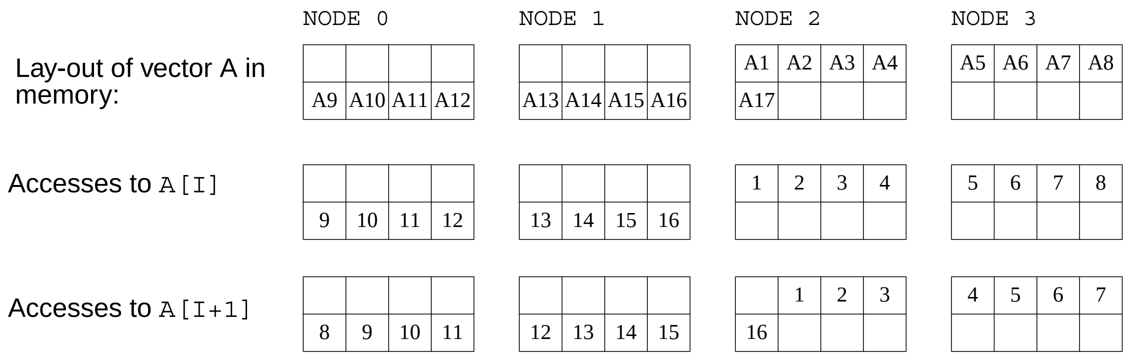


FIGURE 7. Alignment problems when distributing memory vectors in groups of contiguous elements (first formula).

If, however, we use the second formula to distribute the memory vector (Figure 8) the misalignment becomes much more serious. In this case, none of the `A[I]` accesses and `A[I+1]` accesses align in any node. If a mapping vector generates no remote traffic for the `A[I]` accesses then the same mapping vector makes all `A[I+1]` accesses remote.

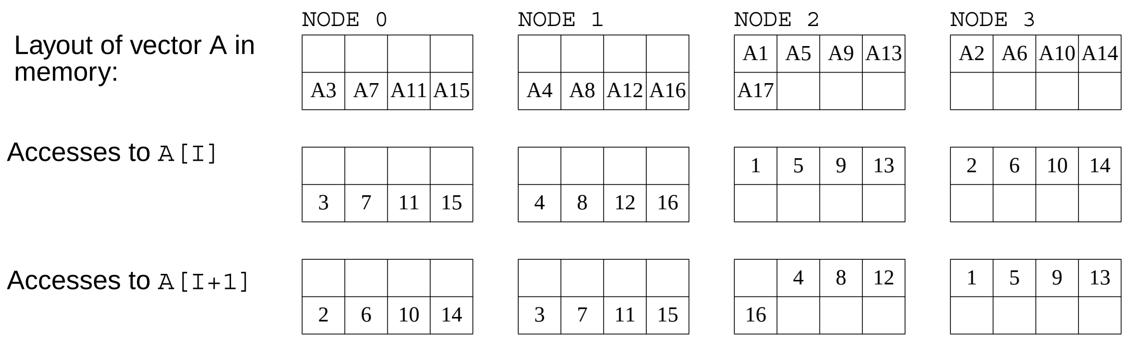


FIGURE 8. Alignment problems when distributing consecutive elements around the nodes (second formula).

For FFT and MXM, just after the points where DIVA exhibits minimal traffic, we have a sharp increase in traffic. This is because, these interleavings effectively map the memory vectors of the application entirely in one node. For VPENTA the sharp increase of traffic for interleavings greater than 64, 32, and 16 for the 2, 4, and 8 nodes respectively, is due to alignment problems.

The behavior of BTRIX, EMIT, and GMTRY is more unpredictable. BTRIX has both distribution and alignment problems. EMIT and GMTRY make extensive use of gather and scatter operations which are largely responsible for the anomalies in their traffic behavior. The traffic behavior of EMIT is especially unstable at large interleavings for 2 nodes. This behavior is due to EMIT's small vectors (64 elements) fitting entirely in one node. Because each DIVA node, in the 2 node case, has 64-element physical vector registers, memory vectors can be accessed with no remote traffic when they are on the same chip. If, however, related memory vectors map onto different nodes then the resulting traffic is considerable.

The results of this section give us a heuristic to select a memory interleaving for an application: we simply select the one that is going to distribute memory vectors of the dominant stride and vector length across the nodes. The best way to distribute these vectors depends on their alignment properties. Many times, distributing the elements in contiguous parts proves to be effective. In the absence of reference patterns similar to that in Figure 8, distributing consecutive elements across the nodes can also lead to minimal traffic. To select an interleaving we need to know the predominant stride and vector length of the programs. For many cases the compiler can provide these values. For the cases where this is not feasible, profiling can be used.

3.6 Benchmarks

We use vector versions of the NAS parallel benchmarks [9] for our evaluations. The NAS parallel benchmarks are a suite of five kernels and three pseudo-applications. In this paper, we use three of the five kernels (IS, FT and MG) and the three pseudo-applications (LU, SP and BT). We run small problem sizes of these benchmarks for ease of tracing and simulation. Also, we run the applications in single thread mode. These benchmarks are in fact quite amenable to high level parallelization and do not really belong in our target application set. However, applications that belong in our target set tend to be large proprietary codes such as NASTRAN or GAUSSIAN which are extremely difficult to get access to or work with. We picked the NAS parallel applications instead, because of availability, ease of use and general familiarity.

A characteristic of these applications is that they generate their own dataset (practically they have no input data). This makes the large COMA caches quite effective, since there are very few cold misses due to the write-validate policy. In Table 2 we present some statistics for these benchmarks. For each one we list the problem size and the memory references of the corresponding trace. These benchmarks have a multitude of strides and vector lengths. Here, we present the most important ones for each application.

Benchmarks	Problem Size	Memory References (Millions)	Dominant Strides	Dominant Vector lengths
BT	16x16x16, 4 iterations	53	1, 17	14
FT	64x64x64	101	1	64
IS	64K keys	26.5	1	128
LU	16x16x16, 20 Iter.	65	> 1024, 1, 17, 255	128, 1 to 14, 120, 116
MG	128x128x128, 4 Iter.	594	1, 2	128, 64, 32, 16
SP	16x16x16, 20 Iter.	69	1, 17	14, 128

Table 2: NAS benchmark statistics

In Figure 9 we present the traffic results for these benchmarks. Similarly to the graphs in Figure 6, the horizontal axis represents different interleavings (the interleaved block size ranges from 1 to 256 words) and the vertical axis represents remote traffic as a percent of the total traffic (100%) required for data transfers between memory and vector registers. The graphs are divided in 3 sections from left to right for 2, 4, and 8 nodes.

We present four sets of results for DIVA systems: NUMA DIVA and CC-NUMA DIVA each with first choice selection and best choice selection of SETMV arguments. These cases are referred to as "NUMA first," "CC-NUMA first," "NUMA best," and "CC-NUMA best" in the graphs. We also present the traffic for STD. COMA ("COMA") and STD. CC-NUMA ("CC-NUMA base") baseline systems. The traffic results of these two cases are independent of the interleaving used for DIVA and they are represented by straight lines. We select in advance an interleaving for these cases that produces the least traffic. For the particular set of benchmarks the interleavings for STD. CC-NUMA and STD. COMA where very large, often comparable in size to the memory capacity of the nodes. In Table 3 we present the cache sizes for the STD. COMA system. The cache sizes of the CC-NUMA DIVA and the STD. CC-NUMA are 1/16th of the corresponding COMA caches.

Application	**Active Memory Size (rounded up)**	COMA **Cache size 2 nodes**	COMA **Cache Size 4 nodes**	COMA **Cache Size 8 nodes**
BT	64 MByte	32 MByte	16 MByte	8 MByte
FT	16 MByte	8 MByte	4 MByte	2 MByte
IS	4 MByte	2 MByte	512 KByte	256 KByte
LU	8 MByte	4 MByte	1 MByte	512 KByte
MG	64 MByte	32 MByte	16 MByte	8 MByte
SP	4 MByte	2 MByte	1 MByte	512 KByte

Table 3: Cache sizes for COMA

The results in Figure 9 show that there is a performance gap between first choice and best choice selection of SETMV arguments. As we have discussed, we expect a real compiler to fall somewhere in between. The remote-data cache in CC-NUMA DIVA provides significant reductions in traffic for all the benchmarks. The bottom-line of the graphs is that the best cases of CC-NUMA DIVA produce less traffic than both STD. CC-NUMA and STD. COMA for three benchmarks (FT, IS, and MG). For SP, CC-NUMA DIVA produces less traffic than STD. CC-NUMA for 4 and 8 nodes.

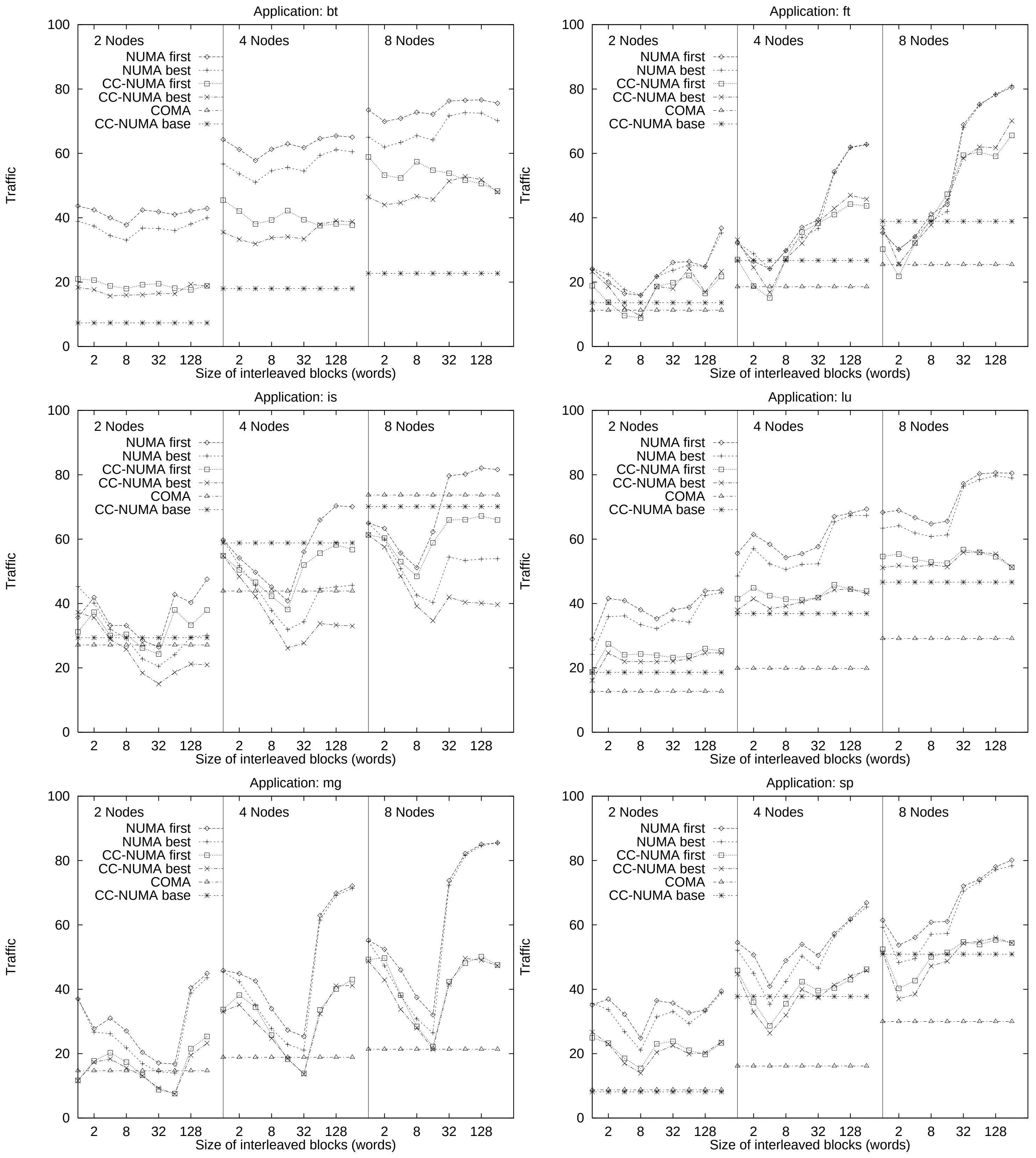


FIGURE 9. Benchmark Traffic

3.7 Optimizations

Data placement presents a significant opportunity in optimizing DIVA programs. The results presented in the previous section were for unoptimized programs. Instead, the data placement of these programs was specific to the CRAY supercomputers that were used to collect the traces. Often, prime number array dimensions (good for avoiding bank conflicts in supercomputers) produced misalignment problems in DIVA.

Since we do not have a compiler capable of DIVA-specific data placement, we implemented simple source code transformations to explore the performance advantages of custom data placement. Our intention was to allocate data structures in memory to reduce misalignment of memory vectors. In its general form, this is not a trivial problem. We followed a very simple approach: for three of the kernels and three of the benchmarks, without changing the structure of the programs, we re-allocated their multi-dimensional arrays so that some (but not all) of the dimensions became powers-of-two. This provided significant benefits in terms of traffic reduction. In essence, allocating with powers-of-two dimensions resulted in statistically much less misalignment of memory vectors.

The results for the optimized kernels (BTRIX, EMIT, GMTRY, and VPENTA) are shown in Figure 10. In Figure 11 we show the results for the optimized benchmarks (BT, LU, and SP). In these graphs we changed only the DIVA results. The STD. CC-NUMA and STD. COMA results come from the unoptimized versions of the programs (the optimized benchmarks exhibit worse cache behavior than before). With this simple optimization CC-NUMA DIVA (and in some cases NUMA DIVA) produces less traffic than the STD. CC-NUMA and almost in all cases (except LU) it produces at most as much traffic as STD. COMA.

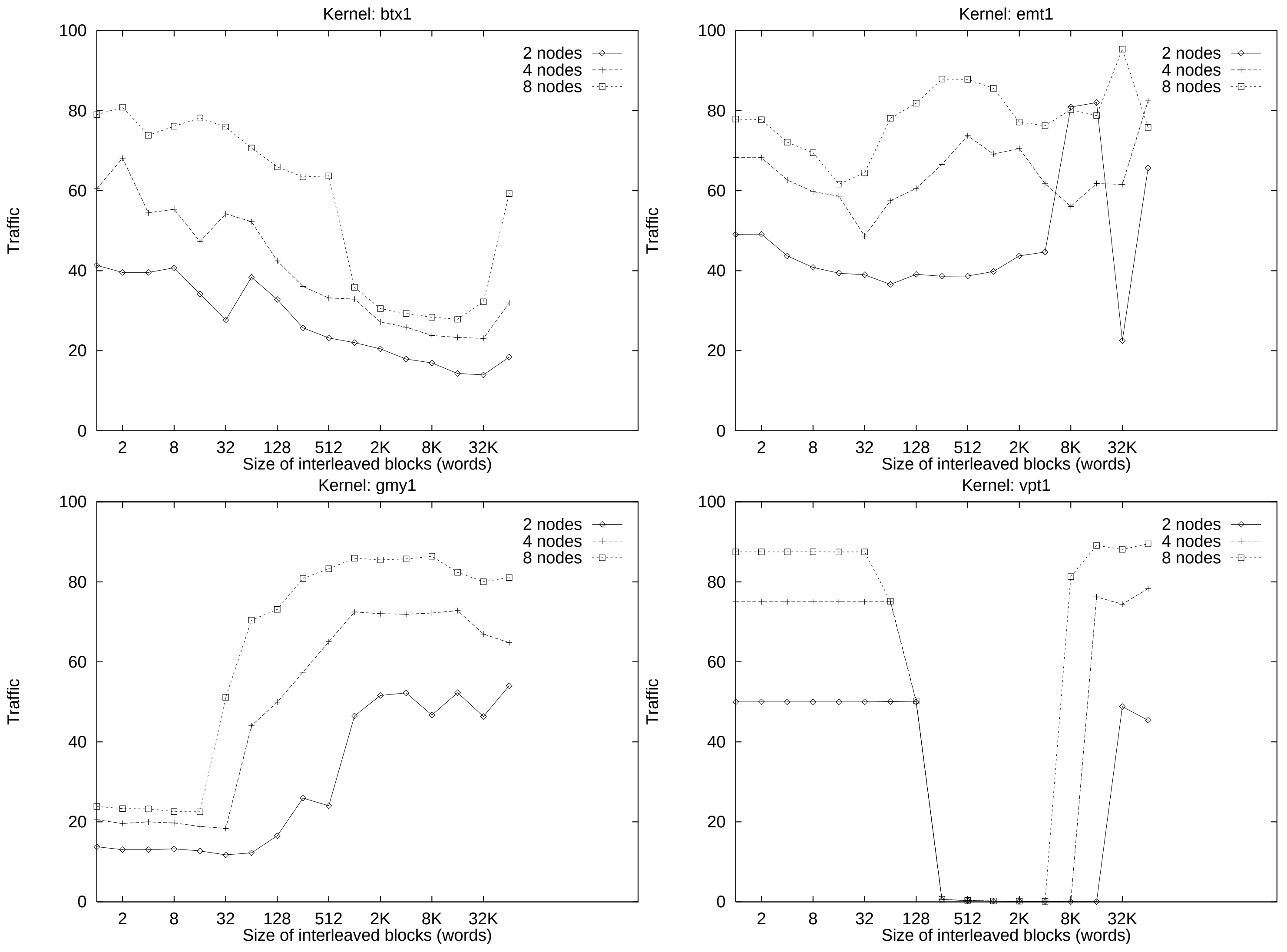
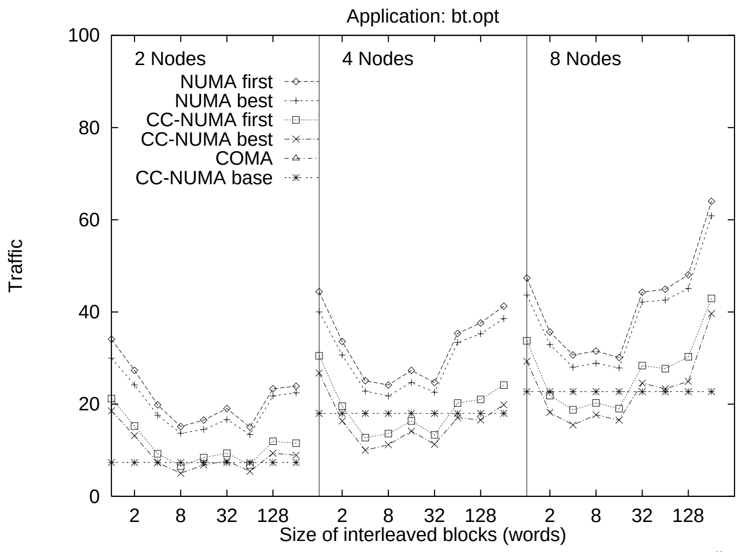
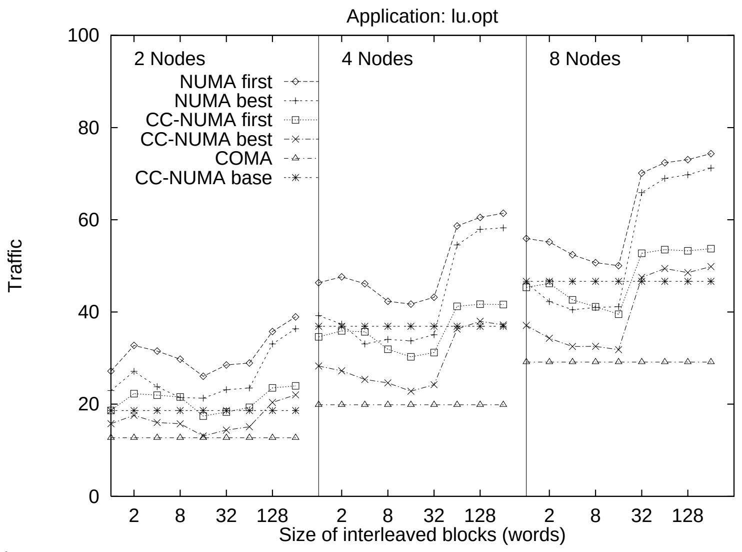


FIGURE 10. Optimized kernel traffic





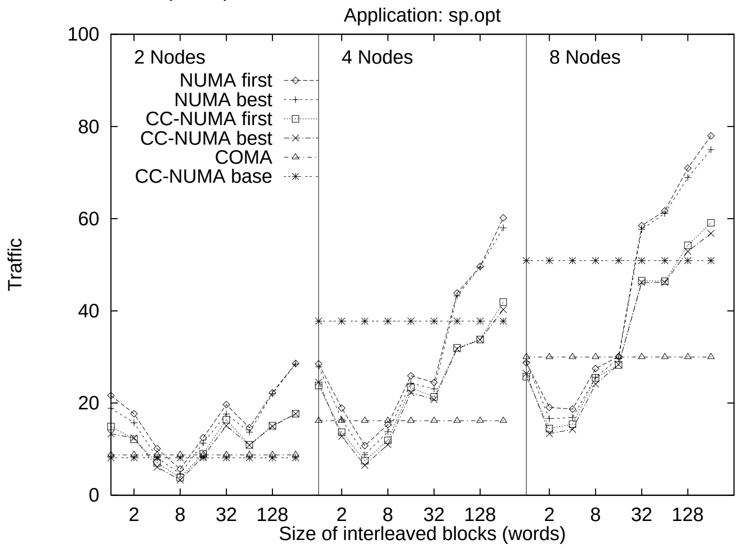


FIGURE 11. Optimized benchmark traffic

4 Conclusions

The paper considers a future where processors and local memory are tightly packaged together, possibly on the same chip. We expect that when this happens the bandwidth/latency of a processor to its local memory will be orders of magnitude superior to its bandwidth/latency to remote memory. Under such conditions, applications that fit in local memory would perform extremely well. Applications that may be parallelized in a distributed fashion, where each thread fits in local memory and there is very little communication between the threads, would also perform extremely well. However, applications that do not fit in local memory and are not amenable to parallelization in a distributed fashion would be greatly limited by the external traffic required. An important class of applications including several large proprietary codes belong in this category. We propose a novel approach for running such applications when they are dominated by vector computations.

We propose placing the application on as many nodes as needed to hold its entire dataset and using the nodes together as one large vector processor. The physical vector registers on the individual processors combine together to form architectural vector registers referenced by the application. A significant innovation is that we allow variable mappings of architectural elements to physical vector elements. The mappings are selected to reduce remote accesses. We propose to use mapping vectors to specify the correspondence of architectural to physical elements at any instant. We describe an implementation of an instruction that creates mapping vectors and we present heuristics for selecting its arguments leading to traffic efficient mapping vectors. Memory interleaving, also, has a significant effect on the amount of remote traffic and we propose heuristics for selecting appropriate interleavings for applications. Custom data placement is very promising for reducing remote traffic but we have not dealt with the problem in its general form.

We evaluated these ideas through simulations using traces generated on CRAY vector machines. Two types of DIVA systems are evaluated: with cache (CC-NUMA DIVA) and without cache (NUMA DIVA). The two DIVA systems are compared to two baseline systems. One baseline system is STD. CC-NUMA, where one node runs the entire application using the memory of other nodes. The other baseline system is COMA where all local memory is treated as one huge cache. DIVA systems can also be run in a COMA fashion and we plan to extend our evaluation to include such systems. The comparisons show that with good mapping vector and memory interleaving selection, the DIVA systems result in significantly lower traffic than CC-NUMA on several codes. For some cases the baseline COMA system has lower remote traffic than the DIVA systems. However, these results are for programs that were unoptimized for DIVA.

Remote traffic in DIVA is quite sensitive to data placement. Better alignment of data arrays could result in lower traffic. We present some preliminary results on the reduction in remote traffic when arrays are aligned on power-of-two boundaries. With such simple source code transformations, DIVA produces less traffic than the baseline COMA system as well.

There are several directions for future research in DIVA. Some of the important ones are developing more sophisticated approaches for mapping vector selection, investigating simple array alignment techniques for reducing traffic, and exploring different ideas for memory interleaving.

5 Acknowledgements

We would like to thank Jim Schwarzmeier, head of the Performance Evaluation Group at CRAY Research and Jim Goodman of the University of Wisconsin for their support. We would also like to thank Greg Faanes and Steve Scott of CRAY Research, as well as Antreas Moshovos and Babak Falsafi of the University of Wisconsin for comments on drafts of this paper.

6 References

[1] CRAY Research Inc., CRAY YMP C-90 Functional Description Manual, HR-04028, March 1990.

[2] CRAY Research Inc., CRAY T3D System Architecture Overview, 1993.

[3] Steven L. Scott, “Synchronization and Communication in the T3E Multiprocessor,” ASPLOS-VII, October 1996

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

[5] Doug Burger, James R. Goodman, and Alain Kägi, “Memory Bandwidth Limitations of Future Microprocessors.” In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 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] David Patterson, Tom Anderson, and Kathy Yelick, “The Case for IRAM.” In *Proceedings of HOT Chips 8*, Stanford, California, August 1996.

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

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

[10] James R. Goodman. “Using Cache Memory To Reduce Processor-Memory Traffic.” In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[11] Silicon Graphics Origin Technology. http://www.sgi.com/Products/hardware/servers/technology/index.html.

[12] Tony Brewer, “A Highly Scalable System Utilizing up to 128 PA-RISC Processors.” http://www.convex.com/tech_cache/ps/SPP_Arch.times.ps.

[13] Erik Hagersten, Anders Landin, and Seif Haridi. “DDM–A Cache-Only Memory Architecture.” *IEEE Computer*, 25(9):44–54, September 1992.

[14] E. Rosti, E. Smirni, T.D. Wagner, A.W. Apon, and L.W. Dowdy. “The KSR1: Experimentation and Modeling of Poststore.” In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1993.

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