The Architecture of the DIVA Processing-In-Memory Chip

Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett,
Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen,
Chang Woo Kang, Ihn Kim, Gokhan Daglikoca
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
{draper,jchame,mhall,steele,jtb,jlacoss,granacki,jaewook,chunchen}@isi.edu, {ckang,ihnkim}@usc.edu,
giskard@computer.org

ABSTRACT

The DIVA (Data IntensiVe Architecture) system incorporates a collection of Processing-In-Memory (PIM) chips as smart-memory co-processors to a conventional microprocessor. We have recently fabricated prototype DIVA PIMs. These chips represent the first smart-memory devices designed to support virtual addressing and capable of executing multiple threads of control. In this paper, we describe the prototype PIM architecture. We emphasize three unique features of DIVA PIMs, namely, the memory interface to the host processor, the 256-bit wide datapaths for exploiting on-chip bandwidth, and the address translation unit. We present detailed simulation results on eight benchmark applications. When just a single PIM chip is used, we achieve an average speedup of 3.3X over host-only execution, due to lower memory stall times and increased fine-grain parallelism. These 1-PIM results suggest that a PIM-based architecture with many such chips yields significantly higher performance than a multiprocessor of a similar scale and at a much reduced hardware cost.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures; C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiple Data Stream Architectures (Multiprocessors)*

General Terms

Design, Performance

Keywords

processing-in-memory, memory bandwidth, architecture

1. INTRODUCTION

A recent trend in computer architecture combines processing logic with memory in intelligent processing-in-memory (PIM) chips to address the well-known performance gap between processor and memory speeds [2, 7, 8, 12, 15, 16, 17, 20, 21, 22, 25, 27, 28, 30]. Many previous architectural solutions to the processor-memory gap such as multithreading, prefetching, and speculation, seek to reduce or tolerate memory latency, at the expense of increased memory bandwidth requirements [3]. PIMs instead dramatically improve memory bandwidth, by 10-100X over conventional DRAM systems, because internal processors can be directly connected to the memory banks. Latency to on-chip logic is also reduced, down to less than one half that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

For the last four years, the authors have been developing one such PIM-based system called Data IntensiVe Architecture (DIVA). The ultimate goal of the DIVA project is to design and build a prototype workstation-class system where PIMs serve as smart-memory co-processors for an otherwise conventional system. In this paper, we describe the prototype DIVA PIM chip, shown in Figure 1, which we have recently fabricated.

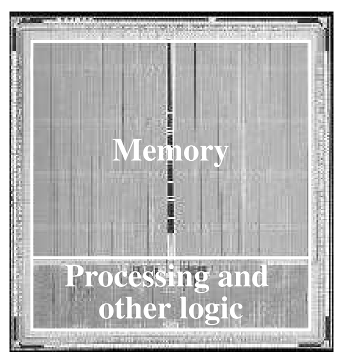


Figure 1: Microphotograph of a DIVA PIM.

DIVA targets two important classes of bandwidth-limited applications: multimedia and irregular applications, including sparse-matrix and pointer computations. Multimedia applications perform repeated computations on streams of data, often with little temporal data reuse. As processors exploit increased parallelism, multimedia applications become memory bound [23]. Performance of applications with irregular data accesses is also dominated by memory stalls

ICS'02, June 22-26, 2002, New York, New York, USA.

since such applications usually have neither temporal nor spatial reuse of data needed to make effective use of cache [5]. DIVA accelerates both classes of applications by performing computation directly in memory, requiring novel underlying hardware structures, described in this paper. Streaming multimedia applications obtain high bandwidth to on-chip memories through a 256-bit wide datapath, while irregular applications benefit from very low latency accesses to memory. As a result, much of the traffic between the host processor and memory is eliminated.

Our experience with the DIVA PIM chip has important implications for future architectures that seek to maximize memory bandwidth. We demonstrate that simple but powerful hardware mechanisms can yield significant performance improvements on bandwidth-limited applications. Many of these hardware features, including address translation, the memory interface and memory-to-memory interconnect, are specifically oriented towards architectures such as DIVA, in which PIMs are smart-memory co-processors to a conventional host. Many other features are suitable for conventional processors and embedded systems-on-a-chip, such as the design of the WideWord unit.

In two previous papers, we presented the DIVA system architecture, memory model and simulated performance improvements due to coarse-grain parallelism in PIMs for 3 programs [9], and we described system software requirements and memory management functionality [10]. This paper focuses on the DIVA PIM device and makes the following unique contributions.

- It is the first detailed description of the DIVA PIM microarchitecture.
- It pinpoints some of the design issues that must be considered in future architectures for exploiting memory bandwidth.
- It presents simulation results demonstrating an average speedup of 3.3X on 8 programs as compared to a conventional host. The speedups are due to up to a 95% reduction in memory stall time, and, for 4 of the programs, an average speedup of 9.94X due to the WideWord unit as compared to scalar PIM execution.

The remainder of the paper is organized as follows. The next section summarizes the DIVA system architecture, to set the context for the PIM microarchitecture discussion. Section 3 describes the microarchitecture in detail. Section 4 presents a set of simulation results on eight programs. Section 5 presents the status of the DIVA project. Section 6 presents related work, and Section 7 concludes the paper.

2. SYSTEM ARCHITECTURE OVERVIEW

The DIVA system architecture was specifically designed to support a smooth migration path for application software by integrating PIMs into conventional systems as seamlessly as possible. DIVA PIMs resemble, at their interfaces, commercial DRAMs, enabling PIM memory to be accessed by host software either as smart-memory co-processors or as conventional memory. In Figure 2, we show a small set of PIMs connected to a host processor through *nearly* conventional memory control logic (see Section 3.1 for details on required modifications). A separate memory-to-memory interconnect enables communication between memories without involving the host processor.

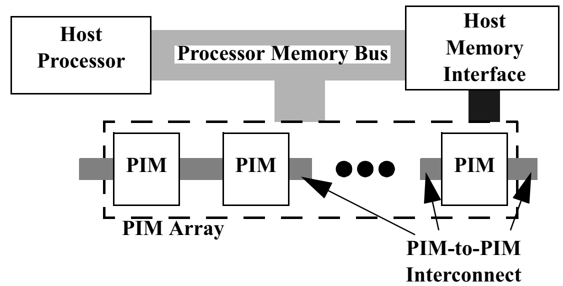


Figure 2: DIVA system architecture.

Spawning computation, gathering results, synchronizing activity, or simply accessing non-local data is accomplished via parcels. A parcel is closely related to an active message as it is a relatively lightweight communication mechanism containing a reference to a function to be invoked when the parcel is received [29]. Parcels are distinguished from active messages in that the destination of a parcel is an object in memory, not a specific processor.

Parcels are transmitted through a separate PIM-to-PIM interconnect to enable communication without interfering with host-memory traffic. This interconnect must support the dense packing requirement of memory devices and allow the addition or removal of devices from the system. For system sizes of the scale expected for DIVA (on the order of 32 PIM chips), this combination of requirements favors a one-dimensional network [14]. Future generations of DIVA-like systems that contain large numbers of PIM chips will require a more complex interconnection network and are the topic of future research.

Parcels, application code, and data contain virtual addresses. To translate these addresses without the overhead of maintaining conventional page tables at each node, we classify DIVA memory according to usage [9]: (1) *global memory* visible to the host and PIM nodes; (2) *dumb memory* allocated as conventional pages in a host application's virtual space and untouched by PIM node processing; and, (3) *local memory* used exclusively by PIM node routines. To condense translation information, rather than page tables, we use segments, each of which is defined by segment registers, as discussed in Section 3.4. In addition to local segments, a node maintains translation information for its portion of global memory. Remote addresses are translated via the concept of a home node, which is guaranteed to have the translation [26]. Thus, each node's portion of global memory includes objects for which it is the home node. The major advantages of this approach are that translation may be accomplished rapidly, and translation information on each PIM scales well.

Memory management functionality is distributed among the host's standard operating system, augmented with support for PIMs, and run-time kernels on PIM processors. Unlike standard multiprocessor systems, the host, which has a system-level view, remains a central figure in system-level scheduling, disk I/O operations, and memory management. The PIM run-time kernel must collaborate with the host on system-level operations, such as loading PIM programs and data, memory management of PIM-visible segments, and PIM context switches between different user programs. The challenge in this collaboration is that two views of memory must be maintained. For dumb pages and for disk I/O of PIM-visible segments, the host sees memory as standard 4Kbyte pages; the PIM run-time kernel instead views PIM-visible memory as variable-sized segments [10].

3. DIVA PIM MICROARCHITECTURE

Each DIVA PIM chip is a VLSI memory device augmented with general-purpose computing and communication hardware. Although a PIM may consist of multiple nodes, each of which are primarily comprised of a few megabytes of memory and a node processor, Figure 3 shows a PIM with a single node, which reflects the focus of the initial research that is being conducted. Nodes on a PIM chip share a host interface and a single PIM Routing Component (PiRC). The host interface implements the JEDEC standard SDRAM protocol so that memory accesses as well as parcel activity initiated by the host appear as conventional memory accesses from the host perspective. The PiRC is responsible for both routing parcels off chip via the PIM-to-PIM interconnect and directing parcels on chip.

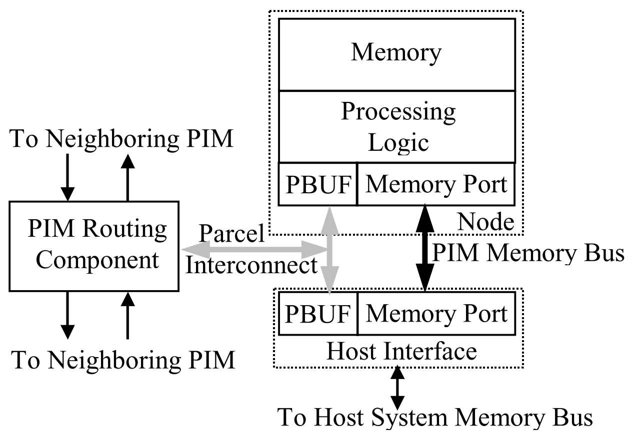


Figure 3: DIVA PIM chip architecture.

Figure 3 also shows two interconnects that span a PIM chip for information flow between nodes, the host interface, and the PiRC. Each interconnect is distinguished by the type of information it carries. The PIM memory bus is used for conventional memory accesses from the host processor. The parcel interconnect allows parcels to transit between the host interface, the nodes, and the PiRC. The host interface also contains a parcel buffer (PBUF) for parcel communication between host and PIM. Each PIM node also has a PBUF, for node-to-node parcel communication, as will be discussed in Section 3.3.

Figure 4 shows the major control and data connections within a node. The DIVA PIM node processing logic supports single-issue, in-order execution, with 32-bit instructions and 32-bit addresses. There are two datapaths whose actions are coordinated by a single execution control unit: a 32-bit scalar datapath that performs operations similar to those of standard 32-bit integer units, and a 256-bit WideWord datapath that performs fine-grain parallel operations on 8-, 16-, or 32-bit operands. Both datapaths execute from a single instruction stream under the direction of a single 5-stage DLX-like pipeline [11], complete with register forwarding logic to resolve data dependence hazards. This pipeline fetches instructions from a small instruction cache, which is included to minimize memory contention between instruction reads and data accesses. The instruction set has been designed so both datapaths can, for the most part, use the same opcodes and condition codes, generating a large functional overlap. The scalar datapath is a standard RISC architecture, augmented with a few DIVA-specific functions for coordinating with the wide datapath. The WideWord datapath accesses the scalar registers for addressing operations, as well as for controlling superword operations. Each datapath has its own independent general-purpose register file with 32 registers. Special instructions permit direct transfers between register files without going through memory. Although not supported in the initial DIVA prototype shown in Figure 1, floating-point extensions to the WideWord unit will be provided in future systems.

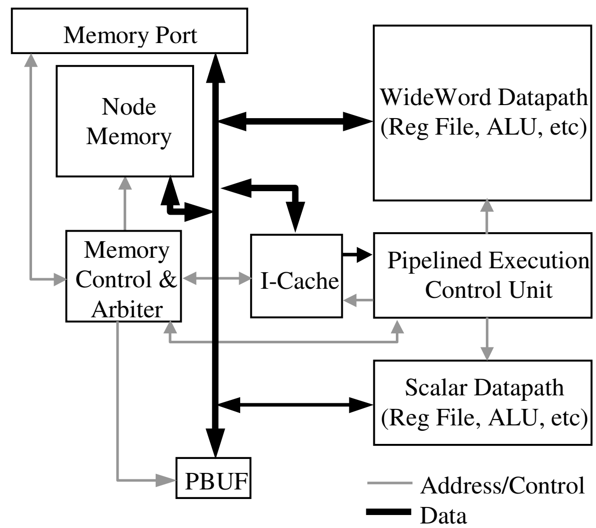


Figure 4: DIVA PIM node organization.

The execution control unit supports supervisor and user modes of processing and also maintains a number of special-purpose and protected registers for support of exception handling, address translation, and general OS services. Exceptions, arising from execution of node instructions, and interrupts, from other sources such as an internal timer or external interrupt signal, are handled by a common mechanism. The exception handling scheme for DIVA has a modest hardware requirement, exporting much of the complexity to software, to maintain a flexible implementation platform. It provides an integrated mechanism for handling hardware and software exception sources.

The following sections present the DIVA PIM node in more detail and highlight some of the unique features of the DIVA microarchitecture. The first subsection focuses on the most distinguishing feature of a PIM as compared to a conventional processor, its memory unit and memory interface. Subsequently, we describe DIVA's WideWord unit, parcel interconnect and address translation mechanism.

3.1 Host Memory Interface and Memory Unit

The host interface and memory unit reflect a number of the challenges in designing a PIM that serves as a smart-memory co-processor to a conventional host. Our underlying goals were to minimize performance penalties to conventional memory accesses as viewed by the host, while maximizing the potential benefit of PIM operations. Although the original design targeted embedded DRAM, the prototype shown in Figure 1 is an SRAM-based design due to challenges in timely access to embedded DRAM fabrication lines. We first describe the resulting design implemented in this prototype chip and then present necessary considerations for a DRAM-based PIM design.

A PIM chip's host interface externally implements the JEDEC SDRAM protocol so that the PIM appears as a conventional SDRAM to the host processor. On-chip, the host interface communicates with an internal memory controller to negotiate access to the embedded memory. In essence, the host interface is a translator between the standard SDRAM protocol and an internal PIM-specific protocol. To satisfy the SDRAM timing requirement, this interface must ensure consistent timing for host memory accesses. At first glance, this may seem difficult to enforce since the PIM node processor may be accessing memory when a host memory request arrives, thereby causing the host access to incur an additional latency. However, a couple of factors allow the PIM to respond with predictable latency as required by the standard. First, the embedded SRAM macro of the prototype chip has a 3.5ns cycle time and 256-bit data bus. Secondly, the internal clock of the PIM is at least twice that of the SDRAM bus (4X for some implementations), so the addition of an arbitration cycle is negligible to the overall memory latency. Refer to Figure 5, which shows a timing diagram for a 3-cycle CAS latency SDRAM burst read operation. The

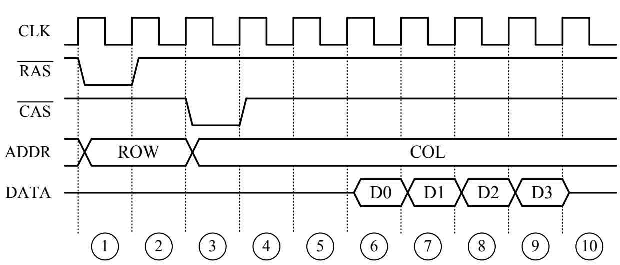


Figure 5: SDRAM burst read timing.

worst-case read latency occurs if the memory is busy satisfying a PIM processor request when the host request arrives. Even in this case, once the CAS strobe of cycle 3 in the figure has been detected, there is only a 4-PIM-cycle latency (2 SDRAM cycles) for the read request to be forwarded from the host interface to the internal node memory controller, serviced, and returned for output onto the SDRAM data bus. This allows the PIM to output the first 64-bit data word in cycle 6, satisfying the SDRAM protocol. Since all accesses to the embedded memory involve 256 bits of data, the succeeding 64-bit data words are readily available for the host interface to output them in cycles 7, 8, and 9. Similar timing applies for write accesses.

The internal node memory controller, shown in Figure 4, consists of two basic components: an *arbiter* and a *memory control unit*. The arbiter performs handshaking between requesters of memory accesses and determines priority of competing requests. Requests for accesses, *i.e.,* reads and writes, may originate from the host interface memory port, PIM processor instruction cache, and/or memory stage of the PIM processor pipeline. Arbitration priorities are formulated as follows: 1) host interface, 2) PIM processor memory stage, and 3) PIM processor instruction cache. The host interface has the highest priority since minimal latency is required for the PIM chip to appear as conventional SDRAM for host processor accesses. The arbiter communicates closely with the memory control unit, which is responsible for generating all control signals to the memory array, such as macro select, write enable, output enable, and address bits. Once a requester has been granted access, the requesting source drives the data bus and associated byte-write-enable signals for write accesses while the memory control unit drives the control signals. For read accesses, the requester simply latches data returned from the memory at the appropriate time.

For future DRAM-based implementations, the PIM chip memory system must be augmented to support refresh operations and page-mode accesses. For refresh operations, the host interface must be able to translate system memory controller refresh operations into internal refresh operations, which is a fairly straightforward exercise. To exploit page-mode accesses, the node memory controller should maintain a *current page address register*. For normal read/write accesses, the address presented with the request is compared against the contents of the current page address register, assuming that a page is currently open. If the portion of the requesting address which designates the DRAM page matches the value of the current page address register, the access is performed as a page-mode access. If the two values are unequal, a random access must be performed, which entails restoring the currently open page and strobing in the new page corresponding to the access request. Simultaneously with this access, the new page address is latched into the current page address register. Also, the current page address register is invalidated upon refresh operations, since refresh operations corrupt the values retained in the DRAM sense amps, which represent the currently open page.

Also, DRAM-based PIM implementations must carefully consider the SDRAM interface requirements. As an example, consider an implementation based on the DRAM macro provided by the IBM Cu-11 process [13]. Like the SRAM macro used in the first DIVA prototype chip, this macro supports a 256-bit data bus with byte-write-enable signals to support writes of data smaller than 256 bits, where needed. The macro page size is 2048 bits. The page-mode cycle time is 5ns, while the random-mode cycle time is 15ns.

If the system memory controller always initiates full burst requests, like the one shown in Figure 5, small modifications can be made to the internal PIM logic to satisfy the SDRAM timing requirements. As soon as a system memory controller RAS cycle is detected, the host interface should alert the internal memory controller to finish its current memory operation and remain idle in anticipation of a host request. Even with the current highest-speed SDRAM standard, 133MHz, there is a 38ns latency between the RAS cycle and when data is required for read operations for a 3-cycle CAS latency implementation. Based on the random-mode cycle times of the IBM DRAM macro mentioned above, this is adequate time for the internal node memory controller to complete its current memory cycle and respond to a host-initiated memory cycle. For systems with intelligent memory controllers that perform page-mode accesses (initiating CAS-only memory operations), the CAS latency must be configured to support the maximum PIM latency. The resulting memory latency penalty is highly system-dependent in this case.

3.2 WideWord Unit

The WideWord unit operates on 256-bit words, enabling applications to exploit fine-grain, or superword-level, parallelism and the increased processor-memory bandwidth available in a PIM node. The WideWord unit has the ability to change operand width on a per-instruction basis, enabling it to treat a WideWord operand as a packed array of objects of 8, 16, or 32 bits in size. With the exception of a few specialized instructions, this characteristic means the WideWord ALU is more generally represented as a number of parallel ALUs, where the number depends on operand size.

Besides conventional arithmetic and logic operations, the WideWord unit also supports a rich set of operations for manipulating data, including rearrangement of data within a WideWord operand, transfers between WideWord and scalar registers and packing and unpacking operations. Furthermore, the WideWord unit supports selective execution of instructions on a per-datapath basis, depending on the state of condition codes. The generality of these three features, as well as the ability to access main memory at very low latency, distinguish the DIVA WideWord capabilities from multimedia ISA extensions such as Intel SSE2 and PowerPC AltiVec, as well as subword parallelism approaches such as MAX [19]. We now discuss each of these in detail, and show examples of their use. In the examples, we use the convention that WideWord instructions and references to WideWord registers are both prepended with a 'w'.

Permutation. To rapidly align and reorganize data in WideWord registers, the WideWord unit has a permutation functional unit, which enables any 8-bit field of the source register to be moved into any 8-bit field of the destination register. A permutation is specified by a permutation vector, which contains 32 indices corresponding to the 32 8-bit subfields of a WideWord destination register, where each index selects which subfield of the source data is moved into that destination field. General permutations are specified such as `wprm wro,wri,wrp`, where wrp specifies the desired permutation vector to be applied to input wri, with the output in wro. Wrp is either constructed through a series of instructions or is loaded from memory. To bypass the cost of constructing or loading general permutations, commonly used permutations are instead specified such as `wprmi wro,wri,sr`, where sr is a scalar register that contains an index into a table of hard-wired permutations, such as shifts, rotates, shuffles, gathers, scatters and reductions.

Figure 6 illustrates the use of permute operations with an example of a reduction sum of the elements of an array (loaded into $wr1$). The reduction sum is performed in $log(n)$ steps, where n is the number of elements in $wr1$ (in the examples, $n = 4$, for simplicity). On each step, the first permutation swaps each even-numbered field $2i$ with its odd-numbered neighbor field $2i + 1$, $0 \leq i < n/2$, storing the result in $wr2$. Then the contents of $wr1$ and $wr2$ are added, resulting in the sum of each pair of even-/odd-numbered elements in each even-numbered field of $wr1$. Finally, all even-numbered partial sums are permuted into the lower half of $wr1$, reducing the problem size by half. After the last step, the sum of all elements is in field zero of $wr1$.

```
// sr1 refers to permutation (1,0,3,2)
// sr2 refers to permutation (0,2,1,3)
wld wr1,&array);       // wr1 = (a0,a1,a2,a3)
// step 1:
wprmi wr2,wr1,sr1;     // wr2 = (a1,a0,a3,a2)
wadd wr1,wr1,wr2;      // wr1 = (a0+a1,a0+a1,a2+a3,a2+a3)
wprmi wr1,wr1,sr2;     // wr1 = (a0+a1,a2+a3,*,*)
// step 2:
wprmi wr2,wr1,sr1;     // wr2 = (a2+a3,a0+a1,*,*)
wadd wr1,wr1,wr2;      // wr1 = (a0+a1+a2+a3,*,*,*)
```

Figure 6: Reduction sum using permutations.

Register Transfers. To enable efficient data movement between scalar and WideWord register files, the WideWord unit supports a set of transfer instructions. The transfer instructions include `mvswr wr,sr`, which replicates the contents of scalar register sr into all fields of WideWord register wr, `mvsw wr,sr,i`, which copies sr only to the immediate i field of wr, and `mvws sr,wr,i`, which copies the contents of sr to immediate i field of wr.

Figure 7 illustrates the use of transfer instructions in a mixed irregular-regular computation where it is advantageous to perform the regular computation in the WideWord unit and the irregular in the scalar ALU. In this example, the multiplication of $A[k] : A[k+3]$ by X can be performed in the WideWord unit, since array A is accessed with stride one. To allow the vector-scalar multiplication to be performed in parallel, X is replicated into a WideWord register. It is not profitable to perform the addition of $Y[R[k]]$ and $A[k] * X$ in the WideWord unit, since it would be necessary to pack $Y[R[k]]$ to $Y[R[k+3]]$ into a WideWord register and check whether $R[k] : R[k+3]$ are distinct values. Nevertheless, the computation of the addresses (base address of Y plus offsets $R[k] : R[k+3]$) can still be performed in parallel, as shown in the example. Finally, the operands are moved to scalar registers and the additions are performed in the scalar unit.

Selective execution. Selective execution is supported for most arithmetic and logic instructions, permutation instructions and some transfer instructions. Under selective execution, only the results corresponding to participating subfields are written back to the destination register specified in the instruction. Therefore, the implementation of selective execution requires that writeback enable bits be associated with each 8-bit subfield of the ALU result, which complicates the register forwarding logic somewhat. The determination of whether a subfield participates in the execution of a given instruction is derived from condition codes, two special-purpose registers, and a field in the instruction. One special-purpose register is a user-settable 32-bit *mask register*, where each bit corresponds to an 8-bit subfield of the operation. The *participation mode register* is a 5-bit register that specifies the condition for selective execution as a combination of condition codes and/or mask register. A 2-bit participation field in the instruction specifies one of four possible extents of selective execution: local participation, where a subfield participates if its local condition (as derived from the participation mode and mask register values and condition codes) is true; leftmost/rightmost participation, where only the leftmost/rightmost subfield with a condition that is true participates; and always participate, where

```
// Original loop:
// for (k = 0; k < N; k ++)
//    Y[R[k]] = Y[R[k]] + A[k] * X

// sr1 = base address of Y (&Y)
// sr2 = address of A[k] (&A[k])
// sr3 = address of X

// compute A[k]*X:A[k+3]*X in WideWord
wld wr4,sr2;           // wr4 = (A[k]:A[k+3])
ld sr4,sr3;            // sr4 = X
mvswr wr5,sr4;         // wr5 = (X,X,X,X)
wmul wr6,wr4,wr5;      // wr6 = (A[k]*X:A[k+3]*X)

// compute addresses &Y[R[k]]:&Y[R[k+3]] in WideWord
mvswr wr1,sr1;         // wr1 = (&Y,&Y,&Y,&Y)
wld wr2,&R[k];         // wr2 = (R[k]:R[k+3])
wsll wr2,wr2,2;        // convert to address offset
wadd wr3,wr1,wr2;      // wr3 = (&Y[R[k]]:&Y[R[k+3]])

// compute Y[R[k]]+A[k]*X in scalar unit
mvws sr10,wr3,0;       // sr10 = &Y[R[k]]
ld sr11,sr10;          // sr11 = Y[R[k]]
mvws sr12,wr4,0;       // sr12 = A[k]*X
add sr13,sr11,sr12;    // sr13 = Y[R[k]] + A[k]*X
st sr10,sr13;          // Y[R[k]] = Y[R[k]] + A[k]*X

// compute Y[R[k+1]]+A[k+1]*X in scalar unit
...
```

Figure 7: Irregular computation using both WideWord and scalar ALUs.

all subfields participate. Although similar designs support some type of conditional operations, the DIVA WideWord unit provides a much richer functionality through the ability to specify selective execution in almost every wide instruction and the use of global condition code information in selection decisions.

This distinction is illustrated in Figure 8. The instruction `wsubcc` subtracts X from elements of array C and sets the condition code for each 32-bit field. Then the subsequent instruction `waddlc`, where `lc` specifies local participation, performs an addition only on those fields for which the GT condition code is set.

```
// Original loop:
// for (k = 0; k < N; k ++)
//    if (C[k] > X)
//       A[k] = A[k] + B[k]

// set participation mode register (PM)
ori r2, r0, "GT"
mtspr PM, r2

wld wr1, &A;              // wr1 = (a0,a1,a2,a3)
wld wr2, &B;              // wr1 = (b0,b1,b2,b3)
wld wr3, &C;              // wr1 = (c0,c1,c2,c3)
ld r1, &X;                // r1 = X
wmvswr wr4,r1;            // wr4 = (X,X,X,X)
wsubcc wr5,wr3,wr4;       // wr5 = (c0-X,c1-X,c2-X,c3-X)
waddlc wr1, wr1, wr2;     // if (C > X) A = A+B
```

Figure 8: Selective update example.

3.3 Parcel Interconnect

Even for applications where the WideWord instructions are not applicable, the WideWord datapath is used to accelerate all parcel communication, as will be discussed here. As described earlier, the PIM Routing Component (PiRC) not only implements the PIM-to-PIM interconnect but also interacts with parcel buffers (PBUFs), the basic on-chip hardware mechanisms supporting parcels. The PBUF has a virtual as well as a physical abstraction. To the application, the PBUF locations appear as regular memory locations that are manipulated through simple loads and stores. At a physical level, the PBUF is a set of memory-mapped registers. Each PIM node contains a PBUF that serves as a port between the on-chip parcel interconnect and the node. Although a PIM node's PBUF could be implemented as special-purpose registers, a memory-mapped mechanism allows a uniform implementation for both node and host PBUF. The PBUF within the PIM chip host interface is memory-mapped into the host processor's address space to permit host and PIM parcel communication.

A parcel consists of a 96-bit header and 256-bit payload. Most of the parcel contents are written by the user program during a parcel launch; however, the system is responsible for generating some fields such as PiRC routing information, source node ID, process identifier, and interrupt status. The user program is responsible for specifying header fields that include the virtual address of the object to which the parcel is directed and a specification of the command to execute on that object. In addition, the user program specifies the 256-bit payload, which consists of arguments for the command task or other data associated with the action specified by the parcel.

Data is written to or read from the PBUF in 256-bit increments via the WideWord registers. The PBUF address space can then be viewed as a set of 256-bit registers. Besides the header and payload registers, there are also status and configuration registers. Although the payload is the only true physical 256-bit register, each register is allocated 256 bits of the address space and is aligned to the least significant bit boundary. At least two register sets are needed: one for sending and one for receiving. In addition, it is desirable to have multiple address mappings (aliases) of these sets to support different access privileges and modes, such as non-launching and launching writes to the send registers, destructive and non-destructive reads from the receive registers, and interrupt capability. The DIVA design includes several aliases to support such mechanisms.

3.4 Address Translation Hardware

The primary functions of the node address translation unit are to translate virtual addresses to physical addresses for those accesses which are locally resident and to provide access protection. The types of accesses generated by a DIVA PIM processor that require translation include instruction fetches and data accesses to memory or memory-mapped devices such as parcel buffers, generated by load or store instructions. Given the simplicity of the segment-based address translation scheme discussed in Section 2, very little hardware support is needed to effect efficient translation. The necessary descriptors for a local memory segment are a physical base address register, offset limit register, and access privilege control bits. For global memory segments, an additional virtual base address register is useful to effect efficient translation, as described below. The initial DIVA architecture provides eight sets of local segment registers and four sets of global segment registers. If an application requires a number of segments that is more than that supported by the translation hardware, the PIM run-time kernel must manage the configuration of the translation hardware to minimize address faults. Like pages in a conventional system, segments and their associated descriptors are generic in nature. It is only through system programming that a segment serves a specific purpose, such as representing user code or data segments.

To distinguish between local and global segments, we arbitrarily, but with little loss of generality, specify that the upper 5 bits of a virtual address generated by a PIM processor indicate the *scope* of the address. The value of the scope field determines what type of translation, if any, is used (see Figure 9). For local translation, bits 5 through 7 are used as an index value to select one of eight sets of local segment descriptors for translation and protection checking. The rest of the virtual address represents an offset from the segment base address. Unlike the table look-up style of local translation, for global translation it is more efficient to determine if the virtual address is contained within the span of a global segment. Thus, if the scope value indicates global translation, a fully-associative lookup is performed using the global segment descriptors. Also, as shown in the figure, a supervisor-level untranslated region that spans the exception handler addresses has been reserved. This feature is useful for kernel code to run diagnostics, such as verifying the operation of the address translation hardware without being incapacitated by related hardware errors.

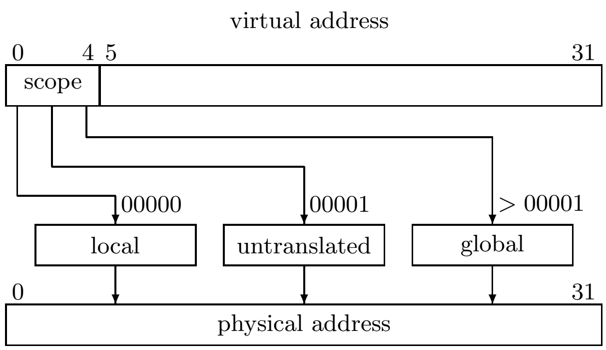


Figure 9: Address translation in DIVA PIMs.

4. EXPERIMENTAL RESULTS

4.1 Applications

To measure the performance potential of the DIVA architecture, we examine in detail eight benchmark applications, summarized in Table 1. These applications span a broad range of domains including scientific computing, databases and image processing. They exhibit both coarse-grain parallelism (which allows computation to be spread across PIMs) and, in some cases, fine-grain parallelism (which can be exploited through execution in the WideWord unit). CG, Neighborhood, Pointer, OO7 and Natural Join exhibit irregular or mixed (regular and irregular) data access patterns, resulting in high memory access overheads on conventional architectures. Cornerturn, Transitive Closure and Template Matching are dense matrix computations with regular access patterns, but memory bandwidth becomes a limiting factor in exploiting available parallelism. These three and CG rely on the WideWord unit to exploit parallelism and PIM bandwidths. Hereon, we use abbreviations for each of the program names, with a suffix -H for host and -P for PIM.

4.2 Simulation Environment and Parameters

To evaluate the DIVA architecture, we developed a system simulator called DSIM, which uses RSIM as a framework, with significant extensions [24]. RSIM is an event-driven simulator that models shared-memory multiprocessors built with state-of-the-art multiple issue, out-of-order superscalar processors. DSIM extensions include a simpler PIM processor with a WideWord unit, the DIVA memory system, the parcel communication mechanism and the PIM-to-PIM interconnect. DSIM supports the full DIVA PIM ISA.

The DSIM host processor is taken directly from RSIM, including the first and second-level caches. The host processor architecture is based on the MIPS R10000 and is configured as a four-issue processor with two integer arithmetic units, two floating-point units and one address unit. Loads are non-blocking. It has a 32Kbyte L1 and a 1Mbyte L2 cache, both two-way associative, with access times of 1 and 10 cycles, respectively. Both L1 and L2 caches are pipelined and support multiple outstanding requests.

The host is connected to the DIVA memory system via a split-transaction, 64-bit bus. The memory system consists of the aggregation of all PIM memories, where each local memory is visible from both host and local PIM processor. DSIM maintains the current open row of each memory bank to determine the memory access type (page or random mode) and simulates arbitration between host and PIM accesses, as described in Section 3.1. The memory latencies seen by the host are 52 cycles for page-mode accesses and 60 cycles for random mode, and include the bus transfer delay, the memory arbitration time and the DRAM access time (4 and 12 cycles for page and random mode, respectively). The memory latencies seen by the local PIM processor, including arbitration and DRAM access times, are 5 and 13 cycles for page and random mode accesses, respectively.

An application library supports a cache-line flush to enforce coherence between the host caches and PIM memory, as well as synchronization and communication functions. These functions are linked with the application, and their execution is simulated by DSIM in the same way as the application code. DSIM also models the parcel mechanism and the PIM-to-PIM interconnect in detail, but we omit further description since this paper focuses on 1-PIM performance.

For these experiments, we make the conservative assumption that the PIM processor runs at half the speed of the host processor. Although the inherent speed of the logic is no slower [13], we make this assumption because the subcomponents of the PIM processing logic run in lock-step, so the resulting clock speed is slower than that of superscalar schemes.

4.3 Performance Compared Against Host

Figure 10 summarizes 1-PIM performance as compared to execution on the conventional host processor. Five of the eight programs speed up significantly compared against host execution, two remain about the same, and one program is slowed down. (All programs speed up when multiple PIMs are used.) Overall, the average speedup is 3.3X. Several factors contribute to these speedups, including the lower memory stall times on the PIM nodes and the benefits of the WideWord unit in exploiting fine-grain parallelism

Program	LOC	Description	Source	Data Set Size	WideWord Usage
Template Matching (TM)	815 (C)	image correlation	Sandia	4-Kbyte image, 32 1-Kbyte templates	parallelism, selective, reuse in registers, page mode
Cornerturn (CT)	177 (C)	matrix transpose	Atlantic Aerospace	32-Mbyte matrix	parallelism, permutation
CG	857 (FORTRAN)	sparse conjugate gradient	NAS	2M double precision elements	parallelism, floating point, page mode
Transitive Closure (TC)	202 (C)	Floyd's all pairs shortest paths	Atlantic Aerospace	256 Kbytes	parallelism, selective, reuse in registers
Natural Join (NJ)	13144 (C)	relational database join	Alphatech	72 Kbytes	
Neighborhood (NH)	2098 (C)	image processing stencil	Atlantic Aerospace	500,000 bytes	
Pointer (P)	252 (C)	random walk	Atlantic Aerospace	4 Mbytes	
OO7	8000 (C++)	object-oriented database query	University of Wisconsin	888 Kbytes	

Table 1: Application description.

and taking advantage of page-mode memory accesses. The remainder of this section examines these factors in detail.

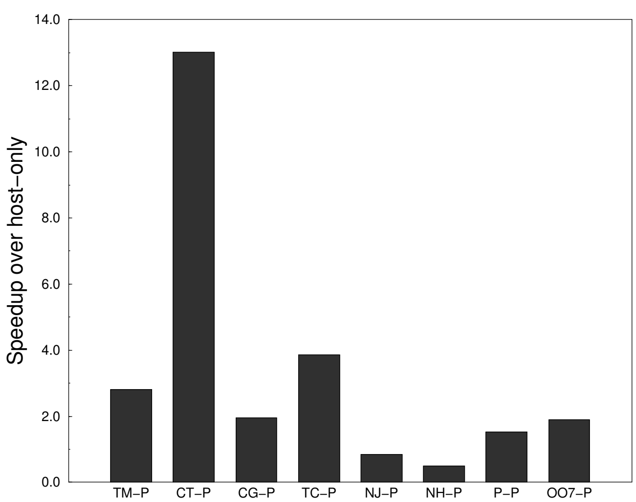


Figure 10: Speedups over host-only execution.

4.4 Reduction in Memory Stall Time

Figure 11(a) shows the memory stall times of host-only execution. PIMs reduce memory stall time in two ways: (1) lower latency to memory; and, (2) higher bandwidth to memory through wide loads and stores. (A third reduction occurs as a result of coarse-grain parallelism across the PIMs, which is not discussed in this paper.) We see from the figure that five of the eight programs spend more than 40% of their time stalled in memory accesses. DIVA achieves a reduction in memory stall time for these five programs ranging from 13.89% for Natural Join to 95% for Cornerturn, as shown in Figure 11(b).

The host version of Template Matching (TM-H) has a memory stall time of only 3% of its total execution time. The data set size of this application fits in the L2 cache, and the working set of each loop fits in the L1 cache; therefore, the data reuse exhibited by TM is effectively exploited. Even though TM-H does not suffer from large memory stall times, the 1-PIM version (TM-P) has even smaller stall times due to the high data bandwidth at the PIM node. The use of the WideWord unit for loading/storing and operating on 256-bit objects, plus the reuse of data in WideWord registers reduces the memory stall time to 20% of that of TM-H.

Cornerturn has a memory stall time of 90.17% when running on the host. This application has very little temporal reuse, since each matrix element is accessed only twice (one read and one write) during the matrix transpose. Thus primarily spatial reuse is exploited in cache, and each new cache line is only reused a few times. In the PIM version, the WideWord datapaths also exploit the available spatial reuse. Furthermore, the WideWord loads/stores and operations on 8 matrix elements at a time also reduce the number of accesses to memory. Finally, the latency seen by the PIM processor (average of 11.57 cycles, since most of the accesses are in random mode) is much lower than that suffered by the host. The combination of these factors reduce the CT-P memory stall time to 4.32% of that of CT-H.

CG also benefits from the lower memory latencies on the PIM node. Since the data set size does not fit in the host caches and the irregular access patterns cause conflict misses, CG-H spends 85.21% of its execution time stalled due to cache misses. Although most of the misses are satisfied at the L2 cache (51.32%), 46% of the stall time is due to accesses to the DRAM. On the PIM, 78% of the memory accesses are page-mode accesses, and the average latency seen by the processor is only 5.91 cycles.

Transitive Closure on the host, TC-H, spends 70% of its execution time stalled due to cache misses, with 47.14% of the misses satisfied at the L1 and 52.81% satisfied at the L2, resulting in an average miss latency of 6.23 cycles. For TC-P, the average memory latency is 5.57 cycles, due to 67% of page-mode accesses. In addition to lower memory latencies, TC-P also has a smaller number of memory accesses since the WideWord unit is used to transfer the data to/from memory and perform the computation. The use of the WideWord unit results in the added benefit of exploiting spatial reuse, since the matrix is accessed with stride one in the row dimension.

Neighborhood shows an increase in memory stall time because the data fits in cache, and thus the memory latency at the PIM is larger than that of the host. The increase in memory stall time plus the fact that the PIM processor runs at half the speed of the host result in a slowdown with respect to host-only execution.

Pointer has no spatial reuse and little temporal reuse, and since the data set size is larger than the L2 cache, P-H stalls for memory for 49.8% of its execution time, with most misses satisfied at the DRAM. P-P has roughly the same number of loads and stores, but the average latency seen by the PIM is much smaller than the memory latency suffered by the host, even though most of the PIM acesses are random-mode accesses.

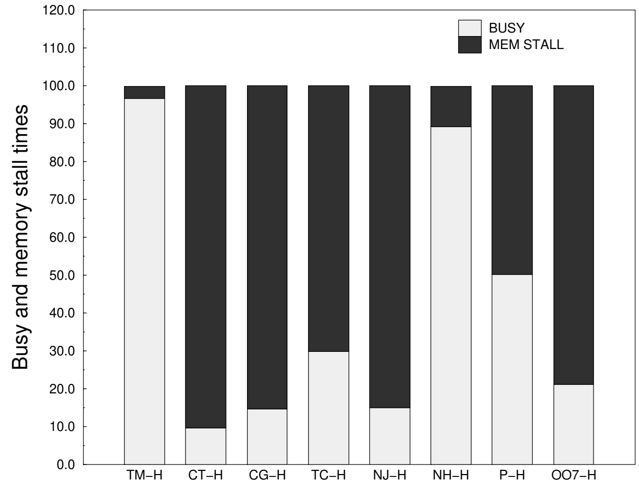
Natural Join exhibits little temporal reuse and high cache miss rates, even though the data set size fits in the L2 cache. NJ-P shows a reduction of 13.8% in memory stall times due to the lower average latency seen by the PIM processor. OO7 also has almost no temporal reuse and OO7-H suffers from a large amount of cache misses. On the PIM version the memory stall time is reduced by 62.8%, again as a result of the smaller on-chip latency.

4.5 Benefits from WideWord and Page Mode

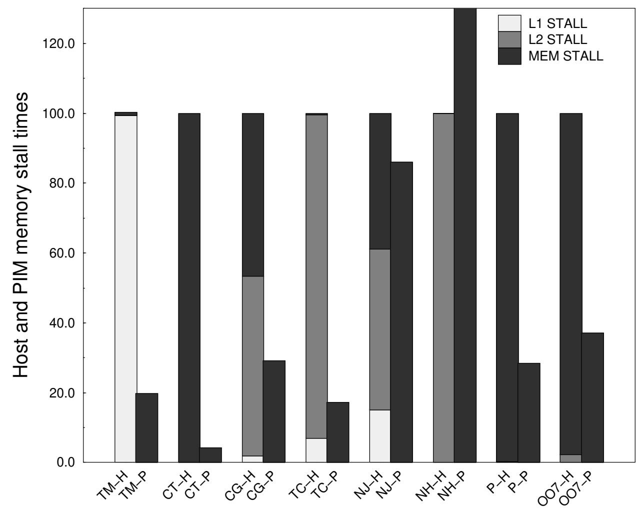
To isolate the benefit of the WideWord unit, we compare scalar versions against versions tuned to take advantage of the WideWord unit and page-mode memory accesses for the four programs that utilize the wide datapaths. These results are shown in Figure 12. Speedups are significant, ranging from 1.19X for CG up to 17.96X for TM, with an average improvement of 9.93X. The features of the instruction set that are exploited are summarized in the final column of Table 1, and described as follows.

TM computes three correlation values between an image and each of 32 templates, each correlation corresponding to a loop nest. The DIVA implementation, which is described in detail in [6], takes advantage of the inherent fine-grain parallelism by operating on 32 8-bit image pixels and 32 8-bit template elements at a time. Since a template is represented as a 32-by-32 matrix of 8-bit elements, an entire template row fits into one WideWord register. Also, since the innermost loop traverses one template row, the entire inner loop computation is transformed into a sequence of WideWord operations on one template row and 32 pixels of an image row, effectively eliminating the innermost loop. The accumulation of the pixel values is achieved by a parallel reduction sum, using permutation operations as in Figure 6, and the result of the reduction sum is added to the correlation value using selective execution as in Figure 8. To exploit temporal reuse in WideWord registers, we applied common loop transformations, particularly unroll-and-jam [4]. In addition, we exploited spatial reuse by shifting an image subrow held in a WideWord register by one pixel, to move the window of the image to be compared against the template. Further performance improvements are obtained by reordering memory accesses and grouping streaming accesses to the dense arrays to achieve page-mode memory access latencies.

The CT implementation performs a hierarchical in-place matrix transpose where the smallest submatrices, of size 8x8, are transposed in WideWord registers. Each 8x8 sub-



a) Busy and memory stall times for host-only execution.



b) Host-only and 1-PIM memory stall times.

Figure 11: Memory stall times.

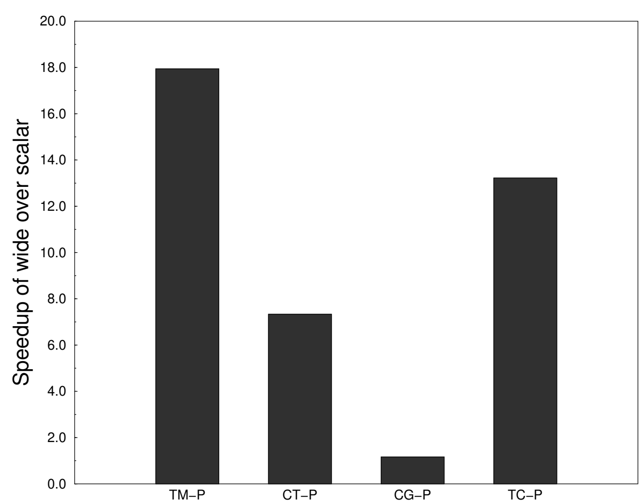


Figure 12: Speedup of WideWord vs. scalar.

matrix is loaded into the WideWord register file (an 8x8 matrix with 32-bit elements requiring 8 WideWord registers), and transposed via a sequence of permutation operations. The transposed submatrix is then stored back in memory. This implementation takes advantage of the large capacity of the WideWord register file, avoiding loads and stores to memory during the transpose of each 8x8 submatrix.

CG's key computation is a sparse matrix-vector multiply. Due to the mixed regular/irregular nature of data accesses, we only exploit fine-grain parallelism in the WideWord unit for the regular portions of the computation. The dense vector accesses are loaded into WideWord registers, and the dense vector multiplies are performed in the WideWord floating-point unit. The accumulates into the sparse matrix are performed sequentially. Selective execution is used to select the field of the WideWord operand that participates in the operation. As in TM, we also reordered memory accesses to achieve page-mode latencies on the dense arrays.

TC uses a dense matrix to represent the distance graph. It exploits fine-grain parallelism by performing WideWord arithmetic operations on eight 32-bit elements of the matrix that are held in WideWord registers. Selective execution using WideWord operation `wmrgcc` merges the contents of two WideWord registers according to condition-code bits, allowing an efficient computation of the minimum value of each pair of elements of two WideWord operands. Similar to TM, we use unroll-and-jam to obtain temporal reuse in the WideWord register file.

5. STATUS

The first DIVA PIM prototype, shown in Figure 1, is an SRAM-based single-node implementation of the DIVA PIM chip architecture and is currently in test. To minimize silicon area of this SRAM-based prototype, we used a 1-Mbyte memory macro. For comparison, a DRAM-based implementation with a 2-Mbyte macro could be fabricated in approximately half the area of the SRAM-based prototype. The current prototype chip implements all features of the DIVA PIM architecture except address translation and floating-point capabilities. A second version of a PIM chip, which not only integrates these functions but achieves a faster clock rate, is due to tape out in the second half of 2002.

The current chip was fabricated through MOSIS in TSMC 0.18μm technology, and the silicon die measures 9.8mm on a side. It contains approximately 2 million logic transistors in addition to the 53 million transistors that implement 8 Mbits of SRAM. The chip also contains 352 pads, 240 signal I/O, and is packaged in a 35mm BGA. Much of the logic was synthesized with Synopsys Design Analyzer, and the entire chip was placed and routed with Cadence Silicon Ensemble. The IP building blocks used in the chip include Artisan standard cells and register files, Virage Logic SRAM, and a NurLogic PLL clock multiplier.

The chip is currently being tested for functionality with the use of pattern generators, which apply test vectors to input pins, and logic analyzer modules, which sense the outputs. Although exhaustive testing has not yet been completed, the chip is correctly executing at 160MHz on the Cornerturn matrix transpose kernel described in Section 4, exercising all major control and datapaths within the PIM processing logic, including the WideWord permutation unit. Even in this limited test setup, the chip performs 1.28 GOPS while dissipating only 800mW. In addition to the processing logic functionality, correct operation of parcels transiting through the PiRC has also been verified.

We will soon begin integrating PIM-based DIMMs into a workstation-class development system, incorporating compiler and system software technology. For the host operating system, we have augmented Linux to include PIM-specific support, such as loading PIM code and data, booting, process management, and the memory management functions outlined in [10]. We have also developed components of the PIM run-time kernel, an augmented version of the RTEMS open-source real-time embedded operating system. We have developed a prototype compiler for the DIVA PIMs, which takes as input sequential Fortran or C code, and produces DIVA executables that exploit both the scalar and WideWord unit. We leverage the SUIF compiler, including extensions described in [18] and our own implementation of transformations described in [6], and a GCC backend for the PowerPC AltiVec. A system-level compiler is an area of future work.

6. RELATED WORK

The DIVA system architecture is focused on achieving the following four goals: (1) developing PIMs that can serve as the only memory in the system, assuming the dual roles of "smart memories" and conventional memory; (2) supporting a wide range of familiar programming paradigms, closely related to parallel computing; (3) targeting applications that are severely impacted by the processor-memory bottlenecks in conventional systems: sparse-matrix and pointer-based applications with irregular memory access patterns, and image and video applications with large working sets; and, (4) developing a VLSI device to exploit memory and communications bandwidth in PIM-based systems while making efficient use of on-chip resources for target applications.

These four goals distinguish DIVA from other PIM-based architectures. Integration into a conventional system affords the simultaneous benefits of PIM technology and a state-of-the-art host, yielding high performance for mixed workloads. Since PIM processors are usually less sophisticated due to on-chip space constraints, systems using PIMs alone in a multiprocessor may sacrifice performance on uniprocessor computations [12, 16, 25, 27], while system-on-a-chip solutions (e.g., the IRAM [22] and the Mitsubishi M32R/D [20]) limit the application domain. DIVA's support for a broad range of familiar parallel programming paradigms, including task parallelism for irregular computations, distinguishes it from systems with restricted applicability (such as to SIMD parallelism [7, 8, 22]), as well as those requiring a novel programming methodology or compiler technology to configure logic [1], or to manage a complex memory, computation and communication hierarchy [15]. DIVA's PIM-to-PIM interconnect improves upon approaches that serialize communication through the host, which decreases bandwidth by adding traffic to the processor-memory bus [8, 21].

With respect to DIVA's WideWord unit, Table 2 compares the features described in Section 3.2 with two commercial multimedia extensions that support superword parallelism, PowerPC AltiVec and Intel SSE2, as well as a previous research design called ASAP [2]. (Most other multimedia extensions support *subword* parallelism, which performs parallel operations on subfields of a machine word.) The ASAP combines WideWord and scalar capabilities in a single unit. This approach eliminates the need for trans-

Capability	**SSE2**	**AltiVec**	ASAP	DIVA
Separate scalar, WideWord units	√	√	×	√
Permutation	immediate	general	general	general, indirect
Register transfers	√	×	n/a	√
Selective execution	limited	limited	√	√

Table 2: Comparison with other superword-level parallelism approaches.

fers between register files, but with register forwarding, it can complicate the pipeline and slow down the clock rate. All other implementations have separate scalar and WideWord units and register files, and other than DIVA, only SSE2 includes transfers between register files. The absence of such capability was reported to be a performance bottleneck in the AltiVec [18]. AltiVec and ASAP support only general permutations, where permutation vectors are read from memory or constructed by instructions. Both SSE2 and DIVA can avoid these costs of deriving a permutation vector through hardwired permutation operations. In the case of SSE2, permutation operations can only be expressed through immediates, so the permutation must be known at compile time. DIVA's hardwired permutation, which is in addition to general permutation, is indirect because it references a scalar register. Hardwired indirect permutations are more powerful than immediate permutations, in that we can use nearby permutations for different iterations of a loop without requiring unrolling (*e.g.,* to do alignment). DIVA provides a detailed reference design and implementation of selective execution, related to the concept discussed in [2], that supports selective execution in almost every wide instruction. By comparison, since the AltiVec does not incorporate selective execution of arithmetic operations, to accomplish the same result as in Figure 8 on the AltiVec would require an additional instruction to commit only those fields of the result of the add for which the condition code is set.

We further consider a performance comparison with the PowerPC AltiVec 74XX. Even with a very aggressive DRAM technology, the 74XX can achieve a peak main memory bandwidth which is only one third that of the PIM DRAM. While the 74XX has better bandwidth for problems which fit into the 256KB on-chip L2 cache, for our benchmarks with high memory stall times, a single DIVA PIM processor will outperform the AltiVec despite a much smaller transistor count on a DIVA PIM. Further, since each DIVA system will include many interconnected PIM chips, the performance advantage will scale with increasing memory size for problems amenable to coarse-grain parallel computation.

7. CONCLUSION

This paper has presented a detailed description of the DIVA PIM microarchitecture. We discuss some of the issues that must be considered in future architectures for exploiting memory bandwidth, particularly the memory interface and controller, instruction set features for fine-grain parallel operations, and mechanisms for address translation. We present simulation results on eight programs, demonstrating an average speedup of 3.3X as compared to a conventional host. The speedups are due to up to 95% reduction in memory stall time, and, for four of the programs, an average speedup of 9.93X due to fine-grain parallelism in the WideWord unit as compared to scalar PIM execution. As a result of these effects, six of the programs show fairly significant speedups over host-only execution with just one PIM, even though the PIM processor is an in-order, single-issue processor running at half the speed of the host, which is an out-of-order 4-issue processor. These 1-PIM speedups suggest DIVA's potential to outperform conventional multiprocessors for certain applications, and at a much reduced hardware cost.

Acknowledgments

The authors wish to thank Peter Kogge and Jay Brockman at the University of Notre Dame for their contributions to the design of the DIVA PIM microarchitecture, and Thomas Sterling for his early contributions to the project. The authors also wish to acknowledge the work of others previously on the DIVA team at USC/ISI who contributed in some way to the work described in this paper, including Jeff Koller, Michael Gorman and Ruoming Pang. The DIVA project is sponsored by DARPA contract F30602-98-2-0180.

8. REFERENCES

[1] J. Babb et al. Parallelizing applications into silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1999.

[2] J. Brockman et al. Microservers: A new memory semantics for massively parallel computing. In *Proceedings of the ACM International Conference on Supercomputing*, pages 454–463, June 1999.

[3] D. Burger, J. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[4] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1994.

[5] J. Carter et al. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.

[6] J. Chame, M. Hall, and J. Shin. Code transformations for exploiting bandwidth in PIM-based systems. In *Proceedings of the ISCA Workshop on Solving the Memory Wall Problem*, June 2000.

[7] D. Elliott et al. Computational RAM: Implementing processors in memory. *IEEE Design and Test of Computers*, pages 32–41, January – March 1999.

[8] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the Terasys massively parallel PIM array. *IEEE Computer*, pages 23–31, Apr. 1995.

[9] M. Hall et al. Mapping irregular applications to DIVA, a PIM-based Data-Intensive Architecture. In *Proceedings of Supercomputing*, Nov. 1999.

[10] M. Hall and C. Steele. Memory management in a PIM-based architecture. In *Proceedings of the ASPLOS Workshop on Intelligent Memory Systems*, Nov. 2000.

[11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 2 edition, 1996.

[12] IBM. http://researchweb.watson.ibm.com/bluegene/.

[13] IBM Microelectronics. http://www.chips.ibm.com/products/asics/products/edram.

[14] C. W. Kang and J. Draper. A fast, simple router for the Data-Intensive Architecture (DIVA) system. In *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, Aug. 2000.

[15] Y. Kang et al. FlexRAM: Toward an advanced intelligent memory system. In *Proceedings of the IEEE International Conference on Computer Design*, Oct. 1999.

[16] P. Kogge. The EXECUBE approach to massively parallel processing. In *Proceedings of the International Conference on Parallel Processing*, Aug. 1994.

[17] P. Kogge, T. Giambra, and H. Sasnowitz. RTAIS: An embedded parallel processor for real-time decision aiding. In *Proceedings of NAECON*, Mar. 1995.

[18] S. Larsen and S. Amarasinghe. Exploiting superword-level parallelism with multimedia instruction sets. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation*, 2000.

[19] R. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, Aug. 1996.

[20] Mitsubishi. http://www.mitsubishi-chips.com/data/datasheets/mcus/m32rdgrp.html.

[21] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A model of computation for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[22] D. Patterson et al. A case for intelligent DRAM: IRAM. *IEEE Micro*, Apr. 1997.

[23] P. Ranganathan, S. Adve, and N. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[24] Rice University. http://www-ece.rice.edu/ rsim.

[25] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[26] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple COMA. In *Proceedings of the Symposium on High-Performance Computer Architecture*, Dec. 1995.

[27] T. Sterling. An introduction to the Gilgamesh PIM architecture. In *Euro-Par*, pages 16–32, Aug. 2001.

[28] T. Sunaga et al. A processor in memory chip for massively parallel embedded applications. *IEEE Journal of Solid State Circuits*, pages 1556–1559, Oct. 1996.

[29] T. von Eicken, D. Culler, S. C. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[30] J. Zawodny, P. Kogge, J. Brockman, and E. Johnson. Cache-in-memory: A lower power alternative. In *Proceedings of the ISCA Workshop on Power-Driven Microarchitecture*, June 1998.