ICS

TECHNICAL REPORT

Architectural Adaptation for Application-Specific Locality Optimizations

Xingbin Zhang, Ali Dasdan, Martin Schulz,
Rajesh K. Gupta[†], and *Andrew A. Chien*

Technical Report #97-09
March 1997

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
E-mails: {zhang,dasdan,mschulz1,achien}@cs.uiuc.edu

[†]Department of Information and Computer Science
University of California
Irvine, CA 92697
E-mail: rgupta@ics.uci.edu

Information and Computer Science
University of California, Irvine

Architectural Adaptation for Application-Specific Locality Optimizations

Xingbin Zhang, Ali Dasdan, Martin Schulz, Rajesh K Gupta[†], and *Andrew A Chien*

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{zhang,dasdan,mschulz1,achien}@cs uiuc edu

[†]Information and Computer Science
University of California
Irvine, CA 92697
rgupta@ics uci edu

Abstract

We propose a machine architecture that integrates programmable logic into key components of the system with the goal of customizing architectural mechanisms and policies to match an application. This approach presents an improvement over traditional approach of exploiting programmable logic as a separate co-processor by preserving machine usability through software and over traditional computer architecture by providing application-specific hardware assists. We present two case studies of architectural customization to enhance latency tolerance and efficiently utilize network bisection on multiprocessors for sparse matrix computations. We demonstrate that using application-specific hardware assists and policies can provide substantial improvements in performance on a per application basis. Based on these preliminary results, we propose that an application-driven machine customization provides a promising approach to achieve high performance and combat performance fragility.

I. Introduction

Technology projections for the coming decade [1] point out that system performance is going to be increasingly dominated by intra-chip interconnect delay. This presents a unique opportunity for programmable logic as the interconnect dominance reduces the contribution of per stage logic complexity on performance and the marginal costs of adding switching logic in the interconnect. However, the traditional *co-processing* architecture of exploiting programmable logic as a specialized functional unit to deliver a specific application suffers from the problem of machine retargetability. A system generated using this approach typically can not be retargeted to another application without repartitioning hardware and software functionality and reimplementing the co-processing hardware. This retargetability problem is an obstacle toward exploiting programmable logic for general purpose computing.

We propose a machine architecture that integrates programmable logic into key components of the system with the goal of customizing architectural mechanisms and policies to match an application. We base our design on the premise that communication is already critical and getting increasingly so [17], and flexible interconnects can be used to replace static wires at competitive performance [6], [9], [20] Our approach presents an improvement over co-processing by preserving machine usability through software and over traditional computer architecture by providing application-specific hardware assists. The goal of application-specific hardware assists is to overcome the rigid architectural choices in modern computer systems that do not work well across different applications and often cause substantial performance fragility. Because performance fragility is especially apparent on memory performance on systems with deep memory hierarchies, we present two case studies of architectural customization to enhance latency tolerance and efficiently utilize network bisection on multiprocessors. Using sparse matrix computations as examples, our results show that customization for application-specific optimizations can bring significant performance improvement (10X reduction in miss rates, 100X reduction in data traffic), and that an application-driven machine customization provides a promising approach to achieve robust, high performance.

The rest of the paper is organized as follows. Section II presents our analyses of the technology trends and the project context, and Section III describes our proposed architecture. We describe our case studies in Section IV and discuss related work in Section V Finally, we conclude with future directions in Section VI.

II BACKGROUND

Technology projections for the coming decade point out a unique opportunity of programmable logic. However, the traditional co-processing approach of exploiting programmable logic suffers from the problem of machine retargetability, which limits its use for general purpose applications.

A. Key Technology Trends

The basis for architectural adaptation is in the key trends in the semiconductor technology. In particular, the difference in scaling of switching logic speed and interconnect delays points out increasing opportunities for programmable logic circuits in the coming decade. Projections by the Semiconductor Industry Association (SIA) [1] show that on-chip system performance is going to be increasingly dominated by interconnect delays. Due to these interconnect delays, the on-chip clock periods will be limited to about 1 nanosecond, which is well above the projections based on channel length scaling [1]. Meanwhile, the unit gate delay (inverter with fanout of two) scales down to 20 pico-seconds. Thus, modern day control logic consisting of 7-8 logic stages per cycle would form less than 20% of the total cycle time. This clearly challenges the fundamental design trade-off today that tries to simplify the amount of logic per stage in the interest of reducing the cycle time [14]. In addition, this points to a sharply reduced marginal cost of per stage logic complexity on the circuit-level performance.

The decreasing delay penalty for (re)programmable logic blocks compared to interconnect delays also makes the incorporation of small programmable logic blocks attractive even in custom data paths. Because the interconnect delays scale down much more slowly than transistor switching delays, in the year 2007, the delay of the average length inter-connect (assuming an average interconnect length of 1000X the pitch) would correspond to approximately three gate delays (see [5] for a detailed analysis) This is in contrast to less than half the gate delay of the average interconnect in current process technology. This implies that due to purely electrical reasons, it would be preferred to include at least one inter-connect buffer in a cycle time. This buffer gate when combined with a weak-feedback device would form the core of a storage element that presents less than 50% switching delay overhead (from 20ps to 30ps), making it performance competitive to replace static wires with flexible interconnect.

In view of these technology trends and advances in circuit modeling using hardware description languages (HDLs) such as Verilog and VHDL, the process of hardware design is increasingly a language-level activity, supported by compilation and synthesis tools [11], [12]. With these CAD and synthesis capabilities, programmable logic circuit blocks are beginning to be used in

improving system performance.

B. Co-processing

The most common architecture in embedded computing systems to exploit programmable logic can be characterized as one of *co-processing*, i.e., a processor working in conjunction with dedicated hardware assists to deliver a specific application. The hardware assists are built using programmable circuit blocks for easy interpretation with the predesigned CPU Figure 1 shows the schematic of a co-processing architecture, where the co-processing hardware may be operated under direct control of the processor which stalls while the dedicated hardware is operational [10], or the co-processing may be done concurrently with software [13] However, a system generated using this approach typically can not be retargeted to another application without repartitioning hardware and software functionality and reimplementing the co-processing hardware even if the macro-level organization of the system components remains unaffected. This presents an obstacle of exploiting programmable logic for general-purpose computing even though technology trends make it possible to do so.

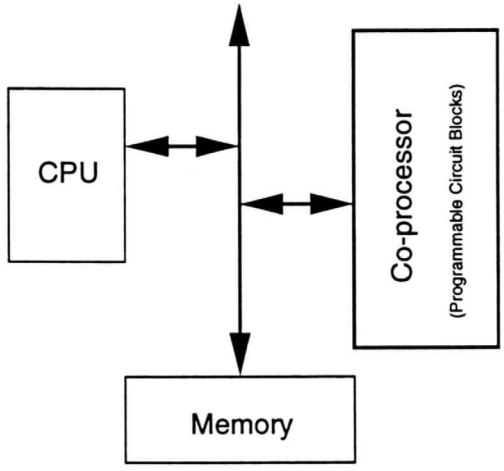

Fig. 1 A co-processing Architecture

III. Architectural Adaptation

We propose an architecture that integrates small blocks of programmable logic into key elements of a baseline architecture, including processing elements, components of the memory hierarchy, and the scalable interconnect, to provide *architectural adaptation* - the customization of architectural mechanisms and policies to match an application. Figure 2 shows our architecture. Architectural adaptation can be used in the bindings, mechanisms, and policies on the

interaction of processing, memory, and communication resources while keeping the macro-level organization the same and thus preserving the programming model for developing applications. Depending upon the hardware technology used and the support available from the runtime environment, this adaptation can be done statically or at run-time.

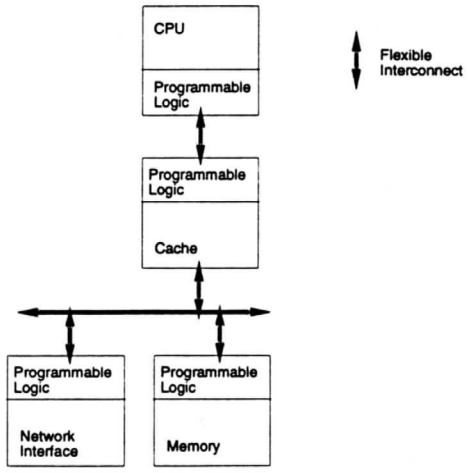

Fig. 2. An Architecture for Adaptation

Architectural adaptation provides the mechanisms for application-specific hardware assists to overcome the rigid architectural choices in modern computer systems that do not work well across different applications and often cause substantial performance fragility. In particular, the integration of programmable logic with memory components enables application-specific locality optimizations. These optimizations can be designed to overcome long latency and limited transfer bandwidth in the memory hierarchy. In addition, because the entire application remains in software while the underlying hardware is adapted for system performance, our approach improves over co-processing architectures by preserving machine usability through software. The main disadvantage of our approach is the potential increase on system design and verification time due to the addition of programmable logic. We believe that the advances in design technology will address the increase of logic complexity.

A. Project Context

Our study is in the context of the MORPH [5] project, a NSF point design study for Petaflops architectures in the year 2007 technology window The key elements of the MORPH (**M**ultiprocess**O**r with **R**econfigurable **P**arallel **H**ardware) architecture consists of processing and memory elements embedded in a scalable interconnect. With a small amount of programmable logic integrated with key elements of the system, the proposed MORPH architecture aims to exploit architectural

customization for a broad range of purposes such as:

- control over computing node granularity (processor-memory association)
- interleaving (address-physical memory element mapping)
- cache policies (consistency model, coherence protocol, object method protocols)
- cache organization (block size or objects)
- behavior monitoring and adaptation

As an example of its flexibility, MORPH could be used to implement either a cache-coherent machine, a non-cache coherent machine, or even clusters of cache coherent machines connected by put/get or message passing. In this paper, we focus on architectural adaptation in the memory system for locality optimizations such as latency tolerance.

IV Case Studies

We present two case studies of architectural adaptation for application-specific locality optimizations. On modern architectures with deep memory hierarchies, data transfer bandwidth and access latency differentials across levels of memory hierarchies can span several orders of magnitude, making locality optimizations critical for performance. Although compiler optimizations can be effective for regular applications such as dense matrix multiply, optimizations for irregular applications can greatly benefit from architectural support. However, numerous studies have shown that no fixed architectural policies or mechanisms, e.g., cache organization, work well for all applications, causing performance fragility across different applications. We present two case studies of architectural adaptation using application-specific knowledge to enhance latency tolerance and efficiently utilize network bisection on multiprocessors.

A Experimental Methodology

As our application examples, we use the sparse matrix library `SPARSE` developed by Kundert and Sangiovanni-Vincentelli (version 1.3 available from `http://www netlib org/sparse/`), and an additional sparse matrix multiply routine that we wrote. This library implements an efficient storage scheme for sparse matrices using row and column linked lists of matrix elements as shown in Figure 3. Only nonzero elements are represented, and elements in each row and column are connected by singly linked lists via the `nextRow` and `nextCol` fields. Space for elements, which is 40 bytes per matrix element, are allocated dynamically in blocks of elements for efficiency. There are also several one dimensional arrays for storing the root pointers for row and column lists.

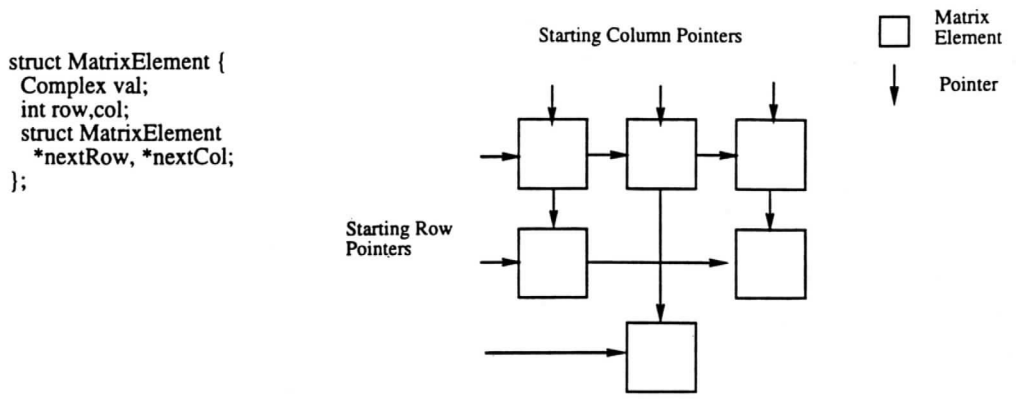

Fig. 3. Data Structures used in the Sparse Library

We perform cycle-based system-level simulation using a program-driven simulator based on MINT [22] that interprets program binaries and models configurable logic blocks behaviorly. The details of our simulation environment are presented in [4]. Table I shows the simulation parameters used. We report our empirical results for current day computer technologies and then use derivative metrics (such as miss rate) to extrapolate potential benefits for future computer systems which will exhibit much higher clock rates and memory sizes. We also manually translated the C routines modeling the customizable logic blocks into HardwareC [18] to evaluate their hardware cost in terms of space and cycle delays. (However, our recent work is focused on automatic translation of these routines to synthesizable blocks [19].)

TABLE I

SIMULATION PARAMETERS

	L1 Cache	L2 Cache
Line Size	32B or 64B	32B or 64B
Associativity	1	2
Cache Size	32KB	512KB
Write Policy	Write back + Write allocate	Write back + Write allocate
Replacement Policy	Random	Random
Transfer Rate	(L1-L2) 16B/5 cycles	(L2-Mem) 8B/15 cycles

B. Architectural Adaptation for Latency Tolerance

Our first case study uses architectural adaptation for prefetching. As the gap between processor and memory speed widens, prefetching is becoming increasingly important to tolerate the memory access latency. However, oblivious prefetching can degrade a program's performance by saturating bandwidth. We show two example prefetching schemes that aggressively exploit application access pattern information.

Figure 4 shows the prefetcher implementation using programmable logic integrated with the L1 cache. The prefetcher requires two pieces of application-specific information. the address ranges and the memory layout of the target data structures. The address range is needed to indicate memory bounds where prefetching is likely to be useful. This is application dependent, which we determined by inspecting the application program, but can easily be supplied by the compiler The program sets up the required information and can enable or disable prefetching at any point of the program. Once the prefetcher is enabled, however, it determines what and when to prefetch by checking the virtual addresses of cache lookups to check whether a matrix element is being accessed.

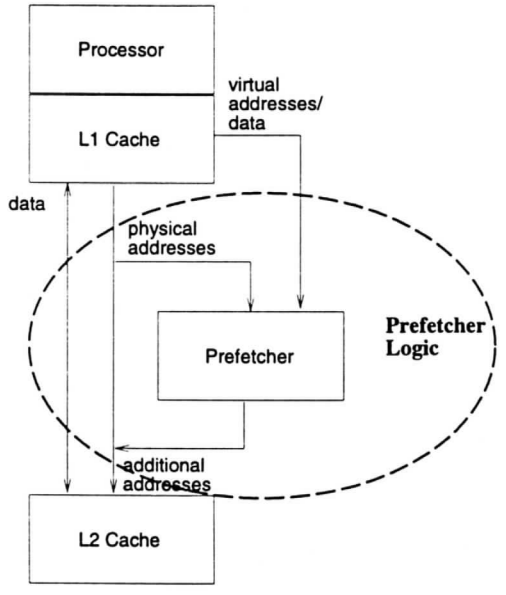

Fig. 4. Organizations of the Prefetcher Logic

The first prefetching example targets records spanning multiple cache lines and for our example, prefetches all fields of an matrix element structure whenever some field of the element is accessed. The pseudocode of this prefetching scheme for the sparse matrix example is shown below, assuming a cache line size of 32 bytes, a matrix element padded to 64 bytes, and a single

matrix storage block aligned at 64-byte boundary. Prefetching is triggered only by read misses. Because each matrix element spans two cache lines, the prefetcher generates an additional L2 cache lookup address from the given physical address (assuming a lock-up free L2 cache) that prefetches the other cache line not yet referenced.

```
GroupPrefetch (vAddr, pAddr, startBlock, endBlock)
{
  /* Check if vAddr is for the matrix and */
  /* prefetch only if the check passes */
  if (startBlock <= vAddr && vAddr < endBlock) {
    /* Determine the starting address of */
    /* not-yet-accessed part */
    if (pAddr & 0x20)
      ptrLoc = pAddr - 0x20;
    else
      ptrLoc = pAddr + 0x20;

    Initiate a request to transfer
    the line at ptrLoc to L1 cache;
  }
}
```

The second prefetching example targets pointer fields that are likely to be traversed when their parent structures are accessed. For example, in a sparse matrix-vector multiply, the record pointed to by the `nextRow` field is accessed close in time with the current matrix element. The pseudocode below shows the prefetcher code for prefetching the row pointer, assuming a cache line size of 64 bytes. Again prefetching is triggered only by read misses, and the prefetcher generates an additional address after the initial cache miss is satisfied using the `nextRow` pointer value (whose offset is hardwired at setup time) embedded in the data returned by the L2 cache.[1]

```
PointerPrefetch (data, vAddr, startBlock, endBlock)
{
  /* Check if vAddr is for the matrix and */
  /* prefetch only if the check passes */
  if (startBlock <= vAddr && vAddr < endBlock) {
    /* Find the row pointer value inside */
    /* the returned cache line (data) */
    ptrLoc = data [24];  /* row pointer offset = 24 */

    Initiate a request to transfer
    the matrix elt at ptrLoc to L1 cache;
  }
}
```

Our prefetching examples are similar to the prefetching schemes proposed in [23], where they

[1] As pointed in [23], the implementation of this prefetching scheme is complicated by the need to translate the virtual pointer address to physical address. We assume that the prefetcher logic can also access the TLB structure. An alternative implementation is to place the prefetcher logic in memory and forward the data of the next record to the upper memory hierarchy. This requires an additional group translation table [23] for address translation.

are shown to benefit various irregular applications. However, unlike [23], using architectural customization enables more flexible prefetching policies, e.g., multiple level prefetch, according to the application access pattern.

C. Architectural Adaptation for Bandwidth Reduction

Our second case study uses a sparse matrix-matrix multiply routine as an example to show architectural adaptation to improve data reuse and reduce data traffic between the memory unit and the processor. The architectural customization aims to send only used fields of matrix elements during a given computation to reduce bandwidth requirement using dynamic scatter and gather. Our scheme contains two units of logic, an address translation logic and a gather logic, shown in Figure 5.

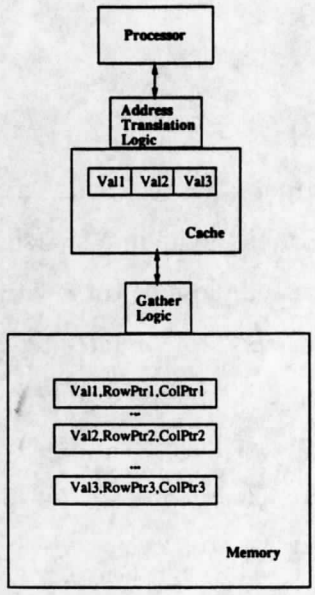

Fig. 5. Scatter and Gather Logic

The two main ideas are prefetching of whole rows or columns using pointer chasing in the memory module and packing/gathering of only the used fields of the matrix element structure. When the root pointer of a column or row is accessed, the gather logic in the main memory module chases the row or column pointer to retrieve different matrix elements and forwards them directly to the cache. The cache, in order to avoid conflict misses, is split into two parts: one small part acting as a standard cache for other requests and one part for the prefetched matrix elements only. The latter part has an application-specific management policy, and can be distinguished by mapping it to a reserved address space. The gather logic in pseudocode is shown below.

```
/* Row gather: pAddr is the start of a row */
Gather(pAddr)
{
  chaseAddr = pAddr;
  while(chaseAddr) {
    forward chaseAddr->val
    forward chaseAddr->row
    chaseAddr = virtual-to-physical(chaseAddr->nextRow)
  }
}
```

Because the data gathering changes the storage mapping of matrix elements, in order not to change the program code, a translate logic in the cache is required to present "virtual" linked list structures to the processor. When the processor accesses the start of a row or column linked list, a prefetch for the entire row or column is initiated. Because the target location in the cache for the linked list is known, instead of returning the actual pointer to the first element, the translate logic returns an address in the reserved address space corresponding to the location of the first element in the explicitly managed cache region. In addition, when the processor accesses the next pointer field, the request is also detected by the translate logic, and an address is synthesized dynamically to access the next element in this cache region. The translate logic in pseudocode is shown below.

```
Translate(vAddr, pAddr, newPAddr)
{
  /* check if accessing start of a row */
  if (startRowRoot <= vAddr && vAddr <= endRowRoot) {
    Initiate prefetch
    return row location in cache
  }
  /* Similarly for column roots */
   ..
  /* Accessing packed rows */
  if (startPackedRows <= pAddr && pAddr <= endPackedRows) {
     offset = pAddr & 63;        /* check which field (record length is 64) */
     if( offset == 24 )          /* row pointer at offset 24 */
       return pAddr + 64;        /* if nextRow, synthesize next address */
     else {
       /* There are two other fields used that reside at offset_1 and _2
          of the original record and at packed_offset_1 and _2 of new layout
       */
       if (offset < offset_2)    /* first used field: val */
          new_offset = offset - (offset_1 - packed_offset_1);
       else                      /* second used field: row */
          new_offset = offset - (offset_2 - packed_offset_2);
       return (pAddr >> 6) * PACKED_SIZE + new_offset;
  }
  /* Similarly for packed columns */
  ...
}
```

As in the case of dense matrix-matrix multiply, blocking of the matrices in the cache is essential

to achieve good performance. However, software blocking for sparse matrix-matrix multiply is not as effective as blocking in the dense case because the additional fields in a matrix element structure reduce the number of rows or columns that can fit in the cache, reducing reuse, and because elements are scattered in memory, increasing conflict misses. Figures 6 and 7 compare the miss rates and total data volume transfered between the memory and processor for the straightforward implementation, a pure software blocking scheme, and customized scatter and gather. (The last bar shows bypassing the cache for the result matrix.) The figures show that the hardware assists provided by customization significantly improve the software blocking by reducing row and column footprints to improve reuse. The final results show a 10X in read miss rates and 100X reduction in data volume compared to the straightforward implementation. The implementation costs of these units are less than 1000 in LSI logic 10K family gates and around 1500 in FPGA CLBs, requiring delays of about 3 cycles.

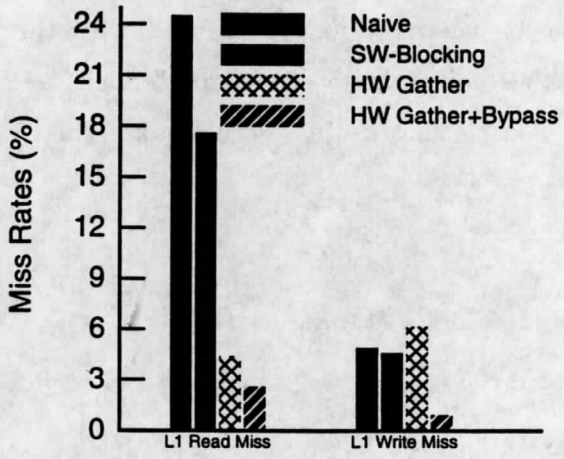

Fig. 6. Cache miss rates comparisons of the naive sparse matrix-matrix multiply, a software blocking scheme, a programmable logic implementation of gather/scatter, and with cache bypass of the result matrix. (For two 460x460 matrices, each with 21660 non-zero elements.)

V. Discussion and Related Work

Traditional computer architectures are designed for best machine performance averaged across applications. Due to the static nature of such architectures, such machines are limited in exploiting an application characteristics unless it is common for a large number of applications. Since no single machine organization fits all applications, the delivered performance is often only a small fraction of the peak machine performance (frequently less than a tenth [5]). Therefore, we believe that there are significant opportunities for application-specific architectural adaptation.

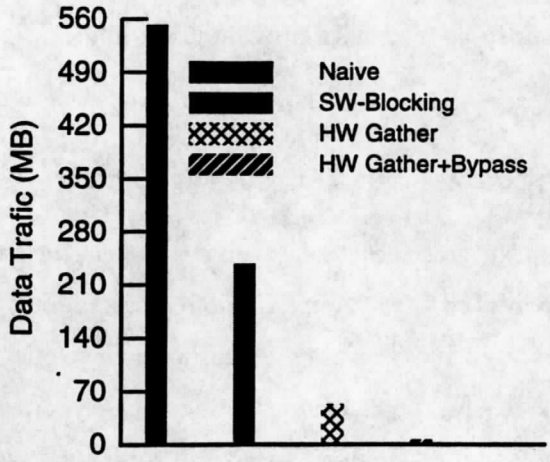

Fig. 7 Data traffic volume comparisons of different schemes in MB. (For two 460x460 matrices multiply, each with 21660 non-zero elements, with total size of elements 1.35 MB.)

In this paper, we have demonstrated mechanisms for latency hiding and required bandwidth reduction that leverage small hardware support as well as do not change the programming model. Following the same methodology, we can build such assists for other applications as well. Among other examples that applications can benefit from are mechanisms for recognition of working set size for a given application that can be used to alter cache update policies or even use a synthesized small victim cache [16], and mechanisms for monitoring access patterns and conflicts in the caches or memory banks and reconfiguring the assists according to these patterns and conflicts. In summary, we expect an adaptable machine to have a large number of application-specific assists that alter architectural mechanisms and policies in view of application characteristics. The application developer with the help of compilation tools selects appropriate hardware assists to customize the machine to match the application without having to repartition the system functionality or rewrite the application.

There are other approaches for locality optimizations. Researchers have proposed processor-in-memory (PIM) [2], [3], [7], [8], [21] as a solution for solving the latency and bandwidth limitations of the memory hierarchy. We rely on more traditional processor-memory structures with customizable components, which we believe will yield a machine with more accessible performance than an organization in which processors are accessing primarily their local on-chip memory, particularly for irregular applications. In addition, by adding a small amount of programmable logic to the memory units, we can yield some benefits of having computational elements within the memory. Researchers have also proposed various specific architectural mechanisms for locality optimizations, for instance, group prefetching [23] and informing memory operations [15], with

mechanism-specific implementations. As shown in our case studies, we believe that integrating programmable logic into memory components provide a flexible implementation framework for these mechanisms.

VI. Conclusions and Future Directions

The increasing dominance of interconnect delays on system performance makes it practical to integrate programmable logic into all key system components, enabling them to be customized to specific applications. As illustrated by two case studies, we believe such architectural adaptation can provide flexible mechanisms for application-specific locality optimizations to combat the increasing gap between processor and memory speed. In addition, system co-design using this approach presents a way to utilize application-specific hardware much more effectively than would be the case when part of an application is implemented in hardware as is the case in co-processing architectures. As future directions, we are studying compilation tools in hardware/software co-design for the architectural assists, IC processing issues for implementing customizable system components, such as cache and scalable interconnects, as well as more, larger application studies.

Acknowledgments

The work was sponsored by support from National Science Foundation CAREER award number MIP 95-01615, NSF Young Investigator Award CCR-94-57809, NSF/DARPA ASC-96-34947 and from NSF EEC 89-43166.

References

[1] Seminconductor technology workshop conclusions. Roadmap, Semiconductor Industry Association, 1993.

[2] Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P S., Sutton, J., Urbanski, J., and Webb, J. iWarp: An integrated solution to high-speed parallel computing. In *Proc. Supercomputing '88* (1988), IEEE Press, pp. 330–341. Orlando, Florida.

[3] Borkar, S., Cohn, R., Cox, G., Gross, T., Kung, H. T., Lam, M., Levine, M., Moore, B., Moore, W., Peterson, C., Susman, J., Sutton, J., Urbanski, J., and Webb, J Supporting systolic and memory communication in iWarp. In *Proc. the 17th Int. Symp. Comp. Arch.* (1990), IEEE, pp. 70–81.

[4] Chien, A., Dasdan, A., Gupta, R., and Zhang, B. Rapid Architectural Design and Validation Using Program-Driven Simulations. In *Symp. High-level Design, Validation and Test (HLDVT)* (Nov. 1996).

[5] Chien, A. A., and Gupta, R. K. Morph: A system architecture for robust high performance using customization. In *Frontiers of Massive-Parallelism* (1996). Annapolis, Maryland.

[6] Choi, L., and Chien, A. A. The design and performance evaluation of the DI-multicomputer. *Journal of Parallel and Distributed Computing* (199x). Submitted for publication.

[7] DALLY, W., GHIEN, A., FISKE, J., FYLER, G., HORWAT, W., KEEN, J., LETHIN, R., NOAKES, M., NUTH, P., AND WILLS, D. The message driven processor: an integrated multicomputer processing element. In *Proc. the 1992 IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors* (1992), pp. 416–19.

[8] DALLY, W J., FISKE, J. A. S., KEEN, J. S., LETHIN, R. A., NOAKES, M. D., NUTH, P R., DAVISON, R. E., AND FYLER, G. A. The message-driven processor. *IEEE Micro* (April 1992), 23–39.

[9] DALLY, W J., AND SONG, P Design of a self-timed vlsi multicomputer communication controller. In *Proc. the Int. Conf. Comp. Design* (1987), IEEE Computer Society, pp. 230–4.

[10] ERNST, R., HENKEL, J., AND BENNER, T Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design & Test of Computers* (Dec. 1993), 64–75.

[11] GAJSKI, D., DUTT, N., WU, C. H., AND LIN, Y. L. *High-level Synthesis: Introduction to chip and system design.* Kluwer, Boston, 1992.

[12] GUPTA, R. K. *Co-Synthesis of Hardware and Software for Digital Embedded Syst..* Kluwer, Boston, 1995.

[13] GUPTA, R. K., AND MICHELI, G. D. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers* (Sept. 1993), 29–41.

[14] HENNESSY, J L., AND JOUPPI, N P Computer technology and architecture: An evolving interaction. *IEEE Computer* (Sep 1991), 18–29.

[15] HOROWITZ, M., MARTONOSI, M., MOWRY, T C., AND SMITH, M. D. Informing memory operations: Providing memory performance feedback in modern processors. In *Proc. the 23rd Int. Symp. Comp. Arch.* (1996).

[16] JOUPPI, N P Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. the Int. Symp. Comp. Arch.* (1990), pp. 364–73.

[17] KOGGE, P M. Summary of the architecture group findings. In *PetaFlops Arch. Workshop (PAWS)* (Apr. 1996).

[18] KU, D., AND MICHELI, G. D. HardwareC - A Language for Hardware Design (version 2.0). CSL Tech. Rep. CSL-TR-90-419, Stanford University, Apr. 1990.

[19] LIAO, S. Y., TJIANG, S., AND GUPTA, R. K. An Efficient Implementation of Reactivity in Modeling Hardware in the CSYN Synthesis and Simulation Environment. In *Proc. the DAC* (1997).

[20] SEITZ, C. L. Let's route packets instead of wires. In *Proc. the 6th MIT Conf. Advanced Research in VLSI* (1990), W J Dally, Ed., MIT Press, pp. 133–37

[21] SPERTUS, E., GOLDSTEIN, S. C., SCHAUSER, K. E., VON EICKEN, T., CULLER, D. E., AND DALLY, W J Evaluation of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5. In *Proc. the 20th Annual Int. Symp. Comp. Arch.* (1993), IEEE, pp. 302–313. Available from `http://www.cs.cornell.edu/Info/People/tve/ucb_papers/isca93.ps`.

[22] VEENSTRA, J E., AND FOWLER, R. J MINT: A front end for efficient simulation of shared-memoy multiprocessors. In *Proc. the 2nd Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Syst. (MASCOTS)* (Jan. 1994), pp. 201–207

[23] ZHANG, Z., AND TORRELLAS, J. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proc. the Int. Symp. Comp. Arch.* (June 1995).