Flexible Parallel Processing in Memory: Architecture + Programming Model

Nael B. Abu-Ghazaleh
Computer Science Dept.
State University of New York
Binghamton, NY 13902–6000
nael@cs.binghamton.edu

Philip A. Wilsey
ECECS Dept.
University of Cincinnati
phil.wilsey@uc.edu

Jerry Potter
Robert Walker
and
Johnnie Baker
Dept. of Mathematics and Computer Science
Kent State University
{potter,walker,jbaker}@mcs.kent.edu

1 Introduction

VLSI technology continues to develop at a staggering rate presenting two challenges to computer designers: (i) how to capitalize on the additional resources that are available on a chip; and (ii) how to evolve computer architecture models that are well matched to the significantly changed physical parameters of new technology and the expanding needs of applications. One of the chief challenges is overcoming the widening gap between processor and DRAM memory. Even allowing for large caches and a heavily pipelined memory, memory will not be able to provide data to a modern superscalar processor at a rate that satisfies cache miss rates for most applications [9, 11, 23].

Integration of processing with DRAM is an emerging solution to the memory bandwidth/latency problem [13]. The arguments for this integration in light of current technology trends are compelling: Moving most, if not all, of the processing to the memory chip exposes the large internal bandwidth to the processing logic at low latencies. Attempts at using PIM as an alternative memory hierarchy for a conventional superscalar architecture have not shown significant improvement in performance over a traditional memory hierarchy [7, 19] because such architectures do not capitalize on the large bandwidth offered by a PIM configuration.

While it is not evident how the available bandwidth can be directly harnessed, it is clear that explicit parallelism is necessary to make efficient use of this bandwidth. The challenge here is twofold: (i) having an architecture that is capable of processing the large bandwidth; and (ii) a programming model/compiler techniques that allow this processing bandwidth to be utilized given the addressing restriction (only one row in a memory bank can be accessed at a time). Several PPIM architectures have been suggested, targeting a wide range of granularity in the parallelism model. More precisely, these architectures range from those exploiting low-level data (stream) parallelism synchronously as per the vector/SIMD model [13, 18] to exploiting high-granularity asynchronous parallelism (multiprocessor chips consisting of full processors with associated memory [12]).

While the vector/SIMD approach satisfies the first requirement on the architecture (of having the processing capability to utilize the memory bandwidth), it fails on the second requirement of having a programming model that allows this processing bandwidth to be used for general purpose algorithms. Conversely, the chip multiprocessor approach offers a flexible programming model, but does not effectively utilize the bandwidth. In addition, it incurs the overhead for the instruction sequencing and control hardware, as well as an instruction store with each datapath. Thus, neither approach appears to effectively support general purpose processing.

In this paper we propose a PPIM that targets efficient exploitation of the internal bandwidth on a PIM, while providing a programming model that allows this bandwidth to be harnessed by a large class of algorithms. The proposed parallelism model is based on a dynamically partitionable distributed Multiple SIMD (MSIMD) [2, 14, 20]. Under this model, a small number of control units concurrently controls the PEs; each PE receives its control from exactly one of the control units. This model, like other model targeting low-level data-parallelism, can directly utilize data/stream parallelism to take advantage of the internal DRAM bandwidth [13, 18]. In addition, the architecture also supports a limited degree of control-parallelism consistent with that present in most algorithms. Thus, the model balances the algorithmic needs with architecture capabilities. While the MSIMD model is not new, its efficient implementation in a PIM environment requires the introduction of several innovative techniques. For example, we present memory organizations that allow the raw DRAM bandwidth to be exploited even when PEs controlled by different controllers share the same memory bank.

The remainder of this paper is organized as follows. Section 2 overviews the design constraints for a PIM. In Section 3 a high-level overview of the architecture is presented. Section 4 presents some techniques for matching the memory organization with the needs of the architecture. Section 5 presents the programming model. Finally, Section 6 presents some conclusions.

2 Background — Constraints of a Processor in Memory

Parallel processing in memory is necessary to allow effective utilization of a PIM. There are several options including, VLIW, vector processing, SIMD, and MIMD (multiple processors, with varying granularity). Many of the classic tradeoffs between these models at the board level persist inside the chip. However, there are significant changes imposed by this new environment: some of the cost for the different tradeoffs are different.[1] Moreover, there are important functional properties imposed by the DRAM organization. Appreciation of the different on-chip parameters is necessary for an accurate evaluation of the architectural decisions. The primary differences are:

- *A different physical process*: One of the important advantages of the classical separation between memory and processing logic is that each could be fabricated using a different process. Accordingly, the logic process evolved to emphasize speed by minimizing the feature sizes and providing a large number of metal layers (typically 3-6) to minimize wire delays. On the other hand, the DRAM process has additional layers of polysilicon used to build 3-D trench/stack capacitors that maximize the density and minimize leakage currents (which are directly related to the refresh rate of the DRAM). As a result, embedded DRAM processes cannot support logic operations at the aggressive logic-only process speeds or densities (because of the additional area needed for wiring using only 3 metal layers), nor can they support DRAM at the same density because larger DRAM cells are needed to provide immunity from the additional noise introduced by the high speed logic. Our working estimates for the DRAM embedded process are 200MHz logic clock, negligible DRAM bus delays but with 25ns DRAM access latency. Moreover, SRAM cells can be built in the DRAM process with 1.5 penalty in speed and area [10].
- *Concurrent DRAM accessibility for a parallel processor*: The aggregate bandwidth promised by the available bits at the rows of a DRAM is not directly exploitable in general. More precisely, the bits available at the sense amplifiers/row buffer correspond to the data in the accessed row [16]. If the memories of multiple processors are mapped to the same DRAM bank (to provide sufficient processing to utilize the row bandwidth, the memory accesses must somehow align to the same row address or the access will have to be sequentialized. It is possible for vector/SIMD smart memories to work within these constraints by having the compiler map vector/strided data to reside within the same DRAM row. On the other hand, large granularity multiprocessor projects can afford to have independently addressable memory banks for each processor (even multiple ones for synchronous access). Our model requires concurrent general accesses to different addresses (as needed by the limited number of controllers).

3 PPIM: Overall Organization

This section describes the high-level architecture and control-organization of the PPIM modules. In order to harvest the largest degree of parallelism, the parallelism model must be flexible enough to support most applications. However, the benefits of the flexibility provided by a sophisticated model must be con-

[1] One important consideration contributing to the success of MIMD organizations at the board-level is the ability to reuse off-the-shelf components for processing nodes; this consideration does not apply for a PIM design.

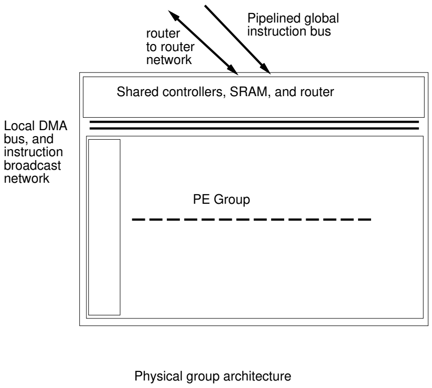

Figure 1. A Physical Process Group in PPIM

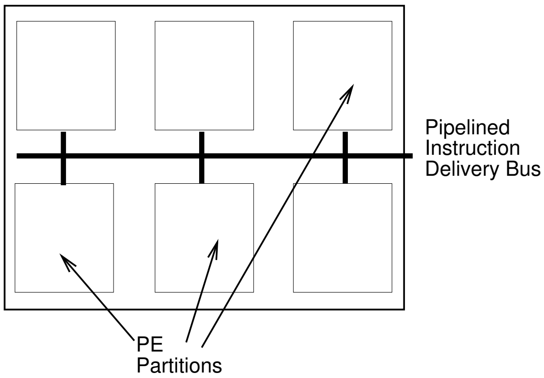

Figure 2. Instruction Delivery Bus

sidered against the cost for supporting it. The model supported by PPIM is a balance between flexibility and overhead. Like the data-parallel approaches, we target low-level data-parallelism as the primary source of available parallelism. However, we also realize that there is a degree of control parallelism present in most applications that renders pure SIMD/vector machines ineffective for general applications. Thus, the model proposed herein is a generally partitionable multiple SIMD [14, 21, 8]: instead of a single thread of control supported by a single controller, we have a small number of controllers available to the program. Each processing element can receive its control from any of the controllers according to its local data. Thus, when a point in the program is reached when different sets of the data require unique processing (control parallelism is present in the application), they can be concurrently supported using the different control units. This model has been shown to be optimal for a variety of algorithms, and it provides efficient support for the associative computing model that will be used to program the PPIM modules (because its data-centric view is conducive to efficient programming for data-intensive algorithms). The cost of an MSIMD configuration when the number of control units is much smaller than the number of processing elements is slightly more than that of SIMD (additional complexity is added to each PE to allow selection among the different control units).

Broadcasting instructions across the chip (or even from off-chip) is not a cost-effective solution because the delay for driving a high fan-out bus across the full chip is large. In addition, the cost for having multiple buses (associated with the different control units) is undesirable. For these reasons, the instruction streams are cached locally as follows. PEs are grouped into physical partitions that are associated with local controllers (instruction sequencing logic and memory). The number of PEs in a partition is restricted only by the need to be reachable in a single cycle from the controller or other shared support logic.

The system can be configured to let each physical partition execute its own programs, resulting in sets of disjoint MSIMD engines. However, here we assume that the same programs are being executed by the controllers across all the partitions. An instruction "page-fault" (requesting an address not available in the current instruction memory) at the controllers causes an instruction block to be fetched from, generally, an off-chip primary instruction store, and delivered to the PE partitions using a pipelined broadcast bus (Figure 2). Since the broadcast bus is shared among the set of controllers, the need for separate broadcast buses for each controller is obviated. There are several reasons why we chose to provide a dedicated instruction delivery bus, rather than having a single bus that implements all off-chip communication:

- operational model: controller flow-control decisions are carried out based on globally shared scalar data but not based on PE data. PEs simply select which branch of control to follow depending on local data—the PE data does not directly affect which control masks the controllers broadcast. Reduction operations that transform parallel data to scalar data (that *can* affect control flow) must be performed on all participating controllers. The broadcast bus facilitates this reduction operation (reduction is the dual of broadcast). Moreover, the central controller at the root of the reduction resides with the primary instruction store and can predict "page-faults" before they occur at the local controllers. The bus also keeps the partitions synchronized.
- broadcast operation needed: The instruction memories of all the controllers are identical. More-

over, they page-fault in lock-step; thus, the broadcast bus can deliver the instructions to all the partitions concurrently.

- maintain separation between instruction and data memory: As we shall see in the next section, the memory systems for instructions and data are accessed differently since the instruction memory is shared per controller, while the data memory is per PE.

The PEs in each partition share a router which implements local communication via a direct memory copy (using the local memory bus). The routers of the different physical partitions are connected by a router network that is used to implement cross-partition communication. Support for block transfer allows this communication to be performed at very high bandwidth. The router network generalizes to implement communication among routers that reside on different chips (and are used to implement I/O as well). Figure 1 shows the architecture of a physical partition.

Two important aspects of the design that have not been discussed are the refresh cycles and off-chip I/O. DRAM memories have to be periodically refreshed or the charge in the DRAM cells will be lost through leakage currents. The synchronous nature of the design makes it possible to interject refresh cycles as needed (more than regular DRAM refresh rates because the logic increases the noise level; in addition, it becomes more difficult to realize the planar layout critical to high DRAM density). Off-chip I/O will be implemented through the router network which has a dedicated high-bandwidth off-chip interface capable of connecting to other PPIMs or other high speed memory (*e.g.*, using the RAMBUS Direct RDRAM technology [17] allowing 1.6 Gb/s transfer rates per channel). When an I/O operation occurs, only the controller requesting it is blocked; other controllers can continue processing as long as there are no inter-controller dependencies. Moreover, it is possible to multithread the controllers such that a context switch to another thread occurs on an I/O operation; this is likely to be useful for disk I/O and similar slow operations, but not, generally, for communication. These solutions will be investigated quantitatively as the details of the router network and delivery bus are finalized.

4 PE Memory System

The memory system must be carefully designed to match the requirements of the PEs and the parallelism model. There are two problems: the first is matching the DRAM organization to the access patterns required by the MSIMD model, while the second is the speed mismatch between DRAM access and logic speed; even if the access patterns can be supported in a single cycle, there is a large discrepancy between the DRAM access time (25ns) to the logic switching speed in the DRAM process (5ns for a conservative 200MHz operation). We begin by discussing the first problem.

A conventional single-bank DRAM is organized into an array of bits organized as r rows, each with c columns (1024 rows by 1024 columns, for example). The sequence of operations that occur to access a word is as follows. The c *bit-line* buses are pre-charged to a voltage level between logical 0, V_{low} and that of logical 1, V_{high} (say $\frac{V_{high}-V_{low}}{2}$). First, the row address is specified, causing all the DRAM memory cells in the selected row to share their charge with the respective bit-lines. The charge in the DRAM cell is large enough to cause a small but detectable voltage swing on the bit-line (on the order of few tens of milli-volts). This change in voltage is detected and amplified to V_{high} or V_{low} by the sense amplifiers. At the end of this step, the contents of the row (1024 bits in our example) are available at the sense amplifiers. The column address specifies the set of consecutive columns (b-bits, depending on the "width" of the DRAM) that correspond to the addressed word. These bits are multiplexed out and made available at the output of the DRAM. Thus, only b bits out of the potential c-bits available at the sense amplifiers are used. Our design seeks to use a higher proportion of this bandwidth.

The PEs are mapped to different column slices in the DRAM array. The MSIMD model requires that each PE be able to follow any of the controllers. If this was allowed in the most general case, then the PEs could be addressing different memory rows at the same time — this is the equivalent of asking an external memory chip for data from different addresses at the same time. The following solutions to this problem are possible: (i) *Use different memory banks*: Each memory bank is addressable separately, allowing random access on a per-bank basis. However, the overhead for having a bank per PE is excessive (32-bit separately addressable rows) and does not yield an efficient solution; (ii) *Multi-port the DRAM banks*: Additional ports can be supplied such that different locations in the same DRAM bank can be read concurrently. Adding even one port to the DRAM banks is extremely expensive as all the addressing/multiplexing hardware and buses must be duplicated in the metal-poor DRAM process. In addition, only a portion of each row is utilized; and (iii) *"Soft" multi-porting*: Accesses that occur to different memory banks are serviced concurrently; if accesses are destined to the same memory bank, they are serialized. This approach requires careful data placement by the compiler. It severely compromises the efficiency of the model if the PEs are allowed to dynamically switch among controllers (if PEs sharing the same DRAM bank are mapped different controllers, then serialization will occur at every concurrent memory access).

In addition to the problems introduced by the memory bank conflicts, the gap between the DRAM and logic speed is too high to allow direct operation in memory; each memory access takes 5 logic cycles even if there are no conflicts. The solution to this problem is to use a Static RAM (SRAM) register set for each PE that operates at the same speed as the logic. Since the register set is small, we can afford to implement a hardware assist for multi-porting on the register set only. More precisely, the specification of the registers for the operations depend on the controller that each PE follows and hardware support for this indexing is implemented. If each value in the register set is used several times, then the number of accesses to the memory are reduced to the degree where the soft multi-porting approach is sufficient. Of course, the tradeoff between SRAM and DRAM is between speed and area; we cannot make the register sizes excessively large, or the memory yield of the chip will be compromised. One of the simulation tasks is determining a good register set size that will allow the efficient utilization of the memory without excessively increasing the chip area.

While these solutions address the data memory system, similar solutions are needed for instructions. In each cycle, one instruction is needed for each controller. Because of the small number of controllers, it is feasible to have separate banks for each, solving the bank conflict problem. However, the instruction fetch speed must still be matched with the logic speed. There are two possible solutions: (i) have an SRAM cache for instructions that can be accessed each cycle. Unfortunately, it is unlikely that a suitable size cache could be built without wasting too many resources (although SRAM is made of 6 transistors to DRAM's 1, the area ratio is significantly larger than that because DRAM uses a special 3-D process to minimize the size. Moreover, SRAM made in the DRAM process incurs an additional size penalty); and (ii) organize instructions in *blocks*; each block made up of $\frac{DRAMAccessTime}{LogicClockTime}$ instructions. These instructions can be fetched in a single memory access by mapping them to a single row in the DRAM bank. Thus, in each access fetch, a set of consecutive instructions are fetched from memory. The fetch of the next set of instructions is thus perfectly pipelined with the execution of the previous set. The compiler has the responsibility of packaging instructions into blocks to minimize block fragmentation.

Thus, each PE consists of an ALU, a set of multiported SRAM registers, and some control steering logic (p to 1 demultiplexers selecting among the p controllers). The steering logic is controlled by special instructions that set a masking register depending on a local test [3]. Load/store instructions take multiple cycles of the logic clock. However, they can be executed concurrently with other instructions, with a simple hardware interlock mechanism at the controller detecting data-dependencies and stalling if necessary (this scheme is used in the MasPar machines [6]). The PE implements a simple two stage pipeline between the fetch of the instruction block and the execution of the current block. Floating point operations can be supported by sharing floating point units among multiple PEs; we have investigated several mechanisms to allow efficient sharing of resources in a similar environment [1]. A fixed number of floating point units will be provided per partition (depending on the partition size). As a point of reference, the Imagine architecture (a similar scale design) incorporates eight floating point units for the whole chip [18].

Provided the assumptions about the degree of control parallelism present in the applications hold and the instruction locality is sufficient to allow the pipelined instruction bus to keep up with the instruction page-fault rate, we have a high performance, low overhead architecture. The instruction streams and instruction sequencing logic are shared among the PEs within the same physical block, making the overhead for supporting the parallelism model small. The architecture is capable of capitalizing on the memory bandwidth at the source, restricted only by the size of the logic overhead that will be tolerated before the memory yield is compromised to a degree that will harm data locality.

5 Programming model and tools

The programming language will be extended from the *M*ultiple *As*sociative *C*omputing (MASC) [15]. There are three reasons for choosing this language. First, the associative languages embody a data-centric model and are thus conducive to efficient programming of the PPIM modules. For example, support of content addressable memory through efficient associative searches is directly supported. The second reason is that the multiple associative computing model has been shown to be optimal for a large number of real algorithms [4, 5, 22]. Finally, the tools and compiler techniques have been developed by some of the PIs and that expertise is therefore directly available for the project. However, considerable modification to the Associative Computing environment (especially to the back-end of the tools) must is needed for efficient implementation on the PPIM infrastructure.

A number of algorithms have been analyzed for the MASC model. Some of the more recent results are the following. In [15], an optimal algorithm is given for a minimal spanning tree for a undirected graph. In [4], a parallel version of the Graham Scan convex hull algorithm with optimal average cost is presented for MASC. Many other algorithms have been analyzed including graph algorithms such as connected components, traveling salesperson algorithms, data base management programs including associative data base and relational data base, a first pass and optimization phase for a compiler for MASC, a context sensitive language interpreter, two rule based inference engines, an associative PROLOG interpreter, a ray tracing program, and some basic air traffic control algorithms. In addition, a number of image processing programs have been analyzed, including convolution, warping, feature recognition, line thinning and thickening, Tukey median filter, histogram equalization, and region growing. For example, a number of numerical application programs such as matrix multiplication and FFT have been developed.

The programming environment and support tools must be retargeted towards the new architecture (including the development of compiler techniques that take advantage of its abilities). At the operational level, PPIM is a multiple SIMD model. The general multiple SIMD model consists of several control units, each of which may sequence a separate control thread. The control units send their control signals to the processing elements which, in turn, choose the proper control stream locally depending on their local data. Thus, where a SIMD control unit is time shared when multiple control threads are present in the application, the multiple threads can be supported concurrently on a multiple SIMD machine. Meanwhile, the processing elements remain little more than simple data-paths.

PPIM is different from traditional MSIMD models in that its controllers are distributed. This distributed scheme can result in controllers operating out of sequence with respect to each other (since one may fault while others do not). However, in the system-wide configuration the controller state remains consistent across physical partitions (a controller page faults across all physical partitions identically). If the programs on the different controllers do not interact, the semantics of the MSIMD model are preserved. However, primitives to synchronize the controllers must be added to guarantee correct implementation of the MSIMD semantics. Thus, explicit support in the environment is needed to maintain correct operation.

For example, consider the problem of scalar data (data that exists in the controller memory space and can be used for control-flow operations.) Since the scalar data is now distributed, mechanisms for maintaining coherency of this data are needed. A read of scalar data at a controller requires that the data be available locally, or an instruction page-fault will occur (scalar data resides in the same space as instructions and is treated identically for the purposes of page-faults). Scalar data can be changed in two ways: (i) through controller write operations; and (ii) as the result of a *reduction* operation. In the first case, since the controller state is identical across partitions, the same value will be written by the same controller across partitions; it is sufficient that the value be reflected written to the local controllers. However, the effect of reduction operations on scalar data is more complicated. Reduction operations are operations that reduce data across the participating PEs to produce a scalar value at the controller that initiates the reduction. For example, a reduce-add operation can be used to sum the errors at each data point in an iterative numerical solver; if the sum of the error falls below the tolerance, the computation can be terminated. The reduction operation must be reduced back across the partitions to a (logically) central controller. Once the reduction is complete, the data must be broadcast back to the distributed controllers. If the scalar data is subsequently used at the controller (as it is in the above example), the controller must stall until the broadcast data is received since the value of the data determines the control flow.

Beyond maintaining the correctness of operation, the programming model must enable efficient use of the architecture. It is important to develop programming constructs and compiler techniques to allow efficient utilization of the available resources. The compiler must be aware of the memory structure, the controller instruction memory blocking scheme and the number of controllers. In addition to the optimizations specific to PPIM's organization, traditional compiler optimization techniques for a parallel machine apply. For example, techniques for mapping the data and managing transitions between algorithm phases have to be developed. When there is a transition between algorithms their assumptions about data distribution may be incorrect. The question the compiler must answer is whether to instigate a data re-map, or to keep the less efficient mapping and use cross-router communication to access non-local data with a loss in performance.

6 Concluding Remarks

In this paper, we presented an overview of a flexible but efficient parallel processor in memory. Like Vector/stream PIMs this architecture allows efficient utilization of low-level data-parallelism. However, unlike these models it also supports a small degree of control parallelism consistent with that present in most algorithms. Thus, it allows a large set of general purpose algorithms to benefit from a processing in memory organization. Some aspects of the architecture are still under development. We are currently building a parameterized detailed simulation model of the physical partition and the PPIM chip. We will use the model to investigate different configurations of the architecture.

References

[1] N. B. Abu-Ghazaleh. *Shared Control: A Paradigm for Supporting Control Parallelism on SIMD-like Architectures.* PhD thesis, University of Cincinnati, July 1997.

[2] N. B. Abu-Ghazaleh and P. A. Wilsey. Managing control asynchrony on SIMD machines — a survey. In M. Zelkowitz, editor, *Advances in Computers.* Academic Press, Dec. 1998.

[3] N. B. Abu-Ghazaleh and P. A. Wilsey. Shared control architectures: Supporting control-parallelism on simd-like architectures. In D. Pritchard and J. Reed, editors, *LNCS 1470 — Proceedings of the 4nd European Parallel Processing Conference (EuroPar '98)*, pages 1089–1099, Sept. 1998.

[4] M. Atwah, J. Baker, and S. Akl. An associative implementation of graham's convex hull algorithm. In *Proceedings of the Seventh IASTED International Conference on Parallel and Distributed Computing Systems*, pages 273–276, 1995.

[5] M. Atwah, J. Baker, and S. Akl. An associative implementation of classical convex hull algorithms. In *Proceedings of the Eighth IASTED International Conference on Parallel and Distributed Computing Systems*, pages 435–438, 1996.

[6] T. Blank. The MasPar MP-1 architecture. In *Proceedings of the 35th IEEE Computer Society International Conference*, pages 20–24, 1990.

[7] N. Bowman, N. Gardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of existing architectures in IRAM systems. In *Workshop on Mixing Logic and DRAM: Chips that think and Remember — ISCA'97*, June 1997.

[8] T. Bridges. The GPA machine: A generally partitionable MSIMD architecture. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Architecutres*, pages 196–203, 1990.

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

[10] IRAM course class page and projects at Berkeley, 1996.

[11] N. P. Jouppi and P. Rangarathan. The relative importance of memory latency, bandwidth and branch limits to performance. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, International Symposium on Computer Architecture (ISCA 97)*, June 1997.

[12] K. Murakami, K. Inoue, and H. Miyajima. PPRAM: (parallel processing RAM): A merged DRAM/logic system-LSI architecture. In *1997 International Conference on Solid State Devices and Materials (SSDM'97)*, Sept. 1997.

[13] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *IEEE Micro*, pages 34–44, March/April 1997.

[14] J. L. Potter. An associative model of computation. In *Proceedings of the Second International Conference on Supercomputing*, volume III, pages 1–8, May 1987.

[15] J. L. Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, and C. Asthagiri. ASC: An associative computing paradigm. *IEEE Computer*, pages 19–26, Nov. 1994.

[16] B. Prince. *High Performance Memories: New Architecture DRAMs and SRAMs – Evolution and Function.* John Wiley and Sons, 1996.

[17] Rambus Homepage. http://www.rambus.com.

[18] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Langunas, P. Mattson, and J. Owens. Bandwidth efficient architecture for media processing. In *Proceedings for the 31st International Symposium on Microarchitecture*, 1998.

[19] A. Salsbury, 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*, pages 90–101, May 1996.

[20] D. E. Schimmel. Superscalar SIMD architecture. *Proc. of the 4th Symposium on the Frontiers of Massively Parallel Computation*, pages 573–576, Oct 1992.

[21] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley Jr., and S. D. Smith. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. on Computers*, C-30(12):934–947, Dec. 1981.

[22] D. Ulm. *The Power of ASC Through Simulations with Other Parallel Models.* PhD thesis, Department of Mathematics and Computer Science, Kent State University, 1998.

[23] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1), Mar. 1995.