Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines

Brian R. Gaeke[1], Parry Husbands[2], Xiaoye S. Li[2], Leonid Oliker[2],
Katherine A. Yelick[1, 2], and Rupak Biswas[3]

[1]Computer Science Division, University of California, Berkeley, CA 94720
[2]NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720
[3]NAS Division, NASA Ames Research Center, Moffett Field, CA 94035

Abstract

The increasing gap between processor and memory performance has led to new architectural models for memory-intensive applications. In this paper, we use a set of memory-intensive benchmarks to evaluate a mixed logic and DRAM processor called VIRAM as a building block for scientific computing. For each benchmark, we explore the fundamental hardware requirements of the problem as well as alternative algorithms and data structures that can help expose fine-grained parallelism or simplify memory access patterns. Results indicate that VIRAM is significantly faster than conventional cache-based machines for problems that are truly limited by the memory system and that it has a significant power advantage across all the benchmarks.

1. Introduction

Many high performance applications run well below the peak arithmetic performance of the underlying machine, with the inefficiency being often attributed to a lack of memory bandwidth. In particular, applications involving sparse matrices, unstructured meshes, sorting, or table lookups are often said to be bandwidth-limited. In this paper, we focus on the memory bandwidth problem within individual processing nodes, and evaluate a novel Intelligent RAM design that mixes logic and DRAM on a single chip. VIRAM is a vector processor designed for multimedia applications; it couples on-chip DRAM for high bandwidth with vector processing to express fine-grained data parallelism [4]. The peak memory bandwidth of VIRAM is 6.4 GB/s, which is 5–10× higher than most cache-based machines. An early study of processor-in-memory technology showed that conventional processor designs did not take advantage of the enormous on-chip bandwidth [19], whereas the explicit parallelism in vector instructions could be used for high arithmetic performance as well as for masking memory latency. Energy consumption and cooling are also concerns for large-scale machines, and the use of fine-grained parallelism provides much better energy efficiency than a high clock rate.

In this paper, we compare the performance of several memory-intensive scientific kernels on VIRAM and other architectures. The purpose is to: 1) evaluate the general idea of processor-in-memory chips as a building block for high performance computing; 2) examine specific features of the VIRAM processor, which was designed for media processing, for use in scientific computing; 3) determine whether on-chip DRAM can be used in place of the more expensive SRAM-based memory systems of vector supercomputers; and 4) isolate features of the architecture that limit performance, showing that the issues are more complex than simply memory bandwidth. We treat each benchmark as a paper-and-pencil description and explore several alternative algorithms to improve performance.

2. VIRAM Architecture

The VIRAM architecture [4] extends the MIPS instruction set with vector instructions that include integer and floating point operations, as well as memory operations for sequential, strided, and indexed (scatter/gather) access patterns. The processor has 32 vector registers, each containing up to 32 64-bit values. Logically, a vector operation specifies that the operation may be performed on all elements of a vector register in parallel. The current micro-architecture is divided into 4 64-bit lanes, so a single vector instruction is executed by the hardware 4 elements at a time.

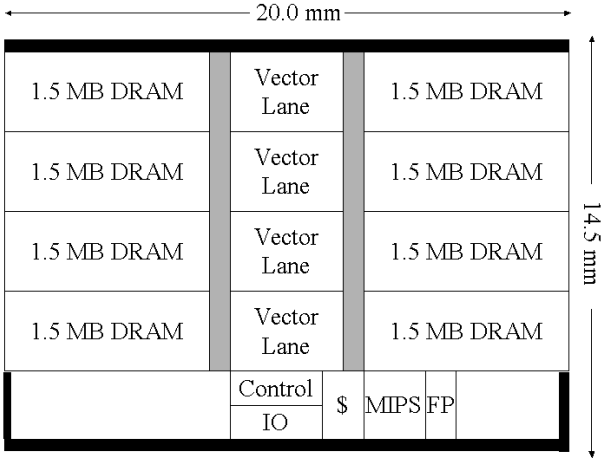

Figure 1: Block diagram of VIRAM

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 **IEEE**

The hardware resources devoted to functional units and registers may be subdivided to operate on 8, 16, 32, or 64-bit data. When the data width (known as the *virtual processor width*) is cut in half, the number of elements per register doubles, as does the peak arithmetic rate. The virtual processor width may be set by the application software and changed dynamically as different data types are used in the application. VIRAM contains two integer functional units and one floating-point unit, and to support conditional execution, a register of flags can mask individual elements during a vector operation.

The VIRAM implementation includes a simple in-order MIPS processor with cache, a floating point unit, a DMA engine for off-chip access, a memory crossbar, and a vector unit which is managed as a co-processor. Figure 1 shows the major blocks in the chip; the shaded area is a full memory crossbar, which allows any lane to access any DRAM bank. The estimated transistor count is over 100 million, while the clock rate is 200MHz and the power consumption is only 2 Watts. There are 12 MB of on-chip DRAM organized into 8 banks, and all of the memory is directly accessible from both scalar and vector instructions. As an academic research project, some compromises were made to simplify the chip design. There is no 64-bit floating-point capability, and the compiler does not generate fused multiply-add instructions; so these aspects of the instruction set specification will be omitted from further consideration. The resulting peak performance for VIRAM is 1.6 GFLOPS for 32-bit floating-point operations, 3.2 GOPS for 32-bit integer operations, and 6.4 GOPS for 16-bit integer operations.

The variable data widths in VIRAM are common to other SIMD media extensions such as Intel's SSE, but otherwise the architecture more closely matches vector supercomputers. In particular, the parallelism expressed in SIMD extensions are tied to the degree of parallelism in the hardware, whereas a floating-point instruction in VIRAM specifies 64-way parallelism while the hardware only executes 8-way. The advantages of specifying longer vectors include lower instruction bandwidth needs, a higher degree of parallelism for memory latency masking, and the ability to change hardware resources across chip generations without requiring software changes.

3. Benchmark Applications

Our benchmarks were chosen to stress the limits of a processor's memory system, but they represent the kernels of real applications of interest in large-scale scientific computing. Most of them are taken from the DARPA Data Intensive Systems (DIS) stressmark suite [8]. In general, data-intensive applications are characterized by low arithmetic operation counts per datum relative to memory access. Many of the problems are further complicated by irregular memory access patterns or control structures. These characteristics often lead to performance scaling deficiencies when executed in parallel and to memory bottlenecks on single processors.

Transitive Closure: The first benchmark problem is to compute the transitive closure of a directed graph in a dense representation [6]. The code taken from the DIS reference implementation used non-unit stride [8], but was easily changed to unit stride. This benchmark performs only 2 arithmetic operations (an add and a min) at each step, while it executes 2 loads and 1 store.

GUPS: This benchmark is a synthetic problem, which measures giga-updates-per-second [9]. It repeatedly reads and updates distinct, pseudo-random memory locations. The inner loop contains 1 arithmetic operation, 2 loads, and 1 store, but unlike transitive, the memory accesses are random. It contains abundant data-parallelism because the addresses are pre-computed and free of duplicates.

Sparse Matrix-Vector Multiplication (SPMV): This problem also requires random memory access patterns and a low number of arithmetic operations. It is common in scientific applications, and appears in both the DIS [8] and NPB [2] suites in the form of a Conjugate Gradient (CG) solver. We have a CG implementation for IRAM, which is dominated by SPMV, but here we focus on the kernel to isolate the memory system issues. The matrices contain a pseudo-random pattern of non-zeros using a construction algorithm from the DIS specification [8], parameterized by the matrix dimension, n, and the number of nonzeros, m.

Histogram: Computing a histogram of a set of integers can be used for sorting and in some image processing problems [8]. Two important considerations govern the algorithmic choice: the number of buckets, b, and the likelihood of duplicates. For image processing, the number of buckets is large and collisions are common because there are typically many occurrences of certain colors (e.g., white) in an image. Histogram is nearly identical to GUPS in its memory behavior, but differs due to the possibility of collisions, which limit parallelism and are particularly challenging in a data-parallel model.

Mesh Adaptation: The final benchmark is a two-dimensional unstructured mesh adaptation algorithm [18] based on triangular elements. This benchmark is more complex than the others, and there is no single inner loop to characterize. The memory accesses include both random and unit stride, and the key problem is the complex control structure, since there are several different cases when inserting a new point into an existing mesh. Starting with a coarse-grained task parallel program, we performed significant code reorganization and data preprocessing to allow vectorization.

Table 1 summarizes the key features of each of our benchmarks. All are memory-intensive: the number of arithmetic/logical operations per step of the algorithm (Ops/step) is never more than the number of memory op-

erations (loads/stores) per step (Mem/step). Most of them involve some amount of irregular memory access, indicated in the table as indexed, although in the case of SPMV and histogram, we consider several different algorithms across which the number and nature of indexed memory operations differ. The table does not capture differences in parallelism, which may limit vector length, or the control irregularity, which leads to less efficient masked operations.

	Width	Mem access	Data size	Total Ops	Ops/ step	Mem/ step
Transitive	32	unit	n^2	n^3	2	2 ld 1 st
GUPS	8,16, 32,64	indexed, unit	$2n$	$2n$	1	2 ld, 1 st
SPMV	32	indexed, unit	$2m + 2n$	$2m$	2	3 ld
Histogram	16,32	indexed, unit	$n + b$	n	1	2 ld, 1 st
Mesh	32	indexed, unit	$1000n$	N/A	N/A	N/A

Table 1. Key features of benchmarks

4. Benchmarking Environment

As a comparison for the VIRAM design, we chose a set of commercial microprocessor systems. Most of these are high-end workstation or PC processors, but we also included a low power Pentium III for comparison. Details of the systems are show in Table 2.

	SPARC IIi	MIPS R10K	P III	P 4	Alpha EV6
Make	Sun Ultra 10	Origin 2000	Intel Mobile	Dell	Compaq DS10
Clock	333MHz	180MHz	600MHz	1.5GHz	466MHz
L1	16+16KB	32+32KB	32KB	12+8KB	64+64KB
L2	2MB	1MB	256KB	256KB	2MB
Mem	256MB	1GB	128MB	1GB	512MB

Table 2. Cache-based machines in our study

The VIRAM chip is scheduled for fabrication in early 2002, so performance reported here is based on a cycle-accurate simulator of the chip. The compiler is based on Cray's vectorizing C compiler, which has been developed with over 20 years of experience in vectorization. The compiler performs several loop transformations and allows users to assert that a loop is free of dependencies. This is important for loops with indexed memory operations that may not be provably vectorizable. The VIRAM version has its own backend that generates a mixture of MIPS scalar instructions and VIRAM vector instructions. While the machine-independent vectorizer is quite sophisticated, the code generator has not gone through the kind of rigorous performance tuning that one would expect from a commercial compiler. In particular, there are cases in which the compiler generates extra boundary checks and redundant loads, and we note two instances below where we somewhat hand-tuned the code.

5. Memory Bandwidth

The best-case scenario for both caches and vectors is a unit stride memory access pattern, as found in the transitive closure benchmark. In this case, the main advantage for IRAM is the size of its on-chip memory, since DRAM is denser than SRAM. VIRAM has 12 MB of on-chip memory compared to 10s of KB for the L1 caches on the cache-based machines. IRAM is admittedly a large chip, but this is partly due to being an academic research project with a very small design team—the 2-3 orders of magnitude advantage in on-chip memory size is due primarily to the memory technology.

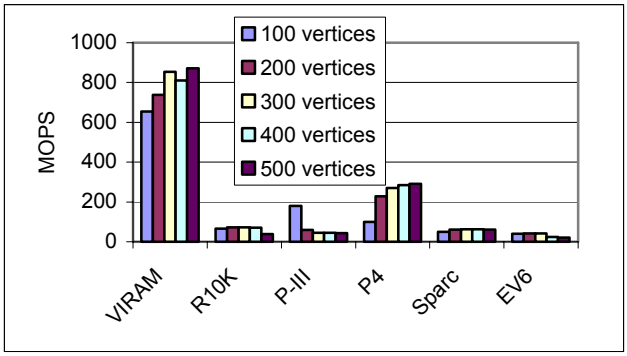

Figure 2. Performance of Transitive Closure

Figure 2 shows the performance of the transitive closure benchmark. Results confirm the expected advantage for VIRAM on a problem with abundant parallelism and a low arithmetic/memory operation ratio. Performance is relatively insensitive to graph size, although VIRAM performs better on larger problems due to the longer average vector length. The Pentium 4 has a similar effect, which may be due to improved branch prediction because of the sparse graph structure in our test problem.

6. Address Generation and Memory Conflicts

A more challenging memory access pattern is one with either non-unit strides or indexed loads and stores (scatter/gather operations). The first challenge for any machine is generating the addresses, since each address needs to be checked for validity and for collisions. VIRAM can generate only 4 addresses per cycle, independ-

ent of the data width. For 64-bit data, this is sufficient to load or store a value on every cycle, but if the data width is halved to 32-bits, the 4 64-bit lanes perform arithmetic operations at the rate of 4 32-bit lanes, and the arithmetic unit can more easily be starved for data. In addition, details of the memory bank structure can become apparent, as multiple accesses to the same DRAM bank require additional latency to charge the DRAM. The frequency of these bank-conflicts depends on the memory access pattern and the number of banks in the memory system.

The GUPS benchmark results, shown in Figure 3, highlights the address generation issue. Although performance improves slightly when moving from 64 to 32 bits, after that performance is constant due to the limits for 4 address generators. Overall, though, VIRAM does very well on this benchmark, nearly doubling the performance of its nearest competitor, the Pentium 4, for 32 and 64 bit data. In fairness, GUPS was the one benchmark in which we tidied up the compiler-generated assembly instructions for the inner loops, which produced a 20-60% speedup.

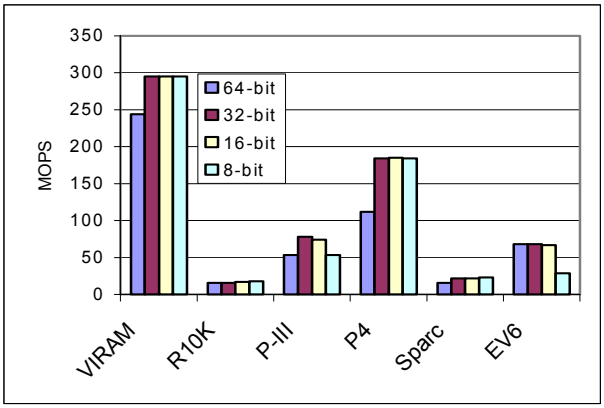

Figure 3. Performance of GUPS

In addition to the MOP rate, it is interesting to observe the memory bandwidth consumed in this problem. GUPS achieves 1.77, 2.36, 3.54, and 4.87 GB/s memory bandwidth on VIRAM at 8, 16, 32, and 64-bit data widths, respectively. This is relatively close to the peak memory bandwidth of 6.4 GB/s.

7. Exploiting Fine-Grained Parallelism

Nearly all modern processors use fine-grained parallelism for performance, especially to mask memory latency. In VIRAM, the use of parallelism is explicit in the instruction set, which allows for a simple, low-power implementation, but places the burden of discovering parallelism on the application programmer and compiler. Our last three benchmarks, SPMV, Histogram, and Mesh, while ostensibly just as memory-intensive as the first two, required more work on our part to take advantage of the on-chip memory bandwidth provided by VIRAM. The issues are slightly different across the benchmarks: SPMV is limited by the degree of parallelism, whereas Histogram and Mesh have parallelism, but not pure data parallelism. We describe each of these vectorization problems below.

7.1 SPMV

For our SPMV benchmark, we set the matrix dimension to 10,000 and the number of nonzeros to 177,782, i.e., there were about 18 nonzeros per row. The computation is done in single precision floating-point. The pseudo-random pattern of nonzeros is particularly challenging, and many matrices taken from real applications have some structure that would have better locality, which would especially benefit cache-base machines [11].

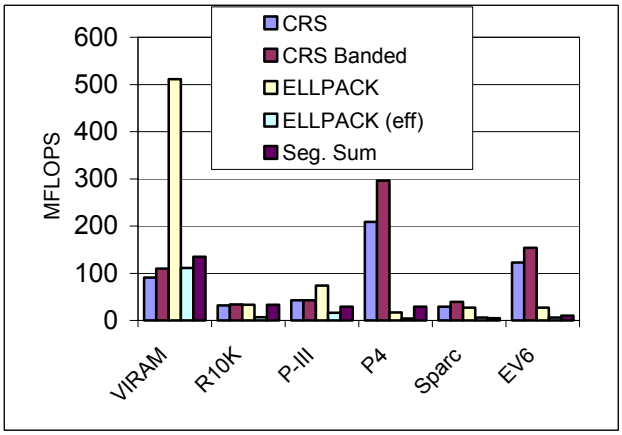

Figure 4. Performance of SPMV

We consider 4 different algorithms for SPMV, reflecting the best practice for both cache-based and vector machines. The performance results are shown in Figure 4. *Compressed Row Storage* (CRS) is the most common sparse matrix format, which stores an array of column indices and non-zero values for each row; SPMV is then performed as a series of sparse dot products. The performance on VIRAM is better than some cache-based machines, but it suffers from lack of parallelism. The dot product is performed by recursive halving, so vectors start with an average of 18 elements and drop from there. Both the P4 and EV6 exceed VIRAM performance for this reason. *CRS-banded* uses the same format and algorithm as CRS, but reflects a different nonzero structure that would likely result from bandwidth reduction orderings, such as reverse Cuthill-McKee (RCM) [7]. This has little effect on IRAM, but improves the cache hit rate on some of the other machines.

The *Ellpack* (or Itpack) format [13] forces all rows to have the same length by padding them with zeros. It still has indexed memory operations, but increases available data parallelism through vectorization across rows. The raw Ellpack performance is excellent, and this format should be used on VIRAM and PIII for matrices with the longest row length close to the average. If we instead

measure the effective performance (eff), which discounts operations performed on padded zeros, the efficiency can be arbitrarily poor. Indeed, the randomly generated DIS matrix has an enormous increase in the matrix size and number of operations, making it impractical.

The *Segmented-sum* algorithm was first proposed for the Cray PVP [5]. The data structure is an augmented form of the CRS format and the computational structure is similar to Ellpack, although there is additional control complexity. We modified the underlying Ellpack algorithm that converts roughly 2/3 of the memory accesses from a large stride to unit stride. The remaining 1/3 are still indexed references. This was important on VIRAM, because we are using 32-bit data and have only 4 address generators as discussed above.

7.2. Histogram

This benchmark builds a histogram for the pixels in a 500×500 image from the DIS Specification. The number of buckets depends on the number of bits in each pixel, so we use the base 2 logarithm (i.e., the pixel depth) as the parameter in our study. Performance results for pixel depths of 7, 11, and 15 are shown in Figure 5. The first five sets are for VIRAM, all but the second (Retry 0%) use this image data set. The first set (Retry) uses the compiler default vectorization algorithm, which vectorizes while ignoring duplicates, and corrects the duplicates in a serial phase at the end [22]. This works well if there are few duplicates, but performs poorly for our case. The second set (Retry 0%) shows the performance when the same algorithm is used on data containing no duplicates. The third set (Priv) makes several private copies of the buckets with the copies merged at the end [1]. It performs poorly due to the large number of buckets and gets worse as this number increases with the pixel depth.

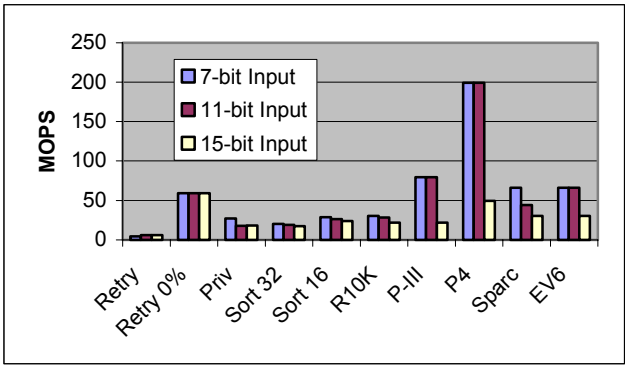

Figure 5. Performance of Histogram

The fourth and fifth algorithms use a more sophisticated sort-diff-find-diff algorithm [16] that performs in-register sorting. Bitonic sort [3] was used because the communication requirements are regular and it proved to be a good match for VIRAM's "butterfly" permutation instructions, designed primarily for reductions and FFTs [23]. The compiler automatically generates in-register permutation code for reductions, but the sorting algorithm used here was hand-coded. The two sort algorithms differ on the allowed data width: one works when the width is less than 16 bits and the other when it is up to 32 bits. The narrower width takes advantage of the higher arithmetic performance for narrow data on VIRAM.

Results show that on VIRAM, the sort-based and privatized optimization methods consistently give the best performance over the range of bit depths. It also demonstrates the improvements that can be obtained when the algorithm is tailored to shorter bit depths. Overall, VIRAM does not do as well as on the other benchmarks, because the presence of duplicates hurts vectorization, but can actually help improve cache hits on cache-based machines. We therefore see excellent timings for the histogram computation on these machines without any special optimizations. A memory system advantage starts to be apparent for 15-bit pixels, where the histograms do not fit in cache, and at this point VIRAM's performance is comparable to the faster microprocessors.

7.3. Mesh Adaptation

This benchmark performs a single level of refinement starting with a mesh of 4802 triangular elements, 2500 vertices, and 7301 edges. In this application, we use a different algorithm organization for the different machines: The original code was designed for conventional processors and is used for those machines, while the vector algorithm uses more memory bandwidth but contains smaller loop bodies, which helps the compiler perform vectorization. The vectorized code also pre-sorts the mesh points to avoid branches in the inner loop, as in Histogram. Although the branches negatively affect superscalar performance, presorting is too expensive on those machines. Mesh adaptation also requires indexed memory operations, so address generation again limits VIRAM. Figure 6 shows the performance.

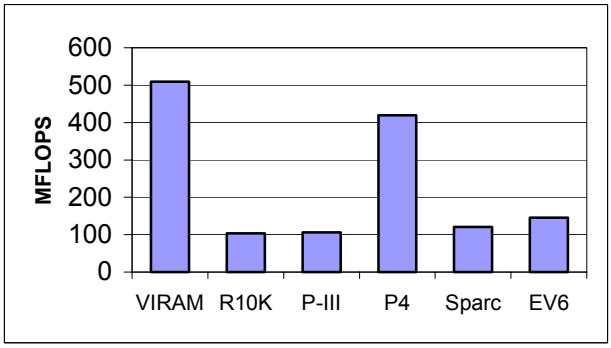

Figure 6. Performance of Mesh Adaptation

8. Summary of Benchmark Characteristics

An underlying goal in our work was to identify the limiting factor in these memory-intensive benchmarks. The graph in Figure 7 shows the memory bandwidth used on VIRAM and the MOPS rate achieved on each of the benchmarks using the best algorithm on the most challenging input. GUPS uses the 64-bit version of the problem, SPMV uses the segmented sum algorithm, and Histogram uses the 16-bit sort.

While all of these problems have low operation counts per memory operation, as shown in Table 1, the memory and operation rates are quite different in practice. Of these benchmarks, GUPS is the most memory-intensive, whereas Mesh is the least. Histogram, SPMV and Transitive have roughly the same balance between computation and memory, although their absolute performance varies dramatically due to differences in parallelism. In particular, although GUPS and Histogram are nearly identical in the characteristics from Table 1, the difference in parallelism results in a very different absolute performance as well as relative bandwidth to operation rate.

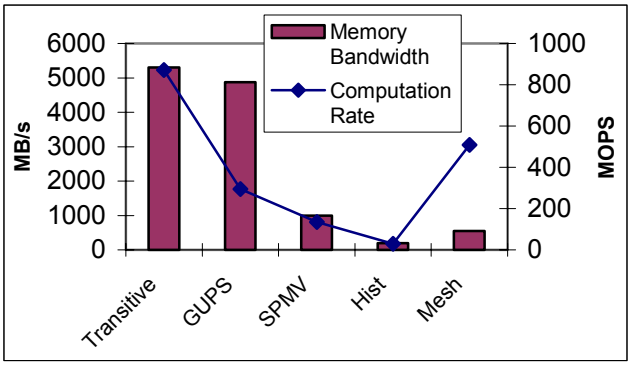

Figure 7. Memory bandwidth vs. MOPS

9. Power and Performance

Figure 8 shows the summary of performance for each of the benchmarks across machines. The y-axis is a log scale, and IRAM is significantly faster than the other machines on all applications except SPMV and Histogram.

An even more dramatic picture is seen from measuring the MOPS/Watt ratio, as shown in Figure 9. Most of the cache-based machines use a small amount of parallelism, but spend a great deal of power on a high clock rate. Indeed a graph of Flops per machine cycle is very similar. Only the Pentium III, designed for portable machines, has a comparable power consumption of 4 Watts compared to IRAM's 2 Watts. The Pentium III cannot compete on performance, however, due to lack of parallelism.

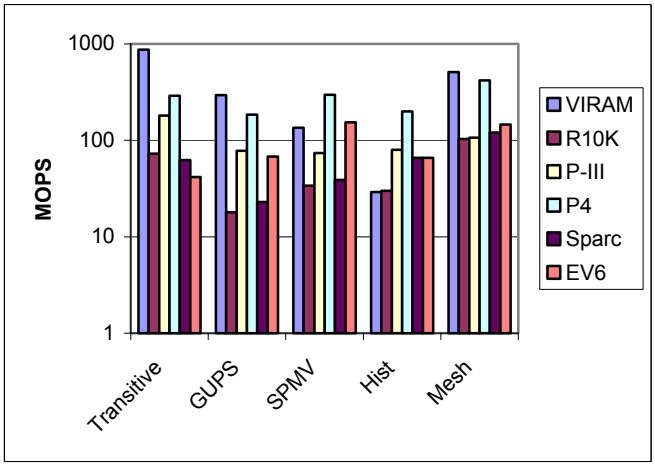

Figure 8. Performance across machines

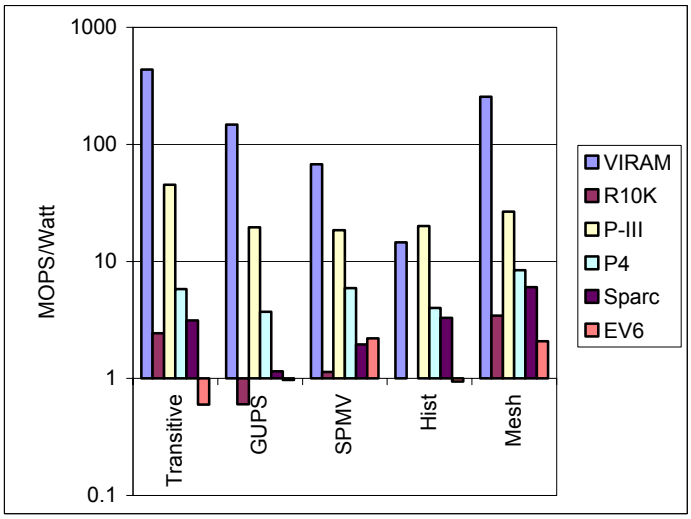

Figure 9. Power efficiency

10. Related Work

The VIRAM processor is one example of a system that uses mixed logic and DRAM [14,15]. Other examples include the DIVA project [10], the HTMT project [21], and the Mitsubishi M32R processor [17]. The DIVA project is directly addressing the use of this technology in a multiprocessor, so their focus has been on scalability [10]. The HTMT processor uses fine-grained multi-threading and forward-looking hardware technology. The support for control parallelism in HTMT might be an advantage for the Histogram and Mesh benchmarks, although it requires more complex hardware support.

The Imagine processor does not use embedded DRAM, but is similar to IRAM in its use of fine-grained data parallelism and support for media processing [12]. The

SLIIC group at ISI has done a similar study to ours, including the M32R, Imagine, and IRAM, but for a different set of signal processing and defense applications [20].

11. Conclusions and Future Work

In this work, we used a set of memory-intensive benchmarks to compare the performance of conventional cache-based microprocessors to a mixed logic and DRAM processor called VIRAM. Our experience with these benchmarks suggest that VIRAM is significantly faster than conventional processors for problems that are limited only by DRAM bandwidth and latency, and because VIRAM achieves high performance through parallelism rather than a fast clock rate, the advantages are even larger if one is interested in building a power-efficient multiprocessor system.

While memory is important in all of our benchmarks, simple bandwidth was not sufficient. Most of the benchmarks involved irregular memory access patterns, so address generation bandwidth was important, and collisions within the memory system were sometimes a limitation. Although the histogram and mesh adaptation problems are parallelizable, the potential for data sharing even within vector operations limits performance. The need to statically specify parallelism also requires algorithms that are highly regular. In SPMV, this lead to data structure padding; in mesh adaptation and histogram, there was some pre-sorting to group uniform data elements together.

Although we have concentrated on the memory systems within a single node, we believe these results indicate that IRAM would be a reasonable building block for large-scale multiprocessors. More work is needed to understand how well a system of IRAM processors would be balanced, given current networking technology. Finally, of course, it will be indispensable to re-run our benchmarks on the real VIRAM hardware once the system is available.

Acknowledgements

The authors would like to thank Hyun Jin Moon and Hyun Jin Kim for their work on the transitive benchmark, as well as Christoforos Kozyrakis, David Judd, and the rest of the IRAM team for their help on this project.

This work was supported in part by the Laboratory Directed Research and Development Program of the Lawrence Berkeley National Laboratory (supported by the U.S. Department of Energy under contract number DE-AC03-76SF00098), by DARPA under contract DABT63-96-C-0056, and by the California State MICRO Program. The VIRAM chip and compiler were supported by donations from IBM, Cray, and MIPS.

References

1. Y. Abe. Present Status of Computer Simulation at IPP, in *Proceedings of Supercomputing '88*: vol 2, Science and Applications, 1988.
2. D.H. Bailey, J. Barton, T. Lasinski, and H.D. Simon (Eds.). The NAS parallel benchmarks. Tech. Rep. RNR-91-002, NASA Ames Research Center, Moffett Field, 1991.
3. K. Batcher. Sorting networks and their applications. *Proc. AFIPS Spring Joint Compute Conf.*, 1968.
4. The Berkeley Intelligent RAM (IRAM) Project, Univ. of California, Berkeley, at http://iram.cs.berkeley.edu.
5. G.E. Blelloch, M.A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Tech. Rep. CMU-CS-93-173, Carnegie Mellon Univ., Pittsburgh, 1993.
6. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*, MIT Press, 1990.
7. E.Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. *Proc. ACM Natl. Conf.*, 1969, 157-192.
8. DIS Stressmark Suite, v 1.0. Titan Systems Corp., 2000, at http://www.aaec.com/projectweb/dis/
9. B.R. Gaeke. GUPS benchmark manual. Univ. of California, Berkeley, at http://iram.cs.berkeley.edu/~brg/.
10. M. Hall, *et al*. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. *Proc. SC99*, 1999.
11. E. Im and K.A. Yelick. Optimizing sparse matrix-vector multiplication for register reuse. *Proc. Intl. Conf. on Computational Science*, 2001.
12. B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. "Imagine: Media Processing with Streams." IEEE Micro, Mar/April 2001.
13. D.R. Kincaid, T.C. Oppe, and D.M. Young. ITPACKV 2D user's guide. Tech. Rep. CAN-232, Univ. of Texas, Austin, 1989.
14. C. Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Tech. Rep. UCB-CSD-99-1059, Univ. of California, Berkeley, 1999.
15. C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick. Hardware/compiler co-development for an embedded media processor. *Proceedings of the IEEE*, 2001.
16. The MathWorks. How do I vectorize my code? Tech. Note 1109, at http://www.mathworks.com.
17. Mitsubishi 32-bit Microcontrollers, at http://www.mitsubishichips.com.
18. L. Oliker and R. Biswas. Parallelization of a dynamic unstructured algorithm using three leading programming paradigms. *IEEE Trans. Parallel and Distributed Systems*, 11(9):931-940, 2000.
19. D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, R. Thomas, C. Kozyrakis, K. Yelick. Intelligent RAM (IRAM): Chips that remember and compute. *Proc. Intl. Solid-State Circuits Conf.*, 1997.
20. The SLIIC Project, at http://www.east.isi.edu/SLIIC/
21. T. Sterling. A hybrid technology multithreaded (HTMT) computer architecture for petaflops computing. Jet Propulsion Laboratory, Pasadena, 1997, at http://htmt.jpl.nasa.gov.
22. K. Suehiro, H. Murai, Y. Seo. Integer Sorting on Shared-Memory Vector Parallel Computers. In *Proceedings of ICS '98*, 1998.
23. R. Thomas and K.A. Yelick. Efficient FFTs on IRAM. *Proc. Workshop on Media Processors and DSPs*, 1999.