Hardware/Compiler Codevelopment for an Embedded Media Processor

CHRISTOFOROS KOZYRAKIS, STUDENT MEMBER, IEEE, DAVID JUDD,
JOSEPH GEBIS, STUDENT MEMBER, IEEE, SAMUEL WILLIAMS, DAVID PATTERSON, FELLOW, IEEE,
AND KATHERINE YELICK, MEMBER, IEEE

Invited Paper

Embedded and portable systems running multimedia applications create a new challenge for hardware architects. A microprocessor for such applications needs to be easy to program like a general-purpose processor and have the performance and power efficiency of a digital signal processor.

This paper presents the codevelopment of the instruction set, the hardware, and the compiler for the Vector IRAM media processor. A vector architecture is used to exploit the data parallelism of multimedia programs, which allows the use of highly modular hardware and enables implementations that combine high performance, low power consumption, and reduced design complexity. It also leads to a compiler model that is efficient both in terms of performance and executable code size. The memory system for the vector processor is implemented using embedded DRAM technology, which provides high bandwidth in an integrated, cost-effective manner.

The hardware and the compiler for this architecture make complementary contributions to the efficiency of the overall system. This paper explores the interactions and tradeoffs between them, as well as the enhancements to a vector architecture necessary for multimedia processing. We also describe how the architecture, design, and compiler features come together in a prototype system-on-a-chip, able to execute 3.2 billion operations per second per watt.

Keywords**—***Compiler, data parallelism, embedded DRAM, embedded systems, multimedia, scalable hardware, vector processor.*

I. Introduction

The trend toward ubiquitous computing will be fueled by small embedded and portable systems that are able to effectively run multimedia applications for audio, video, image, and graphics processing [1]. The microprocessor needed for such devices is actually a merged general-purpose processor and digital-signal processor, with the programmability of the former and the performance and power budget of the latter. Thus, the challenge is to develop architectures that lead to efficient hardware implementations from a performance, power consumption, and complexity standpoint, while providing an effective programming and code generation environment.

A key to meeting this challenge is the observation that multimedia applications are a rich source of fine-grained, data parallelism. Such programs repeat the same set of basic operations over an input sequence of image pixels, video frames, or audio samples [2]. Data parallelism provides the opportunity for designing high performance, yet powerful and complexity efficient hardware. Since data parallelism is also explicit in the algorithmic description of multimedia applications, there is also an opportunity for a simple programming model that results in compact binary executables.

The current popular approaches to processor architecture, developed originally for desktop and server systems, fail to exploit the data parallelism in multimedia applications in order to provide the combination of performance, power, and complexity needed for embedded devices. Superscalar processors [3] need to rediscover parallelism within an inherently sequential instruction stream. In practice, this limits the number of operations that can be executed concurrently in order to achieve high performance and increases power consumption and design complexity [4]. The latter actually grows in a super-linear fashion with the number of operations issued in parallel in a superscalar processor. Very long instruction word (VLIW) designs [5] require multiple, long instructions to effectively communicate data parallelism to hardware, resulting in increased code size [6]. Both architectural approaches consume power to fetch and decode one instruction for every operation to be executed, which is inefficient when the same operation is repeated over a stream of input data.

Manuscript received January 4, 2001; revised June 15, 2001. This work was supported in part by the Defense Advanced Research Projects Agency under contract DABT63-96-C-0056, by the California State MICRO Program, and by the Department of Energy. IBM, MIPS Technologies, Cray Inc., and Avanti Corp. have made significant software or hardware contributions to this work.

The authors are with the Computer Science Division, University of California, Berkeley, CA 94720-1776 USA (e-mail: kozyraki@cs.berkeley.edu; dajudd@cs.berkeley.edu; gebis@cs.berkeley.edu; samw@cs.berkeley.edu; pattrsn@cs.berkeley.edu; yelick@cs.berkeley.edu).

Publisher Item Identifier S 0018-9219(01)09689-X.

0018–9219/01$10.00 © 2001 IEEE

In this paper, we describe the codesign of a microprocessor and its compiler that take advantage of data parallelism in a high-performance, low-power, low-complexity system called *Vector IRAM* (*VIRAM*). The key features of VIRAM are: a vector instruction set architecture (ISA) that explicitly communicates data parallelism to the hardware in a compact way; an implementation model that allows one to make performance, power, and area tradeoffs without requiring ISA modifications or increased design complexity; a memory system based on embedded DRAM technology that supports the high memory bandwidth needed for media processing in a low-power system-on-a-chip; and a simple compilation model that provides an efficient, high-level programming interface. Although the use of vector processors and compilers is well-understood in the scientific computing domain [7], several hardware and software changes must be made to match the characteristics of multimedia computing in embedded devices.

We also present the VIRAM-1 microprocessor, a prototype chip fabricated to demonstrate the viability of this architecture. Implemented in a mixed logic-DRAM CMOS process, the 130 million transistor design includes a multimedia vector processor and 14 MB of DRAM. Running at merely 200 MHz in order to reduce power consumption, the highly parallel design achieves 3.2 billion 16-bit integer operations per second per watt, or 0.8 billion 32-bit floating-point operations per second per watt.

This paper is organized around some of the key features or issues of multimedia applications and how they are supported in hardware and software. Section II reviews the vector architecture as a model for exploiting fine-grained parallelism and describes the VIRAM implementation approach for scalable vector processors. Section III discusses the narrow data types and operations that are common in multimedia and how the vector architecture can be modified to support them efficiently and elegantly. Section IV addresses the problems associated with vectorizing applications with conditional branches. A common myth about vector architectures is that they require expensive, SRAM-based memory systems. Section V shows how embedded DRAM can provide the necessary memory bandwidth for a rich set of memory access patterns. Finally, Section VI describes the implementation of these ideas in the VIRAM-1 processor and discusses some of the practical issues involved in its development.

II. Explicit Support for Data Parallelism

The presence of parallelism in applications is the key to achieving high performance with all modern microprocessors, for it allows the hardware to accelerate applications by executing multiple, independent operations concurrently [8]. The most prevalent form of parallelism available in multimedia applications is data parallelism, as their computationally intensive kernels repeat the same set of operations over streams of input data [2].

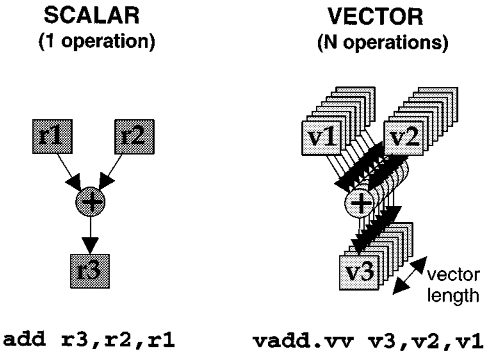


Fig. 1. Although a scalar instruction specifies a single operation, a vector instruction describes a set of independent, yet identical, operations on the elements of two vector registers.

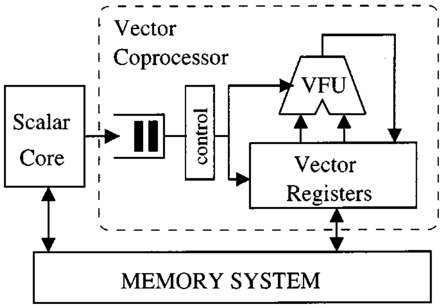


Fig. 2. Simplified view of a vector processor with one functional unit for arithmetic operations (VFU). The functional unit contains a single execution datapath.

The VIRAM architecture relies on vector processing to express and exploit the data parallelism in multimedia programs. Apart from the typical, scalar operations defined in RISC architectures, a vector architecture includes arithmetic, load-store, and logical instructions that operate on vectors, linear arrays of numbers, as shown in Fig. 1. Each instruction specifies a set of operand vectors, a vector length, and an operation to be applied element-wise to the vector operands. Operands are stored in a vector register file, a two-dimensional memory array where each line holds all the elements for one vector. A single vector instruction specifies a number of identical element operations on independent input and output data stored in vector registers. Hence, vector instructions provide a compact description of the data parallelism available in an application. This description is explicit, as it directly states that there are no dependences between the element operations, and hence they can be executed in parallel.

The roles for the hardware (processor) and the compiler in a vector architecture are complementary. The compiler discovers the fine-grain parallelism available in applications, and applies a set of transformations that allow the use of vector instructions. The hardware uses the explicit knowledge of parallelism to deliver high performance at low power consumption.

A. Exploiting Data Parallelism in Vector Hardware

Fig. 2 shows the hardware required for a vector architecture, with the logic for executing vector instructions organized as a coprocessor to an ordinary RISC scalar core.

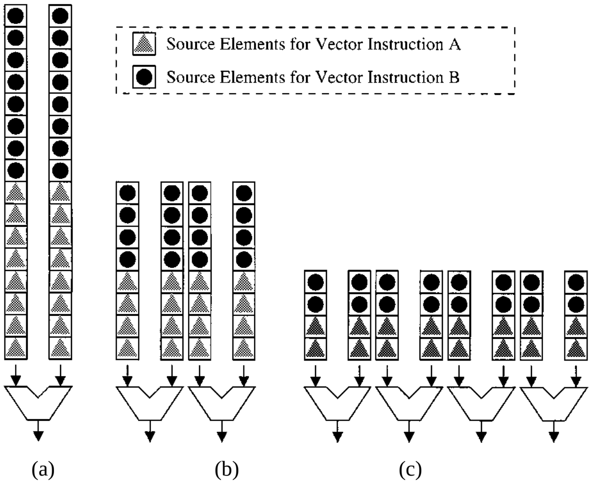

Source Elements for Vector Instruction A
Source Elements for Vector Instruction B
(a)
(b)
(c)

Fig. 3. Execution of two vector instructions in a vector processor with 1 (a), 2 (b), or 4 (c) datapaths per functional unit. Each instruction defines eight element operations. Increasing the number of datapaths per functional unit reduces the execution time, as more element operations can be processed concurrently.

The coprocessor includes the vector register file and a vector functional unit (VFU) that executes vector instructions at the rate of one element operation per cycle. Performance can be improved by allocating additional functional units in order to process multiple vector instructions in parallel. Unlike with superscalar and VLIW architectures, there is no need to linearly increase the instruction fetch and decode bandwidth with the number of functional units. Since a single vector instruction specifies multiple element operations at once, one instruction is typically sufficient to keep a functional unit busy for several clock cycles.

We can also exploit the fact that the element operations in vector instruction are by definition independent, in order to implement high-performance, yet simple vector hardware. Fig. 3 shows that multiple, identical pipelines or execution datapaths can be used within a vector functional unit to accelerate the execution of an instruction by processing multiple of its element operations in parallel. Each datapath receives identical control but different input elements in each clock cycle. Element operations are independent so there is no need for the dependence analysis logic used in superscalar processors, whose complexity scales super-linearly with the number of parallel operations. Element operations are also identical, hence multiple datapaths can be controlled by a single instruction decoder. A VLIW processor, on the other hand, would require a separate instruction decoder for each datapath it includes. It would also require an instruction set update and recompilation in order to exploit an additional datapath. A vector processor can take advantage of the extra datapath without modifications in the binary code, because multiple datapaths can be used for executing a single vector instruction.

Apart from improving performance, parallel datapaths can be used to trade off parallelism for reduced power consumption. Allocating additional datapaths allows the clock frequency for the processor to be reduced while keeping the throughput of operations almost constant. But a reduction in clock frequency allows a significant decrease in the power supply voltage [9]. The power dissipation for CMOS circuits is calculated by the equation

$$\text{Power} = \text{Capacitance} \cdot \text{Voltage}^2 \cdot \text{Frequency}.$$

Although the capacitance of the extra datapaths cancels the benefits from scaling the frequency, the exponential contribution of power supply voltage leads to significant reduction in power consumption. Thus, a vector processor with two datapaths per functional unit, operating at one half the frequency and lower supply voltage, could have the same performance but as a fraction of the power of a processor with a single datapath per functional unit. This architectural technique for power reduction is orthogonal to implementation methods like clock gating and low-power circuit design [10], which can also be used for further power savings.

The lack of dependence in element operations within a vector instruction also simplifies the physical implementation of a vector coprocessor with multiple datapaths per functional unit. Datapath and register files resources can be organized in *vector lanes*, as shown in Fig. 4. Each lane contains a datapath for each functional unit and a vertical partition of the vector register file [11]. The elements of each vector register are distributed across the lanes in a round-robin, interleaved fashion. By storing in each lane the same set of elements from each vector register, no inter-lane communication is necessary to execute the majority of vector instructions, such as vector additions or multiplications.

There are several benefits to the modular, lane-based implementation. A single lane must be designed and verified regardless of the number of lanes allocated in the processor. Scaling a vector processor by allocating the proper number of lanes leads to balanced addition of both register file and execution resources, without requiring redesign of functional units or their control. A four-lane processor, for example, can store vectors twice as long and execute twice as many element operations per cycle as a two-lane processor. Finally,

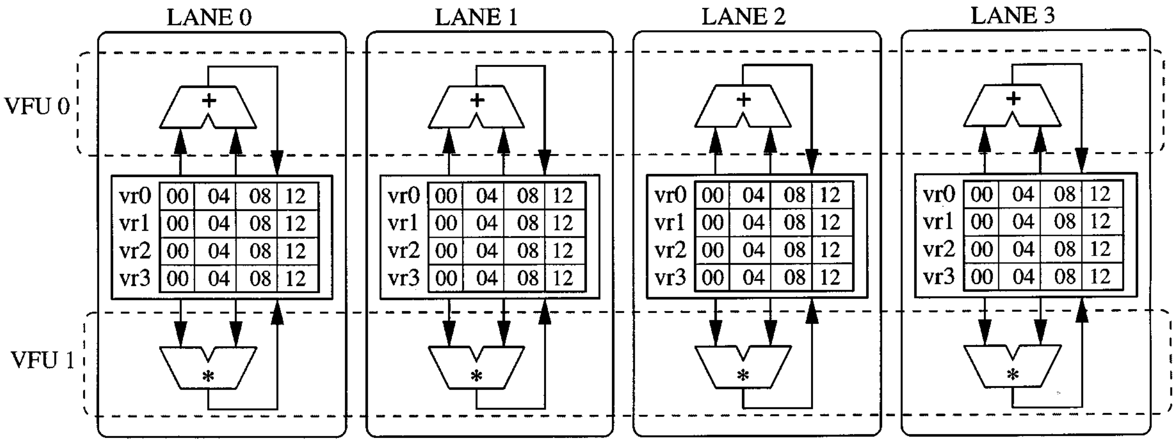


Fig. 4. Organization of a four-lane vector coprocessor, including two functional units for add (VFU 0) and multiply (VFU 1) and four datapaths per unit. The 16 elements for each vector register are distributed across the lanes. All four lanes are identical and receive the same control during the execution of vector instructions.

the locality of communication in a lane-based vector processor, allows hardware scaling without implications due to the high latency of long, cross-chip wires in CMOS technology [12].

In summary, the compact and explicit description of data parallelism with vector instructions, allows the design of vector processors in a modular and scalable manner. Tradeoffs between area and performance, or area and power consumption are possible without the need for binary modifications or additional instruction issue bandwidth, as it is the case with superscalar and VLIW designs.

B. *Compile-Time Vectorization*

For vector hardware to be useful, it is necessary to be able to compile multimedia applications, written in high-level languages such as C and C++, into sequences of vector instructions.

The VIRAM compiler is based on the Cray compiler environment (PDGCS) for vector supercomputers [13]. Fig. 5 presents the compiler flow and the basic tasks performed for vectorization. The front-end implements syntax and semantic analysis of C and C++ applications.[1] The Cray PDGS environment performs vectorization and implementation independent optimizations. The compiler back-end generates executable code for implementations of the VIRAM architecture and schedules code for optimal performance.

The main step in automatic vectorization is to discover loops that are free of *loop-carried dependencies* and, therefore, their iterations can be correctly executed in parallel. It is corresponding operations from each iteration that will be grouped together into a single vector instruction. The compiler computes control and data dependencies for each inner loop and identifies those that are free of dependencies. It also performs loop restructuring such as interchange or distribution (splitting) that enable vectorization [14]. Loop interchange, whereby the inner and outer loops are exchanged, is particularly useful in a loop nest where the inner loop is too short or does not vectorize due to a dependency; the compiler may interchange the loops so that the outer loop is actually vectorized.

The degree of data parallelism identified by the compiler may be much larger than that supported by a particular hardware implementation. The length of a vector register in an implementation, in other words the maximum number of elements it can store, is an upper limit to the number of parallel operations a vector instruction can specify. For any implementation of the VIRAM architecture, this number is available to software through a read-only register, called the maximum vector length (MVL) register. A vectorizable loop is divided by the compiler to produce two nested loops, with the inner loop executing MVL iterations and the outer loop incrementing by MVL on each iteration. This division to handle longer loops is called *strip mining* [15]. The inner loop is converted during code generation into a sequence of vector instructions. Of course, the compiler can also vectorize loops with fewer than MVL iterations by setting the vector length for the corresponding vector instructions to a value less than MVL.

The process of vectorization of applications written in high-level languages is not entirely automatic for most real applications. For example, the C language semantics allow array arguments, which are simply pointers, to be alias. In other words, two pointers may refer to the same array. This requires expensive and fragile interprocedural pointer analysis to fully automate. Instead we allow programmers to add other annotations that convey application specific knowledge and help with vectorization. We use the ANSI-standard C, which allows special comments called *pragmas* to be used for the programmer annotations. Although some annotations are necessary, experience from the scientific community has shown that the programming overhead is quite reasonable, and in particular is much less onerous than manual parallelization or hand-coding in assembly language. In addition, these annotations describe algorithmic properties of the application that are known to

[1] A Fortran front-end for the VIRAM compiler is also available but not used for this work.

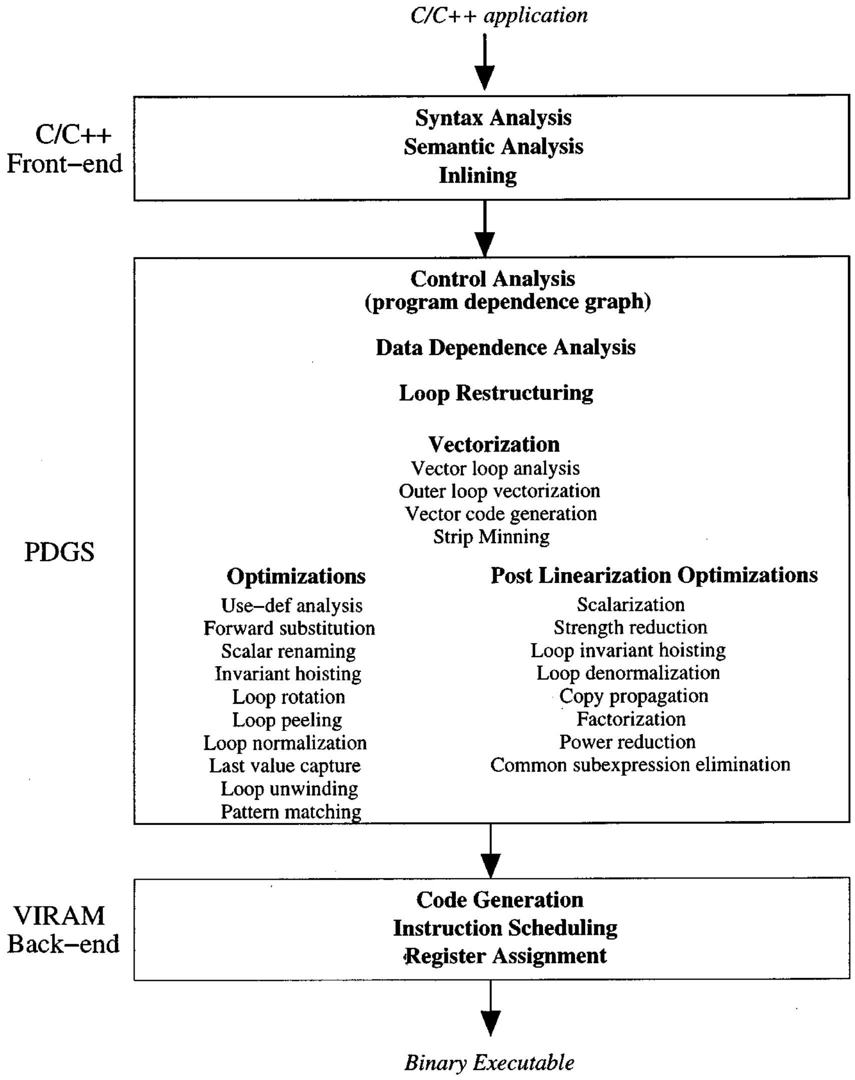


Fig. 5. The VIRAM compilation flow for C and C++ applications. Vectorization and optimizations are performed using the Cray Program Dependence Graph Compiler System (PDGCS).

the programmers of multimedia applications. Alternatively, we could have used the restricted pointer types that are defined in the latest C standard for avoiding aliases.

Fig. 6 shows the performance achieved with compiled code for three single-precision floating-point media kernels. The first is a 64 × 64 matrix multiplication, the second is a saxpy loop ($Y[i] = aX[i] + b$), and the third is a 12-point finite impulse response (FIR) filter ($Y[i] = \sum c[k]X[i-k]$). Matrix multiplication and the FIR filter use outer loop vectorization with programmer annotations. For the FIR filter, the inner loop is unrolled in the C source code. The bars represent the sustained performance (Mflops) for a VIRAM processor with one vector functional unit for floating-point arithmetic and one vector load-store unit, as measured using a near cycle accurate performance model. The first three groups of bars represent performance for an implementation with one, two, four, or eight vector lanes, respectively, with 64-bit datapaths,[2] running at 200 MHz. The last group of bars present the peak hardware performance for each configuration. The graph confirms the compiler's ability to generate efficient vectorized code for such media kernels and to utilize most of the peak hardware performance. In addition, sustained performance scales well as the number of vector lanes increases, even though the same binary executable was used for all configuration.[3]

III. Multimedia Support in a Vector Architecture

Vector processors were originally developed and optimized for supercomputing applications [7], where the typical workload is scientific computations on double-precision (64-bit) floating-point numbers. The data parallelism of multimedia applications is similar to that found in scientific computing. Yet, to enable efficient vectorization of signal processing and graphics applications, the VIRAM architecture includes a set of media-specific features.

[2] Two 32-bit floating-point operations can be executed concurrently on each 64-bit datapath.

[3] The performance for matrix multiplication does not scale well for the case of eight lanes due to limitations in the memory system modeled.

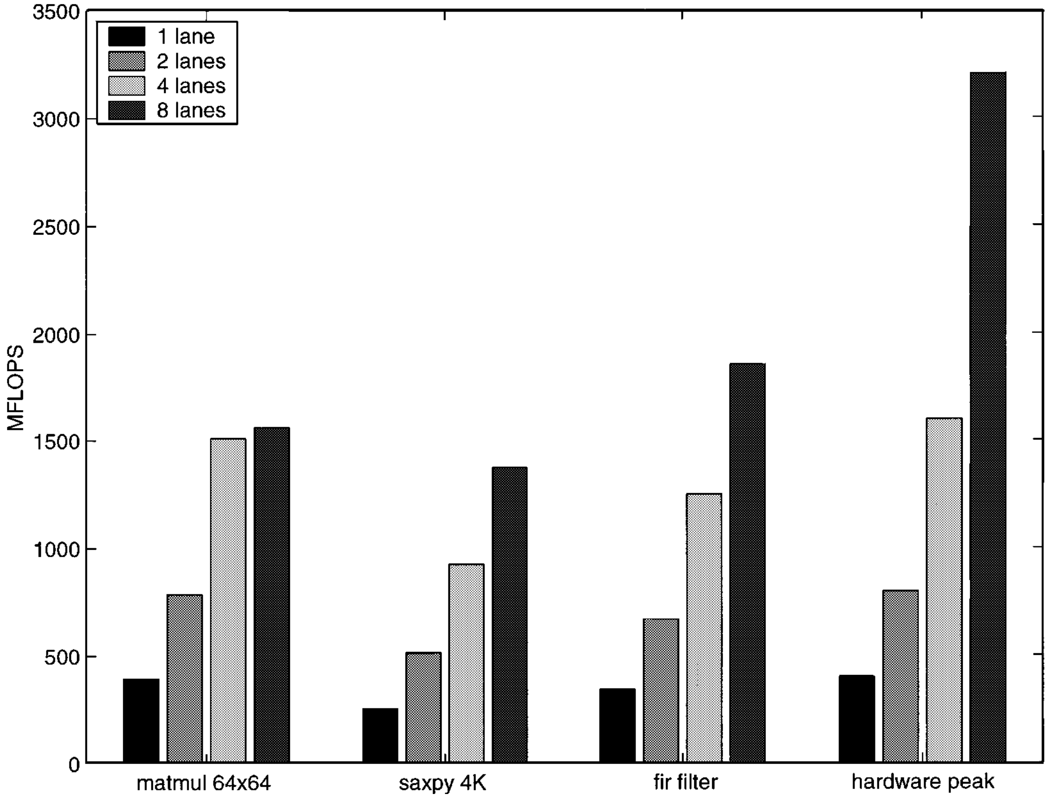


Fig. 6. Compiled performance in Mflops for three 32-bit floating-point multimedia kernels.

A. Support for Narrow Data Types

Multimedia algorithms encode audio or video data using narrow types, such as 8- and 16-bit values in memory. Arithmetic operations producing 16- or 32-bit results are typically sufficient to satisfy any accuracy requirements. The VIRAM architecture supports arithmetic operations on 64-, 32-, and 16-bit data types, and load-store operations on all of these as well as 8 bit values. A vector register can be considered an array of elements of any one of these sizes. Multiple narrow elements can be store within the space for a 64-bit element in every vector register. Similarly, multiple narrow operations can be executed in every 64-bit datapath in the vector lanes. The width of vector elements and arithmetic operations is indicated by a control register called the *virtual processor width* (VPW). By changing the virtual processor width from 32 to 16 bits, for example, each vector register can hold twice as many elements, and each datapath can complete twice as many operations per cycle. In this way, register file and datapath resources can be fully utilized for any of the supported data types. Applications that use narrow data or require lower arithmetic precision can benefit from matching the virtual processor width to their needs.

The VPW register is under compiler control. To avoid excessive overhead by frequently resetting it, the compiler computes the virtual processor width on a per loop nest basis, and sets it as necessary. For each loop nest, it chooses the minimum width that is wide enough for all of the instructions included. Setting VPW to a value wider than the minimum necessary for a specific instruction increases its execution time, but does not affect its correctness. On the other hand, using a VPW narrower than that needed by a certain instruction would lead to an incorrect or inaccurate computation.

Fig. 7 presents the performance of the VIRAM architecture for a set of integer benchmarks that use narrow data types. The four implementations in this case have two vector functional units for integer arithmetic, operate at 200 MHz, and include one, two, four, and eight vector lanes, respectively. The benchmarks are part of the University of Toronto DSP suite [16]. Decrypt implements IDEA decryption, while detect performs horizontal and vertical edge detection on digital images. Convolve implements image convolution using a 3×3 kernel and compose blends two images based on an alpha factor. Finally, colorspace performs RGB to YUV image format conversion. The first two benchmarks use a mixture of 32- and 16-bit operations, while the other three use virtual processor width of 16. The results, presented in sustained MOPS (million operations per cycle), demonstrate the performance benefits from using narrow data types when possible. They also show that the performance improvement using a narrow virtual processor width is orthogonal to that achieved by scaling the number of lanes in the processor implementation.

It is instructive at this point to compare the VIRAM model for narrow data types with the SIMD media extensions, such as the Intel MMX [17] or the PowerPC Altivec [18], which are widely used with superscalar and VLIW processors. SIMD instructions define fixed length, short vector, arithmetic operations by treating a scalar 64-bit register as a vector of narrower data. To execute more then four 16-bit

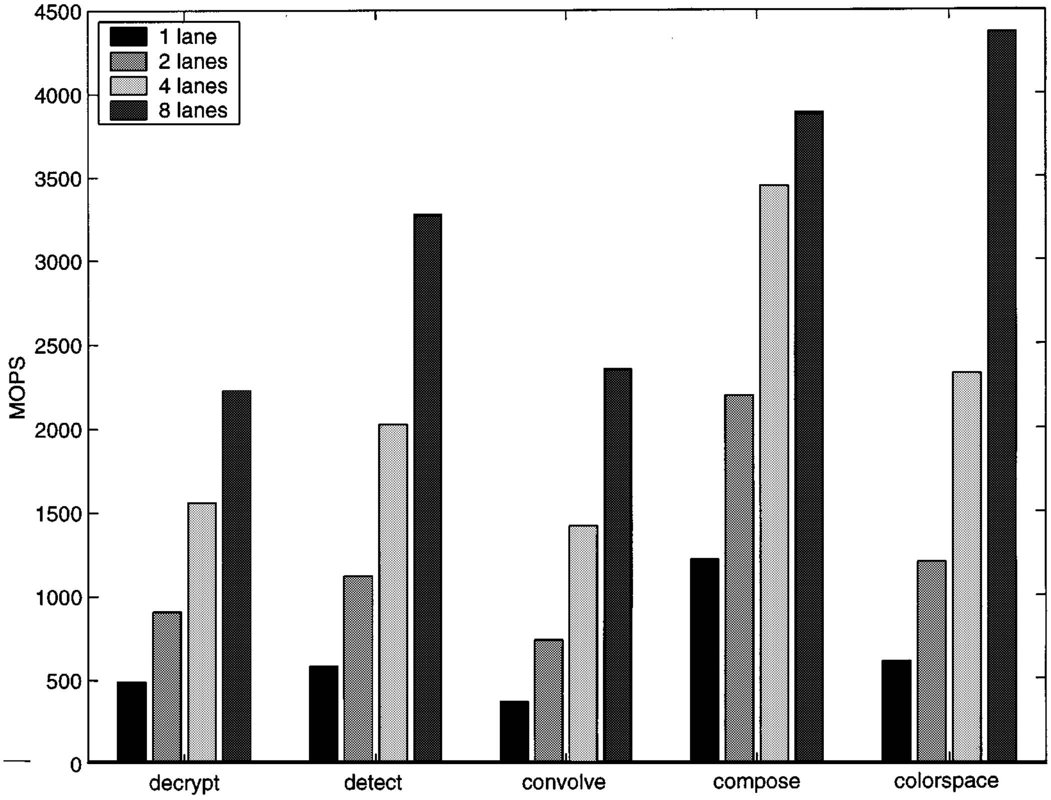


Fig. 7. Compiled performance in MOPS for five integer multimedia benchmarks.

operations per cycle, for example, a processor with SIMD extensions must be able to issue more than one instruction per cycle, or the instruction set must be changed to define operations on longer vectors (more than 64 bits). The vector model of the VIRAM architecture allows implementations to change the number of elements in a vector register (MVL) or the number of vector lanes allocated, without requiring additional instruction issue bandwidth or instruction set updates that lead to recompilation. VIRAM also allows the virtual processor width to be independent and possibly wider than the width of data in memory during load and store operations. Since it is both reasonable and often desirable to load 8-bit values into registers that hold 16-bit elements, the hardware can handle any corresponding conversion or alignment issues. In addition, VIRAM allows vectors to start on unaligned memory locations. Both features are missing from SIMD extensions leading to large software overhead for emulating the functionality of vector load and store instructions. This overhead often masks the benefits from parallel execution of narrow arithmetic operations. Finally, using the VPW register to specify the width in VIRAM reduces the number of unique instruction opcodes used in the instruction set. SIMD extensions, on the other hand, require one opcode per operation per data width supported. In summary, the VIRAM model makes it easier to justify the investment in vectorizing compiler technology, as the same compiler and back-end code generator can be used with a variety of hardware implementations and the overall performance benefits are significantly larger [19].

B. Support for Fixed-Point Arithmetic

Apart from narrow data types, multimedia applications frequently use fixed-point and saturated arithmetic. Fixed-point operations allow decimal calculations within narrow integer formats, which require less hardware resources or can be processed faster than floating-point formats. Saturation replaces modulo arithmetic to reduce the error introduced by overflow in signal processing algorithms. Architecture and hardware support for such features is typically found in digital signal processors (DSPs).

The VIRAM architecture supports both features with a set of vector fixed-point add, subtract, shift, multiply, and multiply-add instructions, which enhance the traditional functionality of these operations with saturation, rounding, and scaling. Fig. 8 presents the fixed-point arithmetic model for multiply-add. The programmable scaling of the multiplication result and the four rounding modes can support arbitrary fixed-point number formats. By setting the width of both the input and the output data for the multiply-add operation to be the same, all operands for this instruction can be stored in regular vector registers. There is no need for the extended precision registers or accumulators commonly found in DSPs, and this omission simplifies the use of these instructions by the compiler. The maximum precision of calculations can be set by selecting the proper virtual processor width. Algorithms with sensitivity to precision can use data types as wide as 32 or 64 bits, while those with limited sensitivity can use narrower data types to benefit from the higher number of operations per cycle when processing narrow numbers.

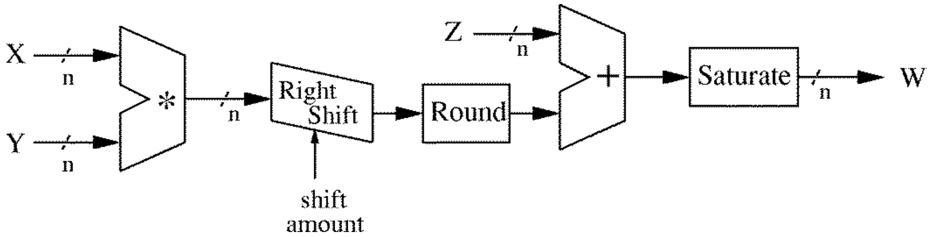


Fig. 8. VIRAM model for fixed-point multiply-add. All operands (X, Y, Z, W) are vector register elements. Only half of the bits (most or least significant) of the multiplier inputs are actually used. Shifting and rounding after the multiplication scales the result in the desired fixed-point format.

Currently, the VIRAM compiler does not generate the fixed-point instructions, because fixed-point data types and operations are not included in the semantics of the C programming language. Instead, programmers need to access these instructions using assembly language or intrinsics written on top of the assembly instructions. The language semantics problem for fixed-point data is not unique to VIRAM. Draft proposals are already available for extending the ANSI standard for the C programming language in order to address such issues. Once consensus has been reached and a standard model has been approved, the vectorizing compiler should have no difficulty utilizing the fixed-point instructions of the VIRAM architecture.

C. Vector Element Permutations

Multimedia applications often include reduction or transformation operations in order to implement kernels such as dot products or the fast Fourier transform (FFT). With traditional vector architectures, automatic vectorization of such kernels is impossible, as their description includes loop-carried dependencies. For example, a dot product is described as

$$Y = \sum X[i].$$

To enable efficient vectorization of such kernels, the VIRAM architecture provides a small set of instructions for vector element permutations.

For the case of dot products, one could introduce an instruction that performs a complete reduction on the elements of a vector register. Several such instructions would be necessary in order to support all useful types of reductions (add, exclusive or, etc.). Instead, VIRAM supports a permutation instruction that can be used to construct reductions of any type in multilane hardware implementations. This instruction moves the second half of the elements of a vector register into another register. By iteratively applying this permutation along with the proper arithmetic operation for the reduction, the elements of a vector register can be reduced to a single number while still utilizing all vector lanes available for as long as possible.[4] For the case of FFT, VIRAM also supports two vector instructions which perform left and right butterfly permutations on the elements of a vector register using a programmable radix [20].

The three vector permutation instructions implement the minimum functionality necessary to vectorize dot products and FFTs. Because of the regularity of their permutation patterns and the fact that the permutation is separated from the arithmetic instructions used for the reductions or the transforms, their implementation, both datapath and control, are simple. For most cases, they require only reading and writing elements within the vector register file partition of each lane. When inter-lane communication is required, it can be accommodated with a simple bus for every two lanes. This bus can also be pipelined if needed. The permutation instructions could be replaced by a single, general instruction that can perform any random permutation of vector elements, like the one used in the Altivec architecture [18]. But such an instruction would require a full crossbar to implement, which is difficult to design, control, and scale with the number of lanes in the design.

To utilize the permutation instructions, the VIRAM compiler recognizes linear recurrences like the one for reductions for operations like addition, multiplication, minimum and maximum, as well as logical and bitwise and, or, and exclusive or. For the add reduction, the generated code sums the vector X using vector adds until the result vector fits in a single vector register. At that point it generates a sequence of instructions that repeatedly move half of the remaining elements to another register and then add the two registers.[5] The speedup for vectorized reduction over the sequential versions is proportional to the number of vector lanes included in the hardware (see the FIR performance in Fig. 6).

For FFT, there are several algorithmic descriptions one can use. Instead of recognizing all of them in the compiler, we provide hand-optimized assembly routines that implement both fixed-point and floating-point FFTs. Using these routines, a four-lane, 200-MHz implementation of the VIRAM architecture can execute 1024 and 256 point single-precision complex FFTs within 37.0 and 9.5 μs, respectively [20].

IV. Support for Conditional Execution

The effectiveness of a vector architecture with multimedia applications depends on the level of vectorization the compiler can achieve with them. Therefore, it is important to pro-

[4] Since each iteration reduces the number of elements in the vector to half, the very last iterations may not have sufficient elements to exploit all the vector lanes available.

[5] The vectorized loop applies operations in a different order than the sequential one. The reordering may change the semantics for nonassociative operations with respect to exceptions and may change the lower bits of the result. Yet, this is not a problem for most applications, especially in multimedia processing.

C code:

```
for (i=0; i<N; i++) {
  if (A[i]<threshold) C[i]=A[i];
  else C[i]=B[i];
}
```

VIRAM assembly without outer strip-mining loop:

```
vld      a, a_address     # Vector load from array A
vld      b, b_address     # Vector load from array B
vcmp.lt  t, a, threshold  # Vector comparison; element i in flag
                          # register t is set if A[i]<threshold
vneg     f, t             # Negate the result of the comparison and
                          # store it in flag register f
vst      a, c_address, t  # Vector store under mask t to array C
vst      b, c_address, f  # Vector store under mask f to array C
```

Fig. 9. C and VIRAM assembly code for a simplified version of the chroma-key kernel.

vide support for vectorizing loops that contain conditional statements, such as those resulting from if statements included in the loop body. Fig. 9 presents a simplified version of the loop for the chroma-key algorithm, which merges two images based on a threshold value. Due to the conditional statement in the loop, each element of A has to be examined separately, and therefore the loop cannot be vectorized using regular vector arithmetic and memory operations.

In order to vectorize loops with conditional statements, VIRAM supports conditional (predicated) execution [21] for vector arithmetic and load-store instructions. The architecture defines 16 vector flag registers, each with a single bit element (mask) for every element in a regular vector register. Virtually all vector instructions operate under mask control with a flag register as a source operand. The mask bits are used to specify if the corresponding element operations should be executed or not. The elements of a vector flag register can be produced by loading them directly from memory or with vector comparison and logical operations. This allows creating proper mask bits even in the case of nested conditional statements. Due to practical restrictions with instruction encoding, a single bit per instruction is used to select between two vector flag registers for conditional execution. The remaining flag registers can be utilized using additional instructions, which perform logical and move operations on flag registers. Fig. 9 shows how the chroma-key loop is vectorized using conditional execution of vector stores. The first store transfers to array C the elements of A that are less than the threshold, while the second one fills the remaining elements of C with values from array B.

Masked execution of vector instructions is not the only way to vectorize conditional statements. Several alternatives exist that vary in execution efficiency and in hardware complexity [22]. They are based on compressing vectors in order to collect all the elements for which the conditional holds into contiguous elements of a register. The compression can be implemented with register-to-register (compress-expand) or indexed memory operations (see Section V). The VIRAM architecture supports all alternatives. The compiler selects between the strategies for vectorizing conditional statements based on a performance model developed for the Cray machine, which also supports all the alternatives. Masked execution of vector instructions is most commonly used, especially for loops that have only a single conditional.

To implement masked operations, the elements of vector flag registers are distributed across lanes just like the elements of regular vector registers. Element operations with a mask bit set to zero can either be skipped or simply ignored by suppressing any register file or memory updates they generate. Skipping operations can potentially reduce the execution time of a vector instruction but complicates multilane implementations, where it is desirable to have identical control for all vector lanes. The VIRAM implementation built at UC Berkeley uses the simpler approach of ignoring updates. This makes conditional execution trivial to implement, but leads to nonoptimal performance for cases where a sparsely populated vector flag register is used.

In addition to conditional execution, the flag registers in VIRAM are used to support software speculation of vector loads and to handle arithmetic exceptions.

V. Memory System Architecture

Most multimedia applications process large, streaming data sets with often limited temporal locality [2]. Therefore, the memory system performance is critical for a media processor. It must be able to provide high memory bandwidth, while tolerating long latencies. On the other hand,

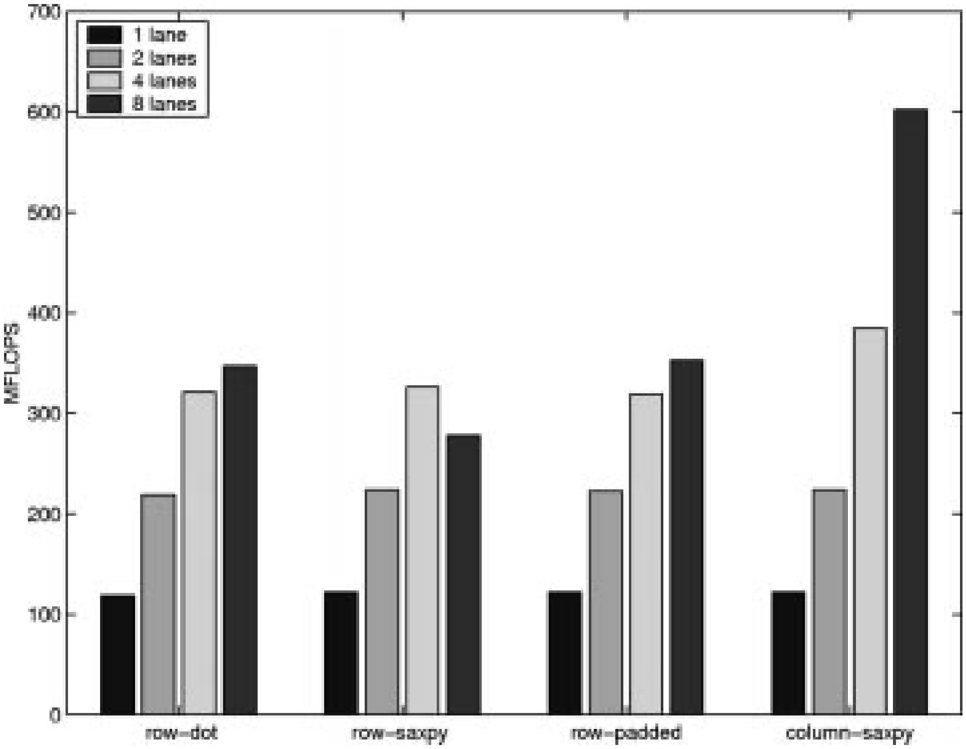


Fig. 10. Performance in Mflops for the different versions of the matrix vector multiplication routine.

the memory capacity required for embedded applications is typically much smaller than that of desktop and server systems. The VIRAM architecture supports a high-bandwidth, integrated memory system by combining vector load-store instructions with embedded DRAM technology.

A. Instruction Set and Compiler Support for Memory Accesses

As a complete vector architecture, VIRAM supports three types of vector load and store instructions: sequential, strided, and indexed. A sequential load will fetch into a vector register a set of data from consecutive memory locations. A strided load will fetch data from memory locations separated by a constant distance (*stride*). An indexed load uses the elements in a vector register as pointers to memory locations with the data to be fetched. The three types of store instructions operate in similar ways. The width of data in memory can be narrower than that of elements in vector registers (VPW) and there is no requirement for the data to be aligned to a memory address that is multiple of its total vector width.

Each vector memory instruction specifies a set of data accesses for vector elements. A large number of element transfers can be executed in parallel in order to utilize high memory bandwidth. The latency for initializing a stream of transfers can be amortized over all the elements fetched by the instruction. Hence, long memory latencies can be tolerated, as long as high bandwidth is available. Furthermore, each memory instruction explicitly specifies its access pattern (sequential, strided, indexed), thus effective prefetching techniques can be utilized.

The use and differences in performance for the three memory access modes can be demonstrated using the matrix-vector multiplication routine (MVM)

```
for $(\mathrm{i}=0;\ \mathrm{i}<\mathrm{N};\ \mathrm{i}++)$
    for $(\mathrm{j}=0;\ \mathrm{j}<\mathrm{N};\ \mathrm{j}++)$
        $\mathrm{Y[i]}+=\mathrm{M[i][j]}*\mathrm{V[j]}.$
```

By vectorizing the inner loop (j loop), we can execute this code as a loop of dot products. Every dot product calculates a single element of the result vector $Y[i]$. Alternatively, we can vectorize the outer loop (i loop), in which case the execution proceeds as a loop of saxpy kernels on the columns of the matrix M. Assuming the matrix is laid out in memory by rows, then the dot product vectorization will require sequential vector loads, while the saxpy one will use strided accesses. The bandwidth for strided accesses is typically lower than that for sequential. Depending on the stride value, multiple addresses may be required to access the data. Even if the memory system can accept multiple addresses per cycle, they may lead to conflicts that reduce the effective bandwidth [23]. On the other hand, the dot product version performs reductions which may not be able to utilize all vector lanes during their last stages. This is only an issue for short reductions, where the cost of the last stages cannot be efficiently amortized.

Fig. 10 presents the performance in Mflops for double-precision MVM on a 64×64 matrix for a 200-MHz VIRAM processor with one vector functional unit for floating-point operations. The first group of bars is for a dot-product implementation with a row-major matrix layout, which uses unit stride accesses. Not surprisingly, with only 64 elements per vector, efficiency is somewhat low due the percentage of time spent on reductions of short vectors. The second group shows the performance of the saxpy version, where the accesses to the matrix with a stride of 64 (matrix dimension). Padding the rows of the array with an extra element, leads to a prime stride of 65 that reduces the frequency of memory conflicts. The performance for this case is shown with the third group of bars. Finally, we can use a column-major layout for the matrix. In this case, vectorizing the outer loop (i loop) leads to a saxpy version with unit stride accesses. This produces the best performance, as there are no memory conflicts and no overhead for reductions. Note that while is may be attractive to consider only using the column-based layout for this routine, there are other common multimedia kernels which have the opposite design constraints, so a row-major layout is preferred for them.

Indexed load and store accesses can be used to vectorize a sparse version of matrix vector multiplication. In a sparse matrix representation, only the nonzero elements in each row are actually stored, along with the corresponding indices for the column. Hence, the loop for this kernel includes the statement

$$Y[i]+=M[j]*V[\mathrm{matrix\_index[j]}].$$

In order to use a dot-product implementation like the one discussed for the dense case, one has to load a set of indices using a sequential load, and then use them as a set of pointers for an indexed load from the source vector V.

With the proper hardware support, indexed vector accesses can be executed at a very fast rate. This includes support for high address generation and translation throughput, and high memory bandwidth for random accesses. A four-lane, 200-MHz VIRAM implementation with the memory system described in Section V-B, can sustain 511 Mflops for sparse MVM on a 10 000 × 10 000 matrix with 178 000 nonzero elements. This throughput is more than an order of magnitude better than the 35 Mflops sustained with a 500-MHz Pentium III superscalar processor with a cache-based memory system. The ability of the VIRAM hardware to issue in parallel and overlap multiple element accesses for a vector indexed load, leads to the significant performance advantage.

B. Embedded DRAM Memory System

For fast execution of load and store instructions, a vector processor requires a memory system that provides high sequential and random access bandwidth. For vector supercomputers, this is typically achieved using SRAM memory or a multichip DRAM system with multiple, wide, high-frequency interfaces. The cost and power consumption of these approaches renders them inappropriate for use with processors for embedded systems.

The memory system for VIRAM is based on embedded DRAM technology (*eDRAM*). Embedded DRAM allows the use of the increasing die densities in order to integrate the logic necessary for a microprocessor with high-capacity DRAM memory [24]. There are several fundamental advantages to this approach. First, a wide, high-frequency interface between the processor and memory becomes economical, because all components are integrated on a single die. The power consumption of the memory system is also reduced, as most memory accesses can be processed on-chip, without driving high capacitance board traces [25]. Embedded DRAM is at least five times denser than SRAM. One can integrate a processor with a DRAM memory system large enough to be used as the main memory, not a cache, for embedded applications such as PDAs, cellular phones, and video cameras. This leads in practice to system-on-a-chip integration. Finally, the use of trench capacitor cells allows mixing DRAM memory with a microprocessor, without a reduction in the speed of the digital circuitry [24].

In order to maximize the bandwidth available for indexed and strided accesses, an eDRAM memory system is organized as multiple, independent memory banks [26]. Each bank has a separate wide interface and a controller that allows initiating one memory access per cycle. A collection of banks can be connected to the vector processor using a high-bandwidth, on-chip, switched fabric instead of a simple bus. Random access bandwidth can be further improved by exploiting the hierarchical structure of eDRAM banks. Each one consists of a number of sub-banks, connected to the bank interface through a shared data and address bus. This technique was introduced in order to reduce the memory access latency, but it can also be used to overlap in time accesses to different sub-banks in a pipelined fashion [27]. Random bandwidth is particularly important to applications with strided and indexed loads and stores, where bank conflicts between element accesses can significantly hurt the overall performance. With DRAM reads and writes taking multiple processor cycles to complete, an access occupies a sub-bank for several cycles, which makes stalls due to conflicts more expensive. Fig. 11 presents the effect of the number of sub-banks on the performance of a four-lane VIRAM implementation with eight DRAM banks for the inverse discrete cosine transform (IDCT). This kernel performs the transformation on every 8 × 8 sub-block of a digital image and utilizes strided loads. Using four sub-banks reduces significantly the frequency of memory conflicts and their corresponding stall cycles, leading to 80% performance improvement over the single sub-bank organization.

The multicycle delay and bursty behavior of DRAM introduce an interesting challenge in designing the load-store unit for the vector processor. In traditional microprocessor systems, the off-chip DRAM memory is hidden behind multiple levels of SRAM-based cache memory. But a vector processor is able to tolerate latency if high bandwidth is available. Embedded DRAM offers both high bandwidth and significantly lower latency than off-chip DRAM. It is, therefore, reasonable to organize the load-store unit so that vector accesses are directly served by DRAM and no die area is occupied for SRAM caches, which are not always the most efficient structure for streaming data.

The bursty behavior of eDRAM can be tolerated using a decoupled organization, where the load-store unit uses data and address queues to hide the variability in access time from the other vector functional units [28]. A simpler alternative, which saves the area and power consumption of the decoupling queues, is the *delayed pipeline* presented in

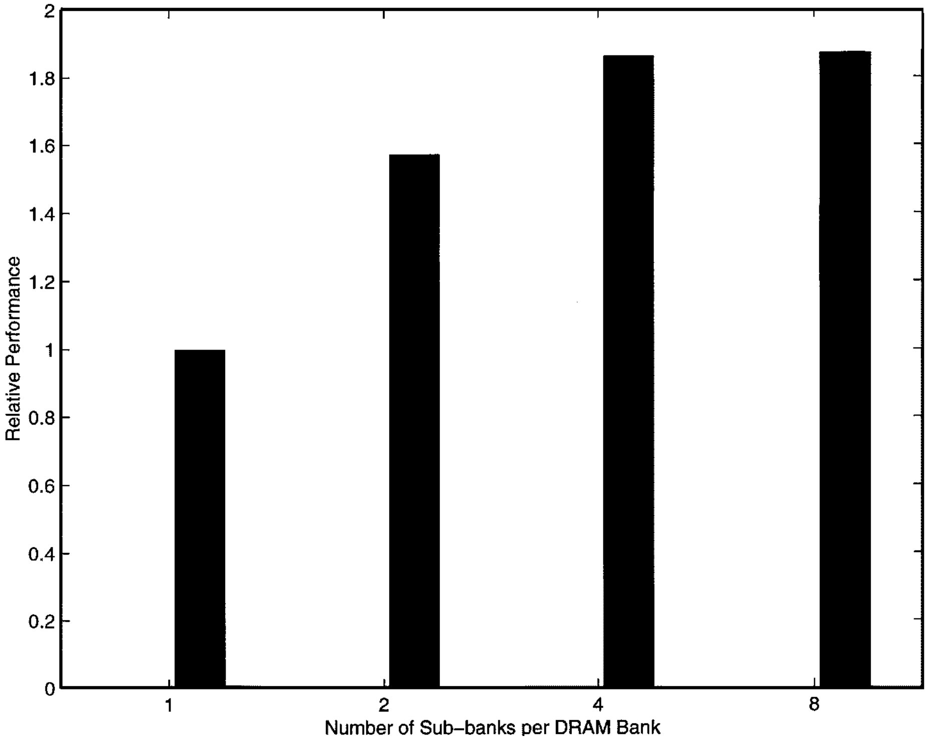


Fig. 11. Effect of the number of sub-banks per DRAM bank on the performance of a four-lane VIRAM processor for the IDCT kernel.

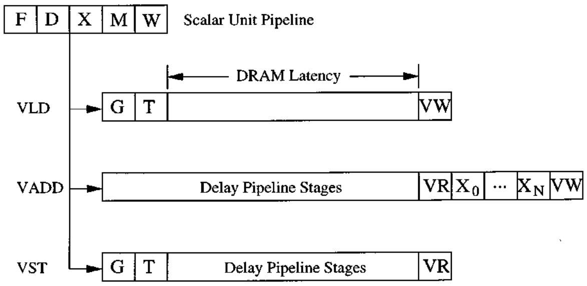


Fig. 12. Delayed pipeline structure for a vector load (vld), store (vst), and add (add). The store and add pipelines have been padded with idle stages in order to align their stage for reading source operands (VR) with that for writing the result (VW) for the load operation that observes the maximum DRAM access latency.

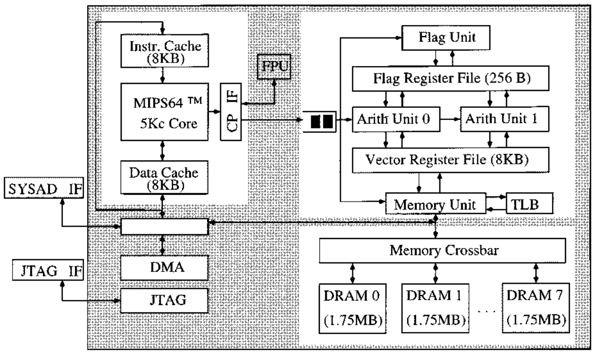


Fig. 13. Block diagram of the VIRAM-1 processor. Its three basic components are the MIPS scalar core, the vector coprocessor, and the embedded DRAM memory system.

Fig. 12. This organization treats DRAM reads and writes as if they always require a row access, which leads to the maximum latency. But this latency is pipelined for both loads and stores. In addition, the pipeline for vector arithmetic operations is also padded with sufficient idle (delay) stages to match the length of the load pipeline. Hence, when a vector add is issued a cycle after the vector load that fetches one of its sources, it can proceed down the corresponding pipeline, without any stall cycles due to the read-after-write dependency and the long DRAM accesses latency. All the VIRAM performance results presented in this paper assume the use of a delayed pipeline, which demonstrates its effectiveness for a wide range of cases. Note that while the delayed pipeline assumes that the time for a row access is always needed, the control signals sent to the memory banks will only initiate the portions of a DRAM access that are actually necessary, depending on whether the requested data are already available at the sense-amplifiers of the corresponding sub-bank. This optimization reduces the power consumed for memory accesses.

VI. Putting it all Together: The VIRAM-1 Processor

To demonstrate in practice the advantages of the architecture and techniques presented in the previous sections, we developed the VIRAM-1 prototype microprocessor [29]. The instruction set for this design is based on the MIPS architecture [30] with the vector instructions implemented as a coprocessor extension. The MIPS architecture was chosen for a number practical reasons and only minor changes would be required to use our vector architecture as a coprocessor to other RISC architectures such as ARM or PowerPC.

Fig. 13 presents the block diagram of the processor. The MIPS core is a single-issue, in-order, 64-bit design with

Table 1
VIRAM-1 Processor Design Statistics

Technology	IBM 0.18μm CMOS process 6 layers copper, deep trench DRAM cell
Area	$270mm^2$
Devices	130 million transistors
Frequency	200 MHz
Power Supply	1.2V (processor), 1.8V (DRAM)
Power Consumption	2 Watts
Integer Performance	1.6/3.2/6.4 Gop/s (64-,32-,16-bit data) 3.2/6.4/12.8 Gop/s (with multiply-add)
Floating-point Performance	1.6 Gflops (32-bit only)

8-kB first-level instruction and data caches. The vector coprocessor contains an 8-kB, multiported, vector register file. Each of the 32 vector registers can store 32 64-bit elements, 64 32-bit elements, or 128 16-bit elements. There are two functional units for arithmetic operations, with only one of them able to perform single-precision floating-point operations due to area constraints. Each functional unit has four 64-bit datapaths and the whole coprocessor is organized in four vector lanes.

The memory system consists of eight embedded DRAM banks with a total capacity of 14 MB. It serves as main memory for both the vector coprocessor and the MIPS core. Each bank has a 256-bit synchronous, pipelined interface and latency of 25 ns row accesses. The interface presents each bank as a single sub-bank structure and does not allow any overlapping of accesses internally. This restriction of the eDRAM banks available to us is not fundamental to the architecture. It has no performance impact on sequential access pattern, but it can limit the sustained performance when strided or indexed accesses are used (see Fig. 11). A crossbar switch with aggregate bandwidth of 12.8 GB/s connects the memory system to the scalar core and the vector coprocessor. The crossbar also connects to a two channel DMA engine which allows communication with external devices or memory over a system bus. Similarly to the on-chip DRAM, any off-chip memory is also considered main memory and its use is controlled by software.

The vector memory unit serves as the interface between the coprocessor and eDRAM, as there is no SRAM cache for the vector load and store accesses. It implements the delayed pipeline with 15 pipeline stages. It is able to exchange up to 256 bits per cycle with the memory system. Four address generators can be used to produce up to four addresses per cycle for indexed and strided accesses. Unlike traditional vector designs, VIRAM-1 includes a multiported, two-level TLB that supports virtual memory and restartable page fault exceptions for vector memory accesses. Overall, the memory unit can sustain up to 64 addresses pending to the memory system at any time.

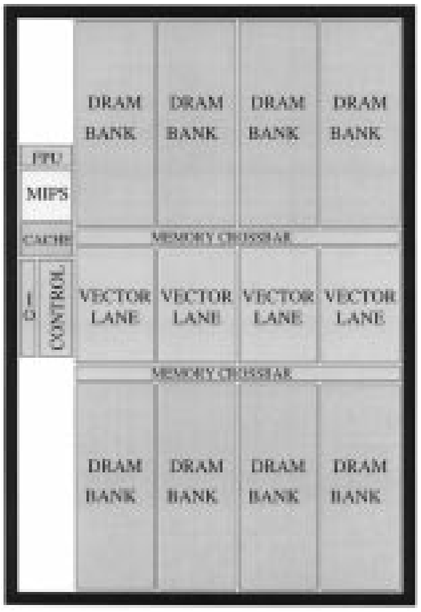


Fig. 14. Floorplan of the VIRAM-1 vector processor (drawn to scale).

Table 1 summarizes the design statistics of VIRAM-1. The clock frequency goal is set to merely 200 MHz to allow for a low power supply and reduced power consumption. Despite this, the performance is high due to the parallel execution of element operations on the four vector lanes. The result is 1.6 billion operations per second per watt for 32-bit integer numbers, with power/performance efficiency being higher for operations on narrower data types or when multiply-add instructions can be used.

Fig. 14 presents the floorplan for VIRAM-1. The die area is 15 mm × 18 mm, while the transistor count is approximately 130 million. The design is particularly modular as it consists of replicated vector lanes and DRAM banks. Modularity is a major advantage both for reducing design complexity and providing a straightforward path for scaling. This key property has allowed the development of such a large design by a team of six graduate students. Alternative designs can be easily produced by properly selecting the mix of lanes and banks with minimal control changes necessary. Another interesting property of VIRAM-1 is that the majority of its die area is used for vector registers, execution datapaths, and DRAM main memory. This area utilization is in contrast with superscalar and VLIW designs where an increasing percentage of the die area is used for SRAM-based caches.

Fig. 15 compares the performance of VIRAM-1 to a number of desktop and embedded processors such as the PowerPC G3 and G4, the Intel Pentium III, and the Mitsubishi M32R/D. The three kernels used are corner turn, coherent sidelobe canceler, and beam steering, which perform signal and image processing for radar applications. Performance is presented as speedup over the PowerPC G3 design running at 400 MHz. Despite its lower clock frequency, VIRAM-1 outperforms the other architectures by an order of magnitude by exploiting parallel execution on vector lanes and the rich memory system resources available.

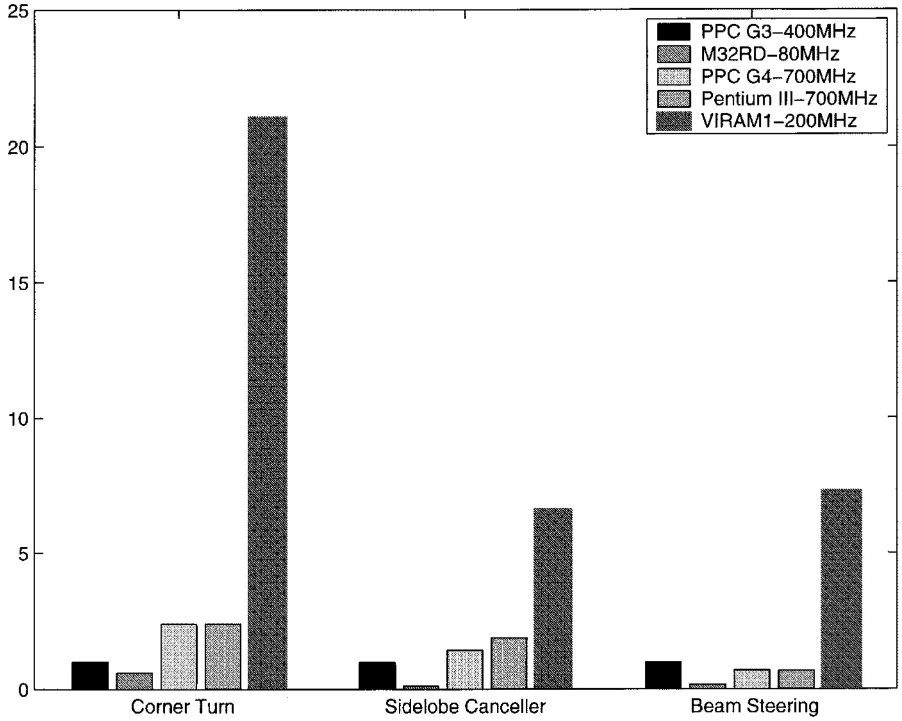


Fig. 15. Performance of VIRAM-1 and a set of superscalar processors for three media kernels for radar applications. Performance is expressed as speedup over the PowerPC G3 processor.

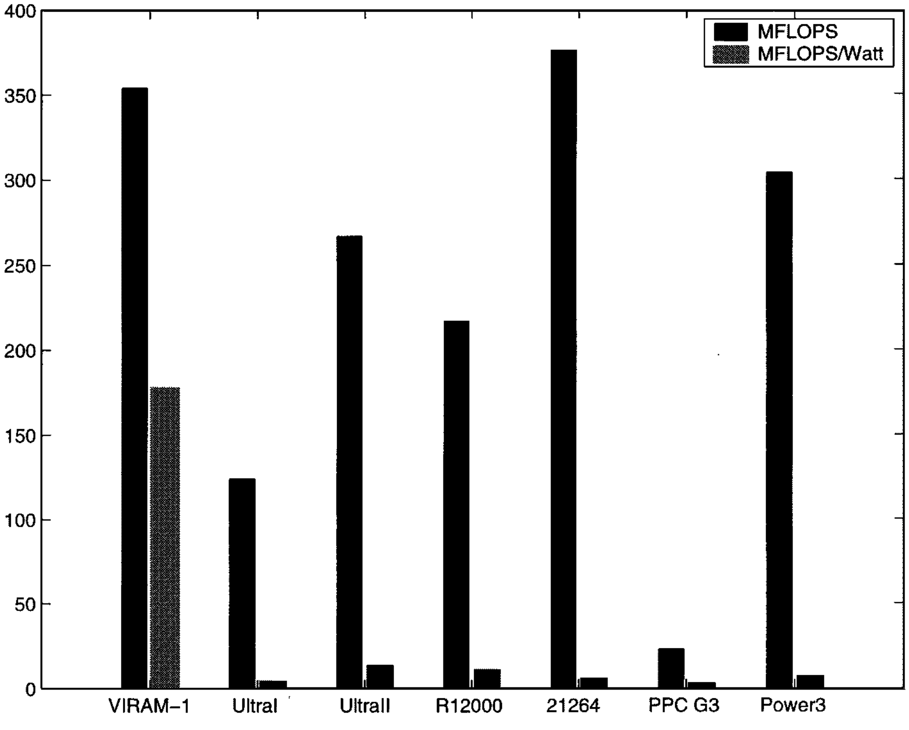


Fig. 16. Performance and power/performance efficiency of VIRAM-1 and a set of high-end superscalar processors for the matrix-vector multiplication routine.

Fig. 16 compares VIRAM-1 to high-end desktop and server superscalar processors for a 100 × 100 matrix-vector multiplication kernel. The other architecture are the Sun Ultra-Sparc I and II, the MIPS R12000, the Alpha 21264, the PowerPC G3, and the IBM Power3–630. Performance, reported in sustained Mflops or Mflops per watt, is achieved with compiled code for VIRAM and with vendor optimized BLAS routines written in assembly for the other processors [31]. Focusing on the bars representing Mflops, one can notice that VIRAM-1 is competitive with the most aggressive superscalar design today, the Alpha 21264, while it clearly outperforms the remaining architectures. If a larger matrix is used, all the superscalar designs will experience a performance reduction due to caching effects. The performance for VIRAM-1 will not be affected, since vector accesses are directly served by on-chip DRAM main memory. When power/performance efficiency is considered (Mflops/watt), the superiority of VIRAM-1 is obvious. It is able to exploit the vector architecture and the eDRAM memory system to achieve both high performance and low power consumption for this highly data parallel kernel. On the other hand, the superscalar architectures rely on parallel instruction issue, hardware speculation, and increased clock frequency. All these techniques lead to significant power consumption between 20 and 80 W, depending on the specific design.

VII. Conclusion

As Moore's Law continues to apply for semiconductor technology, the question is not only how to harness its potential power, but also for what applications. Recent market and device use trends show that embedded and portable systems running multimedia applications such as speech recognition or image processing create an application domain of great importance and large potential growth. While microprocessors for desktop PCs already seem to be fast enough to run office productivity software, embedded applications still require significant advances in the performance, power consumption, and complexity efficiency of general purpose processors. Improvements are not only necessary in order to add features to existing systems, such as improved graphics to a game console, but also to enable new embedded applications such as PDAs with speech recognition.

Vector IRAM demonstrates an approach to riding Moore's Law for multimedia applications on embedded systems. It uses a vector architecture to match the data parallelism of media tasks and embedded DRAM technology to provide a high-bandwidth, integrated memory system. The explicit representation of data parallelism in vector instruction enables VIRAM to be efficient for all four important metrics: performance, power consumption, design complexity, and scalability. It also allows tradeoffs between hardware resources and performance or hardware resources and power consumption. This flexibility is desirable because the cost of additional hardware resources (vector lanes) is automatically reduced with every generation of CMOS technology. Moreover, the VIRAM architecture lends itself to compiling from high-level programming languages with high efficiency, both in terms of the achieved performance and the executable code size.

References

[1] T. Lewis, "Information appliances: Gadget netopia," *IEEE Comput.*, vol. 31, pp. 59–66, Jan. 1998.

[2] K. Diefendorff and P. Dubey, "How multimedia workloads will change processor design," *IEEE Comput.*, vol. 30, pp. 43–45, Sept. 1997.

[3] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proc. IEEE*, vol. 83, pp. 1609–24, Dec. 1995.

[4] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Quantifying the complexity of superscalar processors," Univ. Wisconsin-Madison, Tech. Rep. CS-TR-1996–1328, 1996.

[5] J. A. Fischer, "Very long instruction word architectures and ELI-512," in *Proc. 10th Int. Symp. Computer Architecture*, Stockhold, Sweden, June 1983.

[6] M. Schlansker, T. M. Conte, J. Dehnert, K. Ebcioglu, J. Z. Fang, and C. L. Thompson, "Compilers for instruction-level parallelism," *IEEE Comput.*, vol. 30, pp. 63–69, Dec. 1997.

[7] R. Russel, "The Cray-1 computer system," *Commun. ACM*, vol. 21, pp. 63–72, Jan. 1978.

[8] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1996.

[9] R. W. Brodersen, A. Chandrakasan, and S. Sheng, "Design techniques for portable systems," in *Dig. Tech. Papers Int. Solid-State Circuits Conf.*, San Francisco, CA, Feb. 1993.

[10] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE J. Solid State Circuits*, vol. 27, pp. 473–484, Apr. 1992.

[11] K. Asanovic, "Vector microprocessors," Ph.D. dissertation, Computer Science Division, Univ. California, Berkeley, 1998.

[12] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, pp. 490–504, Apr. 2001.

[13] *Cray Standard C and C++ Reference Manual (004-2179-00)*, Cray Research Inc., 2000.

[14] S. L. Graham, D. F. Bacon, and O. J. Sharp, "Compiler transformations for high performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, 1994.

[15] J. M. Anderson, S. P. Amarasignhe, and M. S. Lam, "Data and computation transformations for multiprocessors," in *Proc. 5th Symp. Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1997.

[16] C. Lee and M. Stoodley, "Simple vector microprocessor for multimedia processing," in *Proc. 31st Int. Symp. Microarchitecture*, Dallas, TX, Dec. 1998.

[17] A. Peleg and U. Weiser, "MMX technology extension to the Intel architecture," *IEEE Micro*, vol. 16, pp. 42–50, Aug. 1996.

[18] M. Phillip, "A second generation SIMD microprocessor architecture," in *Conf. Rec. Hot Chips X Symp.*, Palo Alto, CA, Aug. 1998.

[19] D. Judd, K. Yelick, C. Kozyrakis, D. Martin, and D. Patterson, "Exploiting on-chip memory bandwidth in the VIRAM compiler," in *Proc. 2nd Workshop Intelligent Memory Systems*, Nov. 2000.

[20] R. Thomas, "An architectural performance study of the Fast Fourier Transform on VIRAM," Univ. California, Berkeley, Tech. Rep. UCB//CSD-99–1106, 2000.

[21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the Hyperblock," in *Proc. 25th Int. Symp. Microarchitecture*, Dec. 1992, pp. 45–54.

[22] J. E. Smith, G. Faanes, and R. Sugumar, "Vector instruction set support for conditional operations," in *Proc. 27th Int. Symp. Computer Architecture*, Vancouver, BC , Canada, June 2000.

[23] G. Sohi, "High-bandwidth interleaved memories for vector processors — A simulation study," *IEEE Trans. Comput.*, vol. 42, pp. 34–44, Jan. 1993.

[24] I. S. Subramanian and H. L. Kalter, "Embedded DRAM technology: Opportunities and challenges," *IEEE Spectr.*, vol. 36, pp. 56–64, Apr. 1999.

[25] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, P. Patterson, T. Anderson, and K. Yelick, "The energy efficiency of IRAM architectures," in *Proc. 24th Int. Symp. Computer Architecture*, Denver, CO, June 1997.

[26] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, "A case for intelligent DRAM: IRAM," *IEEE Micro*, vol. 17, pp. 34–44, Apr. 1997.

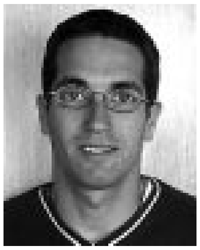
[27] T. Yamauchi, L. Hammond, and K. Olukotun, "The hierarchical multi-bank DRAM: A high-performance architecture for memory integrated with processors," in *Proc. 17th Conf. Advanced Research in VLSI*, Ann Arbor, MI , Sept. 1997.

[28] R. Espasa and M. Valero, "Decoupled vector architecture," in *Proc. 2nd Int. Symp. High-Performance Computer Architecture*, San Jose, CA, Feb. 1996.

[29] C. E. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick, "VIRAM: A media-oriented vector processor with embedded DRAM," in *Conf. Rec. Hot Chips XII Symp.*, Palo Alto, CA, Aug. 2000.

[30] J. Heinrich, *MIPS RISC Architecture*, 2nd ed: Silicon Graphics, Inc, 1998.

[31] J. J. Dongarra, J. D. Croz, S. Hamarling, and R. J. Hanson, "An extended set of Fortran basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 14, pp. 18–32, Mar. 1988.



Christoforos Kozyrakis (Student Member, IEEE) received the B.S. and M.S. degrees in computer science from the University of Crete, Greece, and the University of California, Berkeley (UCB), respectively.

He is a Ph.D. candidate in computer science at UCB. His current research interests include scalable processor architectures, memory hierarchies, multimedia systems, and digital systems design.

Mr. Kozyrakis is the recipient of a Ph.D. Research Fellowship Award by IBM.

David Judd received the B.S. from Stanford University in 1966 and the M.S. degree from the University of Minnesota in 1969.

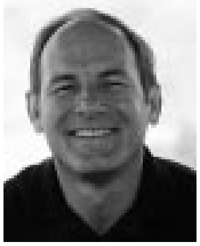
He has 32 years of software development experience on RISC processors at Control Data Corp. (CDC 7600), Cray Research Inc. (Cray-1/2/3, Cray C90/T90, Cray T3D/T3E), and SGI. He was involved with operating system development (SCOPE-2, COS, UNICOS) and compiler development (Cray CF90, C/C++).

Joseph Gebis (Student Member, IEEE) received the B.S. degree in computer engineering from the University of Illinois at Urbana-Champaign.

He is pursuing the Ph.D. degree in computer science from the University of California, Berkeley. His research interests are in computer architecture and circuits design.

Samuel Williams received the B.S. degrees in electrical engineering, applied mathematics, and physics from the Southern Methodist University.

He is pursuing the Ph.D. degree in computer science at the University of California, Berkeley.



David Patterson (Fellow, IEEE) joined the faculty at the University of California, Berkeley, in 1977, where he now holds the Pardee Chair of Computer Science. He led the design and implementation of RISC I, likely the first VLSI Reduced Instruction Set Computer. This research became the foundation of the SPARC architecture, used by Sun Microsystems and others. He was a leader, along with Randy Katz, of the Redundant Arrays of Inexpensive Disks project (RAID), which led to reliable storage systems from many companies. He is co-author of five books, including two with John Hennessy, who is now President of Stanford University. Patterson has been chair of the Computer Science Division at Berkeley, the ACM Special Interest Group in computer architecture, and the Computing Research Association.

His teaching has been honored by the ACM, the IEEE, and the University of California. Patterson shared the 1999 IEEE Reynold Johnson Information Storage Award with Randy Katz for the development of RAID and shared the 2000 IEEE von Neumann medal with John Hennessy. He is a member of the National Academy of Engineering and a Fellow of the ACM.

Katherine Yelick (Member, IEEE) received the Ph.D. degree from the Massachusetts Institute of Technology (MIT), Cambridge, MA, in 1991, where she worked on parallel programming methods and automatic theorem proving.

She is an Associate Professor in the EECS Department at the University of California, Berkeley. Her research in high performance computing addresses parallel data structure design, compiler analyses for explicitly parallel code, and optimization techniques for communication systems and memory systems. Her projects include the Multipol distributed data structure library and the Split-C, Titanium, and UPC parallel languages.

Dr. Yelick won the Geroge M. Sprowls Award for an outstanding Ph.D. dissertation at MIT and has received teaching awards from the EECS Departments at both MIT and Berkeley.