Appears in the 36^{th} Annual International Symposium on Microarchitecture

Universal Mechanisms for Data-Parallel Architectures

Karthikeyan Sankaralingam Stephen W. Keckler William R. Mark Doug Burger

Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu

Abstract

Data-parallel programs are both growing in importance and increasing in diversity, resulting in specialized processors targeted at specific classes of these programs. This paper presents a classification scheme for data-parallel program attributes, and proposes micro-architectural mechanisms to support applications with diverse behavior using a single reconfigurable architecture. We focus on the following four broad kinds of data-parallel programs — DSP/multimedia, scientific, networking, and real-time graphics workloads. While all of these programs exhibit high computational intensity, coarse-grain regular control behavior, and some regular memory access behavior, they show wide variance in the computation requirements, fine grain control behavior, and the frequency of other types of memory accesses. Based on this study of application attributes, this paper proposes a set of general micro-architectural mechanisms that enable a baseline architecture to be dynamically tailored to the demands of a particular application. These mechanisms provide efficient execution across a spectrum of data-parallel applications and can be applied to diverse architectures ranging from vector cores to conventional superscalar cores. Our results using a baseline TRIPS processor show that the configurability of the architecture to the application demands provides harmonic mean performance improvement of 5%–55% over scalable yet less flexible architectures, and performs competitively against specialized architectures.

1. Introduction

Data-parallel programs are growing in importance, increasing in diversity, and demanding increased performance from hardware. Specialized hardware is commonplace in the real-time graphics, signal processing, network processing, and high-performance scientific computing domains. Modern graphics processors can sustain as high as 20 GFlops (at 450Mhz) on programmable hardware [6], which suggests they have at least forty 32-bit floating point units. Software radios for 3G wireless baseband receivers are being developed for digital signal processors and require 15 Gops to deliver adequate performance [29]. Each arithmetic processor in the Earth Simulator contains forty eight vector pipelines and delivers peak performance of up to 8 GFlops. While these domains of data-parallel applications have many common characteristics, they typically show differences in the types of memory accesses, computation requirements, and control behavior.

Most data-parallel architectures target a subset of data-parallel programs, and have poor support for applications outside of that subset. Vector architectures provide efficient execution for programs with mostly regular memory accesses and simple control behavior. However, the vector model is less effective on programs that require computation across multiple vector elements or access memory in an unstructured or irregular fashion. SIMD architectures provide support for communication between execution units (thereby enabling computation across multiple data elements), but are also globally synchronized and hence provide poor support for applications with conditional execution and data dependent branches. MIMD architectures have typically been constructed of coarse-grain processors and operate on larger chunks of data using the single-program, multiple data (SPMD) execution model, with poor support for fine-grain synchronization. Emerging applications, such as real-time graphics, exhibit control behavior that requires fine grain MIMD execution and fine-grain communication among execution units.

Many data-parallel applications which consist of components that exhibit different characteristics are often implemented on specialized hardware units. For example, most real-time graphics processing hardware use specialized hardware coupled with the programmable components for *MPEG4* decoding. The TMS320C6416 DSP chip integrates two specialized units targeted at con-

volution encoding and forward error correction processing. While many of these specialized accelerators have been dedicated to a single narrow function, architectures are now emerging that consist of multiple programmable data-parallel processors that are specialized in different ways. The Sony Emotion Engine includes two specialized vector units–one tuned for geometry processing in graphics rendering and the other specialized for behavioral and physical simulation [19]. The recently announced Sony Handheld Engine integrates a DSP core, a 2D graphics core and an ARM RISC core on a single chip, each targeted at a distinct type of data-parallel computation.

Integrating many such specialized cores leads to increased design cost and area, since different types of processors must be designed and integrated together. Our goal is to determine an underlying set of mechanisms that can be combined in different ways to tailor a data parallel architecture based on application demands. In this paper, we identify and characterize the application demands of different data parallel program classes. While these classes have some common attributes, namely high computational intensity and high memory bandwidth, we show that they also have important differences in their memory access behavior, instruction control behavior and instruction storage requirements. As a result, different applications can demand different hardware capabilities varying from simple enhancements, like efficient lookup tables, to different execution models, such as SIMD or MIMD.

Based on the program attributes identified, we propose a set of general microarchitectural mechanisms for augmenting the memory system, instruction control, and execution core to build a flexible data-parallel architecture. We show that these mechanisms can be combined together in different ways to dynamically adapt the architecture, providing support for a broad spectrum of data-parallel applications. While the mechanisms are universal, since they support each type of DLP behavior determined in our characterization of the application space, and can be applied to diverse architectures ranging from vector processors to superscalar processors, this paper uses the TRIPS architecture [32] as a baseline for performance evaluation. We also show a rough comparison of the performance of these mechanisms to current best-of-breed specialized processors across the application domain space.

The TRIPS processor is well suited for data-parallel execution with its high functional unit density, efficient ALU-ALU communication, high memory bandwidth, and technology scalability. The dataflow style ISA design provides several relevant capabilities, including the ability to map various communication patterns and statically placed dynamically issued execution, that enable a straight-forward implementation of the mechanisms. No major changes to the ISA or programming model is required. Furthermore, since the execution core provides local operand storage at the ALUs and distributed control, the only major modifications required to integrate the mechanisms are the addition of a few extra hardware registers and status bits to maintain local state, and storage tables where necessary. In our previous work we proposed and evaluated one configuration (named S-morph) of the TRIPS processor targeted at data level parallelism (DLP).

The remainder of this paper is organized as follows. Section 2 discusses the behavior and attributes of different classes of data-parallel applications. Section 3 reviews the classic data-parallel architectures and mapping of applications to architectures. Section 4 describes the microarchitectural mechanisms for supporting data-parallel execution. Section 5 evaluates the performance improvements provided by these mechanisms and compares this performance to specialized architectures. Section 6 discusses related work and Section 7 concludes.

2. Application Behavior

Data-parallel workloads can be classified into domains based on the type of data being processed. The nature of computation varies within a domain and across the different domains. The applications vary from simple computations on image data converting one color space to another (comprising 10s of instructions), to complex encryption routines on network packets (comprising 100s of instructions). Four broad categories cover a significant part of this spectrum: digital signal processing, scientific, network/security, and real-time graphics. In this section, we first describe the behavior of these applications categorized by three parts of the architecture they affect: memory, instruction control, and execution core. We then describe our suite of data-parallel programs and present their attributes.

2.1. Program Attributes

At an abstract programming level, data-parallel programs consist of a loop body executing on different parts of the input data. In a data parallel architecture this loop body is typically executed on different execution units, operating on different parts of memory in parallel. We refer to this loop body as a *kernel*. Typically the iterations of a loop are independent of each other and can execute concurrently. Kernels exhibit different types of memory accesses and control behavior, as well as varying computation needs. One example of data-parallel execution is the computation of a 2D discrete cosine transform (DCT) on 8x8 blocks of an image. In this case, parallelism can be exploited by processing the different 8x8 blocks of the image on different computation nodes concurrently. The processing of each instance of the kernel is identical and can be performed in a globally synchronous manner across differ-

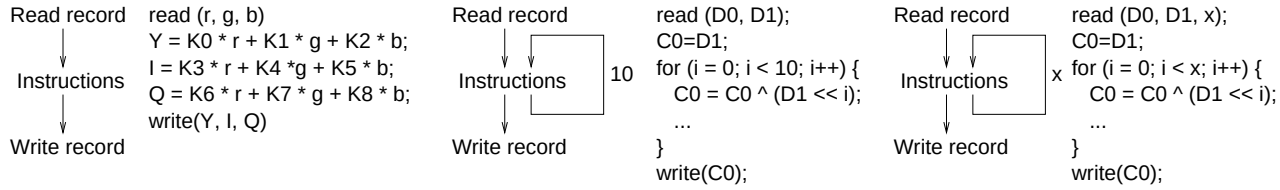

Figure 1. Kernel control behavior.

ent computation nodes. A more complex data-parallel computation is a technique called *skinning* which is used for animation in graphics processing. A dynamically varying number of matrix-vector multiplies are performed at each polygon vertex in a 3D model. The different vertices in the model can be operated upon in parallel, completely independent of each other, but the amount of computation varies from vertex to vertex.

2.1.1. Memory behavior The memory behavior of data-parallel applications can be classified into four different types: (1) regular memory accesses, (2) irregular memory accesses, (3) named constant scalar operands, and (4) indexed constant operands. In characterizing DLP programs we are interested in the frequency of occurrence of each of the four types of accesses in a kernel. The four types of accesses are not exclusive and a kernel may make accesses from all four categories.

- *Regular memory:* Data-parallel kernels typically read from memory in a very structured manner (strided accesses for example). We use the term *record* to refer to a group of elements on which a single iteration of a kernel operates. In image processing, for example, a record may consist of 3 elements, corresponding to 3 primary color components. Because of the regularity of these accesses, microarchitectures can pipeline accesses or amortize the address calculation and other overheads associated with accessing memory, by issuing one instruction to fetch one or more full records.
- *Irregular memory:* Some data-parallel kernels access some parts of memory in a random access fashion similar to conventional sequential programs. One example of such behavior is texture accesses in graphics programs. Unlike regular memory accesses, the overheads of these accesses cannot be amortized by aggregating them. Typical texture data structures for graphics scenes require several megabytes of storage.
- *Scalar constants:* Many operations in data parallel kernels use runtime constants that are unmodified through the full execution of the kernel, such as the constants used in convolution filters applied to an image. The number of coefficients is often small and can thus be stored in machine registers rather than memory.
- *Indexed constants:* Many DLP applications require small lookup tables with the index determined at runtime. Encryption kernels use such lookup tables with between 256 and 1024 8-bit entries to substitute one byte for another byte during computation. These accesses can be frequent in some kernels, reducing performance if they have long access latencies. Storing these tables in the level-1 data caches consumes little storage space, but tremendous cache bandwidth.

2.1.2. Control behavior: The complexity of the control structure in the kernel determines the type of synchronization and instruction sequencing required. Figure 1 shows the three different types of control behavior possible.

- *Sequential instructions:* The simplest kernels contain a sequence of instructions with no internal control flow. A degenerate case is a single vector operation, but the 2D DCT can be transformed into this model by unrolling all of the internal computations of the 8x8 kernel. Each iteration of these kernels executes in the exact same fashion, so these kernels are well-suited for vector or SIMD control. Figure 1a shows this type of control behavior with example RGB to YIQ color conversion kernel pseudo-code.
- *Simple static loops:* A slightly more complex type of control behavior occurs when the kernel contains loops with static loop bounds. Figure 1b shows this type of control behavior with an example encryption kernel pseudo-code. Like the simple instruction sequences, each iteration of the kernel is the same and can be executed in a vector or SIMD style. Such kernels can be unrolled at compile time increasing the code size of the kernel, although for some kernels this transformation results in prohibitively large instruction storage requirements. Architectures that lack any branching support (like some graphics fragment processors) must rely on complete unrolling to execute such loops.
- *Runtime loop bounds:* Figure 1c shows the most generic of control behavior: data dependent branching. Such kernels would require masking instructions to execute on vector and SIMD machines, and are ideally suited to fine-grain MIMD machines, since each processing element can be independently controlled according to the local branching behavior.

Benchmark	Description	Benchmark	Description
Multimedia processing		*Network processing, security (1500 byte packets)*	
convert	RGB to YIQ conversion.	MD5	MD5 checksum.
dct	A 2D DCT of an 8x8 image block.	Rijndael	Rijndael (AES) packet encryption.
highpassfilter	A 2D high pass filter.	Blowfish	Blowfish packet encryption.
Scientific codes			
FFT	1024-point complex FFT.		
LU Decomposition	LU decomposition of a dense 1024x1024 matrix.		
Real-time graphics processing. See [11].			
vertex-simple	Basic vertex lighting with ambient, diffuse, specular and emissive lighting.		
fragment-simple	Basic fragment lighting with ambient, diffuse, specular and emissive lighting.		
vertex-reflection	Vertex shader for a reflective surface.		
fragment-reflection	Fragment shader rendering a reflective surface using cube maps.		
vertex-skinning	A vertex shader used for animation with multiple transformation matrices.		
anisotropic-filtering	A fragment shader implementing anisotropic texture filtering [28].		

Table 1. Benchmark description.

Runtime conditionals, such as simple and nested `if-then-else` statements, can make any of these loop control templates more complex. Data-parallel architectures have traditionally implemented conditionals by using predication, conditional streams [15], or vector masks [33]. Finer partitioning of control, such as provided by a fine-grain MIMD architecture can reduce or eliminate these overheads that conditionals have in highly synchronized architectures.

2.2. Benchmark attributes

Table 1 describes a suite of DLP kernels selected from four major application domains. Table 2 characterizes these kernels according to the computation, memory and control criteria presented previously. The two computation columns list the number of instructions and inherent ILP within the kernel (ILP is the number of instructions in one iteration of a kernel, divided by the dataflow graph height; when the loop bound was variable, the kernel was completely unrolled). The first memory column lists the size of the record (in 64-bit words) that each kernel reads and writes, the second column gives the number of irregular memory accesses, and the third and fourth memory columns describe the use of static coefficients within the kernel and the size of the lookup table for indexed constants, if one is needed. The control column indicates the number of loop iterations within the kernel (if any) and whether the loop bounds are variable across kernel instances, in which case the kernels exhibit data dependent control and prefer a fine grain MIMD execution model. In the *anisotropic-filter* kernel, for example, the number of instructions executed varies from about 150 to 1000 for each instance. In vector or SIMD architectures, which lack support for fine grain branching, each instance would execute all 1000 instructions, using predication or other techniques for nullifying unwanted instructions. Collectively, the benchmarks exhibit wide variation in each of the attributes, demonstrating diversity in the fundamental behavior of DLP applications. We used this application study to drive an identification of attributes and complementary microarchitectural mechanisms.

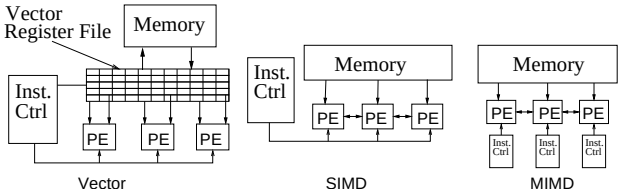

Figure 2. Vector, SIMD and MIMD architectures.

3. Classic data-parallel architectures

Traditionally, different DLP architectures were constructed for applications that exhibited different characteristics. This section gives a brief overview of the basic DLP architecture models and highlights their differences, as shown in Figure 2. These architectures differ principally in their implementations of instruction control and communication between memory and the ALUs. We use a running example of the 2D DCT to illustrate the differences between the architectures.

Vector: Vector architectures, examples of which include the Cray-1 [31], VectorIRAM [17], T0 [2], and Tarantula [9], use global control with a single instruction fetch and decode unit. A vector register file (VRF) serves as a staging area for values from memory into the ALUs and back, while a centralized control unit sequences the vector elements between the VRF and the ALUs. Vector architectures provide efficient regular memory access with the VRF, but without chaining hardware all communication between vector iterations must take place in the VRF. The global synchronization of the ALUs precludes data dependent branch control flow at the ALU level. On a vector machine, the 2D DCT is decomposed into a 1D DCT on the columns, a transposition in the VRF, and then a second 1D DCT on the rows.

SIMD: SIMD architectures, examples of which include the CM-2 [7] and the Maspar MP1 [3], use global control with a single instruction fetch and decode. However, unlike vector architectures, either private memories are present at each node or values are broadcast in a regular manner from a centralized memory. SIMD machines often provide mechanisms for point-to-point communication between neighboring ALUs, but lack vector register files and efficient trans-

	Computation		Memory				Control
Benchmark	**# Inst**	**ILP**	**Record size (words) read/write**	**# Irregular memory accesses**	**# Constants**	**# Indexed scalar constants**	**Loop bounds**
convert	15	5	3/3	-	9	-	-
dct	1728	6	64/64	-	10	-	16
highpassfilter	17	3.4	9/1	-	9	-	-
fft	10	3.3	6/4	-	0	-	-
lu	2	1	2/1	-	0	-	-
md5	680	1.63	10/2	-	65	-	-
blowfish	364	1.98	1/1	-	2	256	16
rijndael	650	11.8	2/2	-	18	1024	10
vertex-simple	95	4.3	7/6	-	32	-	-
fragment-simple	64	2.96	8/4	4	16	-	-
vertex-reflection	94	7.1	9/2	-	35	-	-
fragment-reflection	98	6.2	5/3	4	7	-	-
vertex-skinning	112	6.8	16/9	-	32	288	Variable
anisotropic-filter	80	2.1	9/1	≤ 50	6	128	Variable

Table 2. Benchmark attributes.

position support in the memory system. A more severe limitation for the early SIMD machines was the lack of efficient support for irregular indexed memory accesses. To execute a 2D DCT on an 8x8 block, the image is decomposed evenly among the different processing elements. Each ALU performs some part of the DCT on the piece it owns and then exchanges values with its neighbors to complete the full 2D DCT. SIMD execution has appeared in conventional high performance microprocessors in the form of sub-word parallelism using multimedia extensions like MMX, SSE, Altivec and VIS, newer DSP-oriented processors like Imagine [30] and Intrinsity [27], as well as fragment processing in modern real-time graphics hardware [22, 25].

MIMD: MIMD machines use memory organizations similar to SIMD machines. The processing elements are independently controlled using local instruction control and private instruction memories at each processing element. MIMD processors have appeared in a variety of granularities ranging from iWarp [4] to coarse grained multiprocessors such as the CM-5 [8] or modern SMPs such as the IBM Regatta system [35]. Communication and synchronization have also typically been coarse grained through message passing, shared memory, or dedicated synchronization networks. Modern real-time graphics hardware has moved towards providing a fine grain MIMD execution model for vertex processing [26]. Individual ALUs are locally controlled, and operate in parallel on different vertices. DLP applications, such as the 2D DCT can exploit MIMD execution on a sufficiently fine grained architecture. The 2D DCT MIMD computation is similar to SIMD execution, except that the instructions in the different processing elements are not synchronized at the instruction level. Explicit synchronization instructions are used while exchanging values.

Applications and architectures: The architecture best suited for each application varies. While applications with regular memory accesses and static loops bounds or no control flow prefer a vector or SIMD architecture, the presence of irregular memory accesses or accesses to indexed scalar constants significantly reduces performance on conventional SIMD/vector machines. When the application exhibits data dependent branching as seen in *vertex-skinning* or *anisotropic-filtering*, a fine-grain MIMD architecture is the best choice. A universal data-parallel architecture is one that both supports the execution of applications from every data parallel domain and supports each type of behavior described in this section.

4. Data-Parallel Microarchitectural Mechanisms

Figure 3 shows a block diagram of an abstract microarchitecture. Data-parallel architectures have the following basic requirements: a) an execution substrate with a large number of functional units, b) efficient communication between the functional units to shuffle data, and c) technology scalability. Furthermore, in the previous section we identified the attributes of data-parallel programs affecting each of the main microarchitecture components. The first column of Table 3 summarizes these attributes. The second column lists the proposed mechanisms targeted at different microarchitecture components as shown in the third column. The last column lists the benchmarks that benefit from each mechanism. Two mechanisms are implemented in the memory system: (1) a software managed streamed memory subsystem is used to support high bandwidth regular memory accesses, and (2) a hardware managed cached memory subsystem is used to support efficient irregular memory accesses. The execution core is enhanced with additional local operand storage to efficiently support named scalar operand accesses, and an additional software managed local data storage for accessing indexed named con-

Attributes	Mechanisms	Implemented at	Benchmarks that benefit
Regular memory access	Software managed streamed memory	L2 Memory	All
Irregular memory access	Cached memory subsystem	L1 Memory	fragment-simple, fragment-reflection
Scalar named constants	Local operand storage (Operand revitalization)	Execution core, Register file	convert, dct, highpassfilter, md5, rijndael, all graphics programs
Indexed named constants	Software managed L0 data store at ALUs	Execution core	blowfish, rijndael, vertex-skinning
Tight loops	Local instruction storage (Instruction revitalization)	Execution core, Instruction fetch	All
Data dependent branching	Local program counter control	Instruction fetch, Execution core	vertex-skinning, anisotropic-filtering

Table 3. Data-parallel program attributes and the set of universal microarchitectural mechanisms. Mechanisms in parenthesis indicate TRIPS specific implementations.

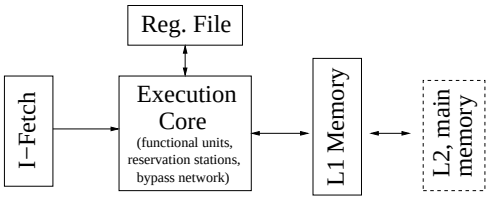

Figure 3. Microarchitecture block diagram.

stants. Finally examining control behavior, instruction storage at each ALU in the execution core is added for supporting short simple loops, and a local program counter at each ALU is added to provide data dependent branching behavior. The next sub-sections describe these microarchitectural mechanisms in detail and their implementation.

4.1. Baseline data-parallel architecture

Several architectures have been proposed with large number of functional units targeted at a subset of data-parallel applications, or to exploit instruction-level parallelism, including the Imagine architecture [30], the RAW architecture [34] and Tarantula [9]. The TRIPS processor using the Grid Processor Architecture family provides a high-performance technology-scalable execution substrate [24, 32]. In this paper we use this processor as the baseline architecture to describe and evaluate the set of microarchitectural mechanisms, and conclude this section with a discussion on applying these mechanisms to other architectures.

The TRIPS processor consists of an array of ALUs connected using a lightweight routed network. Each ALU in the array contains local instruction storage and data storage buffers. Banked instruction and data storage caches are placed around the array of ALUs backed by partitioned secondary level cache banks. The processor follows a block-atomic model of execution where an entire block of instructions is fetched and mapped onto the execution array. A dataflow style ISA that encodes each instructions's placement and its consumers, allows a statically placed but dynamically issued (SPDI) execution model. The dataflow style ISA and the distributed control and local storage inherently provided by the architecture makes the implementation of the mechanisms straight-forward.

4.2. Memory system mechanisms

Software managed cache: Figure 4a shows the configuration of the memory system that provides a high-bandwidth memory system for regular accesses. Portions of the secondary-level cache banks can be reconfigured as a fully software managed cache (SMC). In this configuration, the hardware replacement scheme and tag checks in these cache banks are disabled. The SMC banks each contain a DMA engine that is explicitly programmed by software. These banks are exposed to and are fully managed by the programmer or compiler. Only the regular memory accesses (statically identifiable by the compiler) use the SMC, and they also bypass the L1-cache since temporal locality is poor. The programming abstraction and interface used in Imagine's Stream Register File (SRF) [16] may be used to manage this SMC.

Wide loads: Overhead and latency to access the SMC can be reduced by using a `LMW` (load multiple word) instruction for reads. An `LMW` instruction issued by one ALU fetches multiple contiguous values and sends them to many ALUs in a single row inside the array. To reduce the write port pressure, a store buffer coalesces stores from different nodes together before writing them back to the SMC.

High-bandwidth streaming channels: To deliver these operands at a fast rate to the execution core, dedicated channels are provided from the SMC banks to a corresponding row of ALUs. The array based design provides a natural partitioning of the cache banks to rows of ALUs.

Cached L1-memory: Irregular memory accesses can be efficiently handled by using the level-1 cache and those banks in the level-2 not configured as SMC banks. In applications such as graphics rendering, such a caching mechanism for the irregular texture lookups can provide low latency access [13].

4.3. Instruction Fetch and Control Mechanisms

The branching behavior of data-parallel kernels dictate instruction fetch and control requirements which are: (1) repeated fetching and mapping of kernel instructions to reservation stations, resulting in instruction cache pressure and dynamic cache access power, and (2) MIMD processing support for kernels that exhibit fine grain data dependent

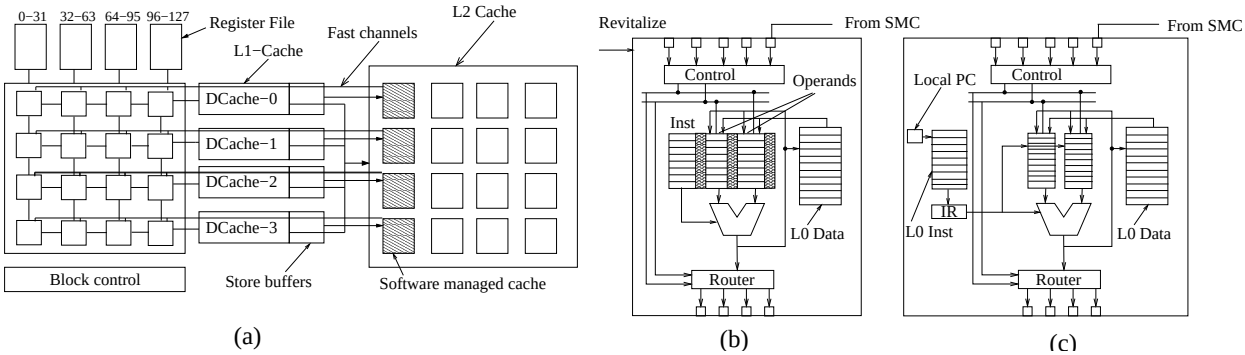

Figure 4. Microarchitectural mechanisms. a) Memory system. b) Instruction, operand revitalization and L0-data storage. c) Local PC and L0-instruction store to provide MIMD execution.

branching. To avoid repeatedly fetching instructions of a loop, the ALUs are enhanced to reuse instructions for successive iterations reading from a local storage. To efficiently support data dependent branching, each ALU is augmented with a local program counter (PC).

Instruction revitalization: In the TRIPS processor, the ALUs already contain local instruction storage. To efficiently support the execution of loops, we augment the ALUs with support for re-using instruction mappings for successive iterations of a loop. This mechanism, which we call *instruction revitalization*, works as follows: before the start of a kernel, a *setup block* executes a *repeat* instruction specifying the run-time loop bounds of the kernel which is saved to a special hardware count register `CTR`. Then the instructions of the kernel are mapped to the execution core and execute their first iteration. When the iteration completes (determined by the block control logic), the `CTR` register is decremented. If the counter has not yet reached zero, the block control logic broadcasts a global revitalize signal to all the nodes in the execution array - which resets the status bits of the instructions in the reservation stations, priming them for executing another iteration. When the `CTR` register reaches zero, the next kernel's execution commences.

To amortize the cost of the global revitalize broadcast delay, blocks are unrolled as much as possible, as determined by the number of the reservations stations, so as to reduce the number of revitalizations. Figure 4b shows the datapath and control path modifications added by this mechanism. The shaded regions next to the reservation stations indicate the status bits required for revitalization. In the TRIPS processor, using instruction revitalization provides a vector/SIMD-like architecture model.

Local program counters: To support fine grain data dependent branching, the execution core is configured as a MIMD processing array by adding local PCs at the ALUs. To simplify the datapath we also add a separate *L0 instruction storage* from which instructions are fetched and executed sequentially. (A slightly more complex, but area efficient implementation is to re-use the local instruction storage already present in the ALUs and use the PC to read this storage.) Prior to executing kernels in a MIMD mode, their instructions are loaded into this store by executing a *setup* block, which copies instructions from memory into this storage and resets the local PC to zero at every ALU. Once this *setup* block terminates, the array of ALUs begin executing in MIMD fashion. Each node independently sequences itself by fetching from its local instruction store. The operand storage buffers are used as read/write registers, providing a simple in-order fetch/register-read/execute pipeline. Figure 4c shows a schematic of the modified ALU datapath to support such a MIMD model. While this MIMD model has a one time startup delay, instruction revitalization incurs a revitalization delay between every iteration.

Multiple nodes can be aggregated together to execute one iteration of a kernel in this MIMD model, providing a logical wide-issue machine for each iteration of the kernel, using the inter-ALU network for fine-grain ALU-ALU synchronization. In this configuration the ALU array can thus be partitioned into multiple dynamically issued cores. Another mode of operation is to execute different kernels on the ALUs, passing values using between them through the inter-ALU network. In real-time graphics processing for example, a rendering pipeline can be implemented by partitioning the ALUs among vertex processing, rasterization, and fragment processing kernels. Since the ALUs are homogeneous and fully programmable, the partitioning of ALUs can be dynamically determined based on scene attributes. This strategy overcomes one of the limitations of current graphics pipelines in which the vertex, rasterization and fragment engines are specialized distinct units.

4.4. Execution core mechanisms

Efficient scalar operand and indexed scalar operand access must be supported for data-parallel execution. For large, statically unrolled loops, reading values from the registers for each iteration of the loop is expensive in terms of power, register file bandwidth, and other overheads of

register file access. Using the memory system for indexed scalar operands incurs cache access overheads and consumes cache bandwidth. Two mechanisms implemented at the execution core support these two types of accesses efficiently.

Operand revitalization: This mechanism reuses register values once they have been received at an ALU, providing persistent register-file like storage at each reservation station. Successive iterations of the loop reuse the values from the reservation stations instead of accessing the global register file. To implement operand revitalization we add status bits to the reservations stations, as shown in Figure 4b.

L0 data storage: A software managed L0 data storage at each ALU provides support for indexed scalar constants (one example is the lookup tables used in encryption kernels). Figures 4b and 4c show the L0 data store, which is accessed using an index computed by some instruction with the result being written to the reservation stations. The index to read the L0 data store is provided by the ALUs and the results are written back into the local registers as shown. For the applications we examined, 2KB was sufficient to store all such constants.

4.5. Summary

While we described these mechanisms using the TRIPS processor as the baseline, they are universal and applicable to other architectures. The SMC, store buffer and the `LMW` instructions can be added in a straightforward manner to conventional wide-issue centralized or clustered superscalar architectures by adding direct channels from the L2-caches to the functional units and augmenting the pipeline to wakeup instructions dependent on the loads when their operands arrive from the SMC. The Tarantula architecture provides similar such support for transfers from the L2 memory to the vector register file, using hardware techniques to generate conflict free addresses to different banks in memory, in contrast to our approach of packing all the regular accesses in a single bank. To support indexed scalar access and irregular memory accesses in this architecture, the L1-cache memory must be addressable using special scatter/gather instructions. Most conventional superscalar processors provide good support for L1-cache memories.

The reservation stations in TRIPS have a one-to-one correspondence to reservation stations in superscalar architectures and both the instruction and operand revitalization mechanisms can be applied to provide instruction and operand re-use. Many DSP processors have implemented zero-overhead branches in different ways to support tight loops. To provide MIMD support local PCs are added and the local ALU control logic modified to fetch from a local instruction store buffer. Conventional SIMD and vector cores conversely have no local storage and thus must be augmented with a local PC and storage buffers to provide a MIMD model of execution.

5. Results

This section presents the compilation strategy, simulation methodology, and the performance evaluation of the mechanisms. The results focus on evaluating and measuring the following: (1) performance improvement provided by each mechanism, (2) benefit from different mechanisms for each application, (3) performance of a flexible architecture constructed using a combination of the mechanisms, and (4) this flexible architecture's performance relative to specialized architectures.

5.1. Simulation methodology

The baseline TRIPS processor executes hyperblocks constructed using the IMPACT compiler, scheduled using our software schedulers. We use an event-driven timing simulator to model the microarchitecture. The different mechanisms were integrated into this simulator for the performance experiments. All the programs were hand-coded in the TRIPS instruction set to exploit these data-parallel mechanisms and then simulated. Where possible we statically unrolled the kernels to fill up the instruction storage across the ALUs. The results show relative speedup (measured in terms of execution cycles) between the baseline and the different machine configurations. The simulations assumed that all data was resident in the software managed cache (SMC) or L2 storage for all applications. Except for *lu*, the datasets of all applications fit entirely in the SMC.

5.2. Baseline TRIPS performance

Our baseline configuration is a mesh interconnect 8x8 array with 64KB SMC banks, one per each row of the processor, a total of 2MB of L2 cache, and a partitioned 64KB L1 data and instruction cache. The functional unit and cache access latencies are configured to match an Alpha 21264. Each node in the processor consists of an integer ALU, integer multiplier, and an FPU with add, multiply, and divide capability. We assumed a 100nm technology with a 10FO4 clock rate, making the hop delay between adjacent ALUs half a cycle.

Table 4 shows the performance of the baseline measured in terms of number of useful computation operations sustained per cycle, not including overhead instructions like address compute and load and store instructions[1]. Only the DSP programs sustain a reasonably high computation throughput, averaging about 11 ops/cycle, while all other applications sustain low throughputs, averaging about 4 ops/cycle.

Since the baseline TRIPS processor is optimized for ILP, converting the data level parallelism in these applications

1 Since we did not have sufficient infrastructure and datasets for a realistic simulation of anisotropic-filtering, we exclude it from all our performance tables and figures.

Benchmark	Ops/cycle	Benchmark	Ops/cycle
convert	14.1	fragment-reflection	4.0
dct	10.4	fragment-simple	2.6
highpassfilter	7.4	vertex-reflection	5.2
fft	3.7	vertex-simple	3.6
lu	0.7	vertex-skinning	5.6
md5	2.8		
blowfish	5.1		
rijndael	7.5		

Table 4. Performance on baseline TRIPS.

Config.	L0 store		Revitalization		Architecture model
	Inst.	Data	Inst.	Ops.	
S	N	N	Y	N	SIMD
S-O	N	N	Y	Y	SIMD+ scalar constant access
S-O-D	N	Y	Y	Y	SIMD+ scalar constant access+ lookup table
M	Y	N	N	N	MIMD
M-D	Y	Y	N	N	MIMD+lookup table

Table 5. Machine configurations.

to ILP results in inefficiencies for DLP programs which include. For example, loops cannot be sufficiently unrolled to provide large enough blocks to efficiently utilize the array of ALUs, and every scalar operand or memory reference must proceed through shared structures such as the L1 cache and the common register file. Since many DLP programs have large demands on these resources, the limited bandwidth prevents the architecture from achieving its potential performance.

5.3. Mechanism Evaluation

The mechanisms described in Section 4 can be combined in different ways according to application requirements to produce as many as 20 different run-time machine configurations of a single flexible architecture. The frequency of each type of memory access, the control behavior of the kernels and the instruction size of kernels, measured in Table 2 determines the ideal combination of mechanisms on the TRIPS processor. In this paper we focus on five machine configurations, shown in Table 5, that cover the application set we examined.

In all five configurations, one memory bank per row is configured to be used as a software managed cache. The SMC banks use the store buffers and the high speed channels to communicate with the execution core. Combining this memory system with an instruction revitalization mechanism creates a baseline model that is similar to SIMD and vector machine. This baseline machine (**S**) can be augmented with operand revitalization to create the **S-O** machine. The **S-O-D** machine adds local L0 data storage to each ALU of the S-O machine. Combining the memory system with local PCs creates a baseline MIMD machine (**M**), and local L0 data storage in addition creates the **M-D** machine.

Figure 5 shows the application speedups obtained by these different machine configurations relative to the baseline. The following paragraphs classify the applications by their preferred configurations. Two benchmarks preferred the S, seven preferred the S-O and four preferred M-D configuration.

- **SIMD execution (S):** *fft* and *lu* are vector-oriented benchmarks and require high memory bandwidth and high instruction fetch rate. Compared to the baseline a four-fold speedup is achieved because of the higher ALU utilization and higher memory bandwidth of the **S** configuration. Adding other mechanisms does not improve performance further, and the routing overhead of MIMD execution degrades performance slightly.
- **SIMD + scalar operand access (S-O):** The performance of many applications is dictated by the frequency of scalar operand access (35 constants in *vertex-reflection* for example). These perform best on the **S-O** machine configuration as shown by the set of 7 programs in Figure 5.
- **SIMD + scalar operand + lookup table access (S-O-D):** *Blowfish, and rijndael* which use reasonably large lookup tables show speedups of 27% and 80% respectively over the **S-O** configuration, but perform poorer than the **M-D** machine.
- **MIMD (M):** The baseline MIMD configuration degrades performance somewhat relative to **S-O-D** for all applications except *vertex-skinning*. This degradation arises because in the MIMD model the load instructions from each ALU must be routed through the network to reach the memory interface. In the previous three **SIMD** configurations, synchronized at block boundaries, a multi-word load instruction could be placed near the memory interface, to behave like a vector fetch unit. Since each node operates independently in the MIMD model, such a schedule is not possible.
- **MIMD + lookup table access (M-D):** The MIMD machine with lookup table support performs best for *md5*, *blowfish*, *rijndael*, and *vertex-skinning*. With local looping control, these programs require far less instruction storage and hence can be unrolled more aggressively providing more parallelism. Because *vertex skinning* uses data dependent branching, the overheads of predicated execution (or conditional vectors) are also removed.
- **Flexibility:** The last single bar labeled *Flexible* in Figure 5 shows the harmonic mean of speedups achieved by a flexible architecture when a subset of mechanisms are combined according to application needs (running *fft* and *lu* on **S**, *convert* through *vertex simple light* on **S-O**, and the rest on **M-D**). Averaged across the different applications, this flexible dynamic tuning pro-

Benchmark	Performance		Reference Hardware	Units
	TRIPS (clock normalized)	Specialized hardware		
convert	19016	960	MPC 7447, 1.3Ghz	iterations/sec
highpassfilter	2820	907	(DSP processor)	iterations/sec
dct	33.9	8.2	Imagine [30] (multimedia processor)	ops/cycle
fft	14.4	28	Tarantula [9]	ops/cycle
lu	10.6	15	(vector core)	ops/cycle
md5	14.6	-	Cryptomaniac [36]	cycles/block
blowfish	6	80		cycles/block
rijndael	12	100		cycles/block
fragment-reflection	86	-	Nvidia QuadroFX 450Mhz	million fragments/sec
fragment-simple	193	1500	(graphics processor)	million fragments/sec
vertex-reflection	434	-	Benchmarked on	million triangles/sec
vertex-simple	418	64	2.4Ghz Pentium4	million triangles/sec
vertex-skinning	207	-		million triangles/sec

Table 6. Performance comparison of TRIPS with DLP mechanisms to specialized hardware.

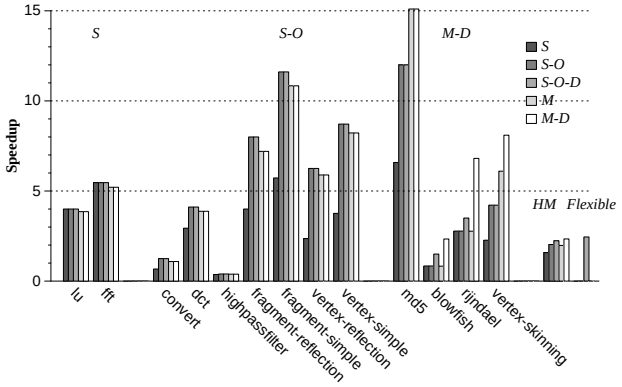

Figure 5. Speedup using different mechanisms, relative to baseline architecture. Programs grouped by best machine configuration.

vides 55% better performance over a fixed **S** configuration, 20% better than fixed **S-O** and 5% better than a fixed **M-D** machine.

5.4. Comparison against specialized architectures

Table 6 shows the results of a rough comparison between the performance of the configurable TRIPS architecture to published performance results on specialized hardware. Columns 2 and 3 show performance, column 4 describes the specialized hardware, and column 5 shows the performance metrics (which vary). For each of the applications we picked the best combination of the mechanisms on the TRIPS baseline. When appropriate, we normalized the clock rate of TRIPS to that of the specialized hardware. Scaling the clock does not violate any microarchitecture assumptions, since the TRIPS processor is designed for potentially faster clock rates than conventional designs.

On the signal processing codes, the TRIPS core in the *S-O* configuration, is 3–20 times faster than the MPC 7447, with the improvement coming from the 16X higher issue-width (4 vs 64). An 8x8 TRIPS core contains roughly four times the number of functional units as the Imagine architecture and performs roughly four times better on *dct*. On the scientific codes, the TRIPS *S* configuration is store bandwidth limited and about a factor of two worse than the Tarantula architecture. On the network processing programs, exploiting the extensive data level parallelism in network flows, the TRIPS *S-O* and *S-O-D* configurations perform an order of magnitude better than specialized hardware, where the packets are processed serially (smaller numbers in the table for these programs indicates better performance). In the *vertex-simple* graphics application, TRIPS outperforms the dedicated hardware primarily because of the much higher issue width and functional unit count. On *fragment-simple* on the other hand the specialized hardware outperforms TRIPS by roughly 8X. Although the exact details on the number of functional units (fixed point + floating point units) on the QuadroFX are not publicly disclosed, we believe part of this high performance can be attributed to the larger number of functional units. The other graphics processing kernels are more complex (using more instructions, more constants, and data dependent branching in one case) than the two we benchmarked, and will perform at best as well as the other kernels, and likely poorer.

6. Related work

Classic vector processors were built using expensive SRAMs for high-speed memory and large vector register files [31, 23, 14]. These machines were designed for programs with regular control and data behavior, but could tolerate some degree of irregular (but structured) memory accesses using scatter and gather operations. Programs with frequent irregular memory references or accesses to lookup tables performed poorly. A number of architectures have been proposed or built to overcome the limitations of the rigid vector execution model and to allow for dynamic instruction scheduling [10, 18, 9]. Removing these limitation still did not make these architectures widely applicable as they provided support only for a subset of data parallel programs. Short vector processing has found its way into commercial microprocessors in the form of instruction extensions such as MMX, SSE2, Altivec and VIS. These architectures have similar requirements of regular control and

data access, and have further restrictions on data alignment. Some of the ISA extensions, such as MMX and SSE2, have poor support for scalar-vector operations, only operating on one sub-word of a MMX/SSE2 register when using a scalar register as one operand.

The early fine-grain SIMD machines like the CM-2 [7] and MasPar MP-1 [3] provided high ALU density but lacked support for fine-grain control and latency tolerance to irregular memory accesses. Addressing some of these problems, the recently announced Intrinsity processor includes a SIMD array with a traditional cache memory system [27]. These architectures provide some of the support provided by the mechanisms we propose, but are not complete data-parallel architectures.

Imagine dubbed a stream processor, is a SIMD/vector hybrid using a SIMD control unit coupled with a memory system resembling a vector machine [30]. Other forms of stream processing are more similar to MIMD execution in which streams of data are pipelined across multiple processors in a highly structured fashion. Examples include graphics pipelines [1] and video processing [5]. Mark et al. describe the motivation and application requirements for MIMD processing in real-time graphics [21]. New fine-grained on-chip MIMD architectures such as Smart Memories [20] and RAW [34] are emerging, targeting this style of stream processing.

7. Conclusions

This paper presents the first comprehensive treatment of programs covering a large spectrum of the DLP application space, including signal processing, scientific, network/security, and real-time graphics applications. While there may be DLP applications outside these domains, the four studied in this paper provide comprehensive coverage over the application space. We identified the key memory, control, and computation demands of DLP applications and characterized the behavior of the DLP application suite.

We then proposed a set of complementary universal microarchitectural mechanisms targeted at the memory system, instruction control, and execution core, that can support each type of DLP behavior. For the memory system, we proposed a streamed software managed cache memory along with a hardware managed level-1 cache. For the execution core and instruction control we proposed local operand storage, local instruction storage, a software managed local storage, and local program counters at each ALU site. These mechanisms can be combined in different ways based on application demand and are powerful enough to provide both a SIMD and MIMD execution model on the same substrate. We found the approach of customizing the architecture resulted in 5%–55% better performance than a fixed yet scalable architecture. The approach in this paper of customizing the architecture to the application has similarities to the philosophy of custom-fit processors [12], but the customization we propose enables different execution models on the same substrate and can be performed after fabrication. When compared to application-specific processors in each of the domains, the architecture built using the mechanisms in this paper achieves performance in a similar range, when normalizing for clock rate and ALU count. While each application specific processor performs well in its own domain, none have significant flexibility to perform well on DLP applications outside its domain.

The mechanisms that we propose are not strictly limited to the TRIPS processor described in this paper. For example the hybrid of SIMD and fine-grain MIMD execution models is a reasonable goal for other DLP architectures. Future systems that must execute multiple classes of DLP applications will benefit by implementing all of the mechanisms and dynamically configuring the architecture based on application needs. However, when only a subset of DLP behavior needs to be supported, the flexibility can be sacrificed for simplicity by implementing a subset of the mechanisms on a fixed architecture by matching the mechanisms to the application attributes. Finally, using either of these approaches, we foresee the appearance of these mechanisms in general purpose processors, targeted at identifying and accelerating applications outside the DLP space, but that exhibit DLP behavior.

While this paper focused on the design and performance evaluation of the DLP mechanisms, we see several directions for future work. First these mechanisms can be evaluated using more detailed metrics, including cycle time, power, and area. Heterogeneous architectures which integrate multiple specialized data-parallel processors, each targeting a distinct type of workload, present a competing design philosophy. Comparison and evaluation of the DLP mechanisms in this paper against heterogeneous architectures should yield useful and interesting results.

Acknowledgments

We thank the anonymous reviewers for their suggestions that helped improve the quality of this paper. This research is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-1892, NSF instrumentation grant EIA-9985991, NSF CAREER grants CCR-9985109 and CCR-9984336, two IBM University Partnership awards, grants from the Alfred P. Sloan Foundation and the Intel Research Council, and an equipment donation from NVIDIA.

References

[1] K. Akeley. Reality engine graphics. In *Proc. of the 20th Ann. Conf. on Computer Graphics*, pages 109–116, 1993.

[2] K. Asanovic, J. Beck, B. Irissou, B. Kingsbury, N. Morgan, and J. Wawrzynek. The T0 Vector Microprocessor. In *Proc. HOT Chips VII*, August 1995.

[3] T. Blank. The Maspar MP-1 architecture. In *Proc. of IEEE Compcon, Spring 1990*, pages 20–24.

[4] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proc. of Supercomputing '88*, pages 330–339, November 1988.

[5] V. Bove and J. Watlington. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2):140–149, 1995.

[6] I. Buck. Data parallel computation on graphics hardware. In *Graphics Hardware 2003: Panel Presentation*, 2003.

[7] The Connection Machine CM-2 Technical Summary, April 1987.

[8] The Connection Machine CM-5 Technical Summary, October 1991.

[9] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *Proc. of the 29th Int'l Symp. on Computer Architecture*, pages 281–292, June 2002.

[10] R. Espasa, M. Valero, and J. E. Smith. Out of order vector architectures. In *Proc. of the 30th Ann. Int'l Symp. on Microarchitecture*, pages 160–170, December 1997.

[11] R. Fernando and M. J. Kilgard. *The Cg Tutorial*. Addison-Wesley Publishing Company, 2003.

[12] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architectures. In *Proc. of the 29th Ann. Int'l Symp. on Microarchitecture*, pages 324–335, December 1996.

[13] Z. S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proc. of the 24th Int'l Symp. on Computer Architecture*, pages 108–120, June 1997.

[14] J. Hennesy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1996.

[15] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient Conditional Operations for Data-parallel Architectures. In *Proc. of the 33rd Ann. Int'l Symp. on Microarchitecture*, pages 159–170, 2000.

[16] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, , and B. Towles. Stream Scheduling. In *Proc. of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, December 2001.

[17] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM. In *Proc. Hot Chips XII*, August 2000.

[18] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *Proc. of the 30th Int'l Symp. on Computer Architecture*, pages 399–409, June 2003.

[19] A. Kunimatsu et. al. Vector Unit Architecture For Emotion Synthesis. *IEEE Micro*, 20(2):40–47, March 2000.

[20] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proc. of the 27th Int'l Symp. on Computer Architecture*, pages 161–171, June 2000.

[21] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proc. of the 30th Ann. Conf. on Computer Graphics*, 2003.

[22] J. L. Mitchell. Radeon 9700 Shading (ATI Technologies, white paper), July 2002.

[23] B. Moore, A. Padegs, R. Smith, and W. Bucholz. Concepts of th System/370 Architecture. In *Proc. of the 14th Int'l Symp. on Computer Architecture*, pages 282–292, June 1987.

[24] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A Design Space Exploration of Grid Processor Architectures. In *Proc. of the 34th Ann. Int'l Symp. on Microarchitecture*, pages 177–189, December 2001.

[25] NVIDIA Corp. NV_fragment_program. In NVIDIA OpenGL Extension Specifications. Jan.2002.

[26] NVIDIA Corp. NV_vertex_program2. In NVIDIA OpenGL Extension Specifications. Jan. 2002.

[27] T. Olson. Advanced Processing Techniques Using the Intrinsity FastMATH Processor. In *Embedded Processor Forum*, May 2002.

[28] M. Pharr and G. Humphreys. Design and Implementation of a Physically-Based Rendering System. Draft edition, August, 2003.

[29] S. Rajagopal, S. Rixner, and J. Cavallaro. A programmable baseband processor design for software defined radios. In *Proc. of the IEEE Int'l Midwest Symp. on Circuits and Systems*, 2002.

[30] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proc. of the 31st Ann. Int'l Symp. on Microarchitecture*, pages 3–13, December 1998.

[31] R. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 22(1):64–72, January 1978.

[32] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture. In *Proc. of the 30th Int'l Symp. on Computer Architecture*, pages 422–433, June 2003.

[33] J. E. Smith, G. Faanes, and R. A. Sugumar. Vector instruction set support for conditional operations. In *Proc. of the 27th Int'l Symp. on Computer Architecture*, pages 260–269, June 2000.

[34] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. L. Jae-Wook Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002.

[35] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, January 2002.

[36] L. Wu, C. Weaver, and T. Austin. CryptoManiac: A Fast Flexible Architecture for Secure Communication. In *Proc. of the 28th Int'l Symp. on Computer Architecture*, pages 110–119, June 2001.