Smart Memories: A Modular Reconfigurable Architecture

Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, Mark Horowitz
Computer Systems Laboratory
Stanford University
Stanford, California 94305
{demon,paaske,jayasena,ronho,billd,horowitz}@leland.stanford.edu

Abstract

Trends in VLSI technology scaling demand that future computing devices be narrowly focused to achieve high performance and high efficiency, yet also target the high volumes and low costs of widely applicable general purpose designs. To address these conflicting requirements, we propose a modular reconfigurable architecture called Smart Memories, targeted at computing needs in the 0.1μm technology generation. A Smart Memories chip is made up of many processing tiles, each containing local memory, local interconnect, and a processor core. For efficient computation under a wide class of possible applications, the memories, the wires, and the computational model can all be altered to match the applications. To show the applicability of this design, two very different machines at opposite ends of the architectural spectrum, the Imagine stream processor and the Hydra speculative multiprocessor, are mapped onto the Smart Memories computing substrate. Simulations of the mappings show that the Smart Memories architecture can successfully map these architectures with only modest performance degradation.

1. Introduction

The continued scaling of integrated circuit fabrication technology will dramatically affect the architecture of future computing systems. Scaling will make computation cheaper, smaller, and lower-power, thus enabling more sophisticated computation in a growing number of embedded applications. This spread of low-cost, low-power computing can easily be seen in today's wired (*e.g.* gigabit ethernet or DSL) and wireless communication devices, gaming consoles, and handheld PDAs. These new applications have different characteristics from today's standard workloads, often containing highly data-parallel streaming behavior [1]. While the applications will demand ever-growing compute performance, power (ops/W) and computational efficiency (ops/$) are also paramount; therefore, designers have created narrowly-focused custom silicon solutions to meet these needs.

This work was supported in part by DARPA contract MDA904-98-R-S855.

ISCA 00 Vancouver, British Columbia Canada

However, the scaling of process technologies makes the construction of custom solutions increasingly difficult due to the increasing complexity of the desired devices. While designer productivity has improved over time, and technologies like system-on-a-chip help to manage complexity, each generation of complex machines is more expensive to design than the previous one. High non-recurring fabrication costs (*e.g.* mask generation) and long chip manufacturing delays mean that designs must be all the more carefully validated, further increasing the design costs. Thus, these large complex chips are only cost-effective if they can be sold in large volumes. This need for a large market runs counter to the drive for efficient, narrowly-focused, custom hardware solutions.

To fill the need for widely-applicable computing designs, a number of more general-purpose processors are targeted at a class of problems, rather than at specific applications. Tri-media [2,3], Equator [4], Mpact [5], IRAM [6], and many other projects are all attempts to create general purpose computing engine for multi-media applications. However, these attempts to create more universal computing elements have some limitations. First, these machines have been optimized for applications where the parallelism can be expressed at the instruction level using either VLIW or vector engines. However, they would not be very efficient for applications that lacked parallelism at this level, but had, for example, thread-level parallelism. Second, their globally shared resource models (shared multi-ported registers and memory) will be increasingly difficult to implement in future technologies in which on-chip communication costs are appreciable [7,8]. Finally, since these machines are generally compromise solutions between true signal processing engines and general-purpose processors, their efficiency at doing either task suffers.

On the other hand, the need for scalable architectures has also led to proposals for modular, explicitly parallel architectures that typically consist of a number of processing elements and memories on a die connected together by a network [9,10]. The modular nature of these designs ensures that wire lengths shrink as technologies improve, allowing wire and gate delays to scale at roughly the same rate [7]. Additionally, the replication consumes the growing number of transistors. The multiple processing elements take advantage of both instruction-level and thread-level parallelism. One of the most prominent architectures in this class is the MIT Raw project [10], which focuses on the development of compiler technologies that take advantage of exposed low-level hardware.

Smart Memories combines the benefits of both approaches to create a partitioned, explicitly parallel, reconfigurable architecture for use as a future universal computing element. Since different application spaces naturally have different communication patterns and memory needs, finding a single topology that fits well with all applica-

tions is very difficult. Rather than trying to find a general solution for all applications, we tailor the appearance of the on-chip memory, interconnection network, and processing elements to better match the application requirements. We leverage the fact that long wires in current (and future) VLSI chips require active repeater insertion for minimum delay. The presence of repeaters means that adding some reconfigurable logic to these wires will only modestly impact their performance. Reconfiguration at this level leads to coarser-grained configurability than previous reconfigurable architectures, most of which were at least in part based on FPGA implementations [11-18]. Compared to these systems, Smart Memories trades away some flexibility for lower overheads, more familiar programming models, and higher efficiency.

Section 2 and Section 3 describe the Smart Memories architecture. To test the flexibility of the architecture, we mapped onto the Smart Memories substrate two machines at different ends of the architectural spectrum: a dedicated streaming processor and a speculative multiprocessor. Section 4 discusses the mapping of these two widely disparate architectures onto one hardware substrate and the simulated relative performance. Section 5 draws conclusions from the architectural proposal and mapping studies.

2. Smart Memories Overview

At the highest level, a Smart Memories chip is a modular computer. It contains an array of processor tiles and on-die DRAM memories connected by a packet-based, dynamically-routed network (Figure 1). The network also connects to high-speed links on the pins of the chip to allow for the construction of multi-chip systems. Most of the initial hardware design work in the Smart Memories project has been on the processor tile design and evaluation, so this paper focuses on these aspects.

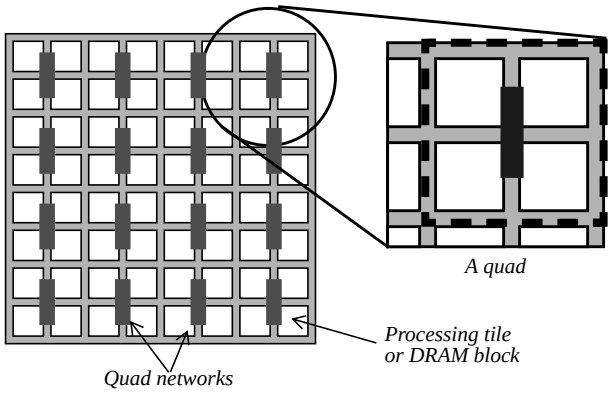

Figure 1. A Smart Memories chip

The organization of a processor tile is a compromise between VLSI wire constraints and computational efficiency. Our initial goal was to make each processor tile small enough so the delay of a repeated wire around the semi-perimeter of the tile would be less then a clock cycle. This leads to a tile edge of around 2.5mm in a 0.1µm technology [7]. This sized tile can contain a processor equivalent to a MIPS R5000 [19], a 64-bit, 2-issue, in-order machine with 64KB of on-die cache. Alternately, this area can contain 2-4MB of embedded DRAM depending on the assumed cell size. A 400mm^2 die would then hold about 64 processor tiles, or a lesser number of processor tiles and some DRAM tiles.

Since large-scale computations may require more computation power than what is contained in a single processing tile, we cluster four processor tiles together into a "quad" and provide a low-overhead, intra-quad, interconnection network. Grouping the tiles into quads also makes the global interconnection network more efficient by reducing the number of global network interfaces and thus the number of hops between processors.

Our goal in the tile design is to create a set of components that will span as wide an application set as possible. In current architectures, computational elements are somewhat standardized; today, most processors have multiple segmented functional units to increase efficiency when working on limited precision numbers [20-24]. Since much work has already been done on optimizing the mix of functional units for a wide application class [2,3,4,25], we instead focused our efforts on creating the flexibility needed to efficiently support different computational models. This requires creating a flexible memory system, flexible interconnection between the processing node and the memory, and flexible instruction decode.

3. Tile Architecture

A Smart Memories tile consists of a reconfigurable memory system; a crossbar interconnection network; a processor core; and a quad network interface (Figure 2). To balance computation, communication, and storage, we allocated equal portions of the tile to the processor, interconnect, and memory.

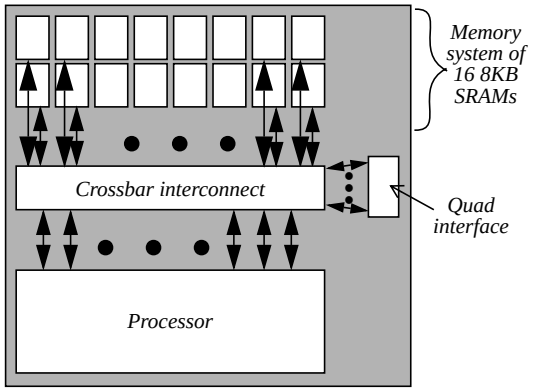

Figure 2. Tile floorplan

3.1 Memory System

The memory system is of growing importance in processor design [26]. Different applications have different memory access patterns and thus require different memory configurations to optimize performance. Often these different memory structures require different control logic and status bits. Therefore, a memory system that can be configured to closely match the application demands is desirable [27].

A recent study of SRAM design [28] shows that the optimal block size for building large SRAMs is small, around a few KB. Large

SRAMs are then made up of many of these smaller SRAM blocks. We leverage this naturally hierarchical design to provide low overhead reconfigurability. The basic memory mat size of 8KB is chosen based on a study of decoder and I/O overheads and an architectural study of the smallest memory granularity needed. Allocating a third of the tile area to memory allows for 16 independent 8KB memory mats, a total of 128KB per tile. Each mat is a 1024x64b logical memory array that can perform reads, writes, compares, and read-modify-writes. All operations are byte-maskable.

In addition to the memory array, there is configurable logic in the address and data paths. In the address path, the mats take in a 10-bit address and a 4-bit opcode to determine what operation is to be performed. The opcode is decoded using a reconfigurable logic block that is set up during the hardware configuration. The memory address decoder can use the address input directly or can be set in auto-increment/decrement streaming mode. In this mode, the mat stores the starting index, stream count, and stride. On each streaming mode request, the mat accesses the next word of the stream until reaching the end of the stream.

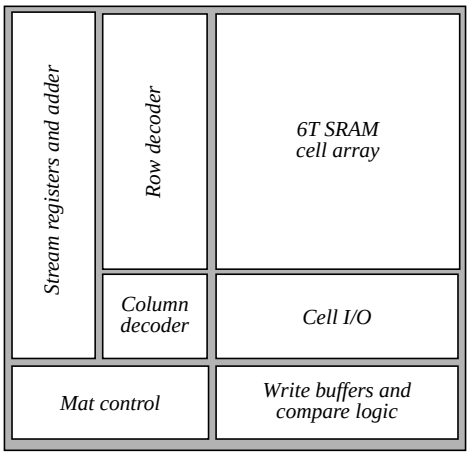

Figure 3. Memory mat detail

In the datapath, each 64-bit word is associated with a valid bit and a 4-bit configurable control field. These bits can be used for storing data state such as cache LRU or coherence bits. They are dual-ported to allow read-modify-write operations each cycle and can be flash cleared via special opcodes. Each mat has a write buffer to support pipelined writes and to enable conditional write operations (*e.g.* in the case of a cache write). Mats also contain logic in the output read path for comparisons, so they can be used as cache tag memory.

For complex memory structures that need multiple accesses to the same data (*e.g.* snooping on the cache tags in a multiprocessor), four of the mats are fully dual-ported. Many applications and architectures also need fully-associative memories which are inefficient and difficult to emulate using mats. Therefore, the tile memory system also contains a 64-entry content-addressable memory (CAM).

The Smart Memories mats can be configured to implement a wide variety of caches, from simple, single-ported, direct-mapped structures to set-associative, multi-banked designs. Figure 4 gives an example of four memory mats configured as a two-way set associative cache with two of the mats acting as the tag memories and two other mats acting as the data memories.

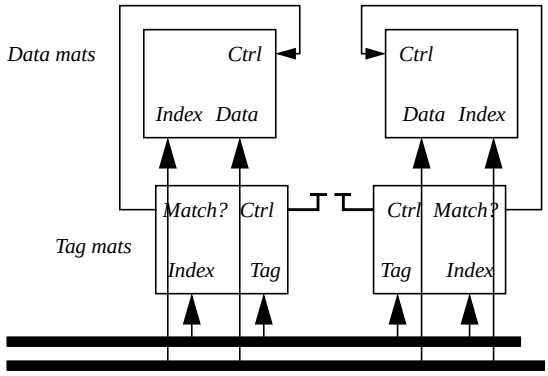

Figure 4. Mats configured as 2-way set-associative cache

The mats can also be configured as local scratchpad memories or as vector/stream register files. These simpler configurations have higher efficiency and can support higher total memory bandwidth at a lower energy cost per access [29-31]. Associated with the memory, but located in the two load-store units of the processor, are direct-memory access (DMA) engines that generate memory requests to the quad and global interconnection networks. When the memory mats are configured as caches, the DMA engines generate cache fill/spill requests. When the mats are configured for streaming or vector memories, the DMA engines generate the needed gather/scatter requests to fill the memory with the desired data.

3.2 Interconnect

To connect the different memory mats to the desired processor or quad interface port, the tile contains a dynamically routed crossbar which supports up to 8 concurrent references. The processor and quad interface generate requests for data, and the quad interface and memories service those requests. The crossbar does not interconnect different units of the same type (*e.g.* memory mat to memory mat communication is not supported in the crossbar).

Requests through the crossbar contain a tag indicating the desired destination port and an index into the memory or unit attached to that port. The crossbar protocol always returns data back to the requestor, so data replies can be scheduled at the time of routing the forward-going request. Requests can be broadcast to multiple mats via wildcards, but only one data reply is allowed. The requests and replies are all pipelined, allowing a requestor to issue a new request every cycle. Arbitration is performed among the processor and quad interface ports since multiple requests for the same mat or quad interface port may occur. No arbitration is necessary on the return crossbar routes, since they are simply delayed versions of the forward crossbar routes.

From circuit-level models of the crossbar and the memories, the estimated latency for a memory request is 2 processor clock cycles.

About half of the time is spent in the crossbar, and the other half is spent in the memory mat. We project that our processor core will have a clock cycle of 20 fanout-of-four inverter delays (FO4s), which is comparable to moderately aggressive current processor designs [7]. In a commodity 0.1µm process, a 20 FO4 cycle time is equivalent to a 1GHz operating frequency.

The quad interconnection network, shown in Figure 5, connects the four tiles in a quad together. The network consists of 9 64-bit multicast buses on which any of the 4 tiles or the global network can send or receive data. These buses may also be configured as half-word buses. In addition to these buses, a small number of control bits are broadcast to update state, atomically stall the processors, and arbitrate for the buses. The quad interface on each tile connects the internal tile crossbar to the quad network, thus mediating all communication to and from the tile.

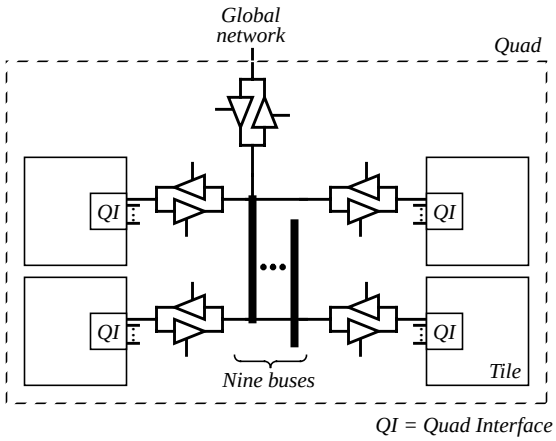

Figure 5. Quad interconnection network

3.3 Processor

The processor portion of a Smart Memories tile is a 64-bit processing engine with reconfigurable instruction format/decode. The computation resources of the tile consist of two integer clusters and one floating point (FP) cluster. The arrangement of these units and the FP cluster unit mix are shown in Figure 6. Each integer cluster consists of an ALU, register file, and load/store unit. This arithmetic unit mix reflects a trade-off between the resources needed for a wide range of applications and the area constraints of the Smart Memories tile [2-5]. Like current media processors, all 64-bit FP arithmetic units can also perform the corresponding integer operations and all but the divide/sqrt unit perform subword arithmetic.

The high operand bandwidth needed in the FP cluster to sustain parallel issue of operations to all functional units is provided by local register files (LRFs) directly feeding the functional units and a shared register file with two read and one write ports. The LRF structure provides the necessary bandwidth more efficiently in terms of area, power, and access time compared to increasing the number of ports to the shared register file [25,32]. The shared FP register file provides a central register pool for LRF overflows and shared constants. A network of result and operand buses transfers data among functional units and the register files.

Optimal utilization of these resources requires that the instruction bandwidth be tailored to the application needs. When ILP is abundant, wide encodings explicitly express parallelism and enhance performance without significantly degrading code density. When ILP is limited, narrow instructions yield dense encodings without a loss in performance. The Smart Memories instruction path, shown at the block level in Figure 7, can be configured to efficiently support wide or narrow instruction encodings.

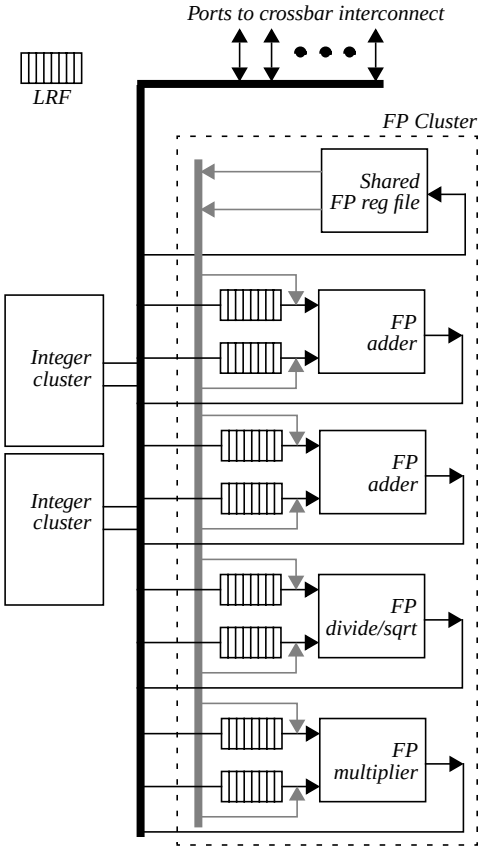

Figure 6. Smart Memories compute resources

A 256-bit microcode instruction format achieves the highest utilization of resources. In this configuration, the processor issues operations to all available units in parallel and explicitly orchestrates data transfers in the datapath. This instruction format is primarily intended for media and signal processing kernels that have high compute requirements and contain loops that can be unrolled to extract ample parallelism. For applications that contain ILP but are less regular, a VLIW instruction format that packs three instructions in a 128-bit packet is supported. This instruction format provides a compromise that achieves higher code density but less parallelism than the microcode, yet higher parallelism but less code density than narrow instructions.

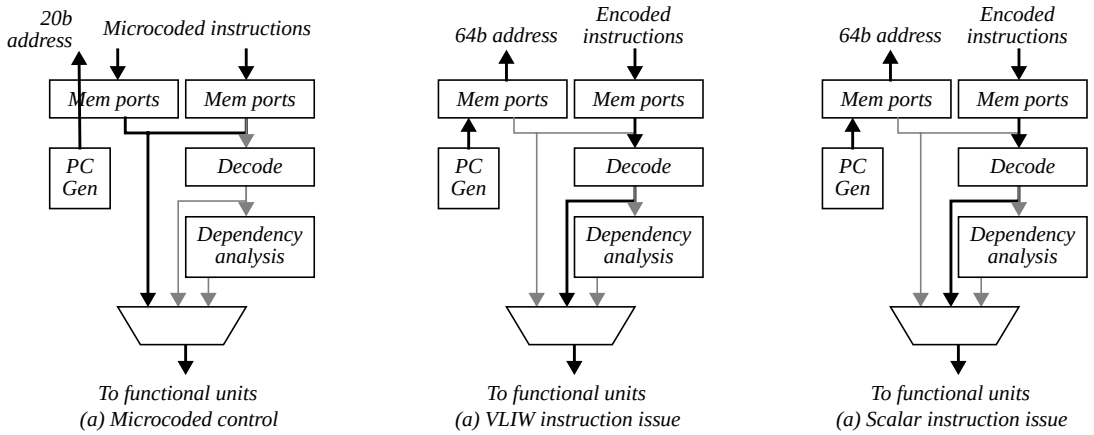

Figure 7. Instruction path

Finally, a 32-bit RISC-style instruction set is available for applications that do not exhibit much ILP. To extract thread-level parallelism of such applications, each tile can sustain two concurrent, independent threads. The two threads on a tile are asymmetric. The primary thread may perform integer or FP computations and can issue up to two instructions per cycle, while the secondary thread is limited to integer operations at single-issue. The secondary thread is intended for light-duty tasks or for system-level support functions. For example, lower communication costs on systems with multiple processing nodes on a chip permit dynamic data and task migration to improve locality and load balance at a much finer grain than is practical in conventional multi-processors. The increased communication volume and resource usage tracking for such operations can easily be delegated to the secondary thread. The two threads are assumed to be independent and any communication must be explicitly synchronized.

For managing interaction with the memory mats and quad interface, the tile processor has two load/store units, each with its own DMA engine as described in Section 3.1. The load/store units, the functional units, and the instruction decode share the 8 processor ports into tile crossbar for communicating with the memory mats and quad interface.

4. Mapping Streaming and Speculative Architectures

One of the goals of the Smart Memories architecture is to efficiently execute applications with a wide range of programming models and types of parallelism. In the early stages of the project, we could not feasibly create, analyze, and map a large number of applications directly onto our architecture, yet we needed to evaluate its potential to span disparate applications classes. Clearly the memory system was general enough to allow changing the sizes and characteristics of the caches in the system as well as to implement other memory structures. However, this is really only part of what we need to support different computation models. To provide some concrete benchmarks, we configured a Smart Memories machine to mimic two existing machines, the Hydra multiprocessor [33] and the Imagine streaming processor [25]. These two machines, on far ends of the architectural spectrum, require very different memory systems and arrangement of compute resources. We then used applications for these base machines to provide feedback on the potential performance of Smart Memories. These results are likely to be pessimistic since the applications were optimized for the existing architecture machine and not for the Smart Memories target machine.

Imagine is a highly-tuned SIMD/vector machine optimized for media applications with large amounts of data parallelism. In these machines, local memory access is very regular, and computation is almost completely scheduled by the compiler. After looking at Imagine, we will explore the performance of Hydra, a single chip 4-way multiprocessor. This machine is very different from Imagine, because the applications that it supports have irregular accesses and communication patterns. To improve performance of these applications the machine supports speculative thread execution. This requires a number of special memory structures and tests the flexibility of the memory system.

4.1 Mapping Imagine

Imagine is a co-processor optimized for high-performance on applications that can be effectively encapsulated in a stream programming model. This model expresses an application as a sequence of kernels that operate on long vectors of records, referred to as streams. Streams are typically accessed in predictable patterns and are tolerant of fetch latency. However, streaming applications demand high bandwidth to stream data and are compute-intensive. Imagine provides a bandwidth hierarchy and a large number of arithmetic units to meet these requirements.

The Imagine bandwidth hierarchy consists of off-chip DRAM, an on-chip stream register file (SRF), and local register files (LRFs) in the datapath. The SRF and LRFs provide increasing bandwidth and allow temporary storage, resulting in reduced bandwidth demands on the levels further away in the hierarchy. The SRF is a 64KB multi-banked SRAM accessed via a single wide port. Streams are stored in the SRF in the order they will be accessed, yielding high bandwidth via the single port. The records of a stream are inter-

leaved among the banks of the SRF. The LRF level consists of many small register files directly feeding the arithmetic units.

The high stream bandwidth achieved through the storage hierarchy enables parallel computation on a large number of arithmetic units. In Imagine, these units are arranged into eight clusters, each associated with a bank of the SRF. Arithmetic resources of a cluster are made up of three adders, two multipliers, and one divide/square-root unit. The eight clusters exploit data parallelism to perform the same set of operations on different records of a stream in parallel. Within each cluster, ILP is exploited to perform parallel computations on the different units. All the clusters execute a single micro-code instruction stream in lock-step, resulting in a single-instruction multiple-data (SIMD) system.

For this study, we map the SRF and LRF levels of Imagine along with its compute resources to the Smart Memories substrate. The arrangement of these resources in Imagine is shown in Figure 8. The LRFs are embedded in the compute clusters and are not shown explicitly.

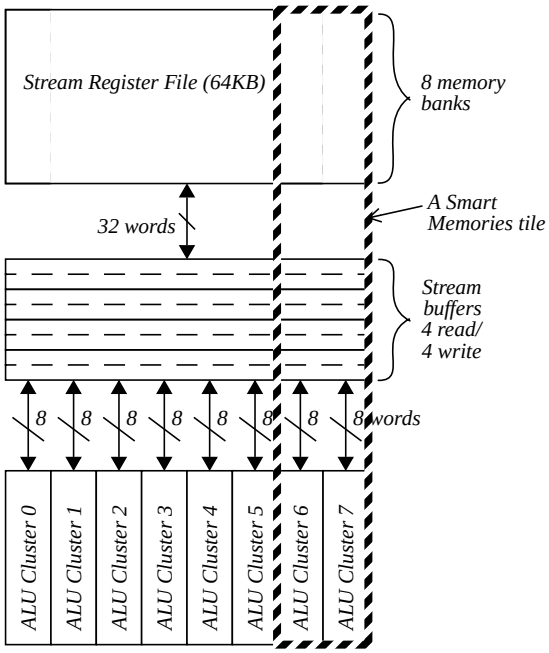

Figure 8. Imagine architecture

The 8-cluster Imagine is mapped to a 4-tile Smart Memories quad. Exploiting the SIMD execution of Imagine clusters, each of the 64-bit Smart Memories datapaths emulate two 32-bit Imagine clusters in parallel. Like Imagine, the mapped implementation is intended to be a co-processor under the control of an off-quad host. In the following sections, we describe the mapping of Imagine to the Smart Memories, the differences between the mapping and Imagine, and the impact on performance.

4.1.1 Mapping the bandwidth hierarchy

In mapping Imagine to Smart Memories, we configure all the memory mats on the tiles as streaming and scratchpad memories. Most of the mats are allocated to the SRF and are configured in streaming mode as described in Section 3.1. Data structures that cannot be streamed, such as lookup tables, are allocated in mats configured as scratchpad memories. Instructions are stored in mats with the decoders configured for explicit indexed addressing. The homogeneity of the Smart Memories memory structure allows the allocation of resources to the SRF and scratchpad to be determined based on the capacity and bandwidth requirements of each on a per-application basis. The LRFs of Imagine map to the almost identical LRF structure of the Smart Memories datapath.

The SRF is physically distributed over the four tiles of a quad, with a total SRF capacity of up to 480KB. Records of a stream are interleaved among the tiles, each active stream occupying the same mat on every one of the four tiles, and different streams occupying different mats. Multiple streams may be placed on non-overlapping address ranges of the same mat at the cost of reduced bandwidth to each stream. This placement allows accesses to a mat to be sequential and accesses to different streams to proceed in parallel. The peak bandwidth available at each level of the hierarchy in Imagine and the mapping is summarized in Table 1. The mapping can sustain bandwidth per functional unit comparable to Imagine at both the SRF and LRF levels.

Level of hierarchy	Imagine cluster	SM FP cluster	Imagine FP unit	SM FP arith unit
SRF BW, sustainable	4	4	0.67	1
SRF BW, burst	8	4	1.33	1
LRF BW	34	23	5.66	5.75

TABLE 1. Comparison of peak BW in words per cycle

4.1.2 Mapping the computation

In the Smart Memories datapath, the majority of computations are performed in the FP cluster where the bandwidth to sustain parallel computation is provided by the LRFs and result buses. Microcode instructions are used to issue operations to all FP units in parallel. The integer units of Smart Memories tiles are used primarily to perform support functions such as scratchpad accesses, inter-tile communication, and control flow operations which are handled by dedicated units in Imagine.

4.1.3 Mapping off-tile communication

Much of the data bandwidth required in stream computations is to local tile memory. However, data dependencies across loop iterations require communication among tiles within the quad. In the mapping, these communications take place over the quad network. Since we emulate two 32-bit Imagine clusters on a tile, the quad network is configured as a half-word network to allow any communication pattern among the eight mapped clusters without incurring a serialization penalty.

Streams that generate or consume data based on run-time conditions require dynamic communication to distribute records among all or a subset of the compute clusters. The communication pattern for these dynamic events, generated by dedicated hardware in Imagine, is determined by a table lookup in the Smart Memories mapping. The broadcast control bits in the Smart Memories quad network distribute status information indicating participation of each cluster in an upcoming communication. These bits combine with state information from previous communications to form the index into the lookup-table.

Gather and scatter of stream data between the SRF and off-quad DRAM, fetch of microcode into the local store, communication with the host processor, and communication with other quads are performed over the global network. The first or final stage of these transfers also utilizes the quad network but receives a lower priority than intra-quad communications.

4.1.4 Evaluation of the Imagine Mapping

To evaluate the performance of the mapping, we conducted cycle-accurate simulations of four kernels by adapting the Imagine compilation and simulation tools. The simulations accounted for all differences between Imagine and the mapping, including the hardware resource differences, the overheads incurred in software emulation of certain hardware functions of Imagine, and serialization penalties incurred in emulating two Imagine clusters on a tile. When an aspect of the mapping could not be modeled exactly using the Imagine tools, we modeled the worst-case scenario. Latencies of 32-bit arithmetic operations were assumed to be the same for both architectures since their cycle times are comparable in gate delays in their respective target technologies. The kernels simulated - a 1024-point FFT, a 13-tap FIR filter, a 7x7 convolution, and an 8x8 DCT - were optimized for Imagine and were not re-optimized for the Smart Memories architecture.

Simulations show that none of the observed kernels suffer a slowdown due to inadequacy of the available SRF bandwidth of four accesses per cycle. However, constraints other than SRF bandwidth lead to performance losses. Figure 9 shows the percentage performance degradation for the four kernels on the mapping relative to Imagine. These performance losses arise due to the constraints discussed below.

Reduced unit mix

The Smart Memories FP cluster consists of two fewer units (an adder and a multiplier) than an Imagine cluster, which leads to a significant slowdown for some compute bound kernels (e.g. *convolve*). Simulations show that simply adding a second multiplier with no increase in memory or communication bandwidth reduces the performance degradation relative to Imagine for *convolve* from 82% to 7%. We are currently exploring ways to increase the compute power of the Smart Memories tile without significantly increasing the area devoted to arithmetic units.

Bandwidth constraints (within a tile)

In the Smart Memories datapath, communication between the FP and integer units and memory/network ports takes place over a limited number of buses. This contrasts with a full crossbar in Imagine for the same purpose, leading to a relative slowdown for the mapping.

Longer latencies

The routed, general interconnects, used for data transfers outside of compute clusters in the Smart Memories architecture, typically have longer latencies compared to the dedicated communication resources of Imagine. While most kernels are tolerant of stream access latencies, some that perform scratchpad accesses or inter-cluster communications are sensitive to the latency of these operations (e.g. *fir*). However, heavy communication does not necessarily lead to significant slowdowns if the latency can be masked through proper scheduling (e.g. *fft*). Other causes of latency increases include the overheads of emulating certain functions in software in the mapping, and serialization delays due to emulating two clusters on a single tile.

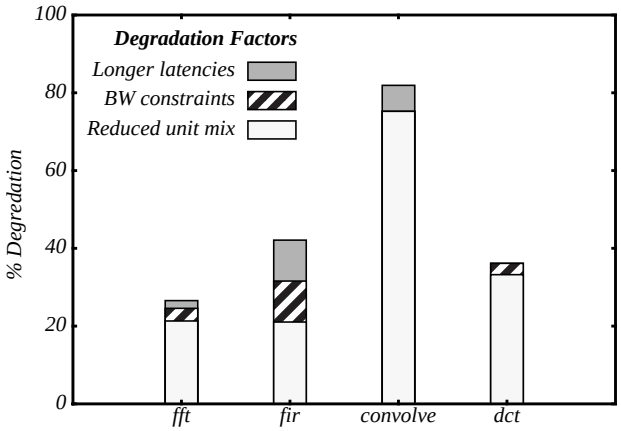

Figure 9. Performance degradation

According to simulation results, the bandwidth hierarchy of the mapping compares well with that of the original Imagine and provides the necessary bandwidth. However, constraints primarily in the compute engines and communication resources lead to an overall performance loss. The increase in run-time over Imagine is moderate: 47% on average and within a factor of two for all the kernels considered. These results demonstrate that the configurable substrate of Smart Memories, particularly the memory system, can sustain performance within a small factor of what a specialized streaming processor achieves.

4.2 Mapping Hydra

The Hydra speculative multiprocessor enables code from a sequential machine to be run on a parallel machine without requiring the code to be re-written [34][35]. A pre-processing script finds and marks loops in the original code. At run-time, different loop iterations from the marked loops are then speculatively distributed across all processors.

The Hydra multiprocessor hardware controls data dependencies across multiple threads at run-time, thereby relaxing the burden on the compiler and permitting more aggressive parallelization. As shown in Figure 10, the Hydra multiprocessor consists of four RISC processors, a shared on-die L2, and speculative buffers which are interconnected by a 256-bit read bus and a 64-bit write-through bus. The speculative buffers store writes made by a processor during speculative operation to prevent potentially invalid data from corrupting the L2. When a processor commits state, this modified data is written to the L2. The read bus handles L2 accesses and fills from the external memory interface while the write-through bus is used to implement a simple cache-coherence scheme. All processors snoop on the write-through bus for potential RAW violations and other speculative hazards.

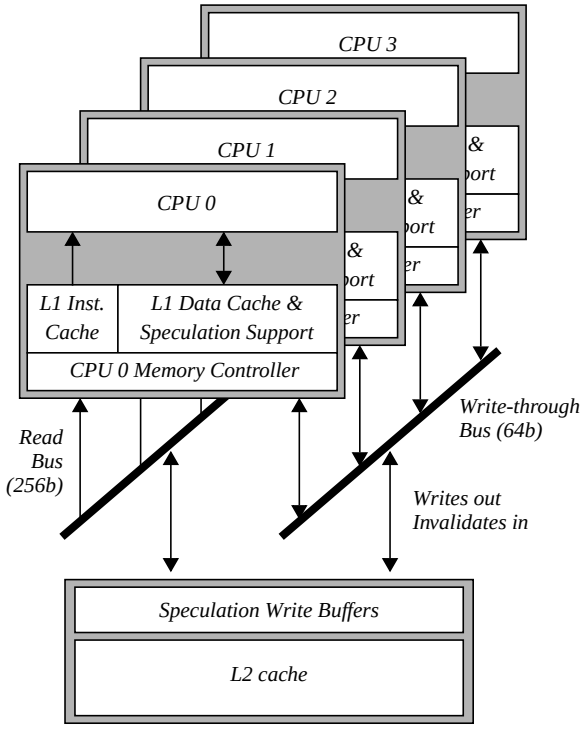

Figure 10. Hydra architecture

When a speculative processor receives a less-speculative write to a memory address that it has read (RAW hazard), a handler invalidates modified lines in its cache, restarts its loop iteration, and notifies all more-speculative processors that they must also restart. When the head (non-speculative) processor commits, it begins work on a thread four loop iterations from its current position and notifies all speculative processors that they must update their speculative rank.

During the course of mapping Hydra we found that performance degradation was introduced through three factors: memory configuration limitations, algorithmic simplifications, and increases in memory access time. Similar to the approach taken with Imagine, we conducted cycle-level simulations by adapting the Hydra simulation environment [35] to reflect the Smart Memories tile and quad architecture.

4.2.1 Memory configuration

In the Smart Memories implementation of Hydra, each Hydra processor and its associated L1 caches reside on a tile. The L2 cache and speculative write buffers are distributed among the four tiles that form a quad. Figure 11 shows the memory mat allocation of a single tile. The dual-ported mats are used to support three types of memory structures: efficient set-associative tags, tags that support snooping, and arbitration-simplifying mats.

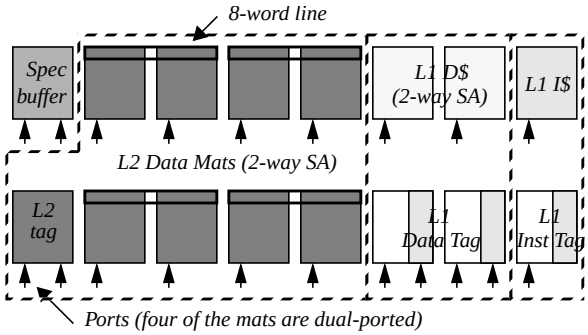

Figure 11. Hydra's tile memory mat allocation

One quarter of the L2 resides on each tile. The L2 is split by address, so a portion of each way is on each tile. Rather than dedicate two mats, one for each way, for the L2 tags, a single dual-ported mat is used. Placing both ways on the same tile reduces the communication overhead. Single-ported memories may be efficiently used as tag mats for large caches, but they inefficiently implement tags for small caches. For example, the L1 data tags are not completely utilized because the tags only fill 2KB. The L1 data tags are dual-ported to facilitate snooping on the write bus under the write-through coherence protocol.

Finally, dual-ported mats are used to simplify arbitration between two requestors. The CAM (not shown) stores indices which point into the speculation buffer mat, which holds data created by a speculative thread. Data may be written to this mat by the tile's processor and then read by a more speculative processor on an L1 miss at the same time. In this case, the dual-ported mat avoids complex buffering and arbitration schemes by allowing both requestors to simultaneously access the mat.

The Smart Memories memory mats architecture causes certain aspects of the mapping's memory configuration to differ from those of the Hydra baseline [36], as detailed in Table 2. Compared to Hydra, the Smart Memories configuration uses lower set-associativity in the L2 and L1 instruction caches to maximize the memory mat utilization. The performance degradation due to lower associativity is at most 6% as shown in Figure 12.

	L1 Cache Hydra	L1 Cache SM	L2 Cache Hydra	L2 Cache SM
Configuration	Separate I&D caches	Separate I&D caches	Central-ized	Distrib-uted
Word Size	32b	64b	32b	64b
Capacity	2K words	2K words	32K words	32K words
Associativity	2-way (D, I)	2-way (D) 1-way (I)	8-way	2-way
Line Size	4 words/ line	4 words/ line	8 words/ line	8 words/ line
Access Time	1-cycle	2-cycles	4-cycles	7-cycles
# Control Bits/ Word	tag: 3 data: 2	tag: 5 data: 2	tag: 2	tag: 2

TABLE 2. Memory configuration comparison

4.2.2 Algorithmic modifications

Algorithmic modifications were necessary, since certain Hydra-specific hardware structures were not available. This section presents two examples and their performance impact.

Conditional gang-invalidation

On a restart, Hydra removes speculatively modified cache lines in parallel through a conditional gang-invalidation if the appropriate control bit of the line is set. This mechanism keeps unmodified lines in the cache as opposed to clearing the entire cache, thus improving the L1 hit rate. Although the conditional gang-invalidation mechanism is found in other speculative architectures, such as the Speculative Versioning Cache [37], it is not commonly used in other architectures and introduces additional transistors to the SRAM memory cell. Therefore, in the Smart Memories mapping, algorithmic modifications are made so the control bits in the L1 tag are not conditionally gang-invalidated.

Under Hydra's conditional gang-invalidation scheme, lines introduced during speculation are marked as valid lines and are invalidated when a thread restarts. In the Smart Memories configuration, lines introduced during speculation are valid for a specified time period and are only permanently marked valid if they are accessed before the processor's next assigned thread commits. Simulations show that this alternative to conditional gang-invalidation decreases performance by up to 12% and requires two extra bits in the tag.

L2 Merge

In Hydra, the L2 and speculative buffers are centrally located, and on an L1 miss, a hardware priority encoder returns a merged line. Data is collected from the L2 and less speculative buffers on a word-by-word basis where the more recent data has priority. However, in Smart Memories the L2 and speculative buffers are distributed. If a full merge of all less-speculative buffers and the L2 is performed, a large amount of data is unnecessarily broadcast across the quad network.

Simulations show that most of the data comes from either the L2 or the nearest less-speculative processor on an L1 miss. Therefore, the L2 merge bandwidth is reduced by only reading data from the L2 and the nearest less-speculative processor's speculative write buffer. Neglecting the different L2 latency under the Smart Memories memory system leads to a performance degradation of up to 25%. The performance degradation is caused by a small number of threads which are restarted when they read the incorrect data on an L2 access.

4.2.3 Access Times

The memory access times in the Smart Memories mapping are larger due to two factors: crossbar delay and delay due to distributed resources. Hydra has a 1-cycle L1 access and a 4-cycle L2 merge, while the Smart Memories configuration has a 2-cycle L1 access and 7-cycle L2 merge. The delay through the crossbar affects the L1 access time, and since the L2 is distributed, the L2 merge time is increased. The 2-cycle load delay slot is conservatively modeled in our simulations by inserting nops without code rescheduling; the resulting performance degradation is up to 14%.

The increased L2 access time has a greater impact on performance than the L1 access time and causes performance degradations greater than 40% on the *m88ksim* and *wc* benchmarks. The performance degradations on the other benchmarks are less than 25%. The increase in the L2 access time is due to the additional nearest-neighbor access on the quad interconnect.

4.2.4 Simulation results

Figure 12 shows the performance degradations caused by the choice of memory configurations, algorithms, and memory access latency. The memory access latency and algorithmic changes contribute the greatest amount of performance degradation, whereas the configuration changes are relatively insignificant. Since the Hydra processors pass data through the L2, the increased L2 latency in Smart Memories damages performance the most for benchmarks that have large amounts of communication between loop iterations, such as *compress*, *m88ksim*, and *wc*.

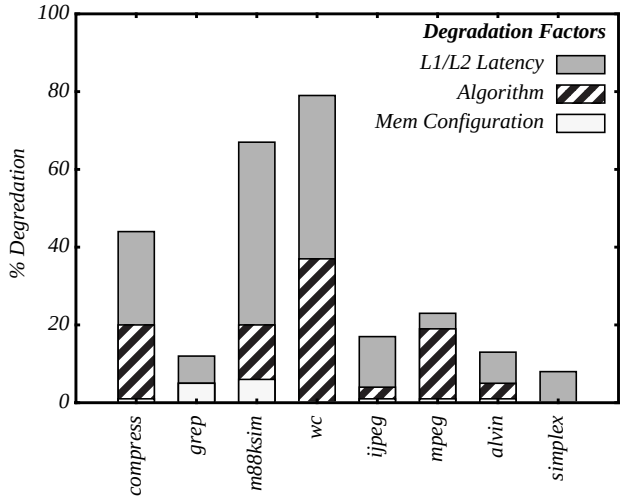

Figure 12. Performance degradation

In Figure 13, the Smart Memories and Hydra speedups are calculated by dividing the execution time of one of the processors in Hydra by the respective execution times of the Smart Memories and Hydra architectures. Scalar benchmarks, *m88ksim* and *wc*, have the largest performance degradations and may actually slow down under the Smart Memories configuration. Since Hydra does not achieve significant speedup on these benchmarks, they should not be run on this configuration of Smart Memories. For example, we would achieve higher performance on the *wc* benchmark if we devoted more tile memory to a larger L1 cache.

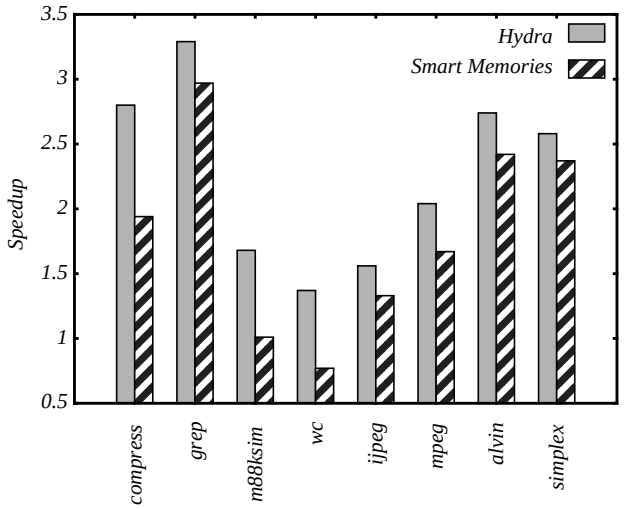

Figure 13. Speedup

5. Conclusion

Continued technology scaling causes a dilemma -- while computation gets cheaper, the design of computing devices becomes more expensive, so new computing devices must have large markets to be successful. Smart Memories addresses this issue by extending the notion of a program. In conventional computing systems the memories and interconnect between the processors and memories is fixed, and what the programmer modifies is the code that runs on the processor. While this model is completely general, for many applications it is not very efficient. In Smart Memories, the user can program the wires and the memory, as well as the processors. This allows the user to configure the computing substrate to better match the structure of the applications, which greatly increases the efficiency of the resulting solution.

Our initial tile architecture shows the potential of this approach. Using the same resources normally found in a superscalar processor, we were able to arrange those resources into two very different types of compute engines. One is optimized for stream-based applications, *i.e.* very regular applications with large amounts of data parallelism. In this machine organization, the tile provides very high bandwidth and high computational throughput. The other engine was optimized for applications with small amounts of parallelism and irregular memory access patterns. Here the programmability of the memory was used to create the specialized memory structures needed to support speculation.

However, this flexibility comes at a cost. The overheads of the coarse-grain configuration that Smart Memories uses, although modest, are not negligible; and as the mapping studies show, building a machine optimized for a specific application will always be faster than configuring a general machine for that task. Yet the results are promising, since the overheads and resulting difference in performance are not large. So if an application or set of applications needs more than one computing or memory model, our reconfigurable architecture can exceed the efficiency and performance of existing separate solutions. Our next step is to create a more complete simulation environment to look at the overall performance of some complete applications and to investigate the architecture for inter-tile interactions.

6. Acknowledgments

We would like to thank Scott Rixner, Peter Mattson, and the other members of the Imagine team for their help in preparing the Imagine mapping. We would also like to thank Lance Hammond and Kunle Olukotun for their help in preparing the Hydra mapping. Finally we would like to thank Vicky Wong and Andrew Chang for their insightful comments.

7. References

[1] K. Diefendorff and P. Dubey. How Multimedia Workloads Will Change Processor Design. *IEEE Computer*, pages 43-45, Sept. 1997.

[2] G. A. Slavenberg, *et al*. The Trimedia TM-1 PCI VLIW Media Processor. In *Proceedings of Hot Chips 8*, 1996.

[3] L. Lucas. High Speed Low Cost TM1300 Trimedia Enhanced PCI VLIW Mediaprocessor. In *Proceedings of Hotchips 11*, pages 111-120, Aug. 1999.

[4] J. O'Donnell. MAP1000A: A 5W, 230MHz VLIW Mediaprocessor. In *Proceedings of Hot Chips 11*, pages 95-109, Aug. 1999.

[5] P. Kalapathy. Hardware-Software Interactions on MPACT. *IEEE Micro*, pages 20-26, Mar. 1997.

[6] C. Kozyrakis, *et al*. Scalable Processors in the Billion-transistor Era: IRAM. *IEEE Computer*, pages 75-78, Sept. 1997.

[7] M. Horowitz, *et al*. The Future of Wires. SRC White Paper: Interconnect Technology Beyond the Roadmap, 1999 (http://www.src.org/cgi-bin/deliver.cgi/sarawp.pdf?/areas/nis/sarawp.pdf).

[8] D. Matzke, *et al*. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, pages 37-9, Sept. 1997.

[9] C. Kaplinsky. A New Microsystem Architecture for the Internet Era. Presented in Microprocessor Forum, Oct. 1999.

[10] E. Waingold, *et al*. Baring It All to Software: Raw Machines. *IEEE Computer*, pages 86-93, Sept. 1997.

[11] S. Hauck, *et al*. The Chimaera Reconfigurable Functional Unit. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 87-96, Apr. 1997.

[12] R. Wittig, *et al*. OneChip: an FPGA Processor with Reconfigurable Logic. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126-35, Apr. 1996.

[13] J. Hauser, *et al*. Garp: a MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12-21, Apr. 1997.

[14] H. Zhang, *et al.* A 1V Heterogeneous Reconfigurable Processor IC for Baseband Wireless Applications. In *Digest of Technical Papers ISSCC 2000*, pages 68-69, Feb. 2000.

[15] H. Kim, *et al.* A Reconfigurable Multi-function Computing Cache Architecture. In *Eighth ACM International Symposium on Field-Programmable Gate Arrays*, Feb. 2000.

[16] E. Mirsky, *et al*. MATRIX: a Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157-66, Apr. 1996.

[17] A. DeHon. DPGA Utilization and Application. In *Proceedings of the 1996 International Symposium on FPGAs*, Feb. 1996.

[18] A. DeHon. Trends Towards Spatial Computing Architectures. In *Proceedings of the International Solid-State Circuits Conference*, pages 362-63, Feb. 1999.

[19] R5000 Improves FP for MIPS Midrange. *Microprocessor Report*, Jan. 22 1996.

[20] R. B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, pages 51-59, July/Aug 1996.

[21] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, pages 42-50, July/Aug 1996.

[22] A. Peleg, *et al*. Intel MMX for Multimedia PCs. *Comm. ACM*, pages 25-38, Jan. 1997.

[23] MIPS Digital Media Extension. Instruction Set Architecture Specification, http://www.mips.com/MDMXspec.ps (current Oct. 21, 1997)

[24] M. Tremblay, *et. al*. VIS Speeds New Media Processing. *IEEE Micro*, pages 10-29, July/Aug., 1996

[25] S. Rixner, *et al*. A Bandwidth-Efficient Architecture for Media Processing. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 3-13, Nov.-Dec. 1998.

[26] R. Sites. It's the Memory, Stupid! *Microprocessor Report*, pages 19-20, August 5, 1996.

[27] J. Williams, *et al.* A 3.2GOPS Microprocessor DSP for Communication Applications. In *Digest of Technical Papers ISSCC 2000*, pages 70-71, Feb. 2000.

[28] B. Amrutur. Design and Analysis of Fast Low Power SRAMs. Ph.D. Thesis, Stanford University, Aug. 1999.

[29] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 364-73, May 1990.

[30] S. Palacharla, *et al*. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24-33, 1994.

[31] K. Farkas, *et al*. How Useful Are Non-blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors? In *Proceedings of the First International Conference on High Performance Computer Architecture*, pages 78-89, Jan. 1995.

[32] S. Rixner, *et al*. Register Organization for Media Processing. To appear in 6th International Symposium on High-Performance Computer Architecture.

[33] L. Hammond, *et al*. A Single-chip Multiprocessor. *IEEE Computer*, pages 79-85, Sept. '97.

[34] L. Hammond, *et al*. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS VIII), pages 58-69, Oct. 1998.

[35] K. Olukuton, *et al*. Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, June 1999.

[36] L. Hammond. The Stanford Hydra Chip. In *Proceedings of Hot Chips 11*, pages 23-31, Aug. 1999.

[37] S. Gopal, *et al*. Speculative Versioning Cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture* (HPCA-4), Feb. 1998.