RaPiD – A Configurable Computing Architecture for Compute-Intensive Applications

Carl Ebeling, Darren C. Cronquist,
Paul Franklin and Chris Fisher

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Technical Report UW-CSE-96-11-03
November, 1996

RaPiD - A Configurable Computing Architecture for Compute-Intensive Applications*

Carl Ebeling, Darren C. Cronquist, Paul Franklin and Chris Fisher

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Abstract

Configurable computers have attracted considerable attention recently because they promise to deliver the performance of application-specific hardware along with the flexibility of general-purpose computers. Unfortunately, configurable computing has had rather limited success to date. We believe that the FPGAs currently used to construct configurable computers are too general to achieve good cost-performance on computationally-intensive applications that demand special-purpose hardware. This paper describes a new architecture called RaPiD (Reconfigurable Pipelined Datapaths), which is optimized for highly repetitive, computationally-intensive tasks. Very deep application-specific computation pipelines can be configured in RaPiD that deliver very high performance for a wide range of applications. RaPiD achieves this using a coarse-grained reconfigurable architecture that mixes the appropriate amount of static configuration with dynamic control.

1 Introduction

Special-purpose architectures have long been used to achieve higher performance at a lower cost than general-purpose processors. But as general-purpose processors have become faster and cheaper, special-purpose architectures have been relegated to a shrinking number of niche applications. By definition, an application-specific architecture speeds up only one application. This inflexibility combined with a high design cost make them unattractive except for very well-defined and wide-spread applications like video processing and graphics.

Configurable computing promises to reverse this trend. The goal of configurable computing is to achieve most of the performance of custom architectures while retaining most of the flexibility of general-purpose computing. This is done by dynamically constructing a custom architecture from an underlying substrate of configurable circuitry[1]. Although the concept of configurable computing is very attractive, success has been remarkably hard to achieve in practice.

Current custom computing machines [2, 13] are constructed from field-programmable gate arrays (FPGAs) [11]. These contain logic blocks which can be configured to compute arbitrary functions, and configurable wiring which can be used to connect the logic blocks as well as registers together into arbitrary circuits. The information that configures an FPGA can be changed quickly so that a single FPGA can implement different circuits at different times. FPGAs thus appear to be ideally suited to configurable computing. Unfortunately, the fine-grained circuit structure which makes them so general has a very high cost: Depending on the circuit being constructed, this cost-performance penalty can range from a factor of 20 for random logic to well over 100 for structured circuits like ALUs, multipliers and memory [6]. Thus, custom computing based on FPGAs is unlikely to compete on applications that involve heavy arithmetic computation.

If we are to achieve good performance for computation-intensive applications, we will need new configurable computing architectures that can ef-

*This work was supported in part by the Defense Advanced Research Projects Agency under Contract DAAH04-94-G0272. D. Cronquist was supported in part by an IBM fellowship. P. Franklin was supported by an NSF fellowship.

[1] Configurable computing is also used to refer to other architecture styles, e.g. dynamically constructing custom functional units in a general-purpose processor similar to defining custom instructions using writable control store [14]. Our research is focused on traditional application-specific architectures.

ficiently implement these arithmetic computations. Our response to this challenge is RaPiD, a coarse-grained configurable architecture for constructing deep computational pipelines. RaPiD is aimed at regular, computation-intensive tasks like those found in digital signal processing (DSP). RaPiD provides a large number of ALUs, multipliers, registers and memory modules that can be configured into the appropriate pipelined datapath. The datapaths constructed in RaPiD are linear arrays of functional units communicating in mostly nearest-neighbor fashion. Systolic algorithms [7], for example, map very well into RaPiD datapaths, allowing us to take advantage of the considerable research on compiling computations to systolic arrays [9, 8]. RaPiD is not limited to implementing systolic algorithms, however; a pipeline can be constructed which comprises different computations at different stages and at different times.

Although the computation bandwidth scales with the size of a RaPiD array, the external memory bandwidth does not, and the amount of computation performed per I/O operation bounds the amount of parallelism that can be attained. RaPiD limits applications to at most two reads and one write per cycle, which we have found to be both necessary and sufficient for the applications we have programmed. Providing even this much bandwidth requires a high-performance memory architecture.

We begin by describing the RaPiD architecture in more detail and then follow with an example of using RaPiD for matrix multiply. This illustrates the need for dynamic datapath control, which is then described in detail. Next we outline a system implementation based on RaPiD and use this to present performance results for a number of RaPiD applications. These results are used to compare RaPiD to some alternatives, notably microprocessors and DSPs. Finally, we conclude with some observations and a discussion of future directions for the RaPiD architecture.

2 RaPiD Datapath Architecture

Our description of the RaPiD architecture will be given in general terms. Implementations of RaPiD will vary according to a number of parameters including data width and data format, number and type of functional units, and number and configuration of busses. The description and examples given here are based on the RaPiD-1 implementation currently being developed.

RaPiD is a linear array of functional units which is configured to form a mostly linear computational pipeline. This array of functional units is divided into identical cells which are replicated to form a complete array. An example cell is shown in Figure 1. This cell comprises an integer multiplier, three integer ALUs, six general-purpose "datapath registers" and three small local memories. A typical single-chip RaPiD array would contain between 8 and 32 of these cells. Although the array is divided into cells, this division is invisible when mapping an application pipeline to the functional units and busses.

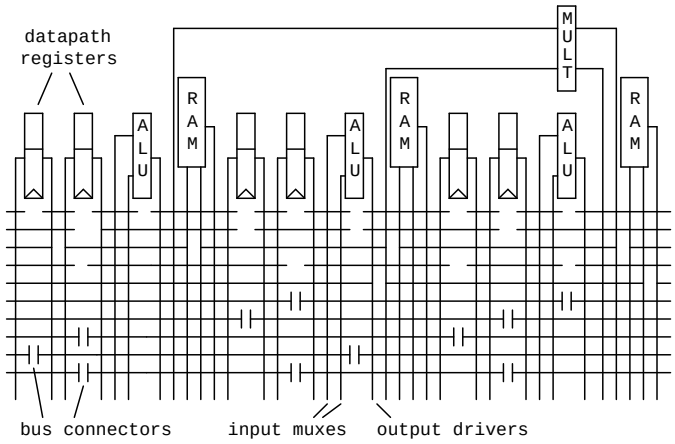

Figure 1: *The basic cell of RaPiD-1 which is replicated left to right 16 times to form a complete RaPiD-1 chip. All busses are 16 bits.*

The functional units are interconnected using a set of segmented busses that run the length of the datapath. The functional units use a multiplexer to select their data inputs from one of the bus segments and a set of tristate drivers to drive their output onto one or more bus segments. Each functional unit output also includes an optional pipeline register. The input multiplexer provides a constant zero and a feedback from the function output. These are used, for example, to implement clear and hold operations for registers, or to configure an ALU as an accumulator in conjunction with the output pipeline register. Operations are assigned to functional units so that bus segments are available to connect units that communicate.

The busses in different tracks are segmented into different lengths to make the most efficient use of the busses. In some tracks, adjacent bus segments can be connected together by a bus connector as shown in Figure 1. This connection can be programmed in either direction via a uni-directional buffer or pipelined with up to three register delays, which allows data pipelines to built in the bus structure itself.

RaPiD operates on signed or unsigned fixed-point 16-bit data which is maintained via shifters in the

multipliers[2]. Different fixed-point representations can be used in the same application by appropriately configuring the shifters in the datapath. An extra tag bit is associated with each data value to indicate whether an overflow has occurred. Once set, the overflow value is propagated to all results. The datapath thus generates no exceptions during operation but incorporates them into the data produced. The tag bit can be programmed for other purposes as well.

The ALUs perform the usual logical and arithmetic operations on 16-bit data. Any two ALUs in a cell can be combined to perform a pipelined 32-bit operation, most typically as a 32-bit add for multiply-accumulate operations. The multiplier inputs two 16-bit numbers and produces a 32-bit result, shifted by a statically programmed amount to maintain the appropriate fixed-point representation. Both 16-bit halves of the result are available as output.

The registers in the datapath are used to store constants and temporary values. A datapath register can be bypassed to connect a bus segment on one track to a bus segment in a different track.

In many applications, the data is segmented into blocks that are accessed once, saved locally and reused as needed, and then discarded. The local memories provided in each cell of the datapath serve this purpose. Each local memory has an address register which acts like a datapath register except for an additional input which provides the self-increment operation. This supports the most common case of simple sequential memory access, but more complex addressing patterns can be accomplished using registers and ALUs in the datapath. During each cycle, a read followed by an optional write is performed to the addressed location. This is not used for read-modify-write operations because the latency would be too long, but rather to implement configurable-length pipeline delays similar to the SILOs found in the Philips VSP [12].

Input and output data enter and exit RaPiD via I/O streams at each end of the datapath. Each stream contains a FIFO filled with data required or with results produced by the computation. The datapath explicitly reads from an input stream to obtain the next input data value and writes to an output stream to store a result. The data for each stream is associated with a predetermined block of memory, and address generation is performed by the I/O streams themselves. The I/O stream FIFOs operate asynchronously: If the datapath reads a value from an empty FIFO or writes a value to a full FIFO, the datapath is stalled until the FIFO is ready.

[2]The data length and format, i.e. fixed vs. floating point, is an implementation parameter. We will assume 16-bit fixed point in this paper.

3 Example: Matrix Multiply

An example exhibiting significant speedup on RaPiD is matrix multiply. Since each base operation performs a multiply-accumulate (MAC), an upper bound on the achievable parallelism is the number of multipliers in the RaPiD array. An ideal mapping achieves the highest utilization of multipliers while minimizing the number of memory accesses per cycle. To generalize results on different RaPiD architectures, we define $RAMsize$ as the number of entries per RAM and $NumCells$ as the number of cells on the RaPiD array. We also assume that a "cell" is a slice of the RaPiD array containing exactly one multiplier, three ALUs, and three RAMs, as shown in Figure 1. RaPiD-1 has $RAMsize = 32$ and $NumCells = 16$.

In this section, we consider multiplying an $L \times M$ matrix $\mathbf{A}$ by an $M \times N$ matrix $\mathbf{B}$ yielding an $L \times N$ matrix $\mathbf{C}$. For illustrative purposes, we first analyze the restricted case of $M \leq RAMsize$ and $N \leq NumCells$ and then extend to matrices of unlimited size. Both of these use a standard systolic approach for implementing matrix multiply.

3.1 Restricted Case

The assumption that $M \leq RAMsize$ and $N \leq NumCells$ allows the entire $\mathbf{B}$ matrix to be stored within the RaPiD RAMs, one column per cell. One cell of the resulting pipeline is configured as shown in Figure 2. The $\mathbf{A}$ matrix is passed through the array in row-major order. Within each cell, the RAM address

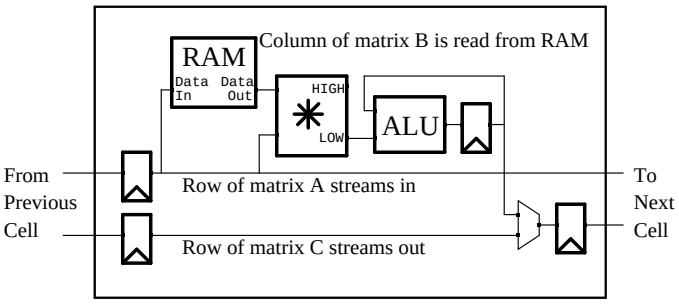

Figure 2: *Netlist for one cell of restricted matrix multiply. The top pipelined bus streams in the* $\mathbf{A}$ *matrix while the bottom bus streams out the resulting* $\mathbf{C}$ *matrix. The top bus also streams the* $\mathbf{B}$ *columns into the RAMs prior to the computation.*

is incremented each cycle, and a register accumulates

the dot product of the stored column and the incoming row. When a cell receives the last element of a row, the resulting product is passed down an output pipeline, the RAM address is cleared, and the cell is ready to compute the product of the next row on the next cycle.

3.2 Unlimited Case

To multiply matrices of unlimited size, the matrices are first tiled according to the available memory and the number of cells. Consider a tiling of the **A** and **B** matrices producing **A** sub-matrices of size $Q \times R$ and **B** sub-matrices of size $R \times S$, as shown in Figure 3. The values of R and S are chosen such that an entire **B** sub-matrix can fit within the RaPiD RAMs, one column per cell (i.e. $R \leq RAMsize$ and $S \leq NumCells$). Then an **A** sub-matrix can be multiplied with a **B** sub-matrix in the same manner as shown in Section 3.1, producing QS intermediate results of a **C** sub-matrix. The value of Q is chosen such that this sub-matrix of partial results can be completely stored within the RaPiD RAMs, one column per cell (i.e. $Q \leq RAMsize$). After a row of **A** sub-matrices is multiplied with a column of **B** sub-matrices, the **C** sub-matrix results are passed down the output pipeline. Since each sub-matrix multiply requires a new **B** sub-matrix to be loaded prior to computation, an extra RAM is used to preload the next **B** sub-matrix while the current sub-matrix multiply is being computed (as illustrated with different shades in Figure 3). One cell of the RaPiD pipeline implementing unrestricted matrix multiply is shown in Figure 4. We want to emphasize that even though the preloading of the next **B** sub-matrix and the computation of the current sub-matrix multiply occur in parallel, RaPiD implements these in a strict lock step manner. Hence, if one operation's request for memory fails, the entire RaPiD array freezes, which effectively stalls not only the operation requesting the unavailable memory but also all independent operations that could have otherwise executed. Although this may seem overly restrictive, leaving the lock step model would require a more complex form of process communication (such as handshaking) potentially leading to significant delays.

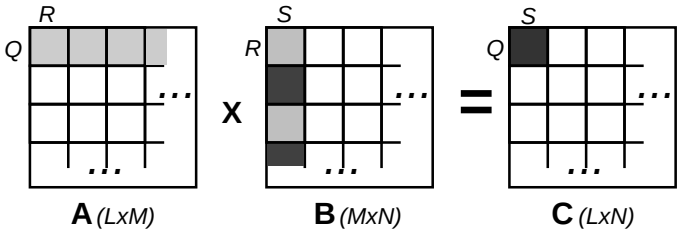

Figure 3: *RaPiD partitions matrices* **A**, **B**, *and* **C** *into tiles of size* $Q \times R$, $R \times S$, *and* $Q \times S$, *respectively. The product of a row of* **A** *sub-matrices and a column of* **B** *sub-matrices produces a single sub-matrix of* **C**.

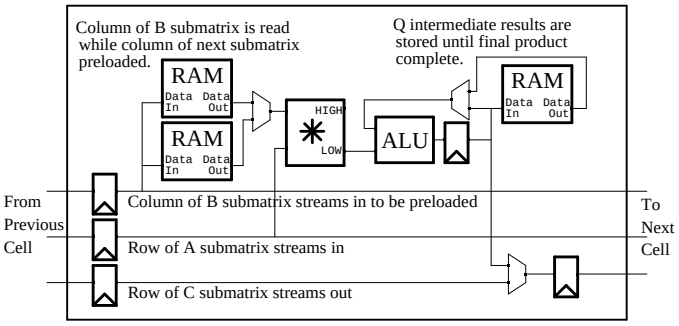

Figure 4: *Netlist for one cell of unrestricted matrix multiply. The top pipelined bus streams in the* next **B** *sub-matrix column to avoid stalling the computation. The middle pipelined bus streams in the* **A** *sub-matrix while the bottom bus streams out the resulting* C *sub-matrix.*

3.3 The Need for Dynamic Control

In the classic systolic array model, the datapath is *statically* configured into a deep pipeline before data is pumped through. However, the algorithm for matrix multiply requires several different phases of configuration, including loading the **B** sub-matrices into the RAMs, performing the dot product of the incoming rows with the stored columns (which includes incrementing or clearing the RAMs at the appropriate cycle), and retiring completed products to an output pipeline. Reconfiguring the datapath statically to perform these different phases would be both time-consuming due to the time hit incurred by static reconfiguration, and difficult since the phases overlap in non-intuitive ways. As a result, dynamic control is required to efficiently implement multiple, overlapping phases of an algorithm. It is this support for limited dynamic control that differentiates RaPiD from other configurable architectures.

4 RaPiD Control Architecture

The signals that control the structure and operation of the RaPiD datapath are divided into static configuration stored in SRAM cells and dynamic control updated every clock cycle. The static programming bits

are used to construct the underlying pipeline structure and the dynamic control is used to schedule the operations of the computation onto the datapath over time, as illustrated in the previous section.

The functionality that can be specified dynamically are functional unit inputs, I/O stream reads and writes, and both RAM and ALU operations. Of the total control information, 27% is dynamic control (about 1600 bits for a 16-cell array). Generating these using a stored-program paradigm would be prohibitively expensive. Two observations allow us to reduce the amount of information needed to generate the dynamic control. First, for a given application most of the functionality that can be specified dynamically is actually static, and hence only a few signals must change on every cycle. Second, since an application's datapath is fairly regular from cell to cell, a dynamic control signal in one cell is typically required in a neighboring cell on a successive cycle. Since we are performing pipelined computations, as data moves through the pipeline the dynamic control signals move with it.

Dynamic control is thus implemented by inserting the current "context" into a pipelined control path that parallels the datapath. This context contains all the information required by the various pipeline stages to compute their dynamic control signals. The control path comprises a set of 1-bit segmented busses similar in structure to the datapath busses, as shown in Figure 5.

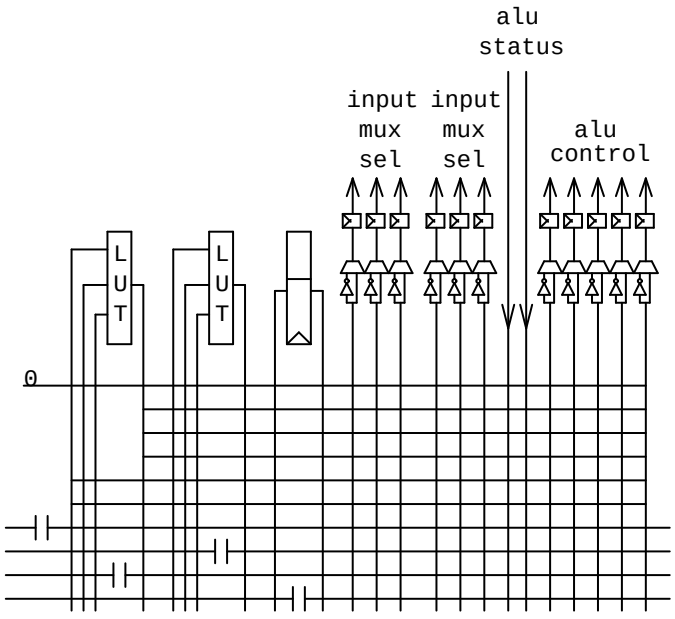

Figure 5: *Dynamic control generation for part of a RaPiD cell. All busses are a single bit.*

Each dynamic control signal is registered and connected through an optional inverter to either the constant 0, if the control is static, or to a control bus if the control is dynamic. In most cases, the control signals needed in the datapath are provided directly by a control bus. However, more complex decoding using 3-input look-up tables (LUTs) can be used to decode several context bits into the appropriate control. The LUTs also contain optional registers that can be used to construct simple finite state machines (FSMs) occasionally required by non-pipelined control. For example, the matrix transpose operation cannot be done efficiently with simple pipelined control since it requires activating one RAM for many cycles until it is exhausted before moving on to the next. Each RAM could have a separate control signal to activate it, but this requires more control lines than are likely to be available. Using a FSM, a stage can remember whether or not it is activated, and one context bit can be used to deactivate one stage and activate the next, requiring only two control lines.

The LUTs are also used to incorporate datapath status information into the dynamic control. For example, the absolute value operation can be implemented by feeding the sign bit from one ALU back into the control path where it selects either the ADD or SUB function in a second ALU.

The number of busses required in the control path varies by application, but it is not large because control signals tend to be reused extensively. The RaPiD-1 architecture provides 15 busses, which is more than enough for the current set of applications.

5 RaPiD System Architecture

Figure 6 shows a complete RaPiD system comprising a RaPiD datapath, the controller, I/O streams and memory. The RaPiD system is configured by programming the controller and I/O streams and loading the configuration information into the RaPiD array. The configuration data is read using one of the input streams. Different configurations can be stored in memory and read very quickly into the on-chip configuration memory.

5.1 Datapath Controller

Computations performed by RaPiD are best described using several nested loops which may run in sequence or in parallel with each other. Together these loops determine the context of the computation. For example, matrix multiply uses three parallel loop nests. One preloads the **B** matrix, another accumu-

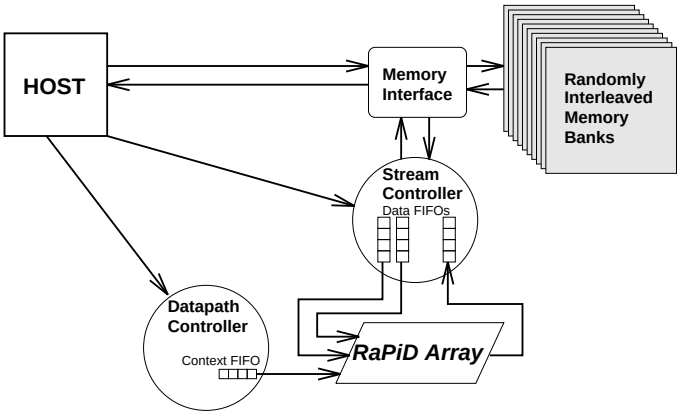

Figure 6: *Block diagram of the overall RaPiD system.*

lates results from the pipelined **A** and stored **B** matrices, and yet another outputs the **C** matrix. These loops must be initiated and synchronized with each other and together provide the control for the datapath.

The datapath controller comprises a set of context generators each of which executes a nested loop or a sequence of nested loops, and produces a context for that loop (Figure 7). The context generators are optimized to execute inner loops at the rate of one

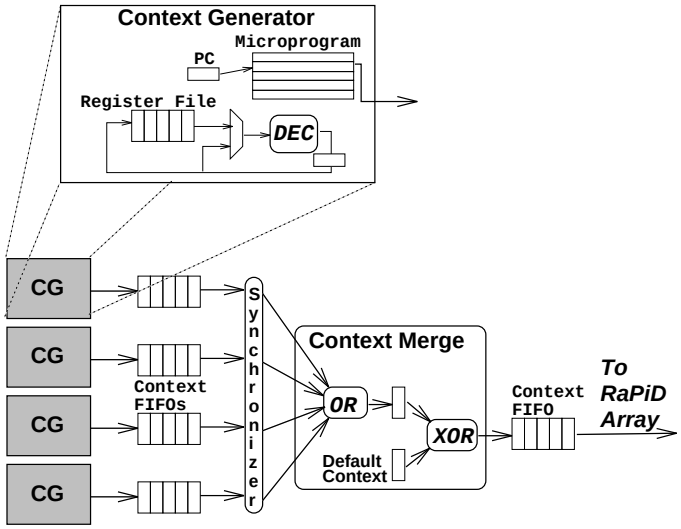

Figure 7: *Block diagram of the RaPiD controller.*

iteration per instruction while simultaneously placing contexts into the context FIFO. Instructions also load constants into registers, or insert SIGNAL and WAIT tags into the output context stream.

The context streams generated by the context generators are merged by a synchronizer which combines matching SIGNAL and WAIT tags from different context FIFOs. This means that the context generators do not communicate explicitly but only indirectly through the synchronizer. After the context streams have been synchronized, they are ORed together, and then the result is XORed with a default context value to produce a final context that is inserted into the context FIFO for the RaPiD array.

The controller typically produces one context for every 1.3–1.5 instructions executed because of the overhead required for loop initialization and outer loop execution. To counteract this potential slowdown, the controller runs twice as fast as the RaPiD array, allowing it to produce at least one context per cycle on average. This faster clock rate is achieved by delaying register writes in the context generators (with forwarding), predicting branches on the non-zero path, and permitting the synchronizer unit to stall a cycle to retire SIG and WAIT instructions.

5.2 I/O Streams

Each I/O stream generates the sequence of addresses used to access memory for reading or writing the values used or produced by the datapath. These addresses are statically determined at compile time and can be described by a multiply-nested loop. These address loops are executed by address generators (Figure 8) that are quite similar in design to the context generators in the controller. The address generators also run at double speed so that addresses can be produced at an average of one per cycle.

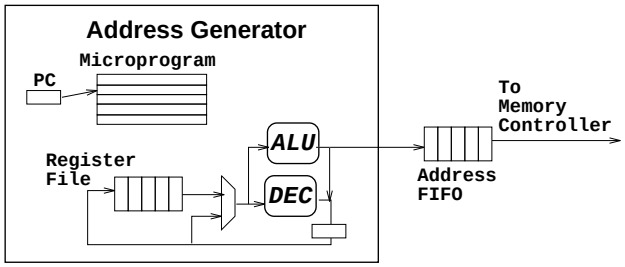

Figure 8: *A RaPiD stream's address generator.*

5.3 RaPiD Memory Architecture

The RaPiD array assumes that up to two reads and one write can be performed to external memory on each cycle. In this section we outline a memory system that can achieve this at a 100MHz clock rate. We make use of two different and seemingly competing techniques. First, we use memories that support

fast burst mode, that is, high-bandwidth data transfer to addresses in the same row. Second, we organize memory into randomly interleaved memory banks [10, 3]. Fast burst mode supports mostly-sequential memory accesses. Data can usually be stored in memory so that accesses are mostly-sequential. For example, all the address streams for matrix multiply are in row-major order[3] and therefore mostly-sequential. Some applications, however, access data with different strides so that the data cannot be placed to give mostly-sequential access. In this case, the random-interleaving provides reasonable performance.

Our proposed memory architecture handles up to three simultaneous address/data transfers via three parallel memory buses, one for each I/O stream. Arbitration is used to determine which streams get to access banks that are in contention. Once a stream has accessed a memory bank, it can transfer data in bursts at one word per cycle. No other stream can access this bank until the data transfer is complete. A stream may generate a request to another bank before it has finished with a current data transfer as long as the current transfer will be complete before the new transfer begins. Single word accesses are thus pipelined using multiple interleaved banks. This memory can be constructed using 100MHz synchronous DRAMs.

Table 1 shows the results of simulating this memory architecture for 256×256 matrix multiply and 8×8 discrete cosine transform (DCT) over a 512×512 image. We chose these two applications because they make high demands on the memory system. Matrix multiply makes an average of 1.5 and a maximum of 3 accesses/sec, while DCT makes a constant 2 accesses/sec. In these applications, like most, data layout can be done so that memory accesses are mostly sequential. However, to show how this memory architecture performs well for non-sequential memory accesses, we have changed the data layout so that one matrix is accessed in column-major order in matrix multiply, and the resulting 8×8 matrices in DCT are transposed. These revised applications are labeled Mat2 and DCT2. The experiments used data FIFOs of length 64, a memory access latency of 8 cycles (80ns), and a memory data transfer rate of 100MHz. The table lists performance hits for different numbers of banks and burst lengths. These results show that this memory architecture with 16 banks and bursts of 1024 can support these RaPiD applications, degrading performance by less than 20%. We believe that further optimization to the memory architecture will reduce this overhead to less than 10% for the usual case of sequential accesses. We also plan to make the memory organization configurable so that it can be tuned dynamically to the application.

Table 1: *The effect of memory system performance on matrix multiply and DCT. Results are given as the percent increase in RaPiD runtime.*

Parameters		Applications			
Banks	Burst	Mat1	Mat2	DCT1	DCT2
16	1024	17%	4%	16%	14%
8	1024	22%	9%	22%	14%
16	512	15%	7%	19%	28%
8	512	18%	18%	23%	32%
16	256	2%	7%	51%	21%
8	256	3%	21%	46%	31%

[3]Since a **B** sub-matrix is stored one-column per cell, row-major order allows for simple pipelined control.

6 RaPiD Performance

This section analyzes the performance of three computationally intensive applications: matrix multiply, motion estimation, and DCT. Each program is evaluated on RaPiD-1, a 16-cell 100MHz array. Static reconfiguration for this architecture is assumed to require 1000 memory accesses and take 2000 cycles (a conservative estimate). We first verify the 100MHz frequency by giving critical cycle times from HSpice simulations and then provide performance results. Since running times are analyzed under an ideal memory interface (no stalls), we also provide measurements of the average number of memory requests per cycle. A detailed account of the implementations of these algorithms can be found in Appendix A.

6.1 RaPiD Cycle Time

The performance goal for the RaPiD-1 chip is 100MHz in a 0.5μ CMOS process. Table 2 lists the delay of the major units in RaPiD as predicted by HSpice on actual layout. The In→Clk delay is the combinational delay of the unit including the input

Table 2: *Unit delays (ns)*

	Pipelined		Comb.
Unit	In→Clk	Clk→Out	Bypass
Multiply1	6.3	1.2	-
Multiply2	5.3	1.6	6.9
ALU	4.3	1.6	5.9
Ram	2.0	3.4	-
Bus Connector	0.5	1.2	1.6
Data Register	1.1	1.6	2.1
Optional Inv.	1.6	1.2	-
3-LUT	2.0	1.2	2.2

multiplexer and the setup time of the register. The Clk→Out delay includes the register output delay, the bypass multiplexer if appropriate, and the bus driver. The bypass delay includes the input multiplexer, internal and bus driver delay for non-pipelined bus connectors. Multiply1 & 2 are the first and second pipeline stages of the multiplier.

Table 3 gives a number of register-to-register paths whose delays are less than 8ns. This indicates that RaPiD will run at 100MHz, even if our HSpice simulations are off by as much as 25%, as long as applications can be placed and routed within this path delay constraint. These constraints are in fact quite generous. The multiplier remains the critical path, but the delay is well-balanced with that needed by other communication paths.

Table 3: *Critical path delays (ns)*

Path	Delay
Register→four busses→Register	7.5
Register→bus→Multiply1	7.9
Multiply1→Multiply2	6.5
Register→two busses→ALU	7.5
Ram→one bus→ALU	7.7
Multiply2→two busses→Ram	6.8
Control Register→one bus→ 3LUT→two busses→Optional Inv.	6.6

The RaPiD-1 chip will contain 16 cells with a total of one million transistors on a die approximately 12mm × 12mm. The future addition of the controller, I/O streams and memory control will add about 20% to this cost.

6.2 Results

6.2.1 Matrix Multiply

Figure 9 shows the number of GOPS (billion of fixed-point "operations" per second) that can be computed on a RaPiD-1 chip configured to perform matrix multiply (an operation is a MAC). Once the pipeline is full (and assuming no memory stalls) RaPiD performs at a sustained rate of approximately 1.6 GOPS with an average of about 1.5 memory accesses per cycle. The jagged lines are a result of lost performance when the partitioning can not be made a factor of the matrix dimensions (e.g. $M = 64$ performs better than $M = 65$ since the latter is not a multiple of the number of RaPiD cells).

Figure 10 shows the distribution of memory requests for a matrix multiply of two 64×64 matrices. Over the course of the computation, eight sets of 512 consecutive writes are required, about half of which induce three memory requests in one cycle.

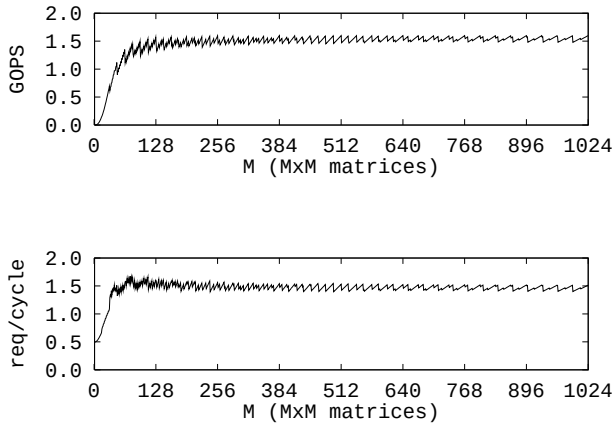

Figure 9: *Matrix Multiply: GOPS and memory accesses per cycle, including cycles and accesses for reconfiguration (an OP is a MAC).*

6.2.2 Discrete Cosine Transform

A 16-cell RaPiD array can efficiently compute an 8×8 2D-DCT by performing two matrix multiplies with the transposed output of the first being the input to the second. Figure 11 shows the number of MAC operations per second that can be computed on RaPiD-1. As with matrix multiply, RaPiD achieves a sustained rate of 1.6 GOPS but with an average of 2.0 memory accesses per cycle. DCT has the highest memory access average of all tested applications since it produces results most efficiently, reading one input and writing one output on every cycle after initialization. This graph is accurate for all image sizes which are multiples of 8; for clarity the slight jagged performance drops on other sized images have been removed from this and the following graphs.

6.2.3 Motion Estimation

Figure 12 shows the number of difference/absolute value/accumulate operations per second that can be computed on a RaPiD-1 chip configured for motion estimation. As with matrix multiply and DCT, RaPiD performs at a sustained rate of 1.6 GOPS but with an average of 0.1 memory accesses per cycle.

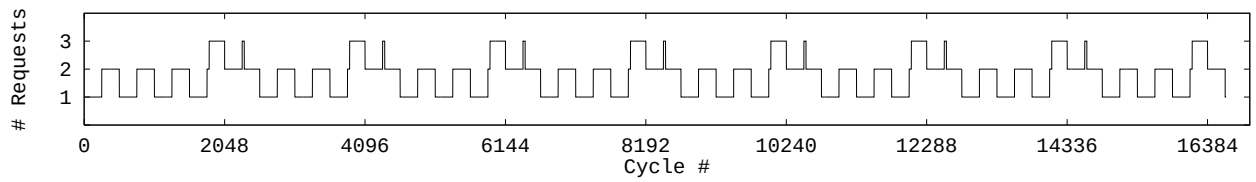

Figure 10: *Distribution of memory requests for a matrix multiply of two* 64×64 *matrices.*

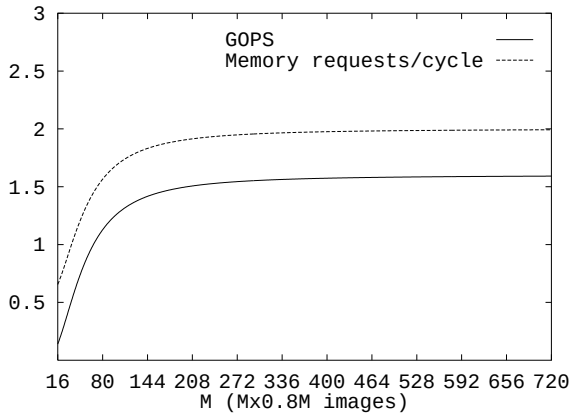

Figure 11: 8×8 *2D-DCT: GOPS and memory accesses per cycle, including cycles and accesses for reconfiguration (an OP is a MAC).*

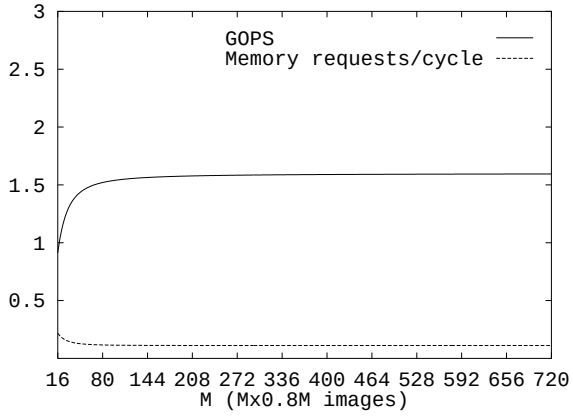

Figure 12: *Motion estimation: GOPS and memory accesses per cycle, including cycles and accesses for reconfiguration (an OP is a difference/absolute value/accumulate).*

6.2.4 Motion Estimation – Double-Gauged

A possible extension to RaPiD requiring little overhead is double gauging the ALUs such that a single 16-bit ALU can be configured as two parallel 8-bit ALUs. This optimization can often double the performance of applications using an 8-bit datapath by effectively doubling the RAM size and the number of ALUs. (However, the number of multipliers, which are not double-gauged, stays constant.) For example, Figure 13 shows that a RaPiD-1 chip computing motion estimation with double-gauged ALUs yields a sustained rate of 3.2 GOPS, as expected.

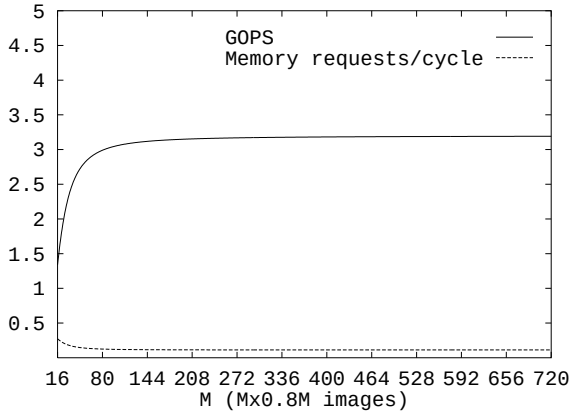

Figure 13: *Double-gauged motion estimation: GOPS and memory accesses per cycle, including cycles and accesses for reconfiguration.*

6.2.5 Real-time Video Performance

Figure 14 shows the performance, in terms of frames per second, of motion estimation and DCT on varying image sizes. On a standard 720×576 image, RaPiD-1 achieves about 12 frames per second when executing both motion estimation and DCT (including 4000 reconfiguration cycles). If the ALUs were

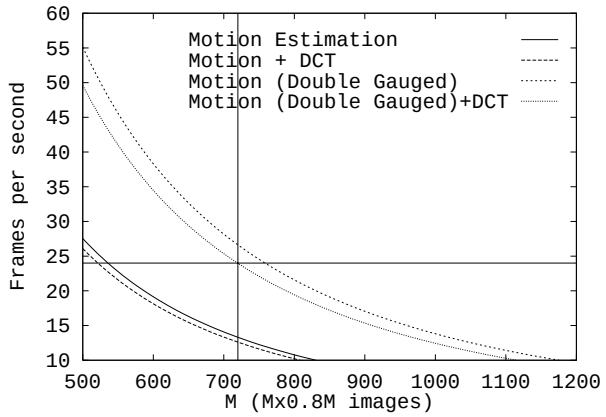

Figure 14: *Number of computable frames per second for motion estimation alone and combined with DCT, including reconfiguration cycles.*

double-gauged, approximately 24 frames per second could be achieved.

6.3 Comparison to Other Architectures

Quantitative comparisons to other architectures are difficult because of differences in technology, application details, data format, and memory systems. For comparison, we cite here performance results for a high-performance digital signal processor and one of the highest performance FPGA-based reconfigurable computing machines.

De Greef et al. derive a motion estimation algorithm highly optimized for DSP-style architectures [5]. In a case study of the 50MHz Texas Instruments TMS320C80 digital signal processor (containing four 32-bit DSPs and one 64-bit RISC processor), they show that 23 TMS320C80 chips can implement motion estimation of 720×576 pixel frames at 25 frames/second. (A 60MHz version would reduce this requirement to 12 chips). Section 6.2.5 showed that a double-gauged RaPiD-1 chip can perform both motion estimation and DCT at 24 frames/second.

The PAM P_1 is an FPGA-based reconfigurable computing machine consisting of 23 Xilinx XC3090 FPGAs, a 4MB local RAM, and a 100MB/s host bus. The PAM project has reported some of the best performance for configurable machines. A single PAM P_1 board can perform 2D-DCT at a rate of 1.4 GOPS (an OP is an MAC/subtract/shift) [4]. Section 6.2.2 showed that RaPiD achieves 1.6 GOPS.

7 Conclusion and Future Directions

RaPiD represents an efficient configurable computing solution for regular computationally-intensive applications. By combining the appropriate amount of static and dynamic control, it achieves substantially reduced control overhead relative to FPGA-based and general-purpose processor architectures. Processors must devote resources to be able to perform irregular and unpredictable computations, while FPGAs must devote resources to construct unpredictable circuit structures. RaPiD is optimized for highly predictable and regular computations which reduces the control overhead. The assumption is that RaPiD will be integrated closely with a RISC engine on the same chip. The RISC would control the overall computational flow, performing the unstructured computations which it does best, while farming out the heavy-duty, brute-force computation to RaPiD.

RaPiD is most closely related to the systolic arrays developed in the 1980s in terms of computation style and application domain. But RaPiD fills the gap between static systolic arrays, which are too inflexible, and programmable systolic arrays like Warp and iWarp [1], which more closely resemble multiprocessors with too much control overhead.

Space has precluded any discussion of programming RaPiD. Programmability is a major problem for configurable computers, which has limited their widespread use. We use a RaPiD programming model based on describing the operation of each pipeline stage for one data element passing through the pipeline. This comprises one "parallel" instruction which executes in a single cycle. Communication between operations is constrained to that which is supported by the physical pipeline. Programs typically comprise several nested loops, sometimes operating in parallel. Our programming language allows these loops to be written separately and composed either sequentially or in parallel according to a specified offset. Although it does require an understanding of the construction of the RaPiD architecture, we have found that programmers can develop a new application in a few hours, once they know the structure of their computation. The next step is to compile higher-level descriptions that do not incorporate the constraints of the RaPiD architecture. Although many recent parallel compiler techniques can be applied, the problem in general is extremely difficult.

The most important open question is how to best incorporate RaPiD into a larger system comprising a general-purpose processor and a more general memory system. One approach is to treat it as a co-

processor as we have described. We believe, however, that RaPiD should be bound much more closely to a general-purpose processor. In this model, it would be viewed as a special functional unit of the processor with its own special path to memory that could include the processor cache where appropriate. In such a model, the granularity of the computation passed to RaPiD could be relatively small, and the configuration information could be contained in the instruction stream and decoded to configure the RaPiD datapath. Such a tight interaction would greatly increase the application domain of RaPiD. Processors incorporating a RaPiD array could be used for both general-purpose computing as well as compute-intensive applications like digital signal processing.

A Application Implementations

This section describes implementations of the applications analyzed in the performance section. In particular, for each application formulas for the number of cycles and memory requests are derived.

A.1 Matrix multiply

We first examine the matrix multiplication of two $M \times M$ matrices **A** and **B**. For a 16-cell RaPiD array, matrix multiply is tiled into **A** sub-matrices of size $Q \times R$ and **B** sub-matrices of size $R \times S$ (see figure 3), where $Q = R = \lceil \frac{M}{\lceil \frac{M}{32} \rceil} \rceil$ and $S = \lceil \frac{M}{\lceil \frac{M}{16} \rceil} \rceil$. These values are chosen to reduce the overhead when M is not a multiple of the RAM size (32) or the number of cells (16). The total number of cycles is broken down into three categories: initialization, computation, and finalization. Initialization uses RS cycles to load a **B** sub-matrix and $S-1$ cycles to fill the pipeline, performing $\frac{S(S-1)}{2}$ MAC operations. Finalization uses $S-1$ cycles to unload the computation pipeline and an additional S cycles to drain the output pipeline, also performing $\frac{S(S-1)}{2}$ MAC operations. The remaining $M^3 - S(S-1)$ MAC operations are executed with a full pipeline, thus consuming a total of $\frac{M^3 - S(S-1)}{S}$ cycles. Hence, the total number of cycles consumed is $RS + 2S + \frac{M^3}{S} - 1$. This equation verifies the quick dissipation of setup overhead with increasing M: the overhead is 12% for $M = 32$ and 2% for $M = 64$. A workload which performs a single matrix multiply has an additional 2000 reconfiguration cycles, yielding an overhead of 53% for $M = 32$ and 12% for $M = 64$.

At least $3M^2$ memory references are required since each input matrix must be read and the output must be written. For the partitioned algorithm described in section 3.2, the entire **A** matrix is read for each column of **B** sub-matrices, the entire **B** matrix is read for each row of **A** sub-matrices, and the output is still written once. Since there are $\frac{M}{16}$ columns of **B** sub-matrices and $\frac{M}{32}$ rows of **A** sub-matrices, the total number of memory references is $M^2\frac{M}{16} + M^2\frac{M}{32} + M^2$.

Matrix multiply influenced the decision to have $RAMsize = 2 * NumCells$, since the amount of local memory plays a role in the average memory accesses per cycle: Decrease $RAMsize$ to $NumCells$ and the average number of memory accesses per cycle jumps to 2.0. Conversely, increase $RAMsize$ to $4 * NumCells$ and the average number of memory accesses per cycle drops to 1.25.

A.2 Discrete Cosine Transform

A 16-cell RaPiD array can efficiently compute an 8×8 2D-DCT by performing two matrix multiplies with the transposed output of the first being the input to the second. Consider an $M \times 0.8M$ image and an 8×8 weight matrix **B**. First, the image is divided into $\frac{0.8M^2}{64}$ sub-images of size 8×8. Then, for each sub-image **A**, $((\mathbf{A} \times \mathbf{B})^T \times \mathbf{B})^T$ is computed by the RaPiD array. Since the **B** matrix is fixed for one image's computation, it is loaded only once: one column per cell in both the first 8 cells and the last 8 cells. The transpose of the first matrix multiply is performed with two RAMs per cell: one to store products of the current sub-image and the other to pass the *transpose* of the previous sub-image's computed products to the next 8 cells (section 4 describes the non-regular control required for the transpose). Using a similar analysis to section A.1 for an 8-cell RaPiD array with no partitioning, multiplying two 8×8 matrices requires $79 + \frac{8^3}{8}$ cycles. However, the latency incurred by transposing the first matrix multiply is an additional 64 cycles. The remaining $(\frac{M}{8})(\frac{0.8M}{8}) - 1$ sub-images overlap their initialization and finalization phases, yielding a total cycle count of $143 + 0.8M^2$. Given a typical image size of $M = 720$ and a reconfiguration cost of 2000 cycles, the setup overhead is only 0.5%. This low overhead is expected since 2D-DCT computes many small matrix multiplies, amortizing reconfiguration cost across many consecutive operations.

Each element of the input must be read once, and the output must be written once, for a total of $2 * 0.8M^2$ memory accesses for DCT.

A.3 Motion Estimation

Motion estimation computes a distance vector yielding the best (i.e. minimum cost via point-to-point

differencing) overlap of a reference block (RB) within a query window (QW). An $M \times 0.8M$ reference frame is divided into $\frac{0.8M^2}{64}$ 8×8 reference blocks, and motion estimation is performed on each RB within a QW (from the query frame) of ± 8 pixels (i.e. a 24×24 pixel query window). The motion estimation algorithm for a 16-cell RaPiD array first divides the reference frame into $\frac{0.8M^2}{256}$ sub-images of size 16×16, each comprising 4 reference blocks. First, a sub-image is loaded into the RB RAMs, one column per cell. Then, a 32×16 sub-image of the query frame is loaded into the QW RAMs, also one column per cell. Each overlap of the 16-entry RB RAM with the 32-entry QW RAM is computed via point-to-point differencing, requiring $17 * 16$ cycles (rows of differences are summed as they pass down the pipeline, and at the end of the array columns are summed and difference vectors are computed). Then, each column of the 32×16 query frame sub-image is shifted down to the next cell, and 32-entries from the next column are read into the first cell. The $17 * 16$ cycle overlap phase is computed again. This process repeats until a total of 16 additional query frame "shifts" have been performed (for a total of 17 overlap computations), completing motion estimation for all four reference blocks in $17 * 17 * 16$ cycles. Then, the next (in row major order) 16×16 sub-image is read into the RB RAMs, and the process repeats $\frac{0.8M}{16}$ times for a total of $\frac{0.8M}{16} * 17 * 17 * 16$ cycles (note that loading the QW and RB RAMs is overlapped with computation and hence consumes no cycles). After each row of RB sub-images, the initial 32×16 entry QW must be loaded, consuming $32 * 16$ cycles (which can not be overlapped). This entire process repeats for each row of sub-images, for a total of $16 * 16 + \frac{M}{16}(\frac{0.8M}{16} * 17 * 17 * 16 + 32 * 16) + 16 = 272 + 32M + 14.45M^2$ cycles, including initialization and finalization. The setup overhead including a 2000 cycle reconfiguration time is 0.3%. Again, this low overhead is expected because the prodigious amount of computation makes the cost of initialization, finalization, and reconfiguration insignificant.

The memory requirements of motion estimation involve reading the reference frame once, but due to overlapping query windows, the query frame must be read twice. In addition, $\frac{0.8M^2}{64}$ difference vectors must be written. Hence, a total of $3\frac{1}{64} * 0.8M^2$ memory accesses are performed.

Acknowledgments

We would like to thank Larry McMurchie and Jeffrey Weener for their contributions to the RaPiD project.

References

[1] M. Annaratone et al. The warp computer: architecture, implementation, and performance. *IEEE Transactions on Computers*, C-36(12):1523–38, 1987.

[2] J. M. Arnold, D. A. Buell, D. T. Hoang, D. V. Pryor, N. Shirazi, and M. R. Thistle. The Splash 2 processor and applications. In *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 482–5. IEEE Comput. Soc. Press, 1993.

[3] G. R. Beck, D. W. L. Yen, and T. L. Anderson. The Cydra 5 minisupercomputer: architecture and implementation. *Journal of Supercomputing*, 7(1-2):143–80, 1993.

[4] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: a performance assessment. In *Parallel Architectures and Their Efficient Use: First Heinz Nixdorf Symposium Proceedings*, pages 119–30. Springer-Verlag, 1993.

[5] E. De Greef, F. Catthoor, and H. De Man. Mapping real-time motion estimation type algorithms to memory efficient, programmable multiprocessor architectures. *Microprocessing & Microprogramming*, 41(5-6):409–23, 1995.

[6] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing.* Ph.D. thesis, Massachusetts Institute of Technology, August 1996.

[7] H.T. Kung. Let's design algorithms for VLSI systems. Technical Report CMU-CS-79-151, Carnegie-Mellon University, January 1979.

[8] P. Lee and Z. M. Kedem. Synthesizing linear array algorithms from nested FOR loop algorithms. *IEEE Transactions on Computers*, 37(12):1578–98, 1988.

[9] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-35(1):1–12, 1986.

[10] B. R. Rau. Pseudo-randomly interleaved memory. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 74–83, 1991.

[11] J. Rose, A. El Gamal, and A Sangiovanni Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–29, 1993.

[12] K. A. Vissers et al. Architecture and programming of two generations video signal processors. *Microprocessing & Microprogramming*, 41(5-6):373–90, 1995.

[13] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard. Programmable active memories: reconfigurable systems come of age. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):56–69, 1996.

[14] Wazlowski-M. et al. PRISM-II compiler and architecture. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16. IEEE Comput. Soc. Press, 1993.