Appeared in Proceedings of 26th International Symposium on Computer Architecture, ISCA 1999

PipeRench: A Coprocessor for Streaming Multimedia Acceleration

Seth Copen Goldstein[†] Herman Schmit[*] Matthew Moe[*] Mihai Budiu[†] Srihari Cadambi[*]
R. Reed Taylor[*] Ronald Laufer[*]
School of Computer Science[†] and Department of ECE[*]
Carnegie Mellon University
Pittsburgh, PA 15213
[†]{seth,mihaib}@cs.cmu.edu
[*]{herman,moe,cadambi,rt2i,rel}@ece.cmu.edu

Abstract

Future computing workloads will emphasize an architecture's ability to perform relatively simple calculations on massive quantities of mixed-width data. This paper describes a novel reconfigurable fabric architecture, PipeRench, optimized to accelerate these types of computations. PipeRench enables fast, robust compilers, supports forward compatibility, and virtualizes configurations, thus removing the fixed size constraint present in other fabrics. For the first time we explore how the bit-width of processing elements affects performance and show how the PipeRench architecture has been optimized to balance the needs of the compiler against the realities of silicon. Finally, we demonstrate extreme performance speedup on certain computing kernels (up to 190x versus a modern RISC processor), and analyze how this acceleration translates to application speedup.

1. Introduction

Workloads for computing devices are rapidly changing. On the desktop, the integration of digital media has made real-time media processing the primary challenge for architects [10]. Embedded and wireless computing devices need to process copious data streaming from sensors and receivers. These changes emphasize simple, regular computations on large sets of small data elements. There are two important respects in which this need does not match the processing strengths of conventional processors. First, the size of the data elements underutilizes the processor's wide datapath. Second, the instruction bandwidth is much higher than it needs to be to perform regular, dataflow-dominated computations on large data sets.

Both of these problems are being addressed through processor architecture. Most recent ISAs have multimedia instruction set extensions that allow a wide datapath to be switched into SIMD operation [17]. The instruction bandwidth issue has created renewed interest in vector processing [14, 24].

A fundamentally different way of addressing these problems is to configure connections between programmable logic elements and registers in order to construct an efficient, highly parallel implementation of the processing kernel. This interconnected network of processing elements is called a *reconfigurable fabric*, and the data set used to program the interconnect and processing elements is a *configuration*. After a configuration is loaded into a reconfigurable fabric, there is no further instruction bandwidth required to perform the computation. Furthermore, because the operations are composed of small basic elements, the size of the processing elements can closely match the required data size. This approach is called *reconfigurable computing.*

Despite reports of amazing performance [11], reconfigurable computing has not been accepted as a mainstream computing technology because most previous efforts were based upon, or inspired by, commercial FPGAs and fail to meet the requirements of the marketplace. The problems inherent in using standard FPGAs include

1. **Logic granularity:** FPGAs are designed for logic replacement. The granularity of the functional units is optimized to replace random logic, not to perform multimedia computations.
2. **Configuration time:** The time it takes to load a configuration in the fabric is called *configuration time*. In commercial FPGAs, configuration times range from hundreds of microseconds to hundreds of milliseconds. To show a performance improvement this startup latency must be amortized over huge data sets, which limits the applicability of the technique.
3. **Forward-compatibility:** FPGAs require redesign or recompilation to gain benefit from future generations of the chip.
4. **Hard constraints:** FPGAs can implement only ker-

nels of a fixed and relatively small size. This is part of the reason that compilation is difficult—everything must fit. It also causes large and unpredictable discontinuities between kernel size and performance.

5. **Compilation time:** Currently the synthesis, placement and routing phases of designs take hundreds of times longer than what the compilation of the same kernel would take for a general-purpose processor.

This paper describes PipeRench, a reconfigurable fabric designed to increase performance on future computing workloads. PipeRench realizes the performance promises of reconfigurable computing while solving the problems outlined above. PipeRench uses a technique called *pipeline reconfiguration* to solve the problems of compilability, reconfiguration time, and forward-compatibility. The architectural parameters of PipeRench, including the logic block granularity, were selected to optimize the performance of a suite of kernels, balancing the needs of a compiler against design realities in deep-submicron process technology.

PipeRench is currently used as an attached processor. This places significant limitations on the types of applications that can realize speedup, due to limited bandwidth between PipeRench, the main memory and the processor. We believe this represents the initial phase in the evolution of reconfigurable processors. Just as floating-point computation migrated from software emulation, to attached processors, to coprocessors, and finally to full incorporation into processor ISAs, so will reconfigurable computing eventually be integrated into the CPU.

In the next section, we use several examples to illustrate the advantages and architectural requirements of reconfigurable fabrics. We introduce the idea of pipeline reconfiguration in Section 3, and describe how this technique solves the practical problems faced by reconfigurable computing. Section 4 describes a class of architectures that can implement pipelined reconfiguration. We evaluate these architectures in Section 5. We cover related work in Section 6, and in Section 7 we summarize and discuss future research.

2. Reconfigurable Computing

2.1. Attributes of Target Kernels

Functions for which a reconfigurable fabric can provide a significant benefit exhibit one or more of the following features:

1. The function operates on bit-widths that are different from the processor's basic word size.
2. The data dependencies in the function allow multiple function units to operate in parallel.

```
for (int i=0; i<maxInput; i++) {
   y[i] = 0;
   for (int j=0; j<Taps; j++)
      y[i] = y[i] + x[i+j]*w[j];
}
```

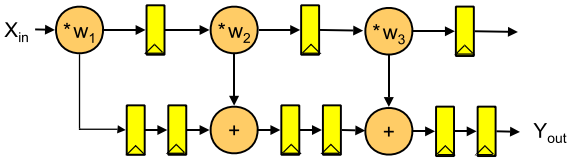

Figure 1. *C code for a FIR filter and a pipelined version for a three-tap filter.*

3. The function is composed of a series of basic operations that can be combined into a single specialized operation.
4. The function can be pipelined.
5. Constant propagation can be performed, reducing the complexity of the operations.
6. The input values are reused many times within the computation.

These functions take two forms. *Stream-based functions* process a large data input stream and produce a large data output stream, while *custom instructions* take a few inputs and produce a few outputs. After presenting a simple example of each type of function to illustrate how a reconfigurable fabric can improve performance, we discuss the ways in which a fabric can be integrated into a complete system.

2.2. A Stream-Based Function: FIR

A reconfigurable fabric can be most effective when used to implement entire pipelines from applications. Here we investigate a simple but prototypical pipeline for implementing a finite-impulse response (FIR) filter. The FIR filter exhibits all but feature 3 from the requirement list in Section 2.1. Figure 1 shows the C code and a hardware implementation. When FIR is mapped to a reconfigurable fabric, the general-purpose multipliers shown in the hardware description are implemented as constant multipliers, where the constants are the `w[i]` values. This results in less hardware and fewer cycles than a general-purpose multiplier.

Figure 2 compares an 8-bit FIR using 12-bit coefficients running on a particular instance of PipeRench to implementations on a Xilinx FPGA using parallel distributed arithmetic (shown as Xilinx PDA in Figure 2) and double-rate bit-serial distributed arithmetic (shown as Xilinx DDA). Both the PipeRench chip and Xilinx FPGA are implemented in 100mm^2 of silicon using a 0.35 micron process.

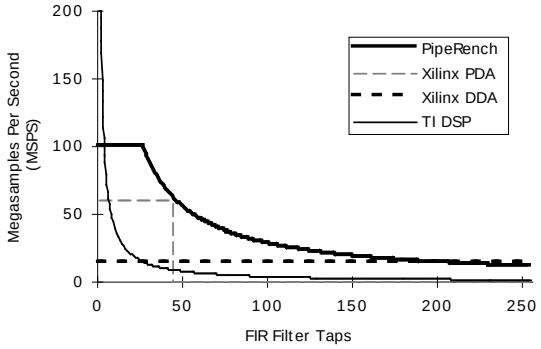

Figure 2. *Performance on 8-bit FIR filters: PipeRench, Xilinx FPGA using parallel and serial arithmetic, and Texas Instruments DSP.*

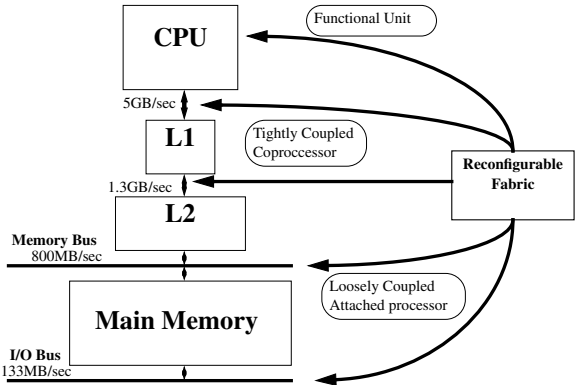

Figure 4. *Possible locations for reconfigurable fabric in memory hierarchy. Bandwidth figures are typical for a 300 MHz Sun UltraSPARC-II.*

```
int
popCount(unsigned n) {
  int sum=0, i;
  for (i=0; i<8; i++)
    sum += (n >> i) & 1;
  return sum;
}
```

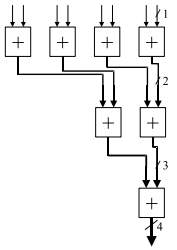

Figure 3. *C code and its hardware implementation for population count.*

The FPGA runs at approximately 60MHz for both applications, while PipeRench's clock is 100MHz. PipeRench outperforms both Xilinx implementations over a broader range of filter sizes. Similarly, PipeRench outperforms the Texas Instruments TMS320C6201, a commercial DSP that runs at 200 MHz and contains two 16x16-bit integer multipliers, on filters larger than a few taps. PipeRench exhibits the same high level of performance as the FPGA. Due to its support for hardware virtualization, as described in Section 3, PipeRench exhibits the same graceful degradations of performance as the DSP.

2.3. Custom Instructions: Population Count Instruction

Most processors, with the exception of vector supercomputers, do not include a native population count instruction and thus it must be implemented in software (see Figure 3). Using a reconfigurable fabric, popCount() can be implemented as a custom instruction giving a raw performance improvement of more than an order of magnitude. The function exhibits three of the qualifying features (1, 2, and 3) from Section 2.1. The reconfigurable computing solution replaces the $O(n)$ loop with an adder tree of height $O(\log n)$. Furthermore, the adders used are significantly narrower than the adders on the processor. The circuit can also be pipelined, so that when executed on a vector it retires one result every cycle.

In evaluating a reconfigurable fabric, it is important to take into account both configuration time and the communication latency and bandwidth between the processor and the fabric. If popCount() is called only once, it makes little sense to configure the fabric to perform the operation since the time to configure the fabric will be larger than the savings obtained by executing popCount() on the fabric.

When popCount() is used outside of a loop and data dependencies require that the result be used immediately after it is computed, the fabric needs direct access to the processor registers. On the other hand, if popCount() is used in a loop, where there are no immediate dependencies on the results, performance can be better if the fabric can directly access memory. In this paper we concentrate on the latter case.

2.4. The Fabric's Place

Reconfigurable fabrics provide the computational datapath with more flexibility. Their utility and applicability is influenced by the manner in which they are integrated into the datapath. We recognize three basic ways in which a fabric may be integrated into a system: as an attached processor on the I/O or memory bus, as a coprocessor, or as a functional unit on the main CPU. (See Figure 4.)

Attached-processor systems, e.g. PAM [1], Splash [4], and DISC [25], have no direct access to the processor. Rather, they are controlled over a bus. The primary feature of attached processors is that they are easy to add to

existing computer systems. However, due to the bandwidth and latency constraints imposed by the bus they can enhance only computations that have a high computation–to–memory-bandwidth ratio. Thus, they are most suited to stream-based functions that require little or no communication with the host processor.

In coprocessor architectures, there is a low-latency, high-bandwidth connection between the processor and the reconfigurable fabric, which increases the number of stream-based functions that can profitably be run on the fabric. Recent examples of such systems include Garp [13] and Napa-1000 [19]. Further specialization occurs when the fabric is on the main processor's data path, as in functional-unit architectures like P-RISC [18], Chimaera [12], and OneChip [26]. All of these allow custom instructions to be executed. The reconfigurable unit is on the processor datapath and has access to registers. However, these implementations restrict the applicability of the reconfigurable unit by disallowing state to be stored in the fabric and in some cases by disallowing direct access to memory, essentially eliminating their usefulness for stream-based processing.

In this paper we describe pipelined reconfigurable architectures, which can be used in any of the fashions described above. However, in order to describe the system we are currently building, we limit ourselves to describing how we would apply it as an attached-processor system. The natural evolution of this fabric to a coprocessor or a function unit would only enhance its applicability.

3. Pipelined Reconfigurable Architectures

In the previous section, we described how application-specific configurations of reconfigurable fabrics can be used to accelerate certain applications. The computation is embedded in a single static configuration rather than in a sequence of instructions, thereby reducing the instruction bandwidth.

The static nature of these configurations, however, causes two significant problems. First, the computation may require more hardware than is available. Second, given more hardware, there is no way that a single hardware design can exploit the additional resources that will inevitably become available in future process generations. In this section, we review a technique called pipeline reconfiguration [20], that allows a large logical design to be implemented on a small piece of hardware through rapid reconfiguration of that hardware. With this technique, the compiler is no longer responsible for satisfying fixed hardware constraints. In addition, the performance of a design improves in proportion to the amount of hardware allocated to that design; as future process technology makes more transistors available, the same hardware designs achieve higher levels of performance.

Pipeline reconfiguration is a method of virtualizing pipelined hardware application designs by breaking the single static configuration into pieces that correspond to pipeline stages in the application. These configurations are then loaded, one per cycle, into the fabric. This makes it possible to perform the computation, even if though the whole configuration is never present in the fabric at one time.

The virtualization process is illustrated in Figure 5, which shows a five-stage pipeline being virtualized on a three-stage fabric. The top portion of this figure shows the five-stage application and the state of each of the stages of the pipeline in five consecutive cycles. The bottom half of the figure shows the state of the physical stages in the fabric that is executing this application. An effective metaphor for this procedure is scrolling on a text window. Once the pipeline is full, every five cycles generates two results from the pipeline. In general, when a v-stage application is virtualized on a device with a capacity of p-stages ($p < v$), the throughput of the implementation is proportional to $(p-1)/v$. Throughput is a linear function of the capacity of the device; therefore performance improves due to both increases in clock frequency and decreases in feature size, without any redesign, until $p = v$. Thereafter, applications' performance continues to gain only through increased clock speed.

Because the configuration of stages happens concurrently with the execution of other stages, there is no loss in performance due to reconfiguration. As the pipeline is filling with data, stages of the computation are being configured ahead of that data. Even if there is no virtualization, configuration time is equivalent to the pipeline fill time of the application and therefore does not reduce the maximum throughput of the application.

In order for this virtualization process to work, the state in any pipeline stage must be a function only of the current state of that stage and the current state of the previous stage. In other words, cyclic dependencies must fit within one stage of the pipeline. Interconnect that directly skips over one or more stages is not allowed, nor are connections from one stage to a previous stage. Fortunately, many computations on streaming data can be pipelined within these constraints. Furthermore, by including structures we call *pass registers*, it is possible to create virtual connections between distant stages.

The primary challenge facing pipeline reconfiguration is configuring a computationally significant pipeline stage in one clock cycle. To do this, we connect a wide on-chip configuration buffer (either SRAM or DRAM) to the nearby fabric allowing a pipeline stage to be configured in one cycle. We use the word *stripe* to describe both the physical stages in the fabric (the *physical stripes*), and the configuration words that are written into them (the *virtual stripes*).

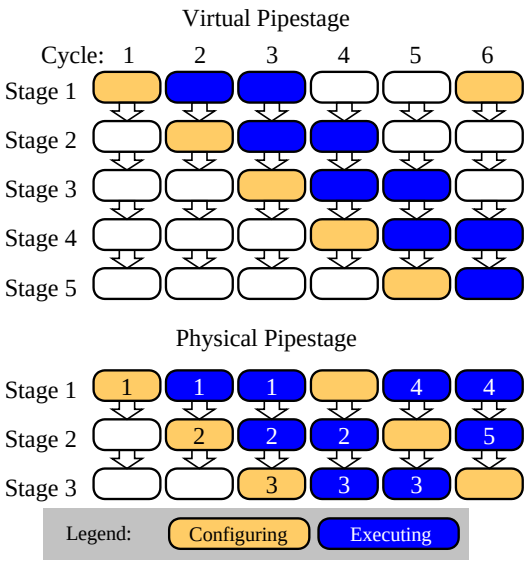

Figure 5. *Pipeline Reconfiguration. This diagram shows the process of virtualizing a five-stage pipeline on a three-stage device.*

Any virtual stripe can be written into any physical stripe. Therefore, all physical stripes must have identical functionality and interconnect.

Before a physical stripe is reconfigured with a new virtual stripe, the state of the resident virtual stripe, if any, must be stored outside of the fabric. Conversely, when a virtual stripe is returned to the fabric, any stored state for the stripe must be restored within the physical stripe [5].

4. PipeRench

In this section, we describe a class of pipeline reconfigurable fabrics, called PipeRench devices, and define critical architectural parameters for this class of fabrics. These architectural parameters are the subject of the performance evaluation described in Section 5.

An abstract view of the PipeRench architectural class is shown in Figure 6. The device is composed of a set of physical pipeline stages, or stripes. Each stripe is composed of interconnect and processing elements (PE), which contain registers and ALUs. An ALU is composed of look-up tables (LUTs) and extra circuitry for carry-chains, zero-detection, etc. The PEs have access to a global I/O bus. Through the interconnect network, the PEs can access operands from registered outputs of the previous stripe as well as registered or unregistered outputs of the other PEs in the stripe. There are no busses that go to a previous stripe; this is required by hardware virtualization (as discussed in [5]) and makes

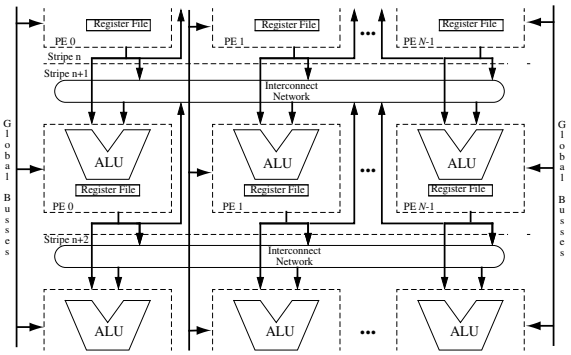

Figure 6. *PipeRench Architecture: PEs and interconnect.*

long feedback loops impossible, since any feedback must be contained within one stripe. The global I/O busses are required because the pipeline stages in an application may be physically located in any of the stripes in the fabric; inputs to and outputs from the application must use a global bus to get to their destination.[1]

All PipeRench devices have four global busses. Two of these busses are dedicated to storing and restoring stripe state during hardware virtualization. The other two are used for input and output. Combinational logic is implemented using using a set of N B-bit wide ALUs. The ALU operation is static while a particular virtual stripe is located in the physical stripe. The carry lines of PipeRench's ALUs may be cascaded to construct wider ALUs. Furthermore, ALUs may be chained together via the interconnect network to build complex combinational functions.

4.1. Pass Register File

We organize each stripe as an array of processing elements (PEs). Each PE contains one ALU and a pass register file. As described in Section 3, there can be no unregistered interconnect between stripes. Furthermore, any state caused by registered feedback within the stripe must be saved and restored. The pass register is designed to provide efficient pipelined (registered) interstripe connections. Each pass register file has one dedicated register that can be used for intra-stripe feedback and therefore must have its state stored and restored.

As illustrated in Figure 7, the output of the ALU can be written to any one of the P registers in the pass register file. If the register is not written by the ALU, the value in the pass register is loaded from the value in the corresponding pass register in the previous stripe. This reduces

[1] By limiting the set of physical stripes that may hold a particular virtual stripe one can eliminate the global busses. This reduces utilization, but may increase clock frequency sufficiently to make it worthwhile.

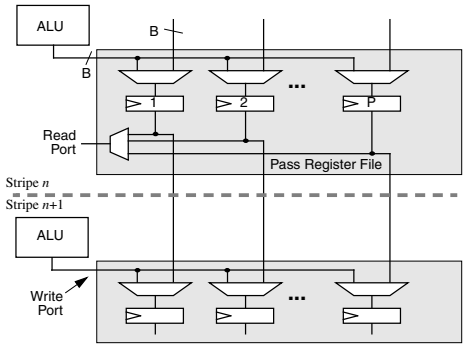

Figure 7. *The pass register interconnect.*

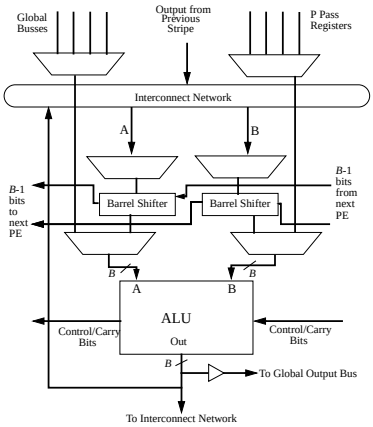

Figure 8. *Complete architectural class.*

the amount of state that can be contained in the pass register file to a single register, because data that travels through the pipeline does not need to be saved and restored. The pass register file also provides a way to route intermediate results computed on one stripe to a stripe somewhere down the pipeline, without wasting ALUs or the interconnect network within the stripe. Like the ALU operation, the specific registers that are written to and read from the pass register file are static while a virtual stripe is resident; different PEs can read and write different registers, but the registers that a particular PE accesses change only when a different virtual stripe configures the physical stripe.

4.2. Interconnect Network

The pass register file provides pipelined interconnect from a PE in one stripe to the corresponding PE in subsequent stripes. If data values need to move laterally within the stripe, they must use the interconnect network, which is illustrated as a horizontal bar in Figure 6. In each stripe, the interconnect network accepts inputs from the each of the PEs in that stripe, as well as one of the registered values from the previous stripe. Like the ALU operations and the pass register files, the interconnect network is programmed during configuration and remains unchanged during the lifetime of the virtual stripe.

The interconnect we evaluate in Section 5 is a full crossbar. This is expensive in terms of hardware, but it makes every design easily placeable by the compiler. Furthermore, a rich network is necessary to achieve good utilization in a reconfigurable fabric [9]. In fact, most fabrics use over 50% of their available area on interconnect. As shown in Section 5, even with a full crossbar we use less than 50% of the area for the interstripe interconnect. Though we use a full crossbar, it connects only PEs to PEs—i.e., it is a B-bit wide, NxN crossbar, as opposed to an $(N$x$B)$x$(N$x$B)$ crossbar. A key to making this interconnect useful is that each PE has a barrel shifter that can shift its inputs up to $B-1$ bits to the left (see Figure 8). This allows our architecture to do data alignments that are necessary for word-based arithmetic as described in [6].

4.3. Physical Implementation

Currently we are planning to design this system in 100mm^2 of silicon in a 0.25 micron process. Half of that area is for the reconfigurable fabric, while the other half is for memory to store virtual stripes, control, and chip I/O. Fifty square millimeters of silicon provides approximately 500kb of virtual configuration storage, which is adequate for very large applications.

4.4. Architectural Parameters

Figure 8 summarizes one of the N PEs in a stripe for our parameterized architecture. In the following section, we explore the following three architectural parameters:

- N: the number of PEs in the stripe;
- B: the width, in bits, of each PE;
- P: the number of B-bit wide registers in the pass register file per PE.

5. Evaluation

In this section we explore the design space of pipelined reconfigurable architecures. Using a compiler and CAD tools, we look at how several kernels perform on implementations of the fabric that differ in the parameters described in Section 4.4.

5.1. Kernels and Applications

Performance and utilization data were gathered for PipeRench implementations of various kernels. The kernels

were chosen based on demand for the applications in the present and near future, their recognition as industry performance benchmarks, and their ability to fit into our computational model.

ATR implements the shapesum kernel of the Sandia algorithm for automatic target recognition [22]. This algorithm is used to find an instance of a template image in a larger image, and to distinguish between images that contain different templates.

Cordic is a 12 stage implementation of the Honeywell timing benchmark for Cordic vector rotations [15]. Given a vector in rectangular coordinates and a rotation angle in degrees, the algorithm finds a close approximation to the resultant rotation.

DCT is a one-dimensional, eight-point discrete cosine transform [16]. **DCT-2D**, a two-dimensional DCT, is an important algorithm in digital signal processing and is the core of JPEG image compression.

FIR is described in Section 2.2. Here we implement a FIR filter with 20 taps and 8-bit coefficients.

IDEA implements a complete eight-round International Data Encryption Algorithm with the key compiled into the configuration [21]. IDEA is the heart of Phil Zimmerman's Pretty Good Privacy (PGP) data encryption.

Nqueens is an evaluator for the Nqueens problem on an 8x8 board. Given the coordinates of chess queens on a chessboard, it determines whether any of the queens can attack each other.

Over implements the Porter-Duff over operator [2]. This is a method of joining two images based on a mask of transparency values for each pixel.

PopCount is described in section Section 2.3.

We also evalute the performance of PipeRench on two complete applications, JPEG and PGP. In each of these applications we assume PipeRench is integrated into the system on the PCI bus, which has a peak memory bandwidth of 132MB/sec.

5.2. Methodology

Our approach is to use CAD tools to synthesize a stripe based on the parameters N, B, and P. We join this automatically synthesized layout with a custom layout for the interconnect. Using the final layout we determine the number of physical stripes that can fit in our silicon budget of 50 mm^2 (5 mm x 10 mm) and the delay characterisitics of the components of the stripe (e.g., LUTs, carry-chain, interconnect, etc.). The delay characterisitics and number of registers are then used by the compiler to create configurations for each of the architectural instances, yielding a design of a certain number of stripes at a particular frequency. We can then determine the overall speed of the kernel, in terms of throughput, for each architectural instance.

The CAD tool flow synthesizes each design point and automatically places and routes the final design. Although the automatic tool flow does not yield the optimal design, we assume that the various points are equally non-optimal, allowing us to compare the designs. Preliminary analysis showed the CAD tools doing quite well, except for the interconnect, which we hand optimize.

The kernels are written in a single-assignment C-like language, DIL, which is intended for both programmers and as an intermediate language for a high-level language compiler that targets reconfigurable architectures. The DIL compiler automatically synthesizes and places and routes our largest designs in a few seconds [3]. It is parameterizable so that we can generate configurations for any pipelined reconfigurable architecure as described in Section 4.

5.3. The Fabric

There are two main constraints that determine which parameters generate realizable fabrics: the width of a stripe and the number of vertical wires that must pass over each stripe. The width of a stripe is influenced by the size and number of the PEs and the number of registers allocated to each PE. We limit the width of a stripe to 4.9mm in order to allow two of them to be placed side by side.[2]

The second constraint is to accomodate the number of vertical wires that pass over the stripes within two metal layers. These wires include those for the global busses, the pass registers, and the configuration bits.

We explore the region of the space bounded by PE bitwidths (B) of 2, 4, 8, 16, and 32 bits; stripe widths (N x B) of between 64 bits and 256 bits; and registers (P) of 2, 4, 8 and 16.[3] Figure 9 shows the computational density (bit-ops/area-time) of the realizable parameters when four and eight registers are allocated to each PE. Interestingly, the result is essentially independent of stripe width. The reason for this is that as the stripe width increases, the amount of area per stripe devoted to interconnect increases, but the total number of stripes decreases—yeilding a constant amount of total area devoted to interconnect. In fact, the total area devoted to interstripe interconnect is less than 50% of the area devoted to the fabric. The total delay from the output of one stripe into the PE of the next stripe remains approximately constant because the wire capacitance of the

[2] Virtualization requires that data be allowed to fbw between any two stripes, including the last physical one and the fi rst physical one. To obtain consistent routing delay times we arrange the stripes in two columns: in one column the data fbws down and in the other it fbws up. This avoids a long path from the last to the fi rst physical stripe.

[3] Some of the wider stripes can be implemented only with eight registers.

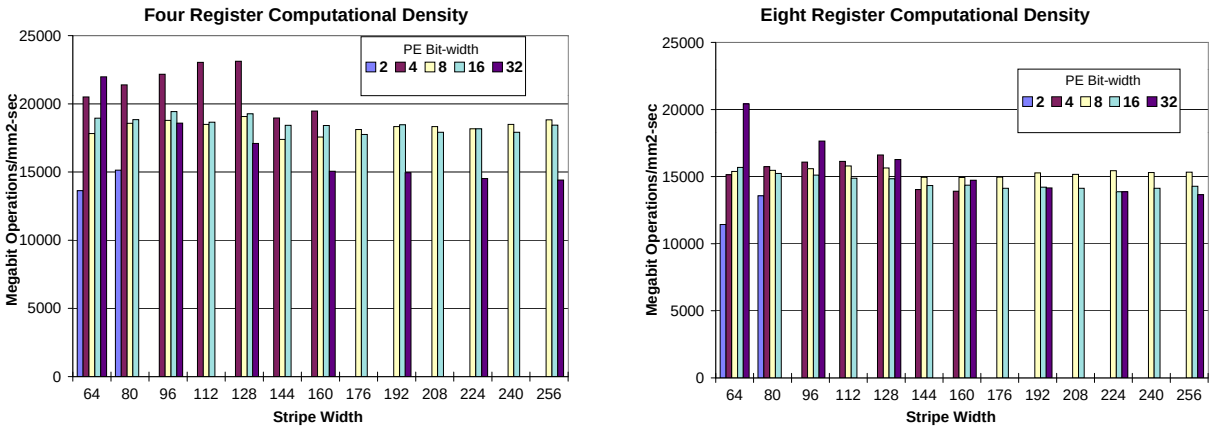

Figure 9. *Computational density.*

interstripe interconnect (5mm long in all cases) dominates the transistor delays.

The computational density does not seem to have a monotonic relationship with PE width. This seems counter-intuitive; as PE size increases, the overhead of configuration decreases and the ability to optimize the PE increases. Therefore, computational density should increase. But our delay metric includes the delay associated the carry chain of one PE, which increases with PE width. The increased carry chain delay counters the reduction in size per bit of the wider PEs causing the computational density to remain relatively constant. On the other hand, if we were to use only logical operations to measure delay, we would observe a near-linear increase in computational density as PE size increases.

Because registers consume substantial area, density goes down as the number of registers increases (compare the two graphs in Figure 9). In fact, since we use registers mainly to implement pipelined interstripe interconnect, they contribute little to computational density. However, as we will see, they are extremely useful in compiling kernels to the fabric.

The last effect we examine is the size of the configuration word. The configuration word size approximately halves as PE widths double. On the other hand, as the width of the stripe increases, the configuration word increases slightly. For 128-bit stripes, the configuration bits for a stripe range from 1280 bits for a 4-bit PE to 164 bits for a 32-bit PE.

5.4. The Compiler

The compilation process maps source written in a dataflow intermediate language (DIL) to a particular instance of PipeRench. DIL is a single-assignment language with C operators and a type system that allows the bit-width of variable to be specified. The compiler converts the source into a dataflow graph and then, through many transformations, creates a final configuration. The important transformations for this study are operator decomposition, operator recomposition, fitting, and place-and-route.

The operator decomposition pass breaks up operators so that they can execute within the target cycle time. For example, a wide adder needs to be broken up into several smaller adders due to the carry-chain delays. The decomposition must also create new operators that handle the routing of the carry bits between the partial sums. For operations that require carry bits, the decomposed version is significantly larger and has additional routing constraints. Thus, as PE size decreases, the penalty for decomposition increases. Currently, the interaction between operator decomposition and place-and-route requires each stripe to have at least six PEs.

The naive decomposition for an operator routes the carry signal on the interstripe interconnect. It also results in sign-extending the single carry bit to the size of the smaller adders. To compensate for this, the operator recomposition pass uses pattern matching to find subgraphs that can be mapped to parameterized modules designed to take advantage of architecture-specific routing and PE capabilities. Most importantly for this study, this slightly reduces the overhead of decomposed carry operations.

The fitting pass matches the wire and operator widths to the size of a PE. This can require the insertion of sign-extension operators to increase the width of wires that are not multiples of a PE width. As the PE width increases, this causes both underutilization of PEs and a larger percentage of sign-extension PEs. Furthermore, routing operations become more complex as extracting bits from wires that are not PE-aligned often involves using an extra PE.

Place-and-route is the key to the compiler. It places and

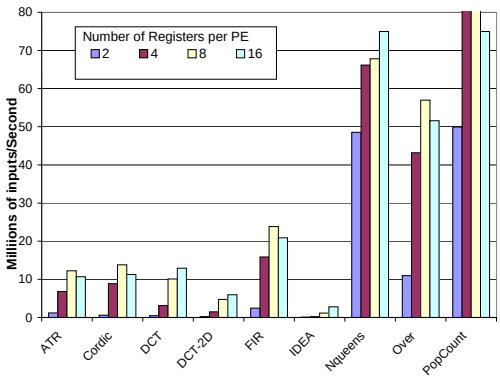

Figure 10. *The harmonic mean of the throughput for all fabric parameters as a function of registers.*

routes the operators in the graph onto stripes under the timing constraint imposed by the target cycle time. Thus, as the clock rate or the delay through the PE increases, the utilization of each stripe can decrease, unless the kernel has sufficient parallelism so that independent operators can be placed in a stripe. This is particularly true of stripes with many PEs.

In addition to assigning operators to PEs and wires to the interconnect, the place-and-route pass assigns wires to the pass registers. If there are insufficient pass registers, the compiler will time-multiplex wires on registers. Time-multiplexing slows the circuit down in order to allow multiple values to reside in a single registers. For example, if two wires are assigned to a single register, then the register holds one wire on the odd cycles and another on the even ones. While time-multiplexing does not increase the circuit size significantly, it does reduce the throughput by a constant factor. For architectures with few registers this is a severe penalty, as time-multiplexing factors of more than ten may be required.

One of the goals for the DIL compiler was compilation speed. It achieves high speed compilation in part by trading off result quality for faster compilation. This affects the results by introducing more time-multiplexing than necessary.

5.5. Compiler/Fabric Interaction

The real question, of course, is not the raw hardware performance available, but how well it can be utilized. Using the parameterizable compiler we compiled configurations for each kernel. Before evaluating the effects of overall width, number of PEs, or number of bits per PE, we narrow down the design space by examing the effect of pass registers. For a given stripe width and bits-per-PE, as the number of registers increase, the computational density decreases. However since pass registers make up an important component of the interstripe interconnect, reducing the number of pass registers increases routing pressure, which decreases stripe utilization and causes the compiler to time-multiplex the values on the registers. As Figure 10 shows, the best balance of computation density with utilization is most often achieved at eight registers. The average time multiplexing factor for all the kernels average across all the fabrics ranges from over 60 for two registers, to 12 at four registers, 2 at eight registers, and 1 at sixteen registers. IDEA and Nqueens have higher factors at eight registers than the other kernels. The rest of the evaluation occurs with eight pass registers per PE, i.e. $P = 8$.

Figure 11 shows the throughput achieved for various stripe widths and PE sizes at eight registers per PE. As can be seen, though the wider PE sizes create fabrics with higher computational density, the natural data sizes of the kernels are smaller, causing 32-bit PEs to be underutilized. On the other end of the spectrum, 2-bit PEs are not competitive due to increased times for arithmetic operations, the lack of raw computational density, and the increased number of configuration bits needed per application.

If we examine the performance of the individual kernels (see Figure 11) we can see that the characteristics of the kernels greatly influence which parameters are best. For example, DCT needs at least 8 PEs in the stripe[4], ruling out 32-bit PEs for all but the widest stripe. The peak at 128 bits occurs because there is a sufficient number of PEs to eliminate time multiplexing. While wider stripes can be utilized because there is sufficient parallelism in the DCT algorithm.

FIR operates mostly on 8-bit and wider numbers. This makes 4-bit PEs less attactive due to the carry chain delay associated with crossing PEs. There is enough parallelism to keep the wider stripes busy. These stripes have fewer registers, which increases the number of stripes in the implementation, thereby reducing its overall throughput.

IDEA takes wide inputs, so stripes of less than 96-bits require substantial time-multiplexing. Unlike DCT and FIR there is not enough parallelism to utilize the wider stripes.

In summary, we need to choose a fabric that is at least 128 bits wide. We also want at least 12 PEs in the stripe. Since not all kernels have sufficient parallelism to utilize wide stripes, we want to choose the narrowest stripe to which all kernels can be compiled. Thus, we choose a 128-bit wide fabric made up of eight-bit PEs with eight registers each.

[4]Eight PEs are required to transpose the data for the two-dimensional DCT.

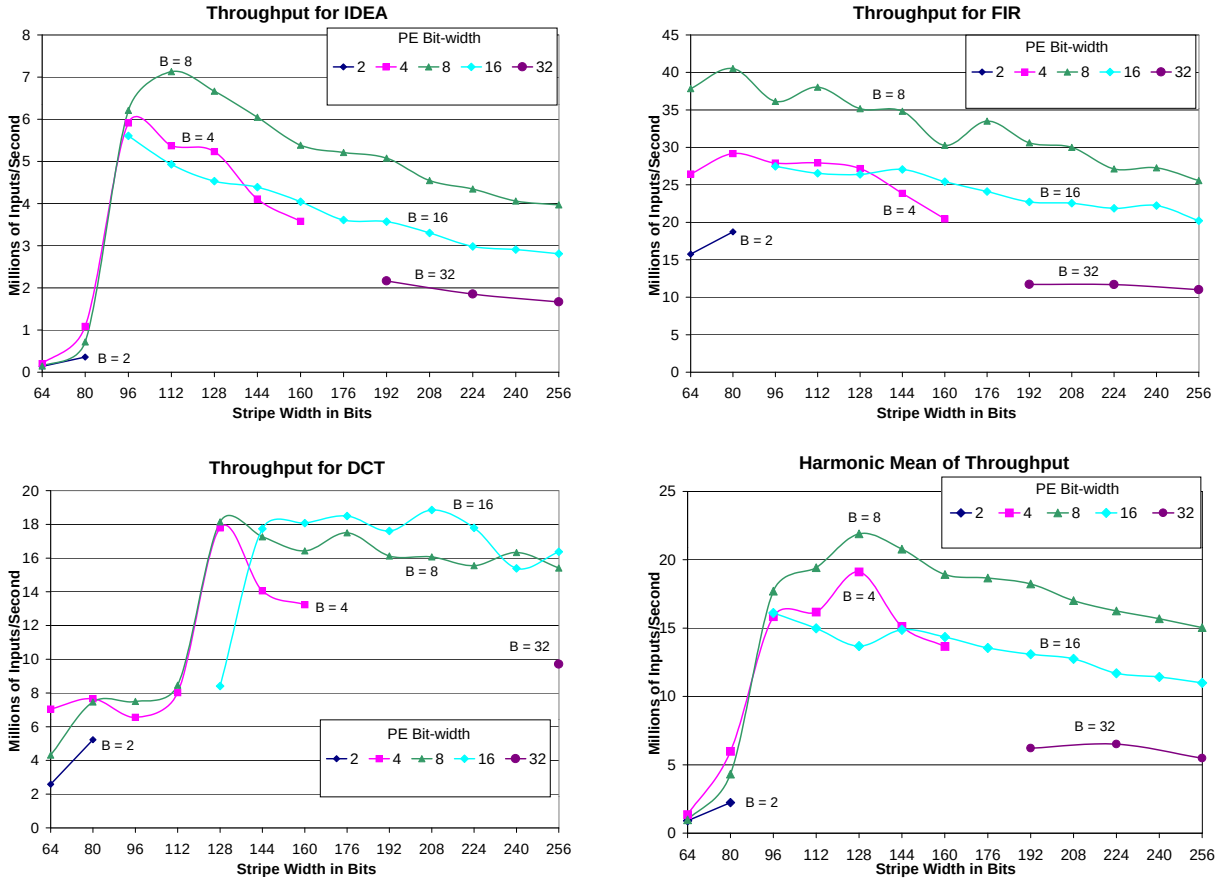

Figure 11. *The throughput for various kernels on a 100MHz PipeRench. The kernels use up to 8 registers.*

5.6. Performance Versus General-Purpose Processors

Using the eight-bit PE 128-bit stripe with eight registers we compare the performance of PipeRench to that of a general-purpose processor, the UltraSparc-II running at 300 Mhz. Figure 12 shows the raw speedup for all kernels. This performance is hard to achieve with PipeRench connected via the I/O bus, but a large fraction of the raw speedup is achievable.

Table 1 shows the speedup from using PipeRench versus doing the entire application on the main processor. For PGP, we replace code for IDEA (accounting for 12% of the application) with invocations of PipeRench, reducing the time for this portion of the code to zero and yielding an average speedup of almost 12%. For JPEG, by running the two-dimensional DCT kernel on PipeRench, we obtain an average improvement of about 7.2%. We also find that the PCI bus imposes no serious bottlenecks on the performance of these applications.

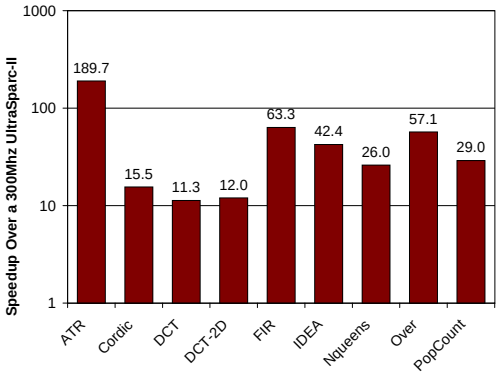

Figure 12. *Speedup of eigth-bit PE, eight registers per PE, 128-bit wide stripe.*

6. Related Work

Numerous other architectural research efforts are focused on efficiently harnessing huge numbers of transis-

Application	Input Size (MB)	Speedup
PGP	0.14	1.07
	9.2	1.12
	18.39	1.12
	27.59	1.12
JPEG	2.02	1.06
	10.13	1.08
	11.74	1.07
	11.75	1.07

Table 1. *Speedups for PGP and JPEG using a 100 MHz 128-bit 53-stripe PipeRench on a 32-bit 33 MHz PCI bus compared to a 330 Mhz UltraSparc-II.*

tors for media-centric computing workloads. The lineage of these systems derives from either FPGAs or existing computer architectures. Those decended from FPGAs are termed "reconfigurable computing systems", and include PRISC [18], DISC [25], NAPA [19], GARP [13], Chimaera [12], One-Chip [26], RAW [23], and RaPiD [8]. None of these reconfigurable computing systems support an architectural abstraction like virtual hardware. In every case, the compiler must be aware of all the system constraints, and if it violates any constraint, it has failed. This makes compilation difficult, slow, and unpredictable. Furthermore, there is no facility in these architectures for forward-compatibility, so that every application needs to be compiled for every new chip. PipeRench offers hardware virtualization, forward compatibility, and easier compilation. Like most of the aforementioned architectures, PipeRench differs from FPGAs in that its basic word size is more than one or two bits and that its interconnect is less general and more efficient for computation.

PipeRench addresses many of the problems faced by other computer architectures. We focus on uniprocessor systems because PipeRench exploits fine-grained parallelism. The most insightful comparisons are to MMX, VLIW, and vector machines.

The mismatch between application data size and native operating data size has been addressed by extending the ISAs of microprocessors to allow a wide data path to be split into multiple parallel data paths, as in Intel's MMX [17]. Obtaining SIMD parallelism to utilize the parallel data paths is nontrivial, and works only for very regular computations where the cost of data alignment does not overwhelm the gain in parallelism. PipeRench has a rich interconnect to provide for alignment and allows PEs to have different configurations so that parallelism need not be SIMD.

VLIW architectures are designed to exploit dataflow parallelism that can be determined at compile time [7]. VLIWs have extremely high instruction bandwidth demands. A single PipeRench stripe is similar to a VLIW processor using many small, simple functional units. But in PipeRench, after the stripe is configured, it is used to perform the same computation on a large data set, thereby amortizing the instructions over more data.

The instruction bandwidth issue has been addressed by vector microprocessors such as T0 [24] and IRAM [14]. The problem with vector architectures is that the vector register file is a physical or logical bottleneck that limits scalability. Allocating additional functional units in a vector processor requires an additional port on the vector register file. The physical bottleneck of the register file can be ameliorated by providing direct forwarding paths to allow chained operations to bypass the register file, as in T0 [24]. This places large demands on the issue hardware. A logical bottleneck is caused by the limited namespace of the register file. This can be addressed by implementing register renaming to avoid false dependencies. Thus, vector microprocessors are subject to the same complexities in issue and control hardware design as modern superscalar processors. All connections in PipeRench are local, and there is no central logical or physical bottleneck. Therefore, the number of functional units can grow without increasing the complexity of the issue and control hardware.

7. Future Work and Conclusions

In this paper we have described a new reconfigurable computing architecture, PipeRench, which emphasizes performance on future computing workloads. PipeRench uses pipelined reconfiguration to overcome many of the difficulties faced by previous attempts to use reconfigurable computing to tackle these important applications. PipeRench enables fast, robust compilers; supports forward compatibility; and virtualizes hardware, removing the fixed size constraint present in other fabrics. As a result, the designer base is broadened, development cycles are shortened, and application developers can amortize the cost of development over multiple process generations.

We first examined computational density of the fabric, by automatically synthesizing hardware based on a number of architectural parameters, including: size of the PE, the number of PEs, and the number of registers. Raw computational density is relatively flat across the space of architectures. The architectural parameters could only be tuned when we had a retargetable compiler and could measure the amount of exploitable computational power in the fabric.

Using the compiler and hardware synthesis flow in tandem, we found that PEs with bit-widths of eight are the best compromise between flexibility and efficiency across a broad range of kernels. When these PEs are arranged in moderately wide stripes (e.g. 128 bits wide) we can

obtain significant performance improvements over general-purpose processors, in some cases achieving improvement of two orders of magnitude. These performance numbers are conservative. Both hardware performance and compiler efficiency can be significantly optimized.

We are currently building a PCI-based board that will include one or more PipeRench chips. Although PipeRench is currently being built into a system as an attached processor, we are examining how to move it closer to the processor. We expect that just as the computing demands of the past decades forced floating-point processors to become floating-point units, the computing workloads of the near future will cause PipeRench to move from an attached processor to reconfigurable unit.

8. Acknowledgements

The authors wish thank the reviewers for the helpful comments. This work was supported by DARPA contract DABT63-96-C-0083. We also received financial support from Altera Corporation, and technical support from STMicroelectronics.

References

[1] P. Bertin and H. Touati. PAM programming environments: Practice and experience. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 133–138, Napa, CA, Apr. 1994.

[2] J. Blinn. Fugue for MMX. *IEEE Computer Graphics and Applications*, pages 88–93, March-April 1997.

[3] M. Budiu and S. Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays (FPGA '99)*, Montery, CA, Feb. 1999.

[4] D. Buell, J. Arnold, and P. Athanas. *SPLASH2: FPGAs in a custom computing machine*. AW, 196.

[5] S. Cadambi, J. Weener, S. Goldstein, H. Schmit, and D. Thomas. Managing pipeline-reconfigurable FPGAs. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.

[6] D. Cherepacha and D. Lewis. A datapath oriented architecture for FPGAs. In *Second International ACM/SIGDA Workshop on Field Programmable Gate Arrays*, 1994.

[7] R. P. Colwell, R. P. Nix, J. J. O'Donenell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of ASPLOS-II*, pages 180–192, Mar. 1987.

[8] D. Cronquist, P. Franklin, S. Berg, and C. Ebling. Specifying and compiling applications for RaPiD. In K. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 116–127, Napa, CA, Apr. 1998. IEEE Computer Society, IEEE Computer Society Press.

[9] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, September 1996.

[10] K. Diefendorff and R. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, 30(9):43–45, September 1997.

[11] S. Hauck. The roles of FPGAs in reprogrammable systems. *Proceedings of the IEEE*, pages 615–638, Apr. 1998.

[12] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pages 87–96, April 1997.

[13] J. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 24–33, April 1997.

[14] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, pages 75–78, September 1997.

[15] S. Kumar and et al. Timimg sensitivity stressmark. Technical Report CDRL A001, Honeywell, Inc., January 1997. `http://www.htc.honeywell.com/projects/acsbench/`.

[16] C. Loeffler, A. Ligtenberg, and G. Moschytz. Practical fast 1-d dct algorithms with 11 multiplications. In *Proc. International Conference on Acoustics Speech, and Signal Processing 1989 (ICASSP '89)*, pages 9880–991, 1989.

[17] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[18] R. Razdan and M. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *MICRO-27*, pages 172–180, November 1994.

[19] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale. The NAPA adaptive processing architecture. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, April 1998.

[20] H. Schmit. Incremental reconfiguration for pipelined applications. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 47–55, Napa, CA, Apr. 1997.

[21] B. Schneier. The IDEA encryption algorithm. *Dr. Dobb's Journal*, 18(13):50, 52, 54, 56, December 1993.

[22] J. Villasenor, B. Schoner, K. Chia, and C. Zapata. Configurable computing solutions for automatic target recognition. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70–79, Napa, CA, Apr. 1996.

[23] E. Waingold, M. Taylor, D. Srikrishna, et al. Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, September 1997.

[24] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, March 1996.

[25] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In P. Athanas and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 99–107, Napa, CA, Apr. 1995.

[26] R. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.