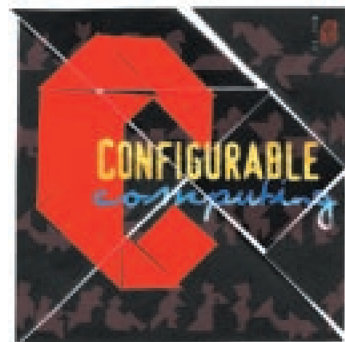
PipeRench: A Reconfigurable Architecture and Compiler



Reconfigurable computing will change the way computing systems are designed, built, and used. PipeRench, a new reconfigurable fabric, combines the flexibility of general-purpose processors with the efficiency of customized hardware to achieve extreme performance speedup.

Seth Copen Goldstein
Herman Schmit
Mihai Budiu
Srihari Cadambi
Matt Moe
R. Reed Taylor
Carnegie Mellon University

Highly specialized embedded computer systems abound, and workloads for computing devices are rapidly changing. General–purpose processors are struggling to efficiently meet these applications' disparate needs, and custom hardware is rarely feasible. The time is ripe for reconfigurable computing, which combines the flexibility of general-purpose processors with the efficiency of custom hardware. PipeRench, a new architecture for reconfigurable computing, and its associated compiler do just that. Combined with a traditional digital signal processor, microcontroller, or general-purpose processor, PipeRench can support a system's various computing needs without requiring custom hardware.

PipeRench is a *reconfigurable fabric*—an interconnected parallel-processing network of logic and storage processing elements. By virtualizing the hardware, PipeRench overcomes the disadvantages of using field-programmable gate arrays as reconfigurable computing fabrics. Unlike FPGAs, PipeRench is designed to efficiently handle computations. Using a technique called *pipeline reconfiguration*, PipeRench improves compilation time, reconfiguration time, and forward compatibility. PipeRench's architectural parameters (including logic block granularity) optimize the performance of a suite of kernels, balancing the compiler's needs against the constraints of deep-submicron process technology.

PipeRench is particularly suitable for stream-based media applications or any applications that rely on simple, regular computations on large sets of small data elements. Conventional processors lag in efficiency because of forced serialization of intrinsically parallel operations; wasted space (small data elements do not use the processor's wide data path); and excessive instruction bandwidth for regular, dataflow-dominated computations on large data sets. A system using a reconfigurable fabric such as PipeRench can

- increase flexibility,
- decrease design time,
- extend system life,
- reduce part count,
- reduce development and maintenance costs, and
- reduce chip fabrication costs through defect tolerance.

PROBLEMS WITH COMMERCIAL FPGAS

Initial performance results with FPGAs were impressive. However, commercial FPGAs have inherent shortcomings, which heretofore made reconfigurable computing impractical for mainstream computing:

- *Logic granularity.* FPGAs are designed for logic replacement. The functional units' granularity is optimized to replace random logic, not to perform multimedia computations.

 0018-9162/00/$10.00 © 2000 IEEE

- *Configuration time.* The time to load a configuration in the fabric ranges from hundreds of microseconds to hundreds of milliseconds. For FPGAs to improve processing speed over that of a general-purpose processor, they must amortize this start-up latency over huge data sets, limiting their applicability.
- *Forward compatibility.* FPGAs require redesign or recompilation to benefit from future chip generations.
- *Hard constraints.* FPGAs can implement only kernels of a fixed and relatively small size. This size restriction makes compilation difficult and causes large, unpredictable discontinuities between kernel size and performance.
- *Compilation time.* A kernel's synthesis, placement, and routing design phases take hundreds of seconds as long as the compilation of the same kernel for a general-purpose processor.

PipeRench overcomes these problems and maintains outstanding performance by virtualizing the hardware.

PIPELINED RECONFIGURATION

A configuration's static nature causes two significant problems: A computation may require more hardware than is available, and a single hardware design cannot exploit the additional resources that will inevitably become available in future process generations. A technique called pipelined reconfiguration implements a large logical configuration on a small piece of hardware through rapid reconfiguration of that hardware.[1] With this technique, the compiler is no longer responsible for satisfying fixed hardware constraints. In addition, a design's performance improves in proportion to the amount of hardware allocated to that design.

Pipelined reconfiguration involves virtualizing pipelined computations by breaking a single static configuration into pieces that correspond to pipeline stages in the application. Each pipeline stage is loaded, one per cycle, into the fabric. This makes performing the computation possible, even if the entire configuration is never present in the fabric at one time.

Figure 1 illustrates the virtualization process, showing a five-stage pipeline virtualized on a three-stage fabric. Figure 1a shows the five-stage application and each pipeline stage's state in seven consecutive cycles. Figure 1b shows the state of the physical stages in the fabric as it executes this application. In this example, virtual pipe stage 1 is configured in cycle 1 and ready to execute in the next cycle; it executes for two cycles. There is no physical pipe stage 4; therefore, in cycle 4, the fourth virtual pipe stage is configured in physical pipe stage 1, replacing the first virtual stage. Once the pipeline is full, every five cycles generates two results for two consecutive cycles. For example, cycles 2, 3, 7, 8, … consume inputs; and cycles 6, 7, 11, 12, … generate outputs.

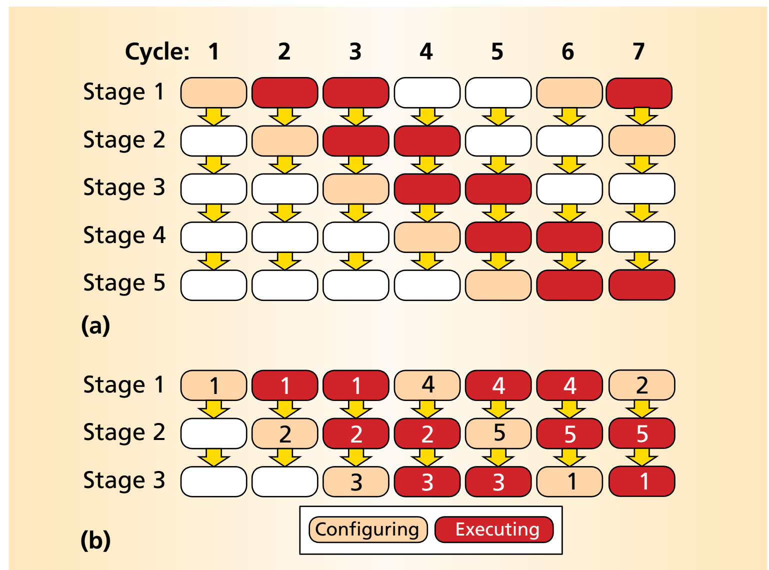


Figure 1. Pipeline reconfiguration showing the virtualization of a five-stage pipeline on a three-stage device: (a) the virtual pipestage and (b) the physical pipeline stage (the numbers in the ovals refer to the virtual pipeline stage).

In general, when PipeRench virtualizes an application with v virtual stages on a device with a capacity of p physical stages ($p < v$), the implementation's throughput is proportional to $(p - 1)/v$. Throughput is a linear function of the device's capacity. Therefore, performance improves not only because of increases in clock frequency but also because of decreases in transistor size, without any redesign, until $p = v$. Thereafter, an application's performance continues to improve only through increased clock frequency.

Because some stages are configured while others are executed, reconfiguration does not decrease performance. As the pipeline fills with data, the system configures stages of the computation before that data. In fact, even if there is no virtualization, configuration time is equivalent to the application's pipeline fill time and so does not reduce the application's maximum throughput. A successful pipelined reconfiguration should configure a physical pipe stage in one cycle. To achieve this, we connected a wide on-chip configuration buffer to the physical fabric. A small controller manages the configuration process.

Virtualization through pipelined reconfiguration imposes some constraints on the kinds of computations that can be implemented on the fabric. The most restrictive is that the state in any pipeline stage must be a function of only that stage's and the previous stage's current state. In other words, cyclic dependencies must fit within one pipeline stage. This restriction also rules out connections between consecutive stages. Nor do we allow connections between every other stage. However, we do allow virtual connections between distant stages.

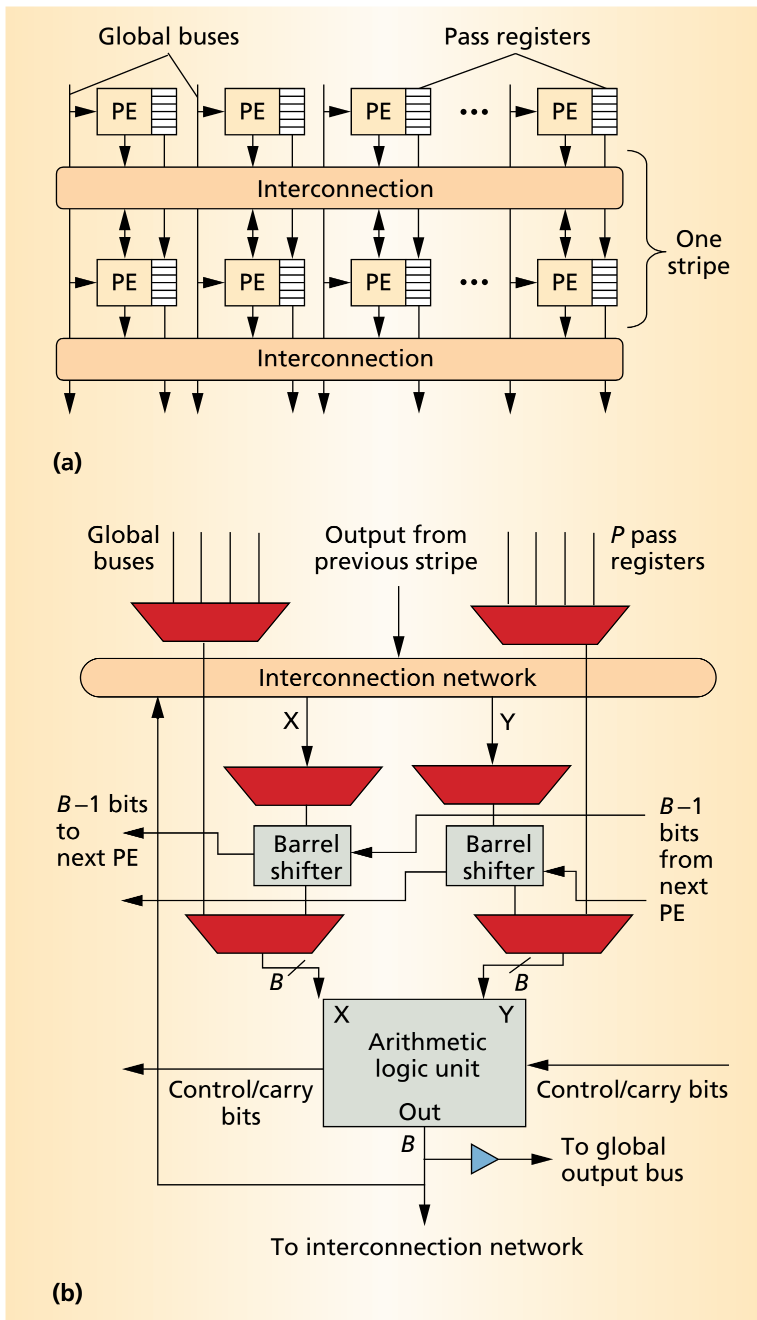


Figure 2. PipeRench architecture: (a) each stripe contains processing elements (PEs) and an interconnection; (b) a detailed view of a PE and its connections.

THE PIPERENCH ARCHITECTURE

In its current implementation, PipeRench is an attached processor. Figure 2a is an abstract view of the PipeRench architectural class, and Figure 2b is a more detailed view of a processing element (PE). PipeRench contains a set of physical pipeline stages called *stripes* (see the "How a Reconfigurable System Handles Computations" sidebar). Each stripe has an interconnection network and a set of PEs.

Each PE contains an arithmetic logic unit and a pass register file. Each ALU contains lookup tables (LUTs) and extra circuitry for carry chains, zero detection, and so on. Designers implement combinational logic using a set of N B-bit-wide ALUs. The ALU operation is static while a particular virtual stripe resides in a physical stripe. Designers can cascade the carry lines of PipeRench's ALUs to construct wider ALUs, and chaining ALUs together via the interconnection network can build complex combinational functions.

Through the interconnection network, PEs can access operands from registered outputs of the previous stripe, as well as registered or unregistered outputs of the other PEs in the same stripe. Because of hardware virtualization constraints, no buses can connect consecutive stripes. However, the PEs access global I/O buses. These buses are necessary because an application's pipeline stages may physically reside in any of the fabric's stripes. Inputs to and outputs from the application must use a global bus to get to their destination.

The pass register file provides efficient pipelined interstripe connections. A program can write the ALU's output to any of the P registers in the pass register file. If the ALU does not write to a particular register, that register's value will come from the value in the previous stripe's corresponding pass register.

The pass register file provides a pipelined interconnection from a PE in one stripe to the corresponding PE in subsequent stripes. For data values to move laterally within a stripe, they must use the interconnection network (see Figure 2a). In each stripe, the interconnection network accepts inputs from each PE in that stripe, plus one of the register values from the previous stripe. Moreover, a barrel shifter in each PE shifts its inputs $B - 1$ bits to the left (see Figure 2b). Thus, PipeRench can handle the data alignments necessary for word-based arithmetic.

PIPERENCH COMPILER

Reconfigurable computing's success depends not only on good hardware fabrics but also on compilers that can quickly create efficient configurations. Taking advantage of PipeRench's hardware virtualization, we have developed a compiler that trades off configuration size for compilation speed. Thus, fitting the design into a limited physical area is not a primary concern.

We parameterized the compiler to facilitate experimenting with different architectures. The compiler begins by reading a description of the architecture. This description includes the number of PEs per stripe, each PE's bit width, the number of pass registers per PE, the interconnection topology, PE delay characteristics, and so on.

The source language is a *dataflow intermediate language*. A single-assignment language with C operators, DIL allows, but doesn't require, programmers to specify the bit width of variables rather than defaulting to

How a Reconfigurable System Handles Computations

A reconfigurable system partitions computations between the fabric and the system's other execution units. The fabric does reconfigurable computations; the system's other execution units (for example, processors) do system computations. The greatest benefit arises when the fabric implements the main computation's computationally intensive portions as a pipelined data path.

The system performs reconfigurable computations by configuring the fabric to implement a circuit customized for each particular reconfigurable computation. The compiler embeds computations in a single static configuration rather than an instruction sequence, reducing instruction bandwidth and control overhead. Because the circuit is customized for the computation at hand, function units are sized properly, and the system can realize all statically detectable parallelism (up to the fabric's size limit).

Reconfigurable computations

A reconfigurable fabric can outperform a general-purpose processor for computations that

- operate on bit widths that differ from the processor's basic word size,
- have data dependencies that let multiple function units operate in parallel,
- contain basic operations that can combine into a single specialized operation,
- can be pipelined,
- enable constant propagation to reduce operation complexity, or
- reuse the input values many times.

Reconfigurable fabrics give the computational data path more flexibility. However, their utility and applicability depend on the interaction between reconfigurable and system computations, the interface between the fabric and the system, and the way a configuration loads onto the fabric.

We divide reconfigurable computations into two categories: *Stream-based functions* process large, regular data input streams, potentially produce a large data output stream, and have little control interaction with the rest of the computation. *Custom instructions* take a few inputs and produce a few outputs, execute intermittently, and have tight control interactions with the other processes. Thus, stream-based functions are suitable for a system where the reconfigurable fabric is not directly coupled to the processor, whereas custom instructions are usually beneficial only when the fabric is closely coupled with the processor.

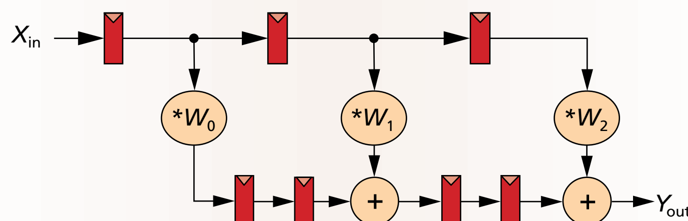
A highly pipelineable streaming function

A reconfigurable fabric is most effective when implementing entire pipelines from applications. A classic example is a finite-impulse response filter. A streaming function, the FIR filter samples an input stream every cycle to convert it into an output stream. Figure A shows the filter's C code and hardware implementation.

The filter's hardware circuit combines bit-value analysis, software pipelining, and retiming techniques. Inputs to the filter are usually between 8 and 16 bits, underutilizing a typical processor's function units. The reconfigurable fabric, on the other hand, constructs adders and multipliers of the proper width. In fact, because of data accumulation, the first adder is smaller than the last. The filter can also exploit tremendous parallelism, in the best case obtaining a new result every cycle through pipelining. When we map the filter to a reconfigurable fabric, we implement the general-purpose multipliers as constant multipliers, where the constants are the tap weights (W_i values). This system requires less hardware and fewer cycles than a general-purpose multiplier. Furthermore, all the data movement is between short local wires.

```
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];
}
```

(1)



(2)

Figure A. A finite-impulse response filter: (1) the C code; (2) the hardware implementation of a three-tap FIR filter.

a standard width for a given data type. DIL can manipulate arbitrary width integer values and (unless directed explicitly to do otherwise) automatically infers wire widths, thereby preventing any information loss due to overflow or conversions. Aside from this one exception, DIL hides all notions of hardware resources, timing, and physical layout from programmers.

After parsing, the compiler inlines all modules (places each function's definition at that function's call site), unrolls all loops, and generates a straight-line, single-assignment program. Then the bit-value inference pass computes the minimum width required for each wire (and implicitly the amount of logic required for computations). Additionally, when the compiler determines that any of a wire's bits are constant, it uses those constants to help simplify the graph.[2]

After the compiler determines each operator's size, the operator decomposition pass decomposes high-

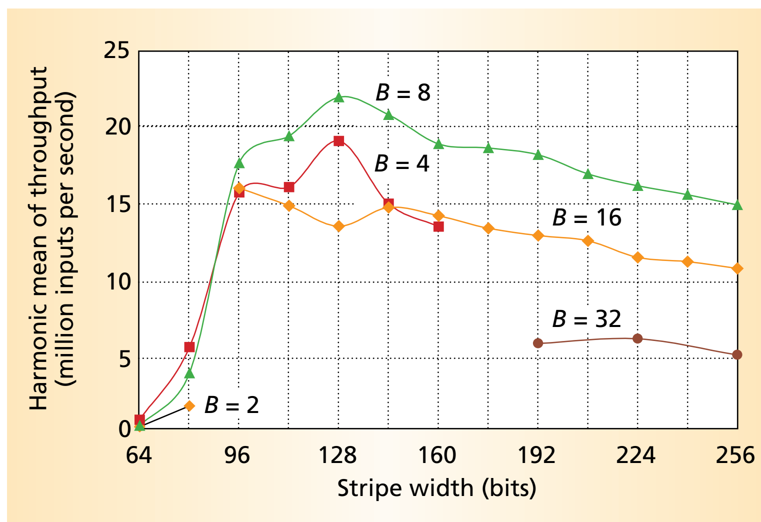


Figure 3. The harmonic mean of throughput for all kernels (each kernel uses up to 8 registers) as a function of stripe width and PE bit width **B***.*

level operators (for example, multiplies become shifts and adds) and decomposes operators that exceed the target cycle time. For example, to avoid carry-chain delays, this pass splits any wide adder (for example, a 45-bit adder) into several smaller adders. This decomposition must also create new operators that handle the routing of the carry bits between the partial sums. Such rather "naïve" decomposition often introduces inefficiencies. Therefore, an operator recomposition pass uses pattern matching to find subgraphs that can it can map to parameterized modules. These modules take advantage of architecture–specific routing and PE capabilities to produce a more efficient set of operators.

The place-and-route algorithm is the key to the compiler's speed.[3] Unlike many place-and-route algorithms, ours is a deterministic, linear-time, greedy algorithm. It runs between two and three orders of magnitude faster than commercial tools yet yields configurations with at most four times as many bit operations. For example, our compiler can handle a 1D discrete cosine transform in 2.4 seconds, but the Synopsys Design Compiler with the Xilinx Design Manager takes 75 *minutes*. The Xilinx configuration has four times fewer bit operations. However, our focus is compiling data paths, and hardware virtualization lets us sacrifice some efficiency for compilation speed.

EVALUATING PIPERENCH'S PERFORMANCE

Using a compiler and CAD tools, we conducted a study to optimize PipeRench's stripe width *N*, PE word width *B*, and number of pass registers per PE *P*. One of the kernels, the International Data Encryption Algorithm (IDEA), illustrates how dynamic compilation can increase performance. In this example, the reconfigurable system actually outperforms existing custom hardware.

Kernels

To evaluate PipeRench's performance, we chose nine kernels on the basis of their importance, recognition as industry performance benchmarks, and ability to fit into our computational model:[4]

- *Automatic target recognition (ATR)* implements the shape-sum kernel of the Sandia algorithm for automatic target recognition.
- *Cordic* implements the Honeywell timing benchmark for Cordic vector rotations.
- *DCT* is a 1D, 8-point discrete cosine transform.
- *DCT-2D* is a 2D discrete cosine transform.
- *FIR* is a finite-impulse response filter with 20 taps and 8-bit coefficients.
- *IDEA* implements a complete 8-round International Data Encryption Algorithm.
- *Nqueens* is an evaluator for the *N* queens problem on an 8 × 8 board.
- *Over* implements the Porter-Duff over operator.
- *PopCount* is a custom instruction implementing a population count instruction.

Methodology

We used CAD tools to synthesize a stripe on the basis of parameters *N*, *B*, and *P* in a 0.25-micron five-metal-layer process. We joined this automatically synthesized layout with a custom layout for the interconnection. Using the final layout, we determined the number of physical stripes that can fit in our silicon budget of 50 mm^2 (5 mm by 10 mm). We also determined the delay characteristics of the stripe's components—for example, LUTs, carry chains, and interconnections. (We reserve another 50 mm^2 for memory to store virtual stripes, control, and chip I/O.) The compiler uses the delay characteristics and the number of registers to create configurations for each architectural instance, yielding a design of a certain number of stripes at a particular frequency. We then determined the kernel's overall speed in terms of throughput, for each architectural instance.

We wrote the kernels in DIL. The DIL compiler automatically synthesizes, places, and routes the design for each parameterized architecture.

Fabric

Consider the region of space bounded by PE bit widths *B* of 2, 4, 8, 16, and 32; stripe widths ($N \times B$) from 64 to 256 bits; and number of registers *P* equaling 2, 4, 8, and 16. Two main constraints determine which parameters generate realizable fabrics: the stripe width and the number of vertical wires that must pass over each stripe. Stripe width depends on the number and size of PEs, as well as the number of registers allocated to each PE. The number of vertical wires depends on the number and width of global buses, pass registers, and configuration bits.

As the number of registers increases, computational density decreases. However, pass registers are an important component of the interstripe interconnection. Reducing their number increases routing pressure, which decreases stripe utilization. The best balance between computational density and stripe utilization typically comes from using eight registers (*P* equals 8).

Figure 3 shows the harmonic mean of the throughput for all the kernels on a 100-MHz PipeRench for various stripe widths and PE sizes. Although wider PE sizes create fabrics with higher computational density, the kernels' natural data sizes are smaller, causing 32-bit PEs to be underused. On the other end of the spectrum, 2-bit PEs are not competitive, because of the increased time for arithmetic operations, the lack of raw computational density, and the increased number of configuration bits needed per application.

Examining each kernel's individual performance, we find that the kernels' characteristics influence which parameters are best. 8-bit PEs strike a balance between utilization and carry-chain delay. For many kernels, enough parallelism is available to keep the wider stripes busy, but these stripes have fewer registers. This means more stripes in the implementation and, therefore, lower overall throughput. Therefore, we built PipeRench as a 128-bit-wide fabric having 8-bit PEs with eight registers each.

Performance and forward compatibility

We compared PipeRench's performance with the performance of an Ultrasparc II running at 300 MHz. Figure 4a shows the raw speedup for each kernel. Although achieving this performance can be difficult with the reconfigurable system connected through the I/O bus, most of the raw speedup is obtainable. In fact, the I/O bandwidth required is inversely proportional to the amount of hardware virtualization that occurs. If bandwidth continues to scale as feature sizes shrink, I/O bandwidth is not an issue. For example, IDEA has a virtualization factor of around 12, which means on average it requires only one input every 12 cycles (that is, fewer than 8 bits per cycle).

As process technology improves, configurations will run faster because of both the increased number of stripes that will fit in a given area and increased clock speeds. Figure 4b shows how each kernel's performance should scale (without recompilation) for process generations through 70 nanometers. This scaling is based on the Semiconductor Industry Association roadmap for both the number of transistors available per unit area and the clock speed. All kernels do not scale in parallel in the same way, because each kernel has a different number of virtual stripes. When a kernel fits completely in the physical fabric, it can no longer benefit from the increased number of stripes, without recompilation. These scaling effects are conservative, as they use the low-volume application-specific integrated circuit (ASIC) transistor counts and clock speeds.

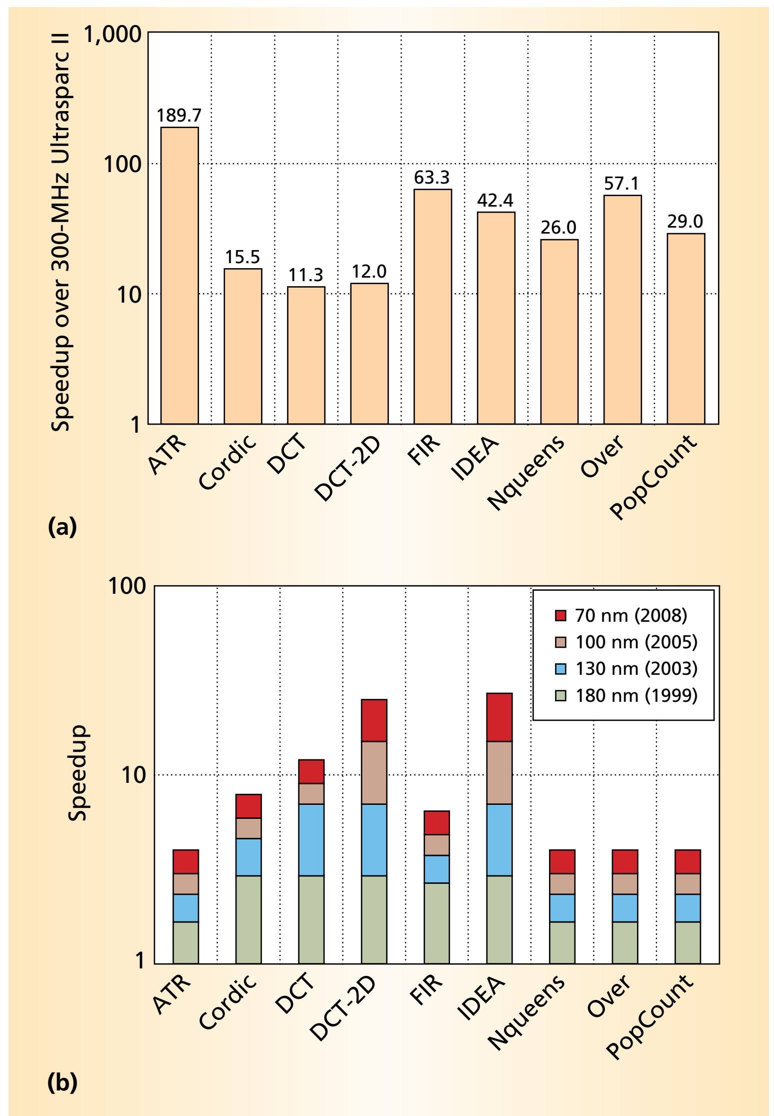


Figure 4. Speedup for eight kernels: (a) raw speedup of a 128-bit fabric with 8-bit PEs and eight registers per PE, and (b) PipeRench performance scaled with process technology based on The National Technology Roadmap for Semiconductors (Semiconductor Industry Association, San Jose, Calif., 1997), including the year projected for each process generation.

Performance in embedded systems

One challenge for reconfigurable computing in embedded systems is that performance depends on creating custom configurations that can incorporate what is typically considered runtime data. For example, IDEA's performance on PipeRench depends on generating a key-specific configuration (one that can encrypt or decrypt only on the basis of a specific key). The compiler can create a complete, single-key optimized IDEA configuration in less than one minute. However, this is still too long for an embedded system, which must be able to work with any key. We can solve this problem by generating a dynamic configuration.

The IDEA block cipher includes three fundamental operations: addition modulo 2^{16}, 16-bit exclusive OR, and 16×16 multiplication modulo $2^{16} + 1$. The 128-bit key generates 36 16-bit subkeys. One operand of every multiplication operation in the algorithm is

Table 1. Comparison of IDEA implementations with other designs.

Processor	Clock speed (MHz)	Clocks per block	Throughput (Mbytes/s)
PipeRench	100	6.3	126.6
Pentium II using MMX[5]	450	358.0	10.0
Pentium[6]	450 (scaled)	590.0	6.1
IDEACrypt Kernel	100	3.0	90.0

a subkey. This multiplication is essential for customizing an IDEA configuration.

Compilation has two components: generating optimized multipliers and generating the rest of the pipeline. With an interface that satisfies the multipliers and the rest of the pipeline, the compiler can perform these two tasks independently. The compiler generates the nonmultiplier portion ahead of time, assuming certain inputs and outputs for the multipliers.

To generate the multipliers themselves, a small controller rapidly returns configuration bits for the stripes that perform the multiplication. This controller has a lookup table that converts constant multiplicands into the necessary configuration bits. PipeRench's regular, small configuration bit stream makes this generation very easy. Using canonical signed digit (CSD) representations for the subkeys, a multiplier takes 2.06 stripes on average. The result is a complete IDEA pipeline of only 177 stripes.

Table 1 compares PipeRench's implementation with two optimized software implementations running on state-of-the-art processors, and a custom VLSI design. PipeRench outperforms the processors by more than 10 times.

Surprisingly, PipeRench also outperforms the 0.25-micron IDEACrypt Kernel from Ascom Systems Ltd.[7] This is due to several factors: First, the PipeRench implementation of IDEA does not include the time taken to generate keys. PipeRench targets streaming-media applications, where key generation involves only a small preprocessing step. Second, because of PipeRench's pipelined nature, it effectively pipelines IDEA into 177 stages, rendering cipher block chaining (CBC) and other chained implementations impractical. Nevertheless, PipeRench's raw throughput is 40 percent faster than that of a fully custom chip.

Reconfigurable computing is changing the face of custom hardware. In the past, "custom hardware" used general-purpose elements; a reconfigurable device customizes the elements. For example, reconfigurable computing lets an IDEA implementation use custom, rather than general-purpose, multipliers for a specific key at runtime. This level of customization can yield tremendous performance improvements. Additionally, because reconfigurable devices can serve many purposes, they will likely become common off-the-shelf parts, further reducing system cost.

At some point in the near future, current processor technology will cease reaping the gains of Moore's law. Reconfigurable computing has the potential to become a vital component in the next generation of computation devices. The current implementation of PipeRench is the first step in realizing this potential. Because PipeRench is an attached processor, bandwidth between PipeRench, the main memory, and the processor is limited. This places significant limitations on the types of applications that can realize speedup. Nevertheless, the attached processor is merely the initial phase in the evolution of reconfigurable processors. Just as floating-point computations migrated from software emulation to attached processors, then to coprocessors, and finally to full incorporation in processor instruction set architectures, so too will reconfigurable computing eventually become an integral part of the CPU. ❖

References

1. H. Schmit et al., "Pipeline Reconfigurable FPGAs," *J. VLSI Signal Processing*, 1999.
2. M. Budiu, "Detecting and Exploiting Narrow Bitwidth Computations," *Proc. 2nd Ann. CMU Symp. Computer Systems*, Carnegie Mellon Univ., Pittsburgh, 1999.
3. M. Budiu and S.C. Goldstein, "Fast Compilation for Pipelined Reconfigurable Fabrics," *Proc. 1999 ACM/SIGDA 7th Int'l Symp. Field Programmable Gate Arrays*, ACM Press, New York, 1999.
4. S.C. Goldstein et al., "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," *Proc. 26th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 28-39.
5. H. Lipmaa, "IDEA: A Cipher for Multimedia Architectures?" *Proc. Selected Areas in Cryptography*, *Lecture Notes in Computer Science*, Vol. 1556, Springer Verlag, New York, 1998, pp. 248-263.
6. B. Preneel, V. Rijmen, and A. Bosselaers, "Recent Developments in the Design of Conventional Cryptographic Algorithms," *Proc. Computer Security and Industrial Cryptography*, *Lecture Notes in Computer Science*, Vol. 1528, Springer-Verlag, New York, 1998, pp. 106-131.
7. Ascom Systec Ltd., "IDEACrypt Kernel," http://www.ascom.ch/infosec/idea/kernel.html.

Seth Copen Goldstein *is an assistant professor in the School of Computer Science at Carnegie Mellon University. His interests include reconfigurable computing systems, with an emphasis on compilation and system architectures. Goldstein has a BSE in electrical engineering and computer science from Princeton University and a PhD in computer science from the University of California at Berkeley. Contact him at seth@cs.cmu.edu.*

Herman Schmit *is an assistant professor at Carnegie Mellon University. His interests include reconfigurable computing systems, field-programmable gate arrays, and IP-based design methodologies. Schmit has a BSE in computer science engineering from the University of Pennsylvania and a PhD in electrical and computer engineering from Carnegie Mellon University. Contact him at herman@ece.cmu.edu.*

Mihai Budiu *is a PhD student in the Department of Computer Science at Carnegie Mellon University. His interests include reconfigurable hardware design and high-level language compilation for systems containing reconfigurable hardware devices. Budiu has a BS and an MS in computer science from the Polytechnica University of Bucharest, Romania. Contact him at mihaib+@cs.cmu.edu.*

Srihari Cadambi *is a PhD student in the Department of Electrical and Computer Engineering at Carnegie Mellon University. His interests include hardware compilation—especially synthesis, technology mapping, and place-and-route for reconfigurable and other novel architectures. Cadambi has a B.Tech in electronics engineering from the Indian Institute of Technology and an MS electrical and computer engineering from the University of Massachusetts. Contact him at cadambi@ece.cmu.edu.*

Matt Moe *is a PhD student in the Department of Electrical and Computer Engineering at Carnegie Mellon University. He is making specialized reconfigurable architectures for different application domains to realize faster and smaller implementations. Moe has a BS and an MS in electrical and computer engineering. Contact him at moe@ece.cmu.edu.*

R. Reed Taylor *is a graduate student in the Department of Electrical and Computer Engineering at Carnegie Mellon University. His interests include reconfigurable hardware, near-Shannon-limit error codes, and cryptography. Taylor has a BS in electrical and computer engineering from Carnegie Mellon University. Contact him at rt2i@andrew.cmu.edu.*