INVITED PAPER

# Fundamental Underpinnings of Reconfigurable Computing Architectures

*This paper introduces a unified framework for understanding advantages and tradeoffs in reconfigurable computing architectures.*

By André DeHon, *Member IEEE*

**ABSTRACT** | Reconfigurable architectures are a distinct point in the larger design space that includes programmable processors and nonprogrammable fixed-function devices. In this paper, we identify the major parameters that distinguish architectures in this design space and draw connections between these parameters and physical requirements (e.g., energy, delay, and area) and application characteristics (e.g., word width, locality). Building on these connections, we identify the fundamental advantages that reconfigurable architectures can offer.

**KEYWORDS** | Computer architecture; field-programmable gate arrays (FPGAs); integrated-circuit (IC) interconnections; locality; microprocessors; multiprocessor interconnection networks; reconfigurable architectures; Rent's rule; spatial computing

## I. INTRODUCTION

At least as far back as Alan Turing, we have known that it is possible to design universal machines that can implement any computation simply by programming the machine after it has been created—universal Turing machine (UTM) [1], [2]. With appropriate technology, we can fabricate computing engines that require only programming after fabrication—postfabrication programming—to be configured to perform any computable operation. With the phenomenal advance in semiconductor technology over the past five decades (i.e., Moore's Law [3], [4]), we can build postfabrication reconfigurable machines with an enormous number of raw resources (e.g., gates, memories) relatively inexpensively. With these resources, we open up a huge design space of potential organizations for these postfabrication reconfigurable machines. In this paper, we identify the primary architectural dimensions in this design space (number of computational operators, balance of instruction and data memories with compute operators, placement of instructions, organization and balance of communication resources) and show how choices in these dimensions impact the characteristics and efficiency of postfabrication computing machines (area, performance, energy).

Postfabrication reconfigurable machines are distinguished from fixed-function architectures [e.g., application-specific integrated circuits (ASICs)] by their universal programmability. A fixed-function machine can perform only a single or limited set of tasks. It cannot perform computational tasks that lie outside this limited set. For example, a dedicated video-compression (e.g., MPEG) component cannot render a portable document format (PDF) file for viewing or printing. A postfabrication programmable device could do both and many more tasks. Since postfabrication reconfigurable devices can be used for a large number of tasks, the fixed cost of designing the device [i.e., nonrecurring engineering costs (NRE)] can be shared across a large set of users, allowing large volume applicability and sales and thereby reducing the portion of the development cost that must be applied to each device sold. Using postfabrication reconfigurable devices, the fabrication time for the physical computing device does not need to be in the critical path from idea to realization; it is only necessary to program the device to obtain an implementation of the new computing task. This reduces the

Manuscript received August 18, 2014; revised November 20, 2014; accepted December 23, 2014. Date of current version April 14, 2015. This work was supported by DARPA/CMO under Contract HR0011-13-C-0005.
Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.
The author is with the Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA 19104 USA (e-mail: andre@ieee.org).

Digital Object Identifier: 10.1109/JPROC.2014.2387696

time to market (TTM) for new ideas and features. Furthermore, since the device is reconfigurable, it can be configured insystem even after the product has been shipped to customers and put into operation; this allows feature additions and bug fixes to occur throughout the lifetime of the device without replacing hardware. This facility can also be used to adapt to device aging and failure during operation (See Section XV.) Nonetheless, postfabrication reconfigurability does have costs. The capabilities to hold programs and reconfigure functionality cost resources (e.g., transistors, wires, silicon area), often result in lower performance, higher area, and higher energy than a fixed-function component.

The key distinguishing feature of postfabrication reconfigurable machines is the configurable bits that instruct the behavior of the machines—the *instructions*. Consequently, most of the key architectural questions when designing postfabrication reconfigurable architectures revolve around instructions: How are the instructions organized? How are they defined? How do they control the computational elements? How many computational elements do they control? Can the behavior of computational elements and, hence, their applied instructions, change during a computation? Another key question resolves around interconnect for data: When there is more than one computational element, how is data movement among computational elements performed and controlled? Answers to these questions have a dramatic impact on the resources and costs of the computation (area, performance, energy).

In particular, we would like to know what instruction organizations minimize:

- area for the computational engine;
- time required to perform the computational task;
- energy required to perform the computational task;
- time required to perform the computational task within a fixed area limit;
- time required to perform the computational task within a fixed power density limit.

Should we implement a single, heavily shared compute element or a large number of compute elements where each performs a single operation during a computation? How do changes in resources (e.g., as provided by Moore's Law capacity growth) and cost metrics impact the choices we might make around instruction organization?

To ground this discussion, we start by giving examples of familiar instruction architectures at the extreme corners of this space (Section II). To make the connection between instruction choices and resources, we first analyze simplified models of these extremes—sequential processors and spatially reconfigurable devices (Section III). This illustrates the large overhead costs for postfabrication configurability in both cases. Efficient postfabrication architectures exploit common application characteristics to reduce costs. We review an essential application characteristic, locality in the form of Rent's Rule, in Section IV. Armed with this model of locality, we show how this can be exploited to reduce the overhead associated with instructions in processors in Section V. We introduce another common application characteristic in the form of instructions that can be reused across different data components (e.g., SIMD, looping) and evaluate its impact in Section VI. We then see how we can exploit locality to reduce the overhead of data movement while potentially increasing performance through parallelism in Section VII. We see there is still a high energy cost associated with the instructions controlling communication as well as a bottleneck to performance in communication, leading us to spatially reconfigurable designs, such as field-programmable gate arrays (FPGAs), that have richer interconnect and place instruction control local to the interconnect resources (Section VIII). To properly evaluate this, we need to introduce a few ideas from very-large-scale integrated (VLSI) complexity theory (Section IX). This allows us to characterize locality optimized designs at the highly parallel extreme (Section X). With this understanding, we can characterize the design space between the extremes and characterize the tradeoffs within this space (Section XI). In Section XII, we illustrate the impact of mismatches between architectural optimization and application characteristics. After exploring homogeneous architectures, we have the tools to also understand the potential benefits of hybrid architectures (e.g., combining a sequential processor with a reconfigurable array) in Section XIII. In Section XIV, we underscore the role of energy as a key limitation in today's systems. Finally, we can return to fixed-function architectures and identify scenarios that allow postfabrication reconfigurable architectures to achieve superior characteristics (lower area, higher performance, lower energy) compared to fixed-function components (Section XV).

## II. EXAMPLES

### A. Stored-Program Processors

At one extreme, we can build *stored-program processors* that reuse a small computing unit in time over a large computation. This small compute unit is paired with a large instruction memory that instructs the behavior of the compute unit on each cycle of operation. The instruction for a single cycle specifies where to find a set of inputs, the operation to perform on them, and where to place the result. Today's canonical example would be an reduced instruction set computing (RISC) processor (e.g., [5]). At the dawn of general-purpose computing, the canonical example was Eckert and Mauchly's EDVAC, which was documented by von Neumann [6], from which we get the common term "von Neumann Architecture." This stored-program processor invested minimal hardware in the actual computing unit and reused it in time, making it a much more viable and economical general-purpose, postfabrication computing machine than its predecessors when resources were scarce, as they were when we built

computers out of vacuum tubes. However, most of the resources in this style of machine go into memory, not computation, and most of the energy goes into memory. The actual computational units comprise only a very tiny fraction of the resources. As a result, the active computational density—the computations that can be performed per-unit area time—is very low. The earliest postfabrication programmable devices we could build on single-chip LSI circuits were processors, starting with the Intel 4004 in 1971.

### B. Reconfigurable Computing Devices

At the opposite extreme, we have a design that uses one computational unit for every operator in a computational graph. The computation is fully parallel, with no operations being sequentialized on computational units. As a result, each compute unit only needs a single instruction or configuration setting to tell what operation to perform, and we need instructions to control how the compute units are interconnected into the computational graph. Today's canonical example of such a reconfigurable machine is the FPGA, where the configuration bits, the instructions, control the behavior of "gates" and the interconnect between the gates. These configuration instructions do not typically change during a computation. In a broad sense, ENIAC, the predecessor to EDVAC, was similarly configured—with the configuration being performed by program switches, physical plug-board wiring between the computational units, and switches for configuring tables [7]. This parallel form of computation was extravagant in 1946 when ENIAC was unveiled, not to mention laborious to configure; as a result, the resource savings of stored program processors was considered advanced at the time. The parallelism also meant the machine was faster than its stored program successors. It was not possible to build this kind of fully parallel, spatially configured design in VLSI until 1983 [8]. However, decades of Moore's Law Silicon growth have made it more viable to build them today.

### C. Terminology

In a broad sense, the term *reconfigurable architecture* could be applied to the entire space of post-fabrication programmable devices. We have traditionally called the spatially configurable end, as typified by the FPGA, *reconfigurable*. As we will see, there is a continuum between these extreme positions making a strict distinction less clear. It is more valuable to understand the continuum than to draw arbitrary, sharp boundaries.

## III. SIMPLE MODELS AT EXTREMES AND SIMPLE IDEALS

The real machines suggested in the previous section are useful grounding as concrete examples of choices in instruction architecture. However, the real machines come with a large number of differences and features that, while important, can distract from the essence. Consequently, in this section, we consider more simplified and ideal models that illustrate the key architectural effects in play.

For the sake of simple illustration, let us assume our computation can be viewed as a circuit netlist of 4-input gates and state elements. Fig. 1(a) shows an example netlist (full adder) with seven 2-input gates. We will consider architectures that can implement a netlist up to a specified number of gates $N$.

### A. Processor

The extreme stored-program processor architecture (Section II-A) uses a single programmable gate over a series of $N$ cycles—one per gate in the netlist per cycle—in order to evaluate the circuit netlist. The gate is fully programmable and requires $2^4$ bits to specify the full truth table for a 4-input function. We call this programmable gate a lookup table (LUT) since it uses the four input bits to select a bit value from the truth table. We will refer to a 4-input LUT as a 4-LUT throughout this paper. We use an $N$-bit

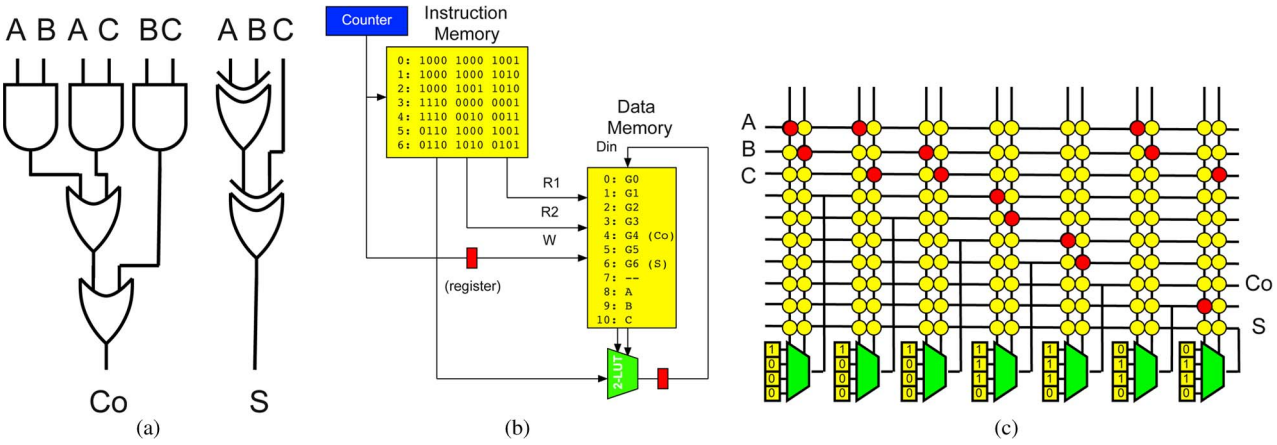


**Fig. 1.** *Example circuit mappings. (a) Circuit netlist. (b) Processor implementation. (c) Reconfigurable implementation.*

data memory to store the outputs of each gate in the original netlist. To evaluate the gate, the processor:

1) reads the values of the four inputs to the gate—the outputs form the four predecessor gates that are stored in the data memory;
2) evaluates the gate on these four bits to produce an output value for the gate;
3) writes this output value into the data memory in a designated location.

To properly evaluate the netlist, the instruction sequence should be ordered so that a gate is only evaluated after its predecessors have been evaluated; that is, the sequence should be a topological ordering of the circuit netlist. To control each gate evaluation, this processor needs an *instruction* that specifies the gate function and the location in data memory for the four data inputs. This will require $2^4 + 4\log_2(N)$ bits in each instruction. We will assume that each output is always written into a location based on its sequence and, therefore, does not require separate control. It will take a sequence of $N$ such instructions in order to evaluate the entire netlist. To apply this sequence, we add an *instruction memory* that is logically $N$ words deep, where each word is $(2^4 + 4\log_2(N))$-bits wide. A counter can then be used as an address to sequence through the $N$ instructions to evaluate the entire netlist. Fig. 1(b) shows how an $N = 7$ instance of such a processor (shown with 2-LUTs for simplicity) is programmed to implement the example netlist shown in Fig. 1(a).

### B. Spatially Reconfigurable

The extreme spatially reconfigurable architecture uses $N$ of the 4-input LUTs to evaluate the $N$ gates in the netlist. As before, each 4-LUT needs $2^4$ bits to specify its function. For the spatial case, we need a way to configurably wire up the four inputs to each gate from the outputs of the $N$ gates. Conceptually, the easiest way to do this is with an $N$-input, $4N$-output crossbar. The crossbar is a set of input wires that crosses a set of output wires with a programmable junction at every wire crossing (Fig. 2). If the input should be connected to the output, the junction is configured to connect the crossing input wire to the crossing output wire. We maintain the invariant that only one such programmable junction is connected—programmed into the "ON" state—for a given output; that is, there is no conflict for the output. Each gate input is associated with a crossbar output. To program up the crossbar, each gate input specifies the crossbar input (gate output) that it wants to receive—equivalently, the crosspoint junction connected to the output wire that should be programmed "ON." This requires $\log_2(N)$ bits for each input, or $4\log_2(N)$ interconnect instruction bits for each gate. Fig. 1(c) shows how an $N = 7$ instance of this spatially reconfigurable architecture (also shown with 2-LUTs for simplicity and consistency) is programmed to implement the example netlist shown in Fig. 1(a).

As we will soon see, the crossbar is a very expensive interconnect structure, and there are much better options to use (Sections VIII and X). We start with the crossbar because it enables a very simple description and conceptual model to illustrate the essence of a spatially reconfigurable architecture.

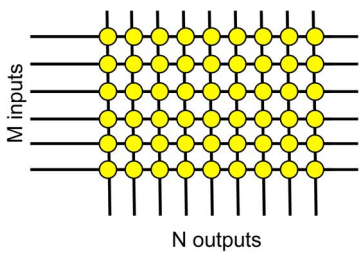


**Fig. 2.** *N* × *M crossbar.*

### C. Analyzing Architectural Characteristics

To develop meaningful comparisons, we must model the memories, switches, and interconnect. For these first two extremes, it suffices to focus on the memory bits and crossbar crosspoints since memories and crossbars will dominate the area, delay, and energy of these implementations. As we explore more sophisticated architectures, we will need to account for wiring complexity (Section IX).

Interconnect wires contribute the largest factor to delay and energy. Consequently, we will need to quantify the size of structures to understand wire length. A key relation to note is that the energy required to switch a wire is linear in the length of the wire.[1] With proper buffering, the delay of a wire is also linear in the length of the wire. For today's VLSI technology, wire capacitance dominates transistor gate capacitance, so we will focus our illustrative modeling on wire capacitance.

*1) Memories:* We consider a memory block where we store $M$, $W$-bit words in one large array. For large memories, the area of the memory is roughly linear in the number of bits stored $(W \cdot M)$. A more precise model for the area of an arbitrary $W$-bit word from a random-access memory (RAM) is [9]

$$A_{\text{rmem}}(W, M) = \left(\sqrt{WMA_{\text{bit}}} + FP\left(\frac{\log_2(M)}{2}\right)\right)^2. \quad (1)$$

Here, $FP$ is the full pitch for wires, including the wire width and the spacing between wires, and $A_{\text{bit}}$ is the area for one RAM bit. The logarithmic term is for the address

[1]Think of the wire as a parallel plate capacitance; the capacitance grows in linear proportion with the length of the wire. Dynamic switching energy is proportional to $C_{\text{wire}}V^2$. With $V$ fixed, energy scales as capacitance, which scales as wire length.

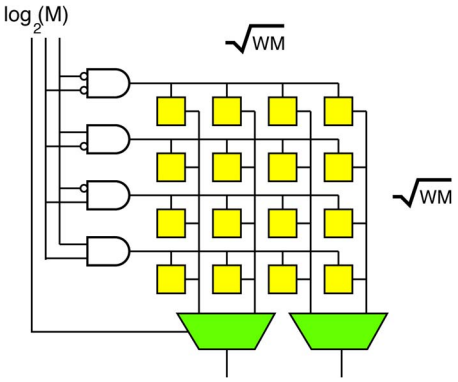


**Fig. 3.** *Memory array (shown with M = 8, W = 2).*

decoder and output selector. We arrange the $W \cdot M$ bits into a $\sqrt{WM} \times \sqrt{WM}$ array (Fig. 3). This means the address lines that access the memory and the word lines that bring data in and out of the memory are of length $\sqrt{WM}$, meaning the delay and energy of a memory reference is also proportional to $\sqrt{WM}$

$$D_{\text{rmem}}(W, M) = 2\sqrt{A_{\text{rmem}}(M, W)} \quad (2)$$

$$E_{\text{rmem}}(W, M) = E_u(\log_2(M) + 2(2W + 2))\sqrt{WMA_{\text{bit}}}. \quad (3)$$

Here, $E_u$ is the energy switched per-unit length of wire.

Sequentially accessed memories, as appropriate for the instruction memories, can avoid the cost of addressing. A simple shift register can activate the appropriate rows and control the output multiplexer. The energy switched for a sequentially accessed memory is

$$E_{\text{smem}}(W, M) = E_u(2(2W + 1))\sqrt{WMA_{\text{bit}}}. \quad (4)$$

The area of the sequential memory is

$$A_{\text{smem}}(W, M) = WMA_{\text{bit}} + \sqrt{WM}A_{\text{shift}} + \sqrt{\frac{M}{W}}A_{\text{shift}} + (\sqrt{WM} - W)A_{\text{mux}}. \quad (5)$$

Instead of using an address decoder, we can use a shift register across the width and height of the array ($A_{\text{shift}}$ terms) for addressing. When we read $W$ bits at a time from the array, we only need one select bit per word ($\sqrt{WM}/W = \sqrt{M/W}$) across the width of the memory array. Finally, we need multiplexing to route the selected word to the output.

*2) Crossbars:* An $N \times M$ crossbar needs $N \cdot M$ crosspoints, typically arranged as $N$ rows of $M$ crosspoints (Fig. 2). The area is directly

$$A_{\text{xbar}}(N, M) = N \cdot M \cdot A_{\text{xpoint}}. \quad (6)$$

The typical crosspoint area $A_{\text{xpoint}}$ is around twice the area of a memory bit ($A_{\text{xpoint}} \approx 2A_{\text{bit}}$). The wires are of length $N\sqrt{A_{\text{xpoint}}}$ and $M\sqrt{A_{\text{xpoint}}}$. This makes the crosspoint delay proportional to $(N+M)\sqrt{A_{\text{xpoint}}}$, and the crosspoint energy

$$E_{\text{xbar}}(N, M) = E_u \cdot 2N \cdot M \cdot \sqrt{A_{\text{xpoint}}} \quad (7)$$

*3) Processor:* With the models for memories and crossbars established, we can now build models to characterize the processor. The processor area is dominated by the two memories

$$A_{\text{proc}} = A_{\text{smem}}\left(2^4 + 4\log_2(N), N\right) + A_{\text{rmem}}(1, N). \quad (8)$$

This gives us an area that grows proportional to $N\log(N)$ when supporting $N$ gates. The delay is also dominated by these memories. As long as we sequentially execute all gates, the instruction memory is pipelineable, so cycle time is dominated by the data memory access, which scales with $\sqrt{N}$, for a total circuit evaluation delay that scales as $N^{1.5}$, since we must execute $N$ of these cycles to evaluate the graph. Energy is similarly

$$E_{\text{proc}} = N\left(\left(2^4 + 4\log_2(N)\right)E_{\text{smem}}\left(2^4 + 4\log_2(N), N\right) + 5E_{\text{rmem}}(1, N)\right). \quad (9)$$

We must read each of the four inputs to the 4-LUT and write the result back to memory, requiring five RAMs. Equation (9) means the energy-required scales as $(N\log(N))^{1.5}$, driven by the energy required to read instructions from the instruction memory.

*4) Spatially Reconfigurable Crossbar:* The crossbar-based, spatially reconfigurable architecture is dominated by the crossbar. The area will be $A_{\text{xbar}}(N, 4N)$, the energy $E_{\text{xbar}}(N, 4N)$. This means crossbar area and energy scale are proportional to $N^2$. The per-gate-evaluation delay scales as $5N\sqrt{A_{\text{xpoint}}}$, meaning a depth $d$ circuit has a total circuit evaluation delay $5dN\sqrt{A_{\text{xpoint}}}$.

### D. Discussion

Both designs need $N(2^4 + 4\log_2(N))$ instruction bits. However, the processor adds only a single 4-LUT to the

memory to store instruction and data bits, while the crossbar uses $N$ 4-LUTs and an $N^2$ area crossbar. The crossbar-based spatially reconfigurable design area $(N^2)$ and energy $(N^2)$ scale poorly compared to the processor $(N\log(N)$ area and $(N\log(N))^{1.5}$ energy). When the critical path length $d$ scales slower than $\sqrt{N}$, the spatially reconfigurable crossbar circuit evaluation time scales better $(dN)$ than the processor $(N^{1.5})$; many functions have log-depth circuits,[2] and circuits are typically pipelined such that $d$ is a constant, so the net benefit for the area and energy cost is a reduction in delay.

### E. Ideals

These metrics are generally far from the ideal lower bounds that we might expect. A spatial design, even a custom one, will require some area for each gate, so we cannot expect designs to take an area that is any less than linear in the number of gates; for some design characteristics, we will see that even custom implementations cannot achieve this density (Section IX). Similarly, each gate evaluation will take unit energy, so evaluation energy will, at best, scale linearly with the number of gates. Furthermore, in the best case, we would perform one operation in each time unit on each unit area gate, for a computational density that is constant as $N$ scales.

## IV. LOCALITY

Do they need to come from any of the other gates in the circuit netlist? In both designs, we have considerable freedom to assign gates in the circuit netlist to timeslots (processors) or physical locations (crossbar based, spatially reconfigurable architecture). Can we assign these locations so that producer and consumer gates are close together, reducing the need for long wires?

Consider a subcircuit of $m$ 4-LUTs within a larger circuit of size $N$ $(N > m)$. If the $m$ 4-LUTs are chosen at random, in the worst case, **all** inputs and outputs connect outside the subcircuit, meaning we need $5m$ external connections. However, assuming $m$ is large compared to the input/output (I/O) for the entire enclosing circuit, we can always improve on that grouping. If we instead started with any random $m/5$ gates, we could grab, at most, $4m/5$ predecessors to the $m/5$ gates and add them to make a group of size no greater than $m$ (Fig. 4). If it does not make a group of size $m$, we can fill in the remaining cluster with any gates. This group of $m$ gates would have, at most, $5 \times (4m/5) + m/5 = 4.2m$ external connections. We could likely do even better because: a) some gates will have the same predecessors as others and b) some gates will have all of their successors within the group. This raises the general question: how should we expect the number of external connections $(IO)$ to relate to the number of gates in a subcircuit $(N)$ when we are trying to organize subcircuits to minimize $IO$?

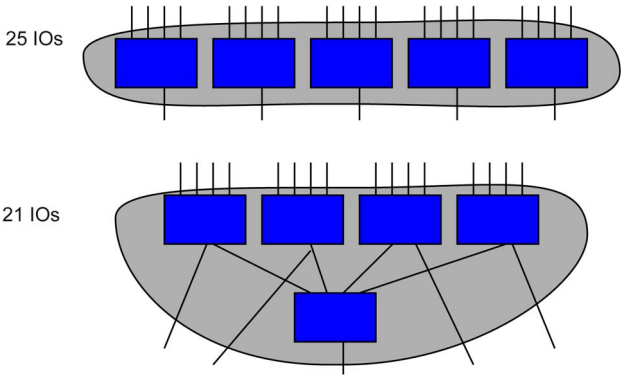


**Fig. 4.** *Subcircuit grouping to reduce external connections.*

In 1960, E. F. Rent at IBM characterized this relationship for the IBM 1401 [12]. Based on the data he collected, he proposed an empirical model that captured the relationship

$$\mathrm{IO} = cN^p \tag{10}$$

where $c$ and $p$ were tuning parameters. Later, Landman and Russo used partitioning algorithms to verify that the Rent model worked well for larger circuits [13]. Bhatt and Leighton use a similar relationship to characterize properties of VLSI layouts of circuits [14]. The constant $c$ can roughly be viewed as characterizing the complexity of the nodes in the graph (e.g., if the nodes are 2-input, 1-output gates, the $c$ might be around 3, whereas if they are $8 \times 8$ multipliers producing a 16b result, $c$ might be around 32). The Rent Exponent, $p$, can be seen as a measure of locality. Designs with essentially no locality have the maximum $p$ of 1.0, as illustrated by our case before where all four inputs and the output of every gate must make an external connection. Designs with a smaller fraction of wires entering and exiting a region, smaller IO, are characterized by a smaller $p$. Very local designs, like an $N$-stage shift register, will have $p = 0$. A typical memory array, with word lines that span the width of the array and bit-lines that span the height, has $p = 0.5$, as shown in Fig. 3.[3] A fast Fourier transform (FFT) has $p = 1$ [15]. Landman and Russo [13], as well as later works, noted that typical designs have $0.5 \leq p \leq 0.7$, with cost-sensitive designs typically closer to $p = 0.5$, and performance-oriented designs closer to $p = 0.7$. Rent locality directly implies the wire-length distributions in a design [16]. If the circuit has a Rent Exponent of less than 0.5, the average wire length is a

[2]The complexity class *NC* characterizes computations that have a polylogarithmic depth [10]. It is an open question of whether all computations can have low depth, but there are circuits known to be P-complete [11].

[3]When $W$ is a small constant, the memory can be implemented as a tree such that it will have $p = 0$.

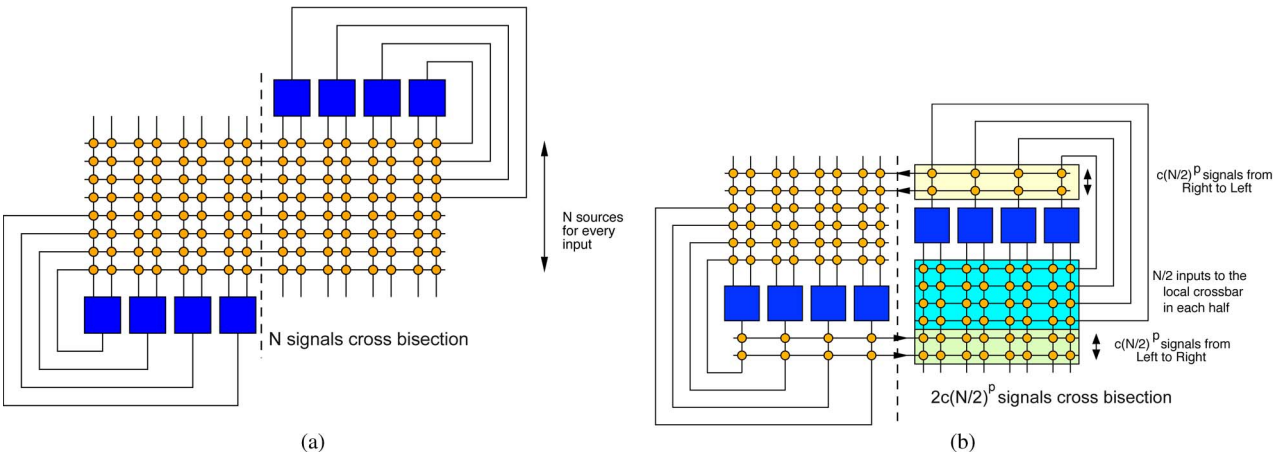


**Fig. 5.** *Reduced bisection wiring from Rent's Rule. (a) Reference crossbar. (b) Reduced interconnect between halves.*

constant independent of the number of gates $N$. For the 2-D circuit layout, when the circuit has a Rent Exponent greater than 0.5, the average wire length scales as $N^{p-0.5}$.

When there is locality ($p < 1$), the crossbar interconnect extreme assumed in Section III-B is excessive. In particular, if we consider breaking the $N$ 4-LUTs into two groups of $N/2$ 4-LUTs, we do not need to send all $N/2$ gate outputs from each half into the other half. We should need only $c(N/2)^p$ wires from each half (see Fig. 5). For small $p$ and large $N$, this can be significantly fewer than $N/2$. For example, if $N = 2 \times 10^6$, $c = 5$, and $p = 0.5$, $N/2$ is $10^6$, whereas $c(N/2)^p$ is 5000. This reduction can be used recursively on each of the halves, and we will explore more efficient configurable networks based on this observation in subsequent sections (Sections VIII, X, and XI).

*Roadmap:* Now that we have seen the fencepost extremes of sequential processors and spatially reconfigurable architectures, we will start with the simple sequential processor and incrementally expand and refine our model, optimizing the processor and working our way back to FPGAs (Section VIII) and a broader set of reconfigurable architectures (Section XI).

## V. DESCRIPTION LOCALITY (PROCESSORS)

This Rent locality can allow us to reduce the size of the instruction memory in the stored-program processor case. The instruction memory stores the circuit netlist. By exploiting Rent locality, we can represent the netlist connectivity more compactly, thereby reducing instruction memory. For designs with locality where $p < 1$, we can partition the circuit into two halves with a small number of edges between the halves as just established. In particular, each half has only $cN^p$ external connections. These external connections in this top partition will need all $\log_2(N)$ bits to describe their source. However, the other connections are contained in smaller subtrees and can use fewer bits. We can recursively subdivide each of the halves to more fully identify and exploit locality (See Fig. 6.) The number of bits required to specify an edge is now proportional to the logarithm of the capacity of the smallest subtree that contains the edge rather than $\log(N)$. We can count the number of bits required by charging each edge for each subtree it must exit, that is, when a graph edge needs to cross out of the top of a tree at level $i$, we need one bit to specify which way the edge connects at that tree level. Thus, the number of instruction bits we need to specify communication $I_{\text{ibits}}$ is

$$I_{\text{ibits}} = \sum_{i=0}^{\log_2(N)} \left( \frac{N}{2^i} \times c(2^i)^p \right) = cN \sum_{i=0}^{\log_2(N)} \left( (2^i)^{(p-1)} \right).$$

The first term $N/2^i$ is the number of subtrees at height $i$ from the leaf, while the second term is Rent's Rule (10) applied to the size of the subtree ($2^i$). For $p = 1$, the term being summed is one, so $I_{\text{ibits}}$ becomes proportional to $N\log(N)$ as we saw in Section III-A when we did not assume any locality. However, when $p < 1$, the exponent $p - 1$ is less than one, making the term being summed a

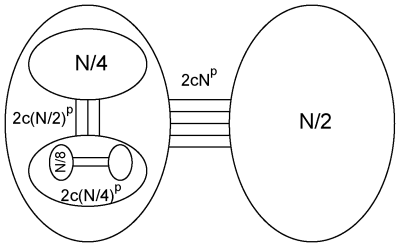


**Fig. 6.** *Recursive bisection wiring by Rent's Rule.*

fraction that decreases geometrically with $i$. As a result, the sum converges to a constant, and we have $I_{\text{ibits}}$ linear in $N$.

*Impact on the Processor Case:* If we exploit this locality when $p < 1$, the instruction memory in the processor (Section III-A) only needs to be linear in $N$. This reduces the area for the processor to linear in $N$, saving a logarithmic factor in $N$. Furthermore, the total bits read from the instruction memory will be proportional to $N$ instead of $N\log(N)$. Here, we assume that we must also specify the destination address in data memory, so we must specify five addresses per instruction. Concretely, the total number of instruction bits becomes:

$$I_{\text{bits}}(N,p) = \left(\frac{5}{1-2^{p-1}} + 2^4\right)N. \tag{11}$$

The first term captures the bits to describe communication, while the second term captures the bits to describe the computation in the 4-LUT. This reduces the area required for the processor in this description locality case $A_{\text{pdesc}}$ to

$$A_{\text{pdesc}} = A_{\text{smem}}(1, I_{\text{bits}}(N,p)) + A_{\text{rmem}}(1,N). \tag{12}$$

Energy becomes

$$E_{\text{pdesc}} = 5NE_{\text{rmem}}(1,N) + I_{\text{bits}}(N,p)E_{\text{smem}}(1, I_{\text{bits}}(N,p)). \tag{13}$$

The total instruction memory energy reduces from proportional to $(N\log(N))^{1.5}$ to $N^{1.5}$. The data memory energy that is proportional to $\sqrt{N}\log(N)$ per gate means a total energy across all $N$ gates that is proportional to $N^{1.5}\log(N)$, which now dominates instruction energy and determines how the energy of operation scales with $N$.

## VI. INSTRUCTION SHARING (WIDE-WORD PROCESSORS)

Equation (11) shows us that the instruction memory is larger than the data memory by a significant constant factor. For $p = 0.7$, $I_{\text{bits}}/N \approx 43$, making the instruction memory almost 43× larger than the data memory. This is driven by the need to store unique instructions for every gate. However, for many regular circuit operations, the instructions can be the same for different data values and can be potentially reused, allowing us to reduce the size of the instruction memory and, consequently, its energy costs.

```
for (j=0;i<IMAGE_HEIGHT;j++)
  for (i=0;j<IMAGE_WIDTH;i++) {
    out[i][j]=0;
    for (wj=WIN_MIN_Y;
         wj<WIN_MAX_Y; wj++)
      for (wi=WIN_MIN_X;
           wi<WIN_MAX_X; wi++)
        out[i][j]+=window[wi+WIN_XOFF]
                         [wj+WIN_YOFF]
                   *in[i+wi][j+wj];
        // boundary condition omitted
        // for simplicity and brevity
    }
```

**Fig. 7.** *Example loop that shares instructions across data elements.*

### A. Looping

A common form of this instruction sharing is looping, where a set of instructions, the loop body, is reused across a large set of data. Low-level image processing or cellular automata are some of the most familiar examples of this kind of looping, applying the same set of operations to each neighborhood region of data (e.g., Fig. 7). For a general formulation, we introduce a separate architecture parameter $I$ for the total number of unique instructions required and allow that to be independent of $N$.

While we use looping to concretely illustrate the opportunity to compress the instruction description, other techniques, such as procedural abstraction, also reduce instruction bits. At the high level used in this analysis, the impact is the same—we can use a smaller number of instructions $I$ that is potentially independent of $N$. Our primary use of $I$ here is to show the magnitude of impact that this sharing can possibly have on implementations. To first order, $I$ can also be viewed as modeling the impact of a first-level instruction cache for a processor, assuming the instruction trace is sufficiently localized that all references are effectively satisfied in this cache.

### B. Word Width

We can also reduce the number of instructions by sharing them across a wide word $W$. One defining property of processors is that they do not operate on single-bit data elements, but rather operate on a set of bits (e.g., 8, 16, 32, 64) grouped into words. This allows them to read many bits from memory, both reducing the number of addresses that must be specified and amortizing out the cost of specifying the address. Assuming the data bits stay fixed, this also reduces the number of instructions, since an instruction is now applied to a group of $W$ bits. The word operations typically apply the same operation to each bit (e.g., bitwise-AND, bitwise-XOR), and even arithmetic operations, such as ADD and SUBTRACT, effectively tell each bit of the datapath to act like an adder bitslice. (See Fig. 8.) This is known as a single-instruction, multiple-data (SIMD)

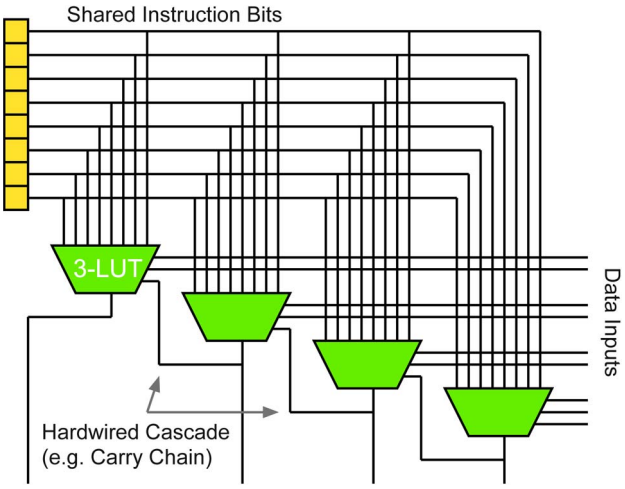


**Fig. 8.** *W = 4 SIMD word control for the 3-LUT programmable gate.*

operation [17] since a single instruction is reused to control the $W$ data bits in the datapath.

### C. Processor Impact

We can reformulate the energy, delay, and area for the processor in terms of the SIMD width $W$ and the total instructions $I$

$$A_p = A_{\text{smem}}(1, I_{\text{bits}}(I,p)) + A_{\text{rmem}}\left(W, \frac{N}{W}\right) \quad (14)$$

$$E_p = 5\left(\frac{N}{W}\right) E_{\text{rmem}}\left(W, \frac{N}{W}\right) + \left(\frac{I_{\text{bits}}(N,p)}{W}\right) \times E_{\text{smem}}(1, I_{\text{bits}}(I,p)) \quad (15)$$

$$D_p = 4\left(\frac{N}{W}\right)\left(\sqrt{A_p} + W\sqrt{A_{4\text{LUT}}}\right). \quad (16)$$

$A_{4\text{LUT}}$ is the area of the 4-LUT without any configuration memory.

Fig. 9(a) compares the energy ratios at $p = 0.7$ to show the impact of limited instructions and SIMD word width. As the "data energy only" curves show, there is a clear crossover point where data energy begins to dominate instruction memory energy. Fig. 9(b) shows how the area scales for the $I = N$, $W = 1$ case and for an instruction-sharing case with $I \leq 128$ and $W = 64$. With no sharing, the area per gate converges to the area for the instruction bits and the data bit $(I_{\text{bits}}(N,p)/N + 1)$. With high sharing, the area per gate converges to the area of the data bit. In today's 22-nm process, if we assume 6-transistor SRAM memory cells, we can fit more than a billion $(2^{30})$ memory cells into a square centimeter of silicon, and a 6-nm process will hold 10 billion. Since a logic gate is larger than a memory bit, this is even smaller than a custom design that directly implements the circuit with a physical gate for every gate in the netlist.

Fig. 9(c) shows how increasing $W$ decreases area, delay, and energy. Except for delay, the benefit saturates when data area and energy dominate the instruction area and energy as noted before. This illustrates that instruction sharing is a net constant effect on area and energy, so it does not change the scaling relations with respect to $N$—the area remains proportional to $N$ and energy proportional to $N^{1.5}\log(N)$.

### D. Takeaway

At the extreme, stored-program processors are about compactness, fitting the computation into the minimum area possible. With these optimizations, for typical computations, data-storage area and energy dominate everything else, including instruction storage and computation. The density of "gates" approaches the density of memory bits [Fig. 9(b)]. However, sequentialization means the computational density—the computation performed per-unit area and time—scales as $N^{-1.5}$. While the word-width operations help the absolute energy costs, the energy per gate evaluation still grows as $\log(N)\sqrt{N}$ due to reads from the data memory—*larger computations become less efficient per gate.*

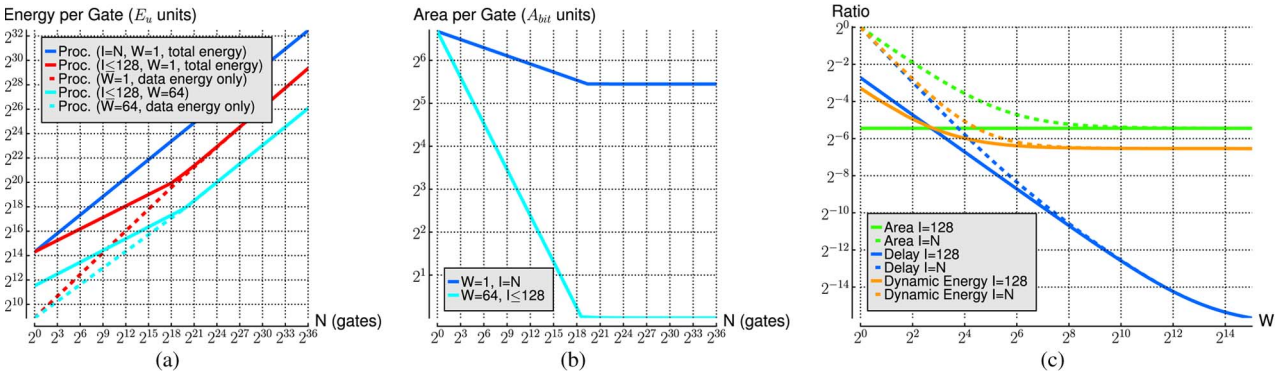


**Fig. 9.** *Processor characteristics at p = 0.7. (a) Energy scaling (15); (b) area scaling (14); (c) energy, delay, and area scaling for N = $2^{30}$ (14)–(16); all ratios to I = N, W = 1 case.*

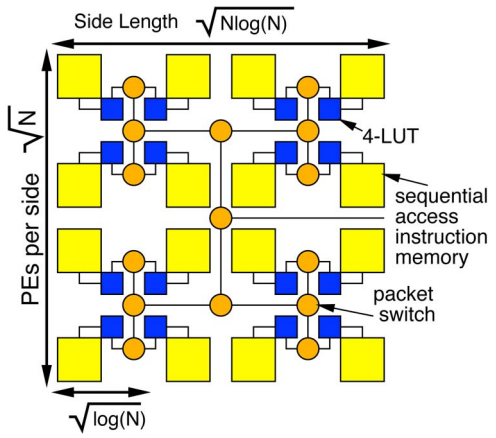


**Fig. 10.** *Sequential communication exploiting data locality.*

## VII. DATA LOCALITY

After exploiting description locality and instruction sharing, the dominant energy component comes from data-memory access. Since we bring the data to a single location to evaluate the gate, we must pay energy proportional to $\sqrt{N}$ for every data fetch from the memory. That is, we are **not** exploiting any locality in the movement of the data.

To create and exploit locality in data movement, we can perform the gate evaluation at different places and arrange the data for minimal movement. We layout the gates in a square grid and use an H-Tree to interconnect them together (Fig. 10).[4] Since the average wire length in a 2-D VLSI circuit is constant when $p < 0.5$ and proportional to $N^{p-0.5}$ when $p > 0.5$ (Section IV), this will reduce the energy requirements per node when $p < 1$ and will come very close to achieving energy linear in the number of gates when $p < 0.5$.

### A. Analysis

To assign gates to processing elements (PEs) at the leaves of the H-Tree (Fig. 10), we perform the recursive bisection of the circuit graph to minimize cut sizes for Rent's Rule (Fig. 6). Assuming the fanout associated with each gate is limited to a constant $k$, the node will need instruction storage space only for its gate operation and the node location of its $k$ successors ($\log_2(N)$ bits for each successor).[5] The bits specifying the location of each successor are used as a packet header to route the output bit through the network to the successor PE. For concreteness, we will assume $k = 4$ to be symmetric with the gate inputs. The PE will only need data-storage space for the four inputs to the 4-LUT. Each PE will have an area proportional to $\log(N)$ driven by the $\log(N)$ instruction bits to specify successors, making the entire structure area grow as $N\log(N)$. All links in the H-tree have a single-bit-wide uplink and downlink, so that the H-tree adds area only linear in the number of leaves supported. The internal nodes in the tree serve as a bit-serial, packet-switched network. We can compute the area of a data local PE, $A_{dlpe}$. Combined with the area of a bit-serial, packet-switched routing node in the H-tree, $A_{sw}$, we can compute the area of this composite data local design $A_{\text{data}}$ as

$$\begin{aligned} A_{dlpe}(N) &= A_{4LUT} + 4A_{\text{bit}} + A_{\text{smem}}(1, k\log_2(N)) \\ A_{\text{data}}(N) &= N \cdot A_{dlpe}(N) + N \cdot A_{sw}. \end{aligned} \tag{17}$$

Evaluation energy at the leaf nodes remains constant per gate, meaning total energy that is linear in the number of gates. We can sequentially access the successor gate address memory. While there are $k\log_2(N)$ bits in each PE to handle the worst-case successor link, locality means that most successors can be described with fewer bits. Consequently, we introduce a separate variable $N_{succbits_i}$ to denote the number of successor bits we actually need to read for PE $i$. As a result, the instruction read energy for PE $i$ ($E_{idpe_i}$) is

$$E_{idpe_i} = N_{succbits_i} \cdot E_{\text{smem}}(1, k\log_2(N)).$$

The bits configuring the 4-LUT never change, so there is no energy required to read them. The total number of bits we need to read across all $N$ nodes is the same as in the description locality case

$$\sum_{i=0}^{N} N_{succbits_i} = \left(\frac{5}{1-2^{p-1}}\right)N. \tag{18}$$

From Rent's Rule, this tells us how many total instruction bits must be read without directly identifying the number of bits that must be read in each PE $N_{succbits_i}$. This brings the total energy for reading instruction memory to

$$E_{\text{idata}} = \left(\frac{5}{1-2^{p-1}}\right)N \cdot E_{\text{smem}}(1, k\log_2(N)). \tag{19}$$

This leaves the energy required to route the data to the successors over the H-tree network. Here, we must account for the number of edges that must be routed to each height in the tree, the bits that specify the destination, and the lengths of the wires at each tree height.

- There are $N/2^i$ subtrees at height $i$ from the leaf.
- By Rent's Rule, we know we have $c(2^i)^p$ edges that must cross out of each of those subtrees. We will

[4]The scaling results derived in this section will not change if we used a constant width mesh instead of the H-Tree.

[5]Any netlist can be transformed into one with bounded fanout with only a constant factor change in the critical path and gate count [18].

take $c = 8$ to capture the left-to-right and right-to-left traffic at the root.
- The packet will need an address specification as well as the data bit. The number of bits in an address will be no greater than $\log(N)$; for simplicity, we make no further attempt to account for the fact that many stages see fewer bits.
- The length of the top wire in the tree is $\sqrt{A_{\text{data}}}$.
- Wire lengths halve every other stage.

Putting this together, we obtain

$$E_{\text{cdata}} \leq \sum_{i=0}^{\log_2(N)} \left( \frac{N}{2^i} 8(2^i)^p (\log_2(N)+1) 2^{\lceil \frac{i}{2} \rceil} \sqrt{A_{data}/N} \right) \quad (20)$$

$$\leq 8\sqrt{N \cdot A_{\text{data}}}(\log_2(N)+1) \sum_{i=0}^{\left\lceil \frac{\log_2(N)}{2} \right\rceil} \left( (2^i)^{(2p-1)} \right)$$

$$E_{\text{data}} = E_{idata} + E_{\text{cdata}}. \quad (21)$$

For $p > 0.5$, we have

$$E_{\text{cdata}}(p > 0.5) \propto N^{p+0.5} \log^{1.5}(N). \quad (22)$$

At $p = 1$, this energy is $\sqrt{\log(N)}$ larger than the memory case in Section V due to the area increase to hold $N \log(N)$ bits for the description; however, for any $p < 1$, the benefit from locality of data movement is greater, and this scheme has lower energy growth since $N^{p+0.5} < N^{1.5}$. *Picking spatial locations for computations and moving the data minimally to these locations saves energy*. For $p < 0.5$, we have

$$E_{\text{cdata}}(p < 0.5) \propto N \log_2^{1.5}(N). \quad (23)$$

With this high locality, energy is within a $\log^{1.5}(N)$ factor of linear in $N$. If we define $p' = \max(0.5, p)$, we can state both results as

$$E_{\text{cdata}} \propto N^{p'+0.5} \log_2^{1.5}(N). \quad (24)$$

In any case, $E_{\text{cdata}}$ grows faster than $E_{\text{idata}}$, defining the growth rate for $E_{\text{data}}$. Fig. 11 ($N_{\text{PE}} = N$) shows that this always results in lower energy than the processor when there is no instruction sharing ($W = 1, I = N$). This is true for any $p < 1$, with the level of benefit dependent on $p$ (See Fig. 12.) When high instruction sharing is possible in the processor, this scheme, as described so far, will not be a net energy reduction until the circuit size becomes very large (e.g., above $2^{36}$ in Fig. 11).

With a separate 4-LUT for each gate, the circuit evaluation will not be serialized on computation. However, the critical path could be needed to cross the entire chip taking delay proportional to $\sqrt{A_{\text{data}}}$ for each crossing, or in the worst case, a total delay proportional to $d\sqrt{A_{\text{data}}}$. Furthermore, communication between the two halves of the chip must be serialized. A total of $4N^p(\log_2(N)+1)$ bits must cross the bisection in each direction. For $p > 0.5$, this will typically be the rate limiter for the design.

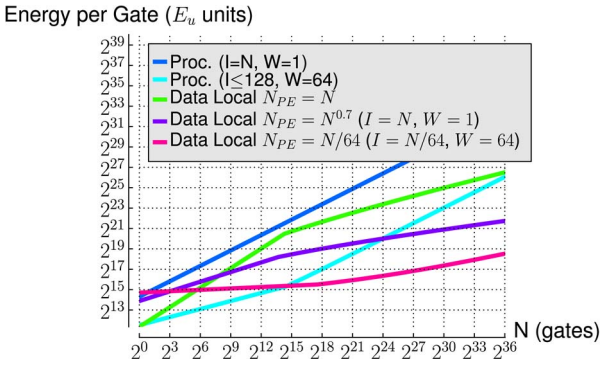


**Fig. 11.** ***Data local energy p = 0.7 (15), (21), (39), (Appendix B).***

## B. Energy-Delay-Area Tradeoff (Packet-Switched, Parallel Processors)

With each gate getting its own PE, the area for this design tends to $N(A_{dlpe} + A_{sw})$. Since the area for the processor design tends to $N \cdot A_{\text{bit}}$ with high instruction sharing (Section VI), this data local design will be larger by a factor of $(A_{dlpe} + A_{sw})/A_{\text{bit}}$. This ratio is technology dependent, but might be 300–400 when $N$ is $2^{24}$ for conventional VLSI. It is the larger size, which translates into longer wires, that makes the pure data local case higher energy than the instruction-sharing cases for smaller circuit graphs (Fig. 11). Between these two extremes, we could use $1 < N_{\text{PE}} < N$ by serializing a group of $N/N_{\text{PE}}$ gates at each leaf PE. We can exploit instruction sharing (Section VI) within the PE and for communication between PEs, so we parameterize the modeling based on $I$ and $W$ (See Apendix B).

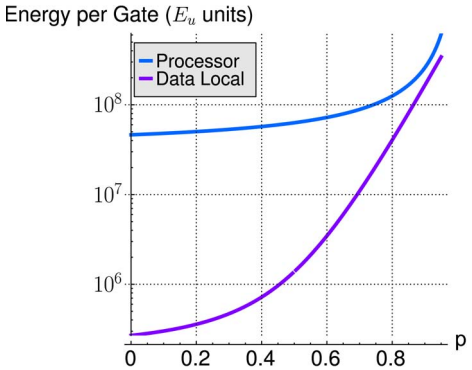


**Fig. 12.** ***Data local energy for $N_{PE} = N = 2^{24}$, $W = 1, I = N$ (15), (21).***

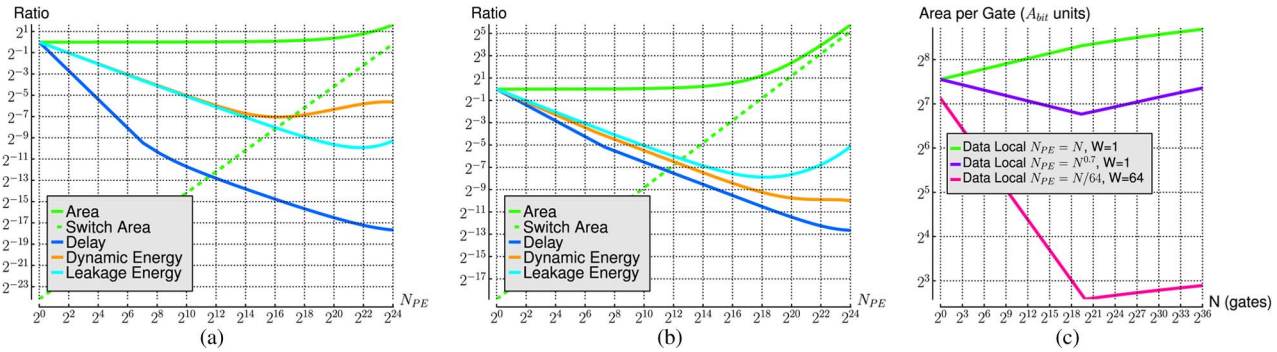


**Fig. 13.** *Data local scaling for $p = 0.7$. (a) Energy, delay, and area for $W = 1$, $I = N$ for $N = 2^{24}$ as a function of $N_{PE}$ (21), (40), (17); (b) energy, delay, and area for $W = 64$, $I \leq 128$ for $N = 2^{24}$ as a function of $N_{PE}$ (38)–(40); (c) area scaling (38).*

Fig. 13(a) and (b) shows how area, delay, and energy change as we vary the number of processing elements. We estimate leakage as the product of area and delay to capture gross scaling; the weighting between dynamic and leakage energy will be process and operating point dependent, so we make no attempt to combine the two.

The dominant components of area in these designs are the instruction memory [$I = N$ case in Fig. 13(a)], data memory [$I \leq 128$ case in Fig. 13(b)], and switches. As we increase the $N_{\mathrm{PE}}$, the instruction and data memory area do not change, but the switch area increases as shown. As a result, at small $N_{\mathrm{PE}}$, where instruction and data memory area still dominate, the switch area, the total area remains flat. Only when the switch area rises to become comparable to memory area does the total area show a noticeable change. This occurs at a lower $N_{\mathrm{PE}}$ in the high-sharing case [Fig. 13(b)] since the instruction memory and, hence, total memory is significantly reduced as previously noted.

As $N_{\mathrm{PE}}$ rises from 1, the instruction and data memories shrink, reducing memory energy. However, since the total area is increasing, the distances where data must be moved in nonlocal cases are increasing. These two competing effects combine to provide an energy-minimizing point that is not at the fully spatial extreme. This means that we can save even more energy with these intermediate points, and do so with an area closer to the processor case. The $N_{\mathrm{PE}} = N^{0.7}$ ($W=1$) and $N_{\mathrm{PE}}=N/64$ ($W=64$) lines in Fig. 11 show how energy scales for the energy-minimizing $N_{\mathrm{PE}}$ selections. The $W=1$ case is always less than the processor, and the $W = 64$ case is less for any size over 32 K gates.

The resulting architecture here is a large set of small PEs that exploit SIMD parallelism. At a gross level, this might look like a general-purpose graphical processing unit (GPGPU). There is a significant difference. Today's GPGPUs do not have a network among the PEs. They store all of their data in a central memory, usually an off-chip DRAM. As a result, they do not exploit communication locality, which is essential in reducing energy. If all data must be moved to a central memory, energy scales as $N^{1.5}$ rather than the smaller $N^{p'+0.5}$ identified here when the design exploits communication locality.

### C. Takeaway

The dominant energy in the processor is data movement. Exploiting data locality reduces energy over the stored-program processor case. It is possible to reduce energy to within a logarithmic factor of linear in $N$ for $p < 0.5$. For larger $p < 1$, energy scales roughly as $N^{p+0.5}(\log(N))^{1.5}$. This comes at the expense of a design that is larger by a constant factor—a concrete example where we can tradeoff area with energy; specifically, we can gain the energy benefits by trading away as little as a factor of four in area density. [See Fig. 13(c).] The computational density improves to $\left(d\sqrt{N\log(N)}\right)^{-1}$ when $p < 0.5$ and $N^{-p}\log^{-1}(N)$ for $p > 0.5$. The bit-serial link between the two halves of the tree is a bottleneck that prevents us from reducing the delay below $4N^p$ for designs with $p > 0.5$.

## VIII. INSTRUCTION LOCALITY (FPGAs)

While the data locality case reduced energy and delay, it still suffered from the need to spend significant area-storing instructions and significant energy-sending headers on the packets. It also had a limited delay benefit due to the serial bottleneck. Instead, we could build a richer network between PEs. In the extreme, we build a tree or mesh that has one wire per signal we want to send so that communications do not need to be sequentialized. This allows us to place the configuration for the interconnect local to the switches that must be configured, and it means the configurations do not need to change during circuit evaluation, thus eliminating instruction energy completely. This resulting architecture is that of an FPGA, with dedicated compute elements, 4-LUTs, for each logical gate in the netlist and dedicated wire links supporting the edges. This is the same instruction configuration as the crossbar-based, spatially reconfigurable architecture (Section III-C4), except that we substitute a more efficient, locality-exploiting

interconnection network for the crossbar. For highly local designs ($p < 0.5$), this has linear area for interconnect, compared to the quadratic area in the crossbar. Compared to the data local designs in the previous section, for highly local designs ($p < 0.5$), we will remove the logarithmic terms, and achieve designs that are fully linear in area and energy, while achieving lower delay. For designs with less locality ($0.5 < p < 1.0$), the energy and area are larger than the data locality case, but the delay is lower.

*Butterfly Fat-Tree Interconnect:* We specifically consider using a butterfly fat-tree (BFT) [19] (or HSRA [20]) for the interconnection network to simplify analysis. (See Fig. 14.) Unlike the H-tree used for the data local case (Fig. 10), fat tree interconnect grows toward the root of the tree, similar to the branches on a real tree. The BFT is designed so that the interconnect at the root of each subtree follows Rent's Rule, growing as $cN^p$, where $N$ is the number of leaves in the subtree. The BFT variant of the fat tree uses switches of constant size [Fig. 14(b)]. At the root of each subtree, each wire is connected to a single switch, such that the number of switches is linear in the subtree IO. The total number of switches needed by the BFT is linear in the number of nodes supported. (See (26).), so linear in circuit size $N$ for any $p < 1$. Note that this is in contrast to the crossbar that required switches proportional to $N^2$.

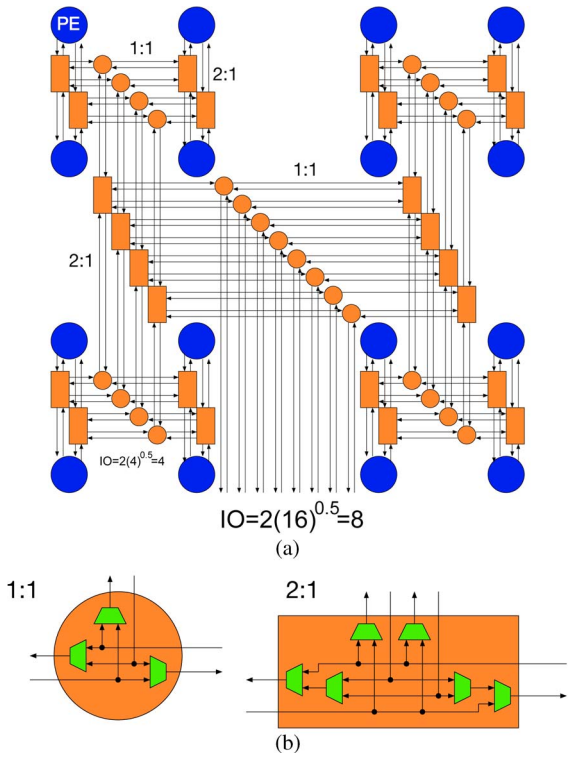


**Fig. 14.** *Butterfly fat tree (BFT). (a) $N = 16$, $c = 2$, $p = 0.5$; (b) switch composition.*

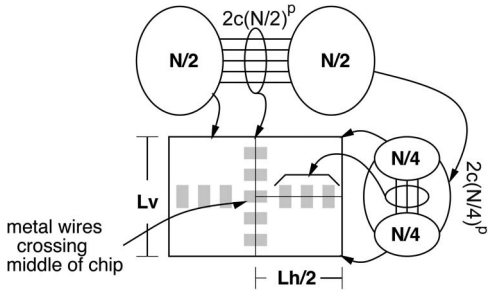


**Fig. 15.** *Wire-driven lower bound on the VLSI area.*

While the switch and leaf resources are linear in $N$, the BFT has more wiring than the data local case. It is necessary to account for the wiring in order to properly characterize the area, delay, and energy of this architecture.

## IX. VLSI WIRING COMPLEXITY

When we have a large number of wires, as we do in a crossbar or in the upper channels of the BFT, we must account for the fact that wires require area. Particularly, if we have a limited number of wiring layers, as we typically do for any particular VLSI fabrication process, large numbers of wires can determine the area of the implementation.

As previously noted, wires will have a minimum width and minimum spacing to adjacent wires determined by the process, defining a minimum pitch $FP$. Both the wire width and spacing are related to the feature size in the process, so the pitch is as well. In fact, most processes are named by their half pitch, or roughly, the width of a minimum size transistor. A collection of $B$ wires placed on a single metal layer requires a width of at least $B \cdot FP$. If we can distribute them across $L$ wire layers, this can be reduced to $(B \cdot FP)/L$. A typical digital VLSI discipline will allocate half of the interconnect wiring metal layers to horizontal connections and half to vertical, making the available wire layers for a bus in a single direction half the routing wiring layers.

Accounting for the wire area is important because wiring, not gates nor switches, can end up determining the area in designs. For example, for any design characterized by a Rent Exponent $p > 0.5$, the lower bound on the chip area is higher than linear in the gate count. Specifically, we know the top cut will require $cN^p$ wires. If we are limited to a fixed number of metal layers $L$, this means the wires that need to cross between the two halves of the chip must take up width $(cN^p \cdot FP)/L/2$. By a similar argument, when we look at the wires for the next tree level, it requires width $2((c(N/2)^p \cdot FP)/L/2)$ in the other dimension. (See Fig. 15.) Together, this means the area for the design must grow at least in proportion to $N^{2p}$, which is greater than linear when $p > 0.5$. This result can be seen as a consequence of the fact that the perimeter of a 2-D region grows as the square-root of the area in the region. If we need to

bring more signals in or out of the region than $\sqrt{N}$, the region size will be determined by the perimeter rather than the capacity to hold gates inside the region. For more on VLSI complexity theory, see [14], [15], [21], and [22].

## X. SPATIALLY RECONFIGURABLE COMPUTATION

Building on the VLSI wiring model, we can characterize the area, delay, and energy for the spatially reconfigurable architecture. We compute area as the sum of the active transistor area and the wiring area.

Starting with the active transistor area, the area in the leaf PE is

$$A_{spe} = A_{4lut} + 2^4 A_{\text{bit}} + ((c-4)4))(A_{mux} + A_{\text{bit}}). \quad (25)$$

The $(c-4)4$ term deals with selecting inputs to the PE from the BFT network [23]. From Fig. 14, we see that each directional wire pair at the top of a subtree is associated with a switch. Each connection needs 3 two-input multiplexers [Fig. 14(b)]. Counting based on wiring at each tree level gives the total switch area

$$\begin{aligned} A_{sws} &= \left( c \sum_{i=0}^{\log_2(N)} \left( 2^{ip} \frac{N}{2^i} \right) \right) (3A_{\text{bit}} + 3A_{\text{mux}}) \\ &\approx N \left( \frac{c}{1-2^{p-1}} \right) (3A_{\text{bit}} + 3A_{\text{mux}}). \end{aligned} \quad (26)$$

Here, $2^{ip}$ is the number of wires in a channel at level $i$, and $N/2^i$ is the number of such channels. Putting these together

$$A_{\text{active}} = NA_{spe} + A_{sws}. \quad (27)$$

For wiring, we must not only account for the $cN^p$ wires at the top level as previously noted (Section IX) but also for all of the wire channels in the horizontal and vertical dimension across all tree levels. We first count the number of wire channels needed across the width of the chip by looking across all overlapping wiring channels

$$\begin{aligned} \text{Wires} &= \left( cN^p + 2cN^p \left( \frac{1}{2} \right)^{2p} + 4cN^p \left( \frac{1}{2} \right)^{4p} + \ldots \right) \\ &= cN^p \sum_{l=0}^{\log_2(N)} 2^{(1-2p)l}. \end{aligned} \quad (28)$$

The exponent increases by a factor of two since we are counting every other stage to account for the contribution of wires to a single dimension. The constant in front doubles since we are doubling the number of wire channels that parallel each other at every other tree stage. To turn this into a length, we consider the wire layers $L$

$$L_{\text{wire}} = \frac{\text{Wires} \times FP}{L/2} = \frac{2FP \times \text{Wires}}{L}. \quad (29)$$

The length of the side of the entire design is thus

$$L_{\text{side}} \leq \sqrt{A_{\text{active}}} + L_{\text{wire}}. \quad (30)$$

The total area is the square of the side length

$$A_{\text{spatial}} = (L_{\text{side}})^2. \quad (31)$$

When $p < 0.5$, $L_{\text{wire}}$ is proportional to $\sqrt{N}$, and the entire area is linear in $N$. When $p > 0.5$, $L_{\text{wire}}$ becomes proportional to $N^p$, making the area proportional to $N^{2p}$. Fig. 16(a) shows how area scales with $N$.

The benefit of this additional area is reduced delay. There is no node or interconnect serialization here, only the critical path delay. In the worst case, all links in the critical path must cross the chip, such that delay scales as the product of the side length $L_{\text{side}}$ and the critical path length $d$

$$D_{\text{spatial}} = d \cdot L_{\text{side}}. \quad (32)$$

This means a delay proportional to $d\sqrt{N}$ for $p < 0.5$ and proportional to $dN^p$ for $p > 0.5$, which is better scaling than the crossbar (Section III-C4). For logarithmic critical paths, the $p < 0.5$ delay can be a factor of $N$ lower than the processor case.

We determine the total wire energy switched by summing up the energy of all the wires

$$E_{\text{spatial}} = \sum_{l=0}^{\log_2(N)} cN^p 2^l \left( \frac{1}{2^l} \right)^p \left( \frac{L_{\text{side}}}{2^{\lceil \frac{l}{2} \rceil}} \right) E_u. \quad (33)$$

The sum ranges over all levels $l$ by computing the wire energy per level. At each level, we must consider the total number of wires at the level and their length.

- At the top level ($l = 0$), there is one channel bundle with $cN^p$ (Rent's Rule) wires that cross the entire chip ($L_{\text{side}}$).
- In general, there are $2^l$ channels in each level.

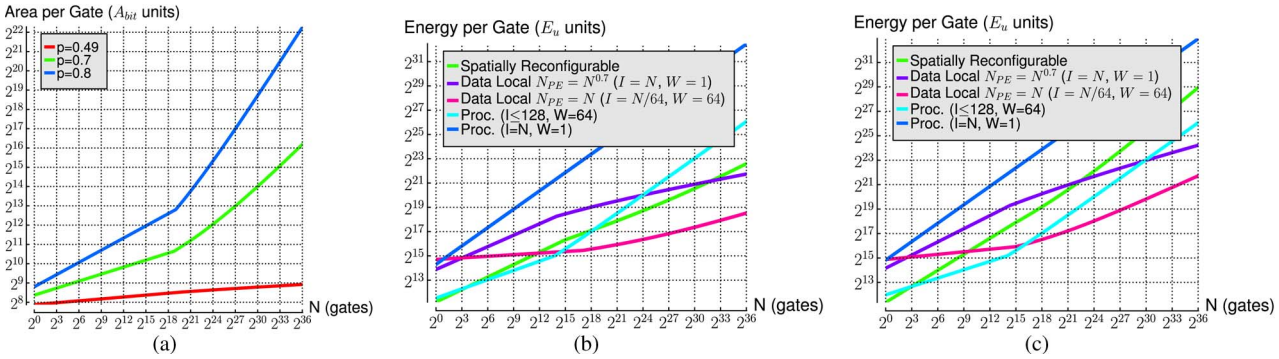


**Fig. 16.** *Spatial area and energy assuming eight metal layers. (a) Area scaling (31), (b) energy compare for $p = 0.7$ (15), (33), (39); (c) energy comparison for $p = 0.8$ (15), (33), (39).*

- By Rent's Rule, the number of wires in each channel at a level is smaller by a factor of $(2^l)^p$.
- Every other channel extends in a different dimension, so the subtree width $(L_{\text{side}}/2^{\lceil l/2 \rceil})$ shrinks to half the previous side length every other level toward the leaf. The ceiling in the exponent $(\lceil l/2 \rceil)$ accounts for this shrink on alternating levels.

When $p < 0.5$, the sum results in a term that is linear in $N$. When $p > 0.5$, the total energy is proportional to $N^{2p}$. This is less than the processor energy of $(N \log(N))^{1.5}$ when $p < 0.75$, but is larger when $p > 0.75$. Fig. 16(b) shows how the energy compares to the processor and data local cases when $p = 0.7$, and Fig. 16(c) shows the comparison with $p = 0.8$. We pick $p = 0.7$ and $p = 0.8$ to illustrate these trends since they are immediately on either side of the $p = 0.75$ breakpoint where the processor and spatial design both scale by roughly $N^{1.5}$. Fig. 21 shows how the energy difference widens as we move further away from the $p = 0.75$ breakpoint. Also note that $p = 0.7$ is the high-end, high-performance Rent Exponent typically observed in circuits.

*Instruction Sharing:* Note that there is no SIMD sharing case for this spatially reconfigurable, instruction locality case. Since instructions never switch, no dynamic energy goes into reading or writing instructions, and so there is no energy to save. The instructions do take up space, but they are small compared to the switches and wiring they control.

For regular designs, it is possible to exploit a form of instruction and datapath sharing by spatially instantiating some number of instances of the common computational graph (e.g., loop body) and reusing those graphs in time with different data. Architecturally, this will demand data memory to hold the data items that share the spatial graph. In modern FPGAs, these data memories show up as embedded RAMs [24]. This implementation raises many additional issues that make direct comparisons across architectures beyond the scope of this paper. See [25] for quantitative experiments illustrating how parallelism tuning with embedded memories can reduce FPGA energy.

*Takeaway:* With a locality-exploiting network, such as the BFT, the spatially configurable design achieves area and energy linear in the number of gates supported when locality is high enough $p < 0.5$, and it avoids the sequential communication bottleneck of the data locality case. *The linear area and energy scaling achieved with this FPGA-style architecture when $p < 0.5$ is optimal to within constant factors.* When $p < 0.5$, this achieves a computational density of $(d\sqrt{N})^{-1}$. This style of storing the instructions local to the resources they control eliminates instruction energy, resulting in lower energy than the processors for any $p < 0.75$. When $p > 0.75$, wiring complexity results in longer interconnection wires and greater energy than the processor case.

While we have developed specific results based on a BFT, similar scaling results can be achieved with a suitably designed mesh. See [26] for details on how to relate tree and mesh results.

## XI. TIME-MULTIPLEXED INTERCONNECT (MULTICONTEXT, CGRA)

While the FPGA-style, spatially reconfigurable computations are achieving optimal area and energy for highly local, $p < 0.5$ tasks, they can be larger and use more energy than the data locality designs (Section VII) and even the processors (Section VI) for large $p$. As we saw, it is the fully spatial wiring area that drives the fully spatial design to consume more area, send data on longer wires, and consequently consume more energy than the data local design with serialized interconnect. In this section, we will see that it is possible to achieve higher bandwidth communication than the data local architecture without demanding substantially more area.

To do so, we consider an architectural variation that combines the best aspects of the spatially reconfigurable design with the data local design. In particular, rather than providing a unique wire for every signal, we provide only as much wiring as we can without exceeding linear switch

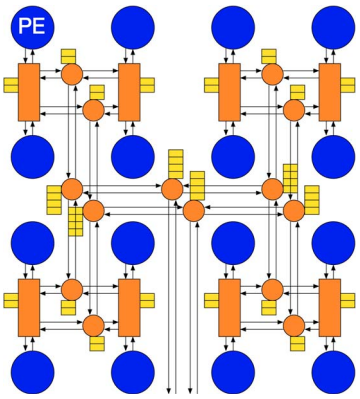


**Fig. 17.** *Time-multiplexed butterfly fat tree (BFT) with $c_{net} = 1$, $p_{net} = 0.25$ supporting $c = 2$ and $p = 0.5$.*

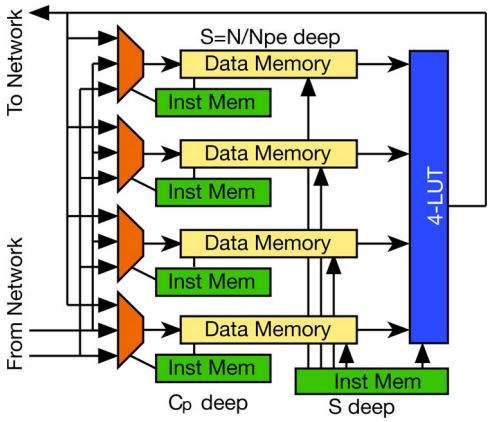


**Fig. 18.** *Multicontext leaf-processing element (PE).*

and wire area. That means we must build the physical network with a limited wiring growth rate, $p_{net} < 0.5$.[6] The root of the tree will still be a bottleneck on communication since it supports $c_{net}N^{p_{net}} < cN^p$. However, this means the design only serializes communication by $(c/c_{net})N^{p-p_{net}}$ rather than completely serializing it by $cN^p$ as in the serial H-Tree data local design.

The data local design requires an $N\log(N)$ area in order to store the successor destinations and spends up to $\log^{1.5}(N)\sqrt{N}$ energy to send instruction bits from their storage at the leaf PEs up to the switches they must control in the interconnect. We can remove these logarithmic terms by placing the instruction bits that control switching local to the network segments they control rather than in the PEs; that is, since we must sequentially reuse the network links in this design, we place small instruction memories in the network along with the switches. (See Fig. 17.) These instruction memories control the behavior of the switches over time. Since $p_{net} < p$, the wires higher in the network will have a larger sharing factor than the wires close to the leaf of the tree. At height $i$ in the tree, the sharing factor is $(c/c_{net})(2^i)^{p-p_{net}}$. This means the instruction memories become deeper as we move toward the root of the tree. Nonetheless, the total instruction memory area remains linear in $N$. The energy from the memories scales as the wire energy, which scales similar to the data local design (20) with the benefit that the total area is linear in $N$ rather than $N\log(N)$. As a result, the basic scaling is

$$A_{tm} \propto N \tag{34}$$

$$p' = \max(0.5, p) \tag{35}$$

$$E_{tm} \propto N^{p'+0.5} \tag{36}$$

$$D_{tm} \propto d\sqrt{N}. \tag{37}$$

[6]Leiserson shows that these $p = 0.5$ fat trees are area-universal in the sense that they are within a polylogarithmic factor of the performance of any network that can be built in the same area [27].

This is the least area, energy, and delay of any of the designs.

As with the data local case, we can also consider clustering multiple gates at the leaf of the tree and serializing them to reduce area. Fig. 18 shows the composition of this leaf PE. $S$ in Fig. 18 captures the level of serialization at the leaf of the network. We can also exploit instruction sharing for this hybrid design. We divide the instruction storage in the PE by $W$ and the instruction storage in the network. This roughly provides a model for coarse-grained reconfigurable arrays (CGRAs) (e.g., [28]–[33]). At word width of one, this is a model for multicontext FPGAs (e.g., [34]–[36]). Appendix C summarizes the modeling for this time-multiplexed design, and many design details are elaborated in [9].

### A. Characteristics

Fig. 19 shows how the energy, delay, and area change as we vary the number of PEs, suggesting a modest sequentialization at the leaves saves energy. As we increase the time multiplexing (decrease $N_{PE}$), the area decreases then flattens out as memory dominates switching. For large time multiplexing, the area increases because we are not exploiting description locality within the PE in this simple formulation. Similarly, as we increase time multiplexing, the serialization increases delay for the $W = 1$ case. However, since the area is decreasing, wire lengths are shorter, causing delay to increase more slowly than the sequentialization factor. In the $W = 64$ case, we actually see the wire length effects dominating the sequentialization effect for modest (up to 16) levels of time-multiplexing. Similarly, energy is reduced by modest (4 or 32 depending on $W$) time-multiplexing. With no time-multiplexing, the wire energy dominates. Time-multiplexing that reduces area, reduces wire lengths, and, hence, wire energy at the cost of adding instruction and data memory energy. As long as the reduction in energy due to wire lengths is larger than the increase due to memories, additional

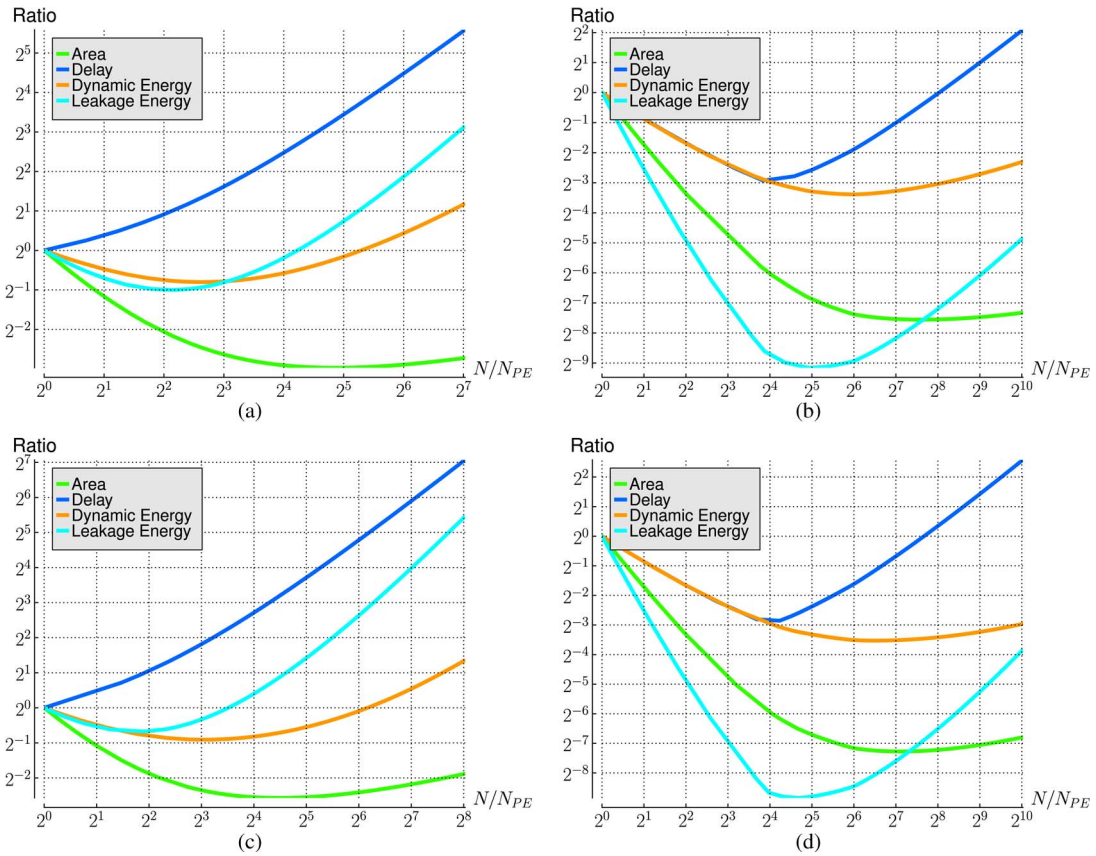


**Fig. 19.** ***Time multiplexed with $p_{net} = 0.49$ at $N = 2^{24}$ (49)–(51), Appendix C. (a) $p = 0.7$, $W = 1$; (b) $p = 0.7$, $W = 64$; (c) $p = 0.8$, $W = 1$; (d) $p = 0.8$, $W = 64$.***

time-multiplexing is a benefit. The memories in the SIMD ($W$ =64) cases grow more slowly than in the $W = 1$ case, so they benefit from greater levels of sequentialization.

Fig. 20 compares energy scaling across the various designs, showing that the time-multiplexed design with $p_{net} = 0.49$ achieves the least energy of all the designs when supporting more than 4096 gates at $p = 0.7$ and $p = 0.8$. The magnitude of the benefit grows with design size. There is an advantage to exploiting SIMD sharing when it is possible, but the benefit is small compared to the advantage gained from the time-multiplexed implementation.

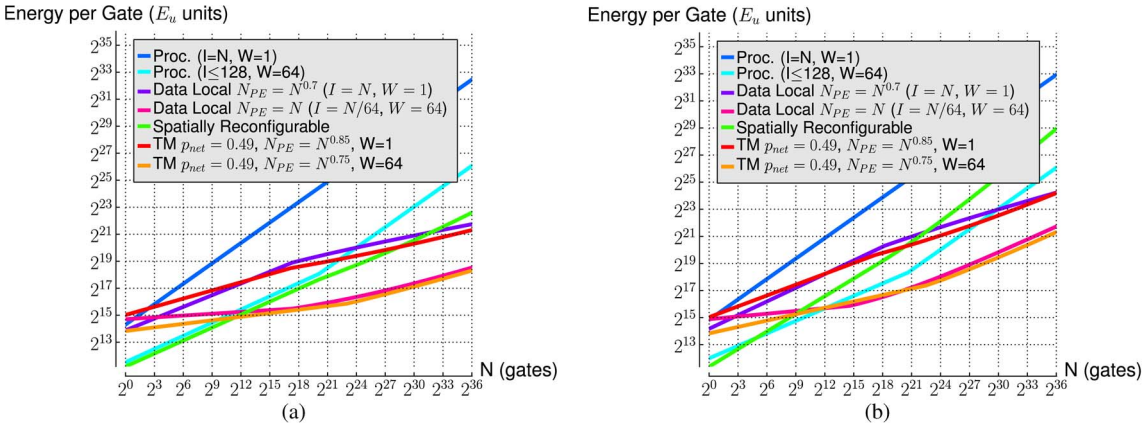


**Fig. 20.** ***Energy scaling comparison (15), (33), (39), (49). (a) $p = 0.7$ and (b) $p = 0.8$.***

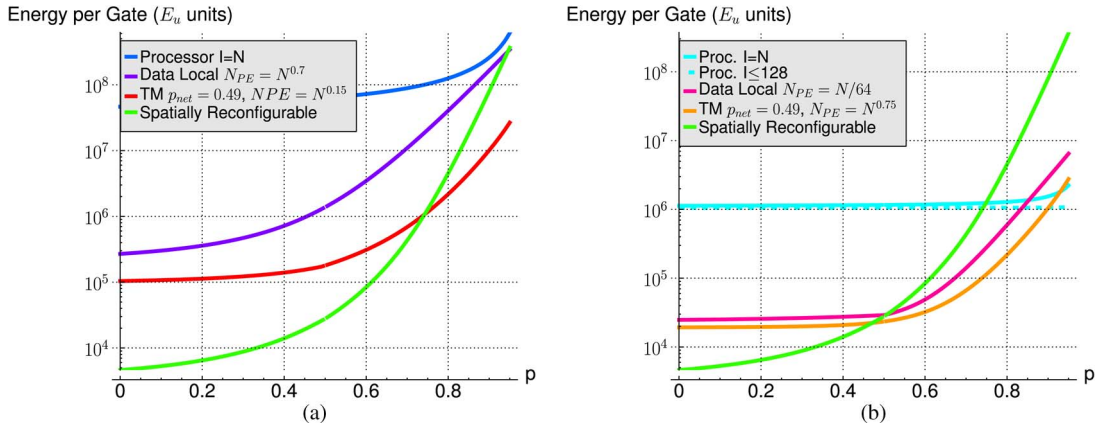


**Fig. 21.** *Energy scaling comparison versus rent exponent p at $N = 2^{24}$ (15), (33), (39), (49). (a) $W = 1$ and (b) $W = 64$.*

Fig. 21 shows that the fully spatial design can have lower energy at low $p$, but as $p$ increases, this time-multiplexed design achieves the lowest energy. This matches our growth expectations with the time-multiplexed energy growing as $N^{p'+0.5}$ and the spatial design energy growing as $N^{2p}$. We deliberately chose $p_{\text{net}} = 0.49$ to reduce the high energy costs in routing when $p > 0.5$.

Fig. 22 shows that the time-multiplexed design achieves this energy efficiency and performance improvement at the expense of area compared to the processor design [Fig. 9(b)]. The area is larger than the processor cases because it adds parallel PEs and switches that are not included in the sequential processor case. Nonetheless, with the time-mulitplexed design where $N_{\text{PE}} < N$, we limit the contribution to switches. In the $W = 1$ case, with $N_{\text{PE}} = N^{0.85}$, the area per gate trends to $410A_{\text{bit}}$ compared to $43A_{\text{bit}}$ for the sequential processor case. In the SIMD case, the area trends to $8A_{\text{bit}}$ compared to one $A_{\text{bit}}$ for the sequential processor case.

Fig. 23 rounds up delay, showing that the time-multiplexed design achieves delay close to the spatial implementation and better than data local and processor alternatives. For the SIMD cases, the time-multiplexed design gives up little performance to gain these energy and area benefits compared to the spatial design. The energy-minimizing time-multiplexed case sacrifices more speed as we saw in Fig. 19, but remains substantially faster than the sequential processor cases.

Table 1 rounds up the scaling comparison and provides point density and energy comparisons.

### B. Takeaway

Using time multiplexing, it is possible to exploit both instruction locality and wire sharing to further reduce area and energy. This is especially useful when the communication locality is low ($p > 0.5$). This achieves the best energy scaling identified, linear in $N^{p'+0.5}$ both with and without instruction sharing. The area scales as $N$, and

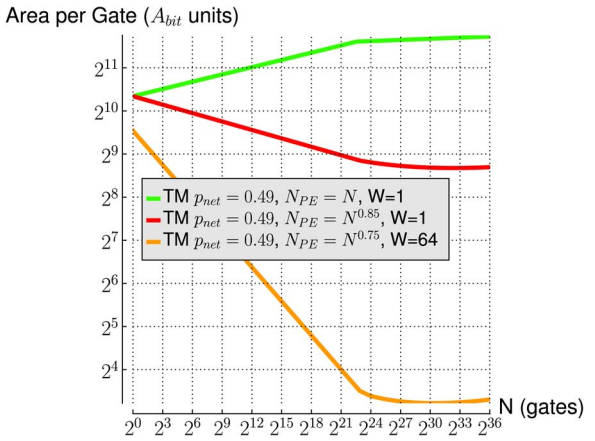


**Fig. 22.** *Multicontext area for $p = 0.7$ assuming eight metal layers (51).*

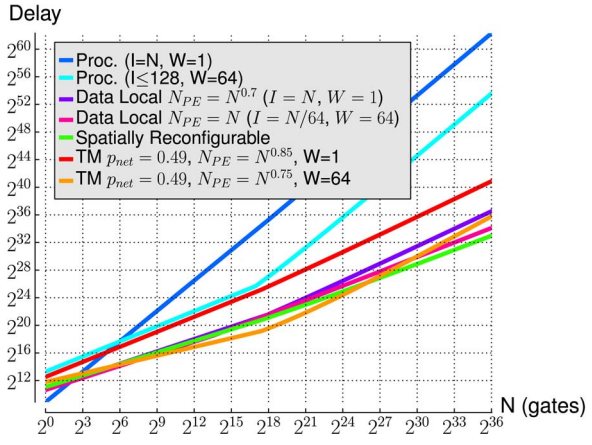


**Fig. 23.** *Delay comparison for $p = 0.7$ assuming eight metal layers $d = 10$ (16), (32) (40), (50).*

Table 1 Comparison of Scaling and Sample Densities at $N = 2^{24}$ Across Architectures

| | | | | Architecture | Description Local | Data Local | Instuction Local Spatial | Instuction Local Multicontext |
|---|---|---|---|---|---|---|---|---|
| | | | | Section | §V | §VII | §X | §XI |
| Scaling | | | | Total Area | $N$ | $N\log(N)$ | $N^{2p'}$ | $N$ |
| | | | | Total Energy | $N^{1.5}\log(N)$ | $N^{p'+0.5}\log^{1.5}(N)$ | $N^{2p'}$ | $N^{p'+0.5}$ |
| | | | | Delay | $N^{1.5}$ | $N^{p+0.5}+d\sqrt{N}$ | $d\cdot N^{p'}$ | $N^{p}+d\sqrt{N}$ |
| | $p$ | $I$ | $W$ | | | | | |
| Area/Gate (Memory Bit Area Units) | 0.5 | $N$ | 1 | | 34 | 122 | 440 | 280 |
| | | 128 | 64 | | 1.0 | 6.4 | 440 | 8.3 |
| | 0.7 | $N$ | 1 | | 44 | 122 | 4200 | 450 |
| | | 128 | 64 | | 1.0 | 6.4 | 4200 | 10 |
| Energy/Gate-Eval (Technology dependent $E_u = C_{unit\_wire}V^2$ units) | 0.5 | $N$ | 1 | | $6\times10^{7}$ | $3\times10^{6}$ | $3\times10^{4}$ | $2\times10^{5}$ |
| | | 128 | 64 | | $1\times10^{6}$ | $3\times10^{4}$ | $3\times10^{4}$ | $2\times10^{4}$ |
| | 0.7 | $N$ | 1 | | $9\times10^{7}$ | $1\times10^{6}$ | $4\times10^{5}$ | $7\times10^{5}$ |
| | | 128 | 64 | | $1\times10^{6}$ | $8\times10^{4}$ | $4\times10^{5}$ | $7\times10^{4}$ |
| Compute Density (Gate-Evals/Area-Time) | 0.5 | $N$ | 1 | | $2\times10^{-15}$ | $9\times10^{-10}$ | $2\times10^{-10}$ | $6\times10^{-11}$ |
| | | 128 | 64 | | $2\times10^{-11}$ | $4\times10^{-8}$ | $2\times10^{-10}$ | $6\times10^{-8}$ |
| | 0.7 | $N$ | 1 | | $1\times10^{-15}$ | $9\times10^{-11}$ | $7\times10^{-12}$ | $1\times10^{-12}$ |
| | | 128 | 64 | | $2\times10^{-11}$ | $3\times10^{-9}$ | $7\times10^{-12}$ | $4\times10^{-9}$ |

To keep scaling entries simple, $p' = \max(0.5, p)$ and we omit $\log(N)$ factors that occur at $p = 0.5$ and $p = 1.0$.

computational density scales as $1/(d\sqrt{N})$, the best of all the designs.

### C. Open

As noted at the end of Section X, embedded memories provide another model for exploring the reconfigurable architecture design space between the fully spatial and the fully sequential extremes, particularly when regularity allows instruction sharing. Characterizing the relation between these approaches is an important topic that merits further research.

## XII. MISMATCH ARCHITECTURAL AND APPLICATION PARAMETERS

The characterization shown so far assumes that the application characteristics match the architectural parameters. From the graphs shown [Figs. 9(c), 13(c), and 20], it is clear how the implementation can be more costly if the SIMD word width supported by the architecture ($W_{\text{arch}}$) is smaller than the SIMD width allowed by the task ($W_{\text{app}}$). We will suffer a different kind of inefficiency if the architecture has a smaller natural SIMD word width than the application. For example, if $W_{\text{arch}} = 64$ but $W_{\text{app}} = 16$, we lack the ability to control the gates independently. This may mean we need a larger component to accommodate the task ($N_{\text{arch}} = (W_{\text{arch}}/W_{\text{app}})N_{\text{app}}$), making memories larger and requiring more energy per operation.

Similar inefficiencies occur when the locality is mismatched. We can see directly from the graphs [Figs. 16 and 21] that higher locality (lower $p$) results in less area, energy, and delay. Using a larger $p$ than necessary ($p_{\text{arch}} > p_{\text{app}}$) directly leads to more costly implementations. Similarly, if $p_{\text{arch}} < p_{\text{app}}$, we may again need a larger component to accommodate the task ($N_{\text{arch}} = ((N_{\text{app}})^{p_{\text{app}}}/(N_{\text{app}})^{p_{\text{arch}}})N_{\text{app}} = (N_{\text{app}})^{1+p_{\text{app}}-p_{\text{arch}}}$), which will cost more area, energy, and delay.

Mismatches between task size and the size of the component device can also lead to inefficiencies. For example, if we place a circuit with $2^{18}$ gates on a component designed to support $2^{20}$ gates, we are using a design that is larger than necessary—in addition to the obvious area inefficiency, this can be an energy and delay inefficiency for architectures where energy and delay are driven by $N$. Specifically, the processor with monolithic memory will use $\log(N)\sqrt{N}$ energy per gate even if the actual number of gates evaluated is smaller. With appropriately placed IOs, the locality exploiting designs may be able to spend energy only proportional to the actual design size.

Reference [37, Ch. 36] treats the impact of SIMD and serialization mismatches on computational density, and [38] treats the impact of $p$ mismatches on area. It is often possible to select architectural parameters that limit the inefficiency due to mismatches with the application.

## XIII. HYBRID ARCHITECTURES

The treatment in this paper has also focused on pure building blocks assuming applications have homogeneous characteristics and architectures are built to support a single set of parameters. In practice, applications contain a mix of subcomputations with different characteristics. The architectural points explored here can be seen as component building blocks for the composition of heterogeneous, hybrid architectures that combine portions of each. The famous 90/10 rule from Knuth suggests that 90% of the runtime is spent in only 10% of the code [39]. Such a profile might benefit from an architecture that had two components—one that focused on area minimization for the 90% of the code that runs only 10% of the time, and

another focused on maximizing computational density and minimizing the energy for the 10% of the code that runs 90% of the time. This has led to designs that combine the area efficiency of a processor with the computational density and energy efficiency of a spatially reconfigurable compute engine, starting from Estrin's Fixed+Variable computer [40], through a host of design proposals (e.g., [41], [42]), leading to today's FPGAs with embedded processors, such as Xilinx's Zynq and Altera's Arria and Cyclone V SoC.

## XIV. ENERGY AND POWER DENSITY

In the past, density—how large a computation we could build with limited materials (vacuum tubes, transistors, silicon area)—was our key limitation. As Moore's Law's exponential growth reduced the cost per transistor, performance per cost mattered, making computational density the key concern; high computational density allowed us to achieve higher performance for a given, limited die size (cost). Today, the increasing role of mobile devices makes energy efficiency—the energy cost per operation—the dominant metric for many applications.

Energy is also becoming the critical limiter to performance. During the era of Dennard Scaling [43], as transistors shrunk, we were also able to scale down operating voltages to maintain a constant power density—we were able to put exponentially more transistors onto a silicon die without increasing the power per die or per centimeter squared. However, subthreshold leakage now prevents us from scaling voltage according to ideal Dennard Scaling laws [44], [45]. The result is rising power density to the point where our ability to remove the heat—to cool the device—limits the density and speed of operation. That is, *we can now place more transistors on a silicon die than we can afford to turn on* at their maximum operating speed [46]—a phenomenon known as *dark silicon* [45], [47]. In the extreme, in this new era of dark silicon, energy efficiency determines performance [48]. The architecture that minimizes the energy per gate evaluation will maximize the performance per Watt and, hence, the performance when operating with a limited area in a limited power density envelope. The energy efficiency of reconfigurable architectures (Fig. 21 and Table 1) may be their key asset in years to come.

## XV. BETTER THAN CUSTOM ASIC?

We pay for postfabrication configurability with area—area to store the configuration, area for gates that have more functionality than strictly necessary, and area for wires that may not be used. A large area makes wires longer, increasing delay, and decreasing performance. Longer wires mean more energy to communicate with the data. In cases that are not fully spatial, we also pay energy-reading configurations from memory. For these reasons, our postfabrication reconfigurable designs are less dense, lower performance, and higher energy than a custom integrated circuit (IC) that performs a single function, an ASIC [49]. Nonetheless, the postfabrication device can use its strengths to outperform the ASIC.

The postfabrication capability means the device can be specialized to the instantaneous needs of the task in a way that is not practical for an ASIC [37, Ch. 22]. For example, an FPGA can be programmed to implement a specific set of constant coefficient multiplications for a filter [50], match a particular pattern [51], or perform Boolean constraint propagation on a particular SAT instance [52], when an ASIC must be prepared to handle any set of coefficients, any pattern match, or any SAT instance. This reduces or inverts the traditional ASIC advantage over FPGAs. On a longer time scale, this ability to change the configuration allows rapid adoption of new algorithms and standards. This allows already fabricated products to benefit from the latest conceptual advances, which can also improve the reconfigurable solution relative to an ASIC running old algorithms or standards. If a design bug manifests after the chip has been fielded, an inflexible ASIC solution could be rendered useless, whereas a reconfigurable solution can be corrected in the fielded system.

As we scale to smaller feature sizes and integrate more transistors onto a single design, it becomes increasingly impossible to fabricate every one of the multiple billions of transistors on a silicon die perfectly. A variation in feature dimensions and doping levels leads to a wide variation in transistor characteristics [53]. This means some transistors will be unusable. Other transistors will operate only at high voltages, further aggravating attempts to reduce voltage to reduce energy and power density. The characteristics of devices will change during operation, meaning many transistors will fail during operation. Postfabrication reconfigurable devices can mitigate these yield, variation, and aging problems by identifying the unusable devices and changing the assignment of gates and interconnect to physical 4-LUTs and wiring segments to avoid the unusable devices [54]–[58]. This allows postfabrication architectures to use more aggressive technologies (smaller feature sizes and voltages) [59] and maintain their performance for longer operational lifetimes, also reducing the traditional gap with ASICs.

The VLSI complexity results (Section IX) show that less local designs ($p > 0.5$) can become wire dominated. At the wire-dominated extreme, the additional area overhead for programmable switches does not matter, reducing one of the key costs of postfabrication devices. Moreover, as we have shown (Section XI), in these regimes of operation, it is more energy efficient to time-multiplex the wires—share them among signals—than to dedicate a wire to a single signal. If the ASIC stays with fully spatial wiring, a postfabrication device that time-multiplexes its wires will close the gap and can ultimately be smaller and lower energy than the ASIC.

Of course, a custom ASIC can use all of these techniques too. It can provide, perhaps limited, reconfigurability to changing task needs, evolving standards, and in-circuit repair. It can include spare resources to address yield, variation, and aging. It can time-share its interconnect. However, as it does so, it increases its reconfigurability and, itself, becomes a reconfigurable device.

## XVI. SUMMARY

Postfabrication programmable architectures remove manufacturing as a bottleneck between concepts or repairs and implementation. Design customization becomes an information problem where we specify the instruction bits that configure the generic device to perform a specific task. By exploiting a typical application structure, such as communication locality and wide-word instruction sharing, we can reduce the overhead associated with storing and applying these instruction bits. Within the space of postfabrication programmable architectures, reconfigurable architectures use spatially distributed processing elements and efficient interconnection networks to exploit data and instruction locality. This reduces computation time and computation energy at a cost of area (decreased gate density) compared to stored-program processors. As we continue to scale technology to smaller, error- and variation-prone feature sizes with more raw capacity on inexpensive chips but increasing nonrecurring engineering costs, reconfigurability will play an increasing role in all computational components. ■

## APPENDIX

### A. Symbol

Tables 2–5 summarize the symbols used in this paper.

### B. Models for Data Local Case

Here are the full models for the data local case as we vary the number of processing elements $N_{\mathrm{PE}}$ and as a function of the instruction-sharing parameters $I$ and $W$.

Table 2 Technology Parameters

| Parameter | Description | Intro. |
|---|---|---|
| $E_u$ | Energy per unit length of wire | III-C1 |
| $C_{unit\_wire}$ | Capacitance per unit length of wire | XI-A |
| $V$ | Supply Voltage | |
| $FP$ | Full pitch between wires | III-C1 |
| $L$ | Number of metal layers for routing | IX |
| $A_{bit}$ | Area of SRAM cell | III-C1 |
| $A_{mux}$ | Area of 2:1 multiplexer | |
| $A_{shift}$ | Area of shift register | |
| $A_{xpoint}$ | Area of crosspoint with configuration | III-C2 |
| $A_{sw}$ | Area of bit-serial, packet-switch router in H-Tree | VII-A |
| $A_{4LUT}$ | Area of a 4-LUT without configuration | VI-C |

Table 3 Design Netlist Parameters

| Parameter | Description | Intro. |
|---|---|---|
| $N$ | Number of 4-LUTs in netlist | III |
| $d$ | 4-LUTs in critical path of netlist (circuit depth) | III-C4 |
| $c$ | Constant in Rent's Rule (roughly, the number of inputs and outputs on indivisible elements at the leaves of the decomposition tree) | IV |
| $p$ | Exponent in Rent's Rule | |
| $p'$ | $\max(p, 0.5)$ | VII-A |
| $I$ | Number of unique instructions needed to describe computation when exploiting instruction sharing | VI-A |
| $W$ | SIMD Word width | VI-B |

The PE will have data memory that holds its fraction of the total data $N/N_{\mathrm{PE}}$ and this data memory is as wide as the SIMD word width. It also need not hold more than $I$ instructions

$$A_{dlpe}(N, N_{\mathrm{PE}}) = A_{4\mathrm{LUT}} + A_{\mathrm{rmem}}\left(W, \frac{N}{W \cdot N_{\mathrm{PE}}}\right) + A_{\mathrm{smem}}(1, I(16 + k\log(N))).$$

The total area is the area of all the PEs and switches

$$A_{\mathrm{data}}(N, N_{\mathrm{PE}}) = N_{\mathrm{PE}} \cdot \left(A_{dlpe}(N, N_{\mathrm{PE}}) + A_{\mathrm{switch}}\right). \quad (38)$$

This data local PE requires energy to read its data, $E_{ddpe}$. The implementation also spends energy reading the instructions, $E_{idpe}$, and communicating amongst PEs, $E_{\mathrm{cdata}}$:

$$E_{ddpe}(N, N_{\mathrm{PE}}) = 4\left(\frac{N}{N_{\mathrm{PE}}}\right) \cdot E_{\mathrm{rmem}}\left(W, \frac{N}{W \cdot N_{\mathrm{PE}}}\right)$$
$$E_{idpe}(N, N_{\mathrm{PE}}) = N_{\mathrm{succbits}} \cdot E_{\mathrm{smem}}(1, I(16 + k\log(N)))$$
$$E_{\mathrm{cdata}}(N, N_{\mathrm{PE}}) \leq 8\sqrt{N_{\mathrm{PE}} \cdot A_{dl}}\left(\frac{\log(N)}{W} + 1\right)$$

Table 4 Architecture Parameters

| Parameter | Description | Intro. |
|---|---|---|
| $M$ | Number of memory words stored in a memory bank | III-C1 |
| $W$ | Width of word read from a memory bank | |
| $N_{PE}$ | Number for processing elements in a multiple PE design where a leaf PE is serialized to evaluate multiple 4-LUTs | VII-B |
| $c_{net}$ | number of input (output) connections between PE and physical network | XI |
| $p_{net}$ | Interconnect growth rate in physical tree network | |
| $S$ | Serialization in leaf PE ($N/N_{PE}$) | |

Table 5 Characteristic Models

| Characteristic | Description | Intro. |
|---|---|---|
| $A_{rmem}$ | Area of random access memory | III-C1 |
| $E_{rmem}$ | Energy of word-wide read or write in random access memory | |
| $A_{smem}$ | Area of sequentially accessed memory | |
| $E_{smem}$ | Energy of word-wide read or write in sequentially accessed memory | |
| $A_{xbar}$ | Area of crossbar | III-C2 |
| $E_{xbar}$ | Energy of crossbar | |
| $A_{proc}$ | Area of unoptimized processor | III-C3 |
| $E_{proc}$ | Energy of unoptimized processor | |
| $I_{ibits}$ | Number of instruction bits to specify communication | V |
| $I_{bits}$ | Total number of instruction bits to specify computation and communication | |
| $A_{pdesc}$ | Area of processor exploiting description locality | |
| $E_{pdesc}$ | Energy of processor exploiting description locality | |
| $A_p$ | Area of processor generalized for instruction sharing | VI-C |
| $E_p$ | Energy of processor generalized for instruction sharing | |
| $D_p$ | Delay of processor generalized for instruction sharing | |
| $A_{dlpe}$ | Area of a data local PE | VII-A |
| $A_{data}$ | Area of a data local design | |
| $N_{succbits_i}$ | Number of successor bits that must be read at PE $i$ | |
| $E_{idpe_i}$ | Energy cost of reading instruction bits at PE $i$ | |
| $E_{idata}$ | Total energy reading instruction bits in data local design | |
| $E_{cdata}$ | Total energy communicating data on data local design | |
| $E_{data}$ | Total energy for data local design | |
| $D_{data}$ | Delay data local design | B |
| $A_{spe}$ | Area of spatial PE | X |
| $A_{sws}$ | Total switch area (including instruction memory) | |
| $A_{active}$ | Total active area | |
| $Wires$ | Total wires across width of network | |
| $L_{wires}$ | Width of network due to wires | |
| $L_{side}$ | Length of side of spatial design | |
| $A_{spatial}$ | Area of spatial design | |
| $D_{spatial}$ | Delay of spatial design | |
| $E_{spatial}$ | Energy of spatial design | |
| $A_{tm}$ | Area of time-multiplexed design | XI |
| $D_{tm}$ | Delay of time-multiplexed design | |
| $E_{tm}$ | Energy of time-multiplexed design | |

$$\times \left( \sum_{i=\left\lceil \frac{\log_2(N_{over}N_{PE})}{2} \right\rceil}^{\left\lceil \frac{\log_2(N)}{2} \right\rceil} \left( (2^i)^{(2p-1)} \right) \right)$$

$$E_{data}(N, N_{PE}) = N_{PE} \cdot E_{ddpe}(N, N_{PE}) + E_{idpe}(N, N_{PE}) + E_{cdata}(N, N_{PE}). \quad (39)$$

The total computation time could be limited by the PE serialization $N/N_{PE}$, the interconnect serialization $cN^p$, or the critical path communication latency across the chip $d\sqrt{A_{data}(N, N_{PE})}$. We assume the cycle time is set by the wire lengths in the PE $\sqrt{A_{dlpe(N,N_{PE})}}$

$$D_{data}(N, N_{PE}) \approx \left( \frac{N}{N_{PE}} + cN^p \right) \sqrt{A_{dlpe(N,N_{PE})}} + d\sqrt{A_{data}(N, N_{PE})}. \quad (40)$$

### B. Models for the Time-Multiplexed Case

Here are the full models for the time-multiplexed interconnect case as we vary the number of processing elements $N_{PE}$ and as a function of the instruction-sharing parameters $I$ and $W$.

The serialization of computation at the PE $S$ is

$$S = \frac{N}{N_{PE}}$$

We use a context factor $CF$ to estimate precedent constraint effects [9]. For concrete illustration in this paper, we assume $CF = 4$. The total number of cycles that the PE needs to execute is the greater of the serialization of the computation and the communication between the PE and the network

$$C_p = \left( \frac{CF}{W} \right) \max \left( S, \frac{c}{c_{net}} (S)^{p-p_{net}} \right) \quad (41)$$

The PE must hold the data for the $S$ 4-LUTs and instructions for the 4-LUTs and communication

$$A_{pe} = 4A_{mux}(c_{net}S^{p_{net}}) + 4A_{rmem}(1, S) + A_{4lut} + 4A_{smem}\left( \log(c_{net}S^{p_{net}} + 1) + \log\left(\frac{S}{W}\right), C_p \right) + A_{smem}\left( 4\log\left(\frac{S}{W}\right) + 16, \frac{S}{W} \right).$$

Similarly, the energy per node evaluated in the PE accounts for data reads and writes and instruction reads

$$E_{pe} = 8E_{rmem}(1, S) + 4\frac{C_p}{W \cdot S} E_{smem}\left( \log(c_{net}S^{p_{net}} + 1) + \log\left(\frac{S}{W}\right), C_p \right) + \left(\frac{1}{W}\right) E_{smem}\left( 4\log\left(\frac{S}{W}\right) + 16, \frac{S}{W} \right) + 2 \times 6C_{wire}\sqrt{A_{rmem}(1, S)}\left( \log\left(\frac{S}{W}\right) + 1 \right)$$

We count the switches at each tree level and account for their local memories

$$A_{sws} = \sum_{l=\log(S)}^{\log(N)} \left(\frac{N}{2^l}\right) c_{\text{net}} (2^l)^{p_{\text{net}}} \times \left(3A_{\text{mux2}} + A_{\text{smem}}\left(3, \frac{CF \cdot c}{W \cdot c_{\text{net}}} (2^l)^{p-p_{\text{net}}}\right)\right). \quad (42)$$

The total active area is the switch area and PE area

$$A_{\text{mcactive}} = N_{\text{PE}} \cdot A_{pe} + A_{sws}. \quad (43)$$

Similar to the spatial design, we must calculate the wire contribution to compute the wire lengths

$$MCWires = 2 \sum_{l=\frac{\log(S)}{2}}^{\frac{\log(N)}{2}} \left(\sqrt{\frac{N}{2^{2l}}}\right) c_{\text{net}} (2^{2l})^{p_{\text{net}}} \quad (44)$$

$$L_{\text{mcwire}} = \frac{2 \times FP \times MCWires}{L} \quad (45)$$

$$L_{\text{mcside}} \leq \sqrt{A_{\text{mcactive}}} + L_{\text{mcwire}}. \quad (46)$$

We can then calculate the total energy in wires and the instruction memories local to the switches in the tree

$$E_{\text{mcwire}} = E_u \sum_{l=\log(S)}^{\log(N)} \left(\frac{N}{2^l}\right) (c2^{lp}) \left(\frac{L_{\text{mcside}}}{2^{\lceil(\log(N)-l)/2\rceil}}\right) \quad (47)$$

$$E_{\text{mcimem}} = \sum_{l=\log(S)}^{\log(N)} \left(\frac{N}{2^l}\right) \left(\frac{c_{\text{net}}}{W}\right) (2^l)^{p_{\text{net}}} \left(\frac{c}{c_{\text{net}}} 2^{l(p-p_{\text{net}})}\right) \times E_{\text{smem}}\left(3, \frac{CF \cdot c}{W \cdot c_{\text{net}}} (2^l)^{p-p_{\text{net}}}\right). \quad (48)$$

We must also account for the energy in the clocking network. *CSF* accounts for the fact that the clock switches multiple times per evaluation; we use $CSF = 4$

$$E_{\text{clk}} = CSF \cdot CF \cdot \frac{c}{c_{\text{net}}} \times (E_{\text{tdist}} + E_{\text{sdist}})$$

$$E_{\text{tdist}} = E_u \frac{L_{\text{mcside}}}{2} \sum 2^{l(p-p_t)} \left(\frac{\sqrt{N}}{2^{\frac{l}{2}}}\right)$$

$$E_{\text{sdist}} = E_u \sum_{l=\log(S)}^{l=\log(N)} (2^l)^{p-p_t} \left(\frac{3}{2}\right) N_{sw}(l) \sqrt{A_{swm}(l)}$$

$$N_{sw}(l) = \left(\frac{N}{2^l}\right) c_{\text{net}} (2^l)^{p_{\text{net}}}$$

$$A_{swm(l)} = \sqrt{\left(A_{1:1} + A_{\text{imem}}\left(3, \frac{CF \cdot c}{W \cdot c_{\text{net}}} (2^l)^{p-p_t}\right)\right)}.$$

Finally, we put this altogether to estimate the overall energy, delay, and area for the time-multiplexed designs

$$E_{tm} = N \cdot E_{pe} + E_{\text{mcwire}} + E_{\text{mcimem}} + E_{\text{clk}} \quad (49)$$

$$D_{tm} = \left(\frac{CF \cdot c}{c_{\text{net}}}\right) N^{(p-p_{\text{net}})} \cdot L_{\text{mcside}} \quad (50)$$

$$A_{tm} = (L_{\text{mcside}})^2. \quad (51)$$

## Acknowledgment

The authors would like to thank B. Fugate, B. Gojman, E. Kadric, D. Lakata, N. Roessler, S. Vijayvargiya, and the anonymous IEEE reviewers for providing feedback on early drafts of this paper.

## REFERENCES

[1] A. M. Turing, "On computable numbers, with an application to the entscheidungs problem," in *Proc. London Math. Soc.*, 1936, vol. 42, no. 2.

[2] A. M. Turing, "On computable numbers, with an application to the entscheidungs problem: A correction," in *Proc. London Math. Soc.*, 1938, vol. 43, no. 6, pp. 544–546.

[3] G. E. Moore, "Cramming more components onto integrated circuits," *Electron. Mag.*, p. 4, 1965.

[4] G. Moore, "No exponential is forever: But "forever" can be delayed! [semiconductor industry]," in *Proc. ISSCC*,, Feb. 2003, vol. 1, pp. 20–23.

[5] C. Rowen *et al.*, "A pipelined 32b NMOS microprocessor," in *Proc. ISSCC*, Feb. 1984, pp. 180–181.

[6] J. von Neumann, "First draft of a report on EDVAC," Moore School Elect. Eng., Univ. Pennsylvania, Jun. 30, 1945, Tech. rep.

[7] H. H. Goldstine and A. Goldstine, "The electronic numerical integrator and computer (ENIAC)," *Math. Tables Other Aids Comput.*, vol. 2, no. 15, pp. 97–110, Jul. 1946.

[8] W. S. Carter *et al.*, "A user programmable reconfigurable logic array," in *Proc. IEEE CICC*,, May 1986, pp. 233–235.

[9] A. DeHon, "Wordwidth, instructions, looping, virtualization—The role of sharing in absolute energy minimization," in *Proc. FPGA*, 2014, pp. 189–198.

[10] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Mateo, CA, USA: Morgan Kaufmann, 1992.

[11] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, Eds., *Limits to Parallel Computation: P-Completeness Theory*. New York, NY, USA: Oxford University Press, 1995.

[12] E. F. Rent, "Memorandum to: File, subject: Microminiature packaging-logic block to pin ratio," *IEEE Solid-State Circuits Mag.*, vol. 2, no. 1, pp. 40–41, Winter 2010, reprint of original 1960 IBM memo.

[13] B. S. Landman and R. L. Russo, "On pin versus block relationship for partitions of logic circuits," *IEEE Trans. Comput.*, vol. C-20, no. 12, pp. 1469–1479, Dec. 1971.

[14] S. Bhatt and F. T. Leighton, "A framework for solving VLSI graph layout problems," *J. Comput. Syst. Sci.*, vol. 28, pp. 300–343, 1984.

[15] C. Thompson, "Area-time complexity for VLSI," in *Proc. ACM STOC*, May 1979, pp. 81–88.

[16] W. E. Donath, "Placement and average interconnection lengths of computer logic," *IEEE Trans. Circuits Syst.*, vol. 26, no. 4, pp. 272–277, Apr. 1979.

[17] M. J. Flynn, "Very high speed computing systems," *Proc. IEEE*, vol. 54, no. 12, pp. 1901–1909, Dec. 1966.

[18] H. J. Hoover, M. M. Klawe, and N. J. Pippenger, "Bounding fan-out in logical networks," *J. ACM*, vol. 31, no. 1, pp. 13–18, Jan. 1984.

[19] R. I. Greenberg and C. E. Leiserson, "Randomized Routing on Fat-Trees, earlier MIT/LCS/TM-307," in *Randomness in Computation*, vol. 5. Greenwich, CT, USA: JAI Press, 1988, ser. Advances in Computer Research.

[20] W. Tsu *et al.*, "HSRA: High-speed, hierarchical synchronous reconfigurable array," in *Proc. FPGA*, Feb. 1999, pp. 125–134.

[21] J. E. Savage, "Planar circuit complexity and the performance of VLSI algorithms," in *VLSI Syst. Comput.*, 1981, pp. 61–68.

[22] F. T. Leighton, "New lower bound techniques for VLSI," in *Proc. IEEE Symp. FOCS*, 1981, pp. 1–12.

[23] K. Fujiyoshi, Y. Kajitani, and H. Niitsu, "Design of minimum and uniform bipartites for optimum connection blocks of FPGA," *IEEE Trans. Comput.-Aided Design*, vol. 16, no. 11, pp. 1377–1383, Nov. 1997.

[24] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "Architecture of centralized field-configurable memory," in *Proc. FPGA*, 1995, pp. 97–103.

[25] E. Kadric, K. Mahajan, and A. DeHon, "Kung fu data energy–minimizing communication energy in FPGA computations," in *Proc. IEEE FCCM*, Boston, MA, USA, 2014.

[26] A. DeHon, "Unifying Mesh- and Tree-Based Programmable Interconnect," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 10, pp. 1051–1065, Oct. 2004.

[27] C. E. Leiserson, "Fat-trees: Universal networks for hardware efficient supercomputing," *IEEE Trans. Comput.*, vol. C-34, no. 10, pp. 892–901, Oct. 1985.

[28] D. C. Chen and J. M. Rabaey, "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths," *IEEE J. Solid-State Circuits*, vol. 27, no. 12, pp. 1895–1904, Dec. 1992.

[29] E. Mirsky and A. DeHon, "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proc. IEEE FFCM*, Apr. 1996, pp. 157–166.

[30] C. Ebeling, D. Cronquist, and P. Franklin, "Rapid—Reconfigurable pipelined datapath," in *Proc. FPL*. New York, NY, USA: Springer, no. 1142, Sep. 1996, pp. 126–135, ser. Lecture Notes in Computer Science.

[31] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *Proc. FPGA*, 1999, pp. 135–143.

[32] S. C. Goldstein *et al.*, "Piperench: A reconfigurable architecture and compiler," *IEEE Comput.*, vol. 33, no. 4, pp. 70–77, Apr. 2000.

[33] F.-J. Veredas, M. Scheppler, W. Moffat, and B. Mei, "Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes," in *Proc. FPL*, 2005, pp. 106–111.

[34] A. DeHon, "DPGA utilization and application," in *Proc. FPGA*, Feb. 1996, pp. 115–121.

[35] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proc. IEEE FCCM*, Apr. 1997, pp. 22–28.

[36] T. R. Halfhill, "Tabula's time machine," *Microprocessor Rep.*, Mar. 29, 2010.

[37] S. Hauck and A. DeHon, Eds., *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. New York, NY, USA: Elsevier, 2008, ser. Systems-on-Silicon.

[38] A. DeHon, "Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization)," in *Proc. FPGA*, Feb. 1999, pp. 69–78.

[39] D. E. Knuth, "Empirical study of fortran programs," *Softw. Practice Exper.*, vol. 1, no. 1, pp. 105–133, 1971.

[40] G. Estrin, "Organization of computer sytems: The fixed plus variable structure computer," in *Proc. Western Joint Comput. Conf.*, 1960, pp. 33–40.

[41] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *MICRO*, Nov. 1994, pp. 172–180, IEEE Comput. Soc.

[42] T. Callahan, J. Hauser, and J. Wawrzynek, "The garp architecture and C compiler," *IEEE Comput.*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[43] R. H. Dennard *et al.*, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.

[44] M. Bohr, "A 30 year retrospective on dennard's MOSFET scaling paper," in *Proc. IEEE Solid-State Circuits Soc. Newslett.*, Winter 2007, vol. 12, no. 1, pp. 11–13.

[45] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. ISCA*, 2011, pp. 365–376.

[46] M. Horowitz *et al.*, "Scaling, power, the future of CMOS," in *Proc. IEDM*, Dec. 2005, pp. 7–15.

[47] G. Venkatesh *et al.*, "Conservation cores: Reducing the energy of mature computations," in *Proc. ASPLOS*, 2010, pp. 205–218.

[48] S. H. Fuller and L. I. Millett, Eds., *The Future of Computing Performance: Game Over or Next Level?*, The National Academies Press, Washington, DC, USA, 2011.

[49] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design*, vol. 26, no. 2, pp. 203–215, Feb. 2007.

[50] A. DeHon, "Trends toward spatial computing architectures," in *Proc. IEEE ISSCC*, Feb. 1999, pp. 362–363.

[51] J. Villasenor, B. Schoner, K.-N. Chia, and C. Zapata, "Configurable computer solutions for automatic target recognition," in *Proc. IEEE FCCM*, Apr. 1996, pp. 70–79.

[52] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating boolean satisfiability with configurable hardware," in *Proc. IEEE FCCM*, Apr. 1998, pp. 186–195.

[53] K. Bernstein *et al.*, "High-performance CMOS variability in the 65-nm regime and beyond," *IBM J. Res. Develop.*, vol. 50, no. 4/5, pp. 433–449, Jul./Sep. 2006.

[54] J. Lach, W. H. Mangione-Smith, and M. Potkonjak, "Low overhead fault-tolerant FPGA systems," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 6, no. 2, pp. 212–221, Jun. 1998.

[55] R. Amerson, R. Carter, W. B. Culbertson, P. Kuekes, and G. Snider, "Teramac-configurable custom computing," in *Proc. IEEE FCCM*, Apr. 1995, pp. 32–38.

[56] S. Srinivasan *et al.*, "Toward increasing FPGA lifetime," *IEEE Trans. Dep. Secure Comput.*, vol. 5, no. 2, pp. 115–127, Apr. 2008.

[57] R. Rubin and A. DeHon, "Choose-your-own-adventure routing: Lightweight load-time defect avoidance," *ACM Trans. Reconfig. Tech. Syst.*, vol. 4, no. 4, Dec. 2011.

[58] A. DeHon and N. Mehta, "Exploiting partially defective LUTs: Why you don't need perfect fabrication," in *Proc. ICFPT*, Kyoto, Japan, Dec. 2013.

[59] N. Mehta, R. Rubin, and A. DeHon, "Limit study of energy & delay benefits of component-specific routing," in *Proc. FPGA*, 2012, pp. 97–106.

## ABOUT THE AUTHOR

**André DeHon** (Member, IEEE), received the S.B., S.M., and Ph.D. degrees in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology in 1990, 1993, and 1996, respectively.

From 1996 to 1999, he co-ran the BRASS group in the Computer Science Department at the University of California at Berkeley. From 1999 to 2006, he was an Assistant Professor of Computer Science at the California Institute of Technology. Since 2006, he has been in the Electrical and Systems Engineering Department at the University of Pennsylvania where he is now a Full Professor. He is broadly interested in how we physically implement computations from substrates, including VLSI and molecular electronics, up through architecture, CAD, and programming models. He places special emphasis on spatial programmable architectures (*e.g.* FPGAs) and interconnect design and optimization.