To Appear in *Proc. of The 28th Annual International Symposium on Computer Architecture, June 2001.*

NanoFabrics: Spatial Computing Using Molecular Electronics

Seth Copen Goldstein and Mihai Budiu
Carnegie Mellon University
{seth,mihaib}@cs.cmu.edu

Abstract

The continuation of the remarkable exponential increases in processing power over the recent past faces imminent challenges due in part to the physics of deep-submicron CMOS devices and the costs of both chip masks and future fabrication plants. A promising solution to these problems is offered by an alternative to CMOS-based computing, chemically assembled electronic nanotechnology (CAEN).

In this paper we outline how CAEN-based computing can become a reality. We briefly describe recent work in CAEN and how CAEN will affect computer architecture. We show how the inherently reconfigurable nature of CAEN devices can be exploited to provide high-density chips with defect tolerance at significantly reduced manufacturing costs. We develop a layered abstract architecture for CAEN-based computing devices and we present preliminary results which indicate that such devices will be competitive with CMOS circuits.

1 Introduction

We are approaching the end of a remarkably successful era in computing: the era where Moore's Law reigns, where processing power per dollar doubles every year. This success is based in large part on advances in complementary metal-oxide semiconductor (CMOS)-based integrated circuits. Although we have come to expect, and plan for, the exponential increase in processing power in our everyday lives, today Moore's Law faces imminent challenges both from the physics of deep-submicron CMOS devices and from the costs of both chip masks and next-generation fabrication plants.

A promising alternative to CMOS-based computing under intense investigation is chemically assembled electronic nanotechnology (CAEN), a form of electronic nanotechnology (EN) which uses self-alignment to construct electronic circuits out of nanometer-scale devices that take advantage of quantum-mechanical effects [10, 30]. In this paper we show how CAEN can be harnessed to create useful computational devices with more than 10^{10} gate-equivalents per cm^2. The fundamental strategy we will use is to substitute compile time (which is inexpensive) for manufacturing precision (which is expensive). We achieve this through a combination of reconfigurable computing, defect tolerance, architectural abstractions and compiler technology. The result will be a high-density low-power substrate which will have inherently lower fabrication costs than CMOS counterparts.

Using EN to build computer systems requires new ways of thinking about computer architecture and compilation. CAEN differs from CMOS: CAEN is extremely unlikely to be used to construct complex aperiodic structures. We introduce an architecture based on fabricating dense regular structures, which we call *nanoBlocks*, that can be programmed after fabrication to implement complex functions. We call an array of connected nanoBlocks a *nanoFabric*.

Compared to CMOS, CAEN-based devices have a higher defect density. Such circuits will thus require built-in defect tolerance. A natural method of handling defects is to first configure the nanoFabric for self-diagnosis and then to implement the desired functionality by configuring around the defects. Reconfigurabilty is thus integral to the operation of the nanoFabric. Their nature makes nanoFabrics particularly well suited for reconfigurable computing.

Reconfigurable computing changes as needed the function of programmable logic elements and their connections to storage, building efficient, highly parallel processing kernels, tailored for the application under execution. The network of processing elements is called a *reconfigurable fabric*. The data used to program the interconnect and processing elements is called a *configuration*. Examples of current reconfigurable fabrics are commercial Field Programmable Gate Arrays (FPGAs) such as [39, 2], and research prototypes, e.g. Chimaera [40] and PipeRench [18]. As we show later, one advantage of nanoFabrics over CMOS-based reconfigurable fabrics is that the area overhead for supporting reconfiguration is virtually eliminated. This will magnify the benefits of reconfigurable computing, yielding computing devices that may outperform traditional ones by orders of magnitude in many metrics, such as computing elements per cm^2 and operations per watt.

In the next section we present some recent research results, which indicate CAEN will be a successful technol-

ogy for implementing computing devices. We next analyze how the unique features of CAEN devices can be exploited, and how their limitations can be circumvented. In Section 3 we present an architecture that utilizes the capabilities of CAEN devices without requiring Herculean fabrication technology. In Section 4 we describe our top-level architectural abstraction, the Split-Phase Abstract Machine (SAM), which enables fast compilation of large programs. The simulation results in Section 5 indicate that the SAM abstraction does not hide the efficiency of the nanoFabric.

2 Electronic Nanotechnology

While CMOS fabrication will soon hit a wall due to a combination of economic and technical factors,[1] we are nowhere near the theoretical limits of physical computation[17]. In order to achieve these limits at room temperature we must work in the nanoscale regime, which currently involves various technologies that exploit the quantum-mechanical effects of small devices. Among these are single-electron transistors [9], nanowire transistors [11], quantum dots [37], quantum cellular automata [23], resonant tunneling devices [6], negative differential resistors (NDR) [7], and reconfigurable switches [8, 10]. In all cases, the fabricated devices are on the order of a few nanometers. Given the small sizes involved, these devices must be created and connected through self-assembly and self-alignment instead of lithography.

In this paper we limit ourselves to molecular devices which have I-V characteristics similar to those of their bulk counterparts. For example, the basis of rectification is different between a silicon-based p-n junction diode and a molecular diode, yet they both have similar I-V curves [4]. We choose to look at systems that can be built from nanoscale devices with bulk-semiconductor analogs so that (1) we can apply our experience with standard circuits to the system and (2) we can model the system with standard tools such as SPICE.

While there are still many challenges left in creating fully functional EN computing devices, recent advances indicate that EN could be a very successful post-CMOS technology. Several groups have recently demonstrated CAEN devices that are self-assembled or self-aligned (or both) [10, 28, 16, 32]. Advances have also been made in creating wires out of single-wall carbon nanotubes and aligning them on a silicon substrate [36, 29]. Even more practical is the fabrication of metal nanowires, which scale down to 5nm and can include embedded devices [25, 27]. Combined,these advances compel us to investigate further how to harness CAEN for computing in the post-CMOS age.

CAEN devices are very small: A single RAM cell will require 100nm^2 as opposed to $100,000\text{nm}^2$ for a single laid out CMOS transistor[2]. A simple logic gate or an static memory cell requires several transistors, separate p- and n-wells, etc., resulting in a factor of 10^5 difference in density between CAEN and CMOS. CAEN devices use much less power, since very few electrons are required for switching.

CAEN devices are particularly suited for reconfigurable computing since the configuration information for a switch does not need to be stored in a device separately from the switch itself [10]. On the other hand, A CMOS-based reconfigurable device requires a static RAM cell to controls each pass transistor. Also, two sets of wires are needed in CMOS: one for addressing the configuration bit and one for the actual signals. Perhaps a more realistic comparison of CAEN is to floating-gate technology which also stores the configuration information at the transistor itself. Like traditional transistors, floating-gate transistors also require two sets of wires. Furthermore, A CAEN switch behaves like a diode, but a floating gate transistor is bi-directional, and thus less useful for building programmable logic.

Electronic nanotechnology is quickly progressing and promises incredibly small, dense, and low-power devices. Harnessing this power will require new ways of thinking about the manufacturing process. We will no longer be able to manufacture devices deterministically; instead, post-fabrication reconfiguration will be used to determine the properties of the device and to avoid defects.

2.1 Fabrication and Architectural Implications

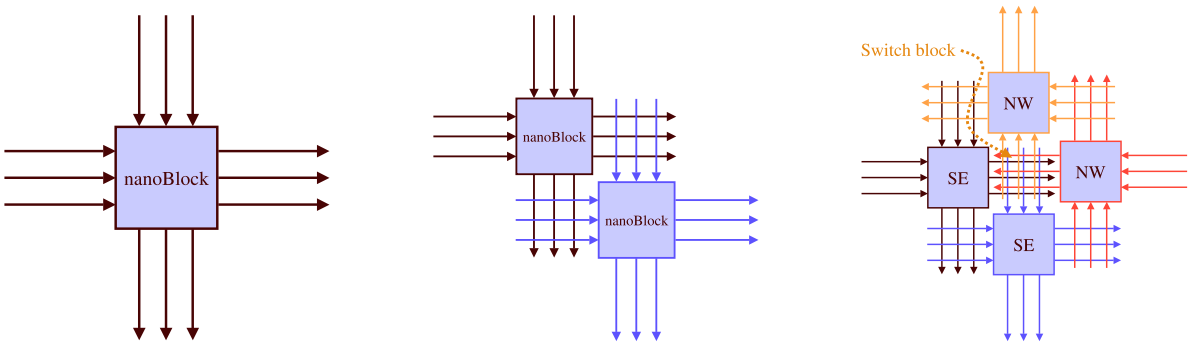
Here we briefly outline a plausible fabrication process. The process is hierarchical, proceeding from basic components (e.g. wires and switches), through self-assembled arrays of components, to complete systems. In the first step, wires of different types are constructed through chemical self-assembly.[3] The next step aligns groups of wires. Also through self-assembly, two planes of aligned wires will be combined to form a two-dimensional grid with configurable molecular switches at the crosspoints. The resulting grids will be on the order of a few microns. A separate process will create a silicon-based die using standard lithography. The circuits on this die will provide power, clock lines, an I/O interface, and support logic for the grids of switches. The die will contain "holes" in which the grids are placed, aligned, and connected with the wires on the die.

Using only self-assembly and self-alignment restricts us to manufacturing simple, regular structures, e.g., rafts of parallel wires or grids composed of orthogonal rafts.

[1] Among the technical issues are ultrathin gate oxides, short channel effects, and doping fluctuations [21].

[2] For the CAEN device we assume that the nanowires are on 10nm centers. A CMOS transistor with a 4:1 ratio in a 70nm process, (even using Silicon on Insulator, which does not need wells) with no wires attached measures 210nm x 280nm. Attaching minimally-sized wires to the terminals increases the size to 350nm x 350nm.

[3] By chemical self-assembly we mean a process by which the components (e.g., wires or devices) are synthesized and connected together through chemical processes. See [24] for an overview.



(a) *Single nanoBlock showing I/O lines*

(b) *Half of a switchblock*

(c) *Switchblock with four surrounding nanoBlocks*

Figure 1. *NanoBlock Connectivity.*

A post-fabrication configuration is used to create useful circuits (See Section 3.3). The small size and non-deterministic nature of the self-assembly will also give rise to high defect densities, which can be bypassed through reconfiguration (See Section 3.2).

While researchers have constructed three-terminal EN devices, the precise alignment required to colocate three wires at the device makes them unsuitable for producing real circuits with inexpensive chemical assembly. We thus assume that CAEN devices will be limited to performing logic using two terminal devices; i.e. diode-resistor logic (see Section 3.1). As the active components will be diodes and configurable switches, there will be no inverters. Because we cannot build inverters, all logic functions will generally compute both the desired output and its complement.

Even more important, the lack of a transistor means that special mechanisms will be required for signal restoration and for building registers. Using CMOS to buffer the signals is unattractive for two reasons: first, CMOS transistors are significantly larger and would decrease the density of the fabric. Second, the large size of CMOS transistors would slow down the nanoFabric. We have succesfully designed and simulated a molecular latch motivated by work in tunnel diodes [1, 26]. The latch is composed of a wire with two inline NDR molecules at either end. The latch combined with a clocking methodology, provides signal restoration, latching, and I/O isolation [31]. The requirement to condition signals will result in circuits which will be either slow (if transistors are used) or deeply pipelined (if latches are used).

The fabrication process also disallows the precise alignment required to make end-to-end connections between nanoscale wires. Our architecture ensures that all connections between nanoscale wires occur by crossing the wires.

3 NanoFabric

The architecture of the nanoFabric is designed to overcome the constraints associated with directed assembly of nanometer-scale components and exploit the advantages of molecular electronics. Because the fabrication process is fundamentally non-deterministic and prone to introducing defects, the nanoFabric must be reconfigurable and amenable to self-testing. This allows us to discover the characteristics of each nanoFabric and then create circuits that avoid the defects and use the available resources. For the foreseeable future, the fabrication processes will only produce simple, regular geometries. Therefore, the proposed nanoFabric is built out of simple, two-dimensional, homogeneous structures. Rather than fabricating complex circuits, we use the reconfigurability of the fabric to implement arbitrary functions post-fabrication. The construction process is also parallel; heterogeneity is introduced only at a lithographic scale. The nanoFabric can be configured (and reconfigured) to implement any circuit, like today's FPGAs; the nanoFabric though has several orders of magnitude more resources.

The nanoFabric is a 2-D mesh of interconnected nanoBlocks. The nanoBlocks are logic blocks that can be programmed to implement a three-bit input to three-bit output Boolean function and its complement (see Figure 1a). NanoBlocks can also be used as switches to route signals. The nanoBlocks are organized into clusters (See Figure 2). Within a cluster the nanoBlocks are connected to their nearest four neighbors. Long wires, which may span many clusters (long-lines), are used to route signals between clusters. The nanoBlocks on the perimeter of the cluster are connected to the long-lines. This arrangement is similar to commercial FPGAs (allowing us to leverage current FPGA tools) and has been shown to be flexible enough to implement any circuit on the underlying fabric.

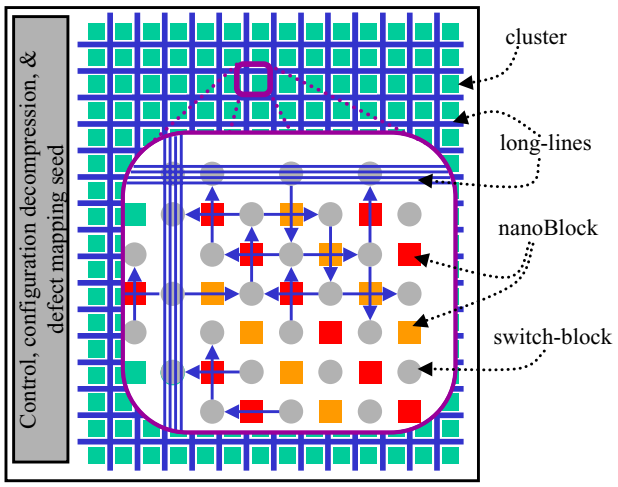


Figure 2. *The layout of the nanoFabric with a partial blowup of a single cluster and some of the adjacent long-lines.*

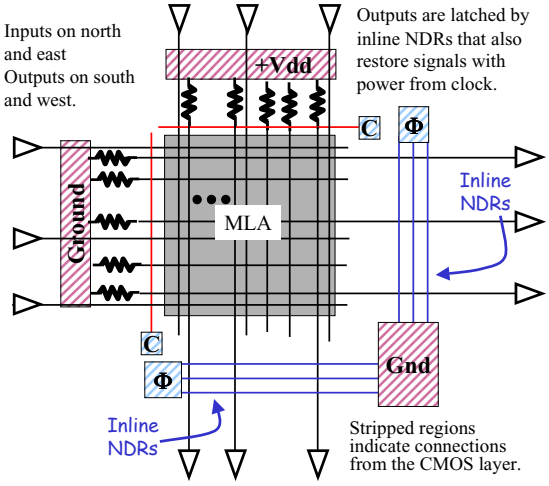


Figure 3. *A schematic of a nanoBlock.*

The nanoBlock design is dictated by fabrication constraints. Each side of the block can have inputs or outputs, but not both. Thus, the I/O arrangement in Figure 1a is required. We have arranged it so that all nanoscale wire-to-wire connections are made between two orthogonal wires; we do not need precise end-to-end alignment. Figures 1 (b) and (c) show how the outputs of one nanoBlock connect to the inputs of another. We call the area where the input and output wires overlap a *switch block*. Notice that the outputs of the blocks are either facing south and east (SE) or north and west (NW). By arranging the blocks such that all the SE blocks run in one diagonal and the NW run in the adjacent diagonal we can map any circuit netlist onto the nanoFabric. Since the nanoBlocks themselves are larger than the minimum lithographic dimension (e.g, greater than one micron), they can be positioned precisely at manufacturing time in the desired patterns.

In addition to the intra-cluster routing there are long-lines that run between the clusters to provide low-latency communication over longer distances. The nanowires in these tracks will be of varying lengths (e.g., 1,2,4, and 8 clusters long), allowing a signal to traverse one or more clusters without going through any switches. This layout is essentially that of an island style FPGA [33]. This general layout has been shown to be efficient and amenable to place-and-route tools [5]. Notice that all communication between nanoBlocks occurs at the nanoscale. The fact that we never need to go between nanoscale and CMOS components and back again increases the density of the nanoFabric and lowers its power requirements.

The arrangement of the clusters and the long-lines promote scalability in several ways. First, as the number of components increases we can increase the number of long-lines that run between the clusters. This supports routability of netlists. Second, each cluster is designed to be configured in parallel, allowing configuration times to remain reasonable even for very large fabrics. The power requirements remain low because we use molecular devices for all aspects of circuit operation. Finally, because we assemble the nanoFabric hierarchically we can exploit the parallel nature of chemical assembly.

3.1 NanoBlock

The nanoBlock is the fundamental unit of the nanoFabric. It is composed of three sections (see Figure 3): (1) the molecular logic array, where the functionality of the block is located, (2) the latches, used for signal restoration and signal latching for sequential circuit implementation, and (3) the I/O area, used to connect the nanoBlock to its neighbors through the switch block.

The molecular logic array (MLA) portion of a nanoBlock is composed of two orthogonal sets of wires. At each intersection of two wires lies a configurable molecular switch. The switches, when configured to be "on", act as diodes. Designing circuits for the MLA is significantly different than for a programmable logic array, which requires an OR and an AND plane. We have preliminary designs for a "standard cell" library using nanoBlocks, e.g. AND, OR, XOR, and ADDER.

Figure 4 shows the implementation of an AND gate, while Figure 5 shows the implementation for a half-adder. On the top part of Figure 5 is a schematic of the portion of the circuit used to generate the sum output. This circuit, which is the XOR of A and B is typical of diode-resistor logic. For example, if A is high and B is low, then their complements (A-bar and B-bar) are low and high respectively. Thus, diodes 1, 2, 5 and 6 will be reverse biased and not conducting. Diode 8 will be forward biased, and will

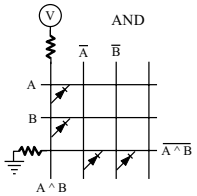


Figure 4. *An AND gate implemented in the MLA of a nanoBlock.*

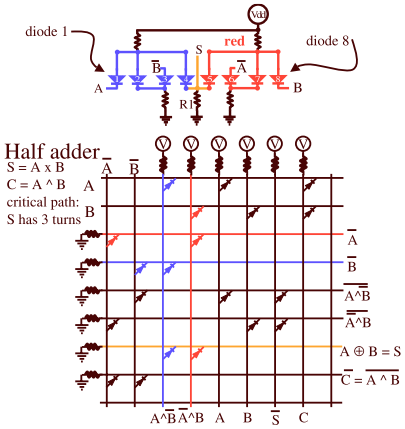


Figure 5. *A half-adder implemented in the MLA of a nanoBlock and the equivalent circuit diagram for the computation of A XOR B = S.*

pull the line labeled “red” in the figure down close to a logic low. This makes diode 4 forward biased. By manufacturing the resistors appropriately (i.e., resistors attached to Vdd have smaller impedances than those attached to Gnd) most of the voltage drop occurs across R1, resulting in S being high. If A and B are both low, then diodes 2, 4, 5 and 7 are back-biased. This isolates S from Vdd and makes it low.

The MLA computes logic functions and routes signals using diode-resistor logic. The benefit of this scheme is that we can construct it by directed assembly, but the drawback is that the signal is degraded every time it goes through a configurable switch. In order to restore signals to proper logic values without using CMOS gates, we will use the molecular latch described in Section 2.1.

Notice that all the connections between the CMOS layer and the nanoBlock occur either between groups of wires or with a wire that is removed from all the other components. This improves the device density of the fabric. To achieve a specific functionality the cross-points are configured to be either open-connections or to be diodes.

The layout of the MLA and of the switch block makes rerouting easy in the presence of faults. By examining Figure 5, one can see that a bad switch is easily avoided by swapping wires that only carry internal values. In fact, the rows can be moved anywhere within the block without affecting the circuit, which makes defect tolerance significantly easier than with CMOS[4]. The number of possible ways to arrange the columns/rows in the MLA combined with the configurable crossbar implemented by the switch block makes the entire design robust to defects in either the switch block or the MLA.

3.2 Defect Tolerance

The nanoFabric is defect tolerant because it is regular, highly configurable, fine-grained, and has a rich interconnect.[5] The regularity allows us to choose where a particular function is implemented. The configurability allows us to pick which nanowires, nanoBlocks, or parts of a nanoBlock will implement a particular circuit. The fine-grained nature of the device combined with the local nature of the interconnect reduces the impact of a defect to only a small portion of the fabric (or even a small portion of a single nanoBlock). Finally, the rich interconnect allows us to choose among many paths in implementing a circuit. Thus, with a defect map we can create working circuits on a defective fabric. Defect discovery relies on the fact that we can configure the nanoFabric to implement any circuit, which implies that we can configure the nanoFabric to test its own resources.

The key difficulty in testing the nanoFabric (or any FPGA) is that it is not possible to test the individual components in isolation. Researchers on the Teramac project [3] faced similar issues. They devised an algorithm that allowed the Teramac, in conjunction with an outside host, to test itself [12, 13]. Despite the fact that over 75% of the chips in the Teramac have defects, the Teramac is still used today. The basis of the defect mapping algorithm is to configure a set of devices to act as tester circuits. These circuits, e.g. linear-feedback shift-registers, will report a result which if correct indicates with high probability that devices they are made from are fault free.

The defect mapping process is complicated by the fact there are no known fault-free regions in the nanoFabric at the start of testing and as the defect density increases the chances of finding a good circuit decreases. To address this problem, we will implement in CMOS the components of a basic tester, which will provide the initial fault-free region. A host computer configures testers out of the CMOS testor and portions of the nanoFabric. As the host gains knowledge of the fault free regions of the nanoFabric it replaces more and more of the CMOS tester with configured sections of the nanoFabric.

[4]This is because nanowires have several orders of magnitude less resistance than switches. Thus, the timing of a block is unaffected by small permutations in the design.

[5]Defect tolerance through configuration also depends on shorts being significantly less likely to occur than stuck-open faults. We arrange for this by biasing the synthesis techniques to increase the likelihood of a stuck-open fault at the expense of potentially introducing more total faults.

After a sufficient number of functioning resources have been discovered by the host, phase two of the testing begins. For this phase, the host downloads a configuration, which performs the testing from within the fabric. In other words, the already tested area of the fabric acts as a host for testing the remainder of the fabric. Because the number of tests required to isolate any specific defect does not grow as the total size of the device grows, the computational work needed to test a device is at worst linear in the size of the device. For very large devices, such defect discovery can be accomplished using many parallel independent test machines.

Once a defect map has been generated the fabric can be used to implement arbitrary circuits. The architecture of the nanoBlock supports full utilization of the device even in the presence of a significant number of defects. Due to the way we map logic to wires and switches, only about 20% of the switches will be in use at any one time. Since the internal lines in a nanoBlock are completely interchangeable, we generally should be able to arrange the switches that need to be configured in the ON state to be on wires which avoid the defects.

While the molecules are expected to be robust over time, inevitably new defects will occur over time. Finding these defects, however, will be significantly easier than doing the original defect mapping because the unknown defect density will be very low.

3.3 Configuration

The nanoFabric uses runtime reconfiguration for defect testing and to perform its intended function. Thus, it is essential that the time to configure the fabric scale with the size of the device. There are two factors that contribute to the configuration time. The first factor is the time that it takes to download a configuration to the nanoFabric. The second factor is the time that it takes to distribute the configuration bits to the different regions of the nanoFabric. Configuration decoders are required to serialize the configuration process in each nanoBlock. To reduce the CMOS overhead, initially we intend to only configure one nanoBlock per cluster at a time. However, the fabric has been designed so that the clusters can be programmed in parallel. A very conservative estimate is that we can simultaneously configure one nanoBlock in each of 1000 clusters in parallel.

A molecular switch is configured when the voltage across the device is increased outside the normal operating range. Devices in the switch blocks can be configured directly by applying a voltage difference between the long intercluster lines. In order to achieve the densities presented above, it will also be necessary to develop a configuration approach for the switches in the MLA that is implemented with nanoscale components. In particular, a nanoscale decoder is required to address each intersection of the MLA independently. Instead of addressing accessing each nanoscale wire separately in space we address them separately in the time dimension. This slows down the configuration time, but increases the device density.

Our preliminary calculations indicate that we can load the full nanoFabric, which is comprised of 10^9 configuration bits at a density of 10^{10} configuration bits/cm^2, in less than one second. This calculation is based on realistic assumptions that, on average, fewer than 10% of the bits are set ON and that the configurations are highly compressible [19]. It also significant to note that it is not necessary to configure the full fabric for defect testing. Instead, we will configure only the portions under test.

As the configuration is loaded onto the fabric it will be used to configure the nanoBlocks. Using the configuration decoder this will require ≈ 300 cycles per nanoBlock, or less than 38K cycles per cluster. Thus, the worst-case time to configure all the clusters at a very conservative 10 MHz requires three seconds.

3.4 Putting It All Together

The nanoFabric is a reconfigurable architecture built out of CMOS and CAEN technology. The support system, i.e., power, ground, clock, and configuration wires, I/O, and basic control, is implemented in CMOS. On top of the CMOS we construct the nanoBlocks and long-lines constructed out of chemically self-assembled nanoscale components.

Assuming a 100nm CMOS process and 40nm centers with 128 blocks to a cluster and 30 long-lines per channel, our design should yield 750K clusters/cm^2 (or 1M blocks/cm^2), requiring 3^{10} configuration bits. If the nanoscale wires are on 10nm centers this design yields 1M clusters/cm^2. (If, instead of molecular latches, transistors were used for signal restoration then with 40nm centers for the nanoscale wires we obtain 180K cluster/cm^2.)

SPICE simulations show that a nanoBlock configured to act as a half-adder can operate at between 100MHz and 1GHz. Preliminary calculations show that the fabric as a whole will have a static power dissipation of ≈ 1.2watts and dynamic power consumption of $\approx .4$watts at 100Mhz.

4 The NanoFabric Model

There are two scenarios in which nanoFabrics can be used: as factory-programmable devices configured by the manufacturer to emulate a processor or other computing device, and as reconfigurable computing devices.

In a manufacturer-configured device, user applications treat the device as a fixed processor (or potentially as a small number of different processors). Processor designers will use traditional CAD tools to create designs using standard cell libraries. These designs will then be mapped to a particular chip, taking into account the chip's defects. A finished product is thus a nanoFabric chip and a ROM containing the

configuration for that chip. In this mode, the configurability of the nanoFabric is used only to accommodate a defect-prone manufacturing process. While this provides the significant benefits of reduced cost and increased densities, it ignores much of the potential in a nanoFabric. Since defect tolerance requires that a nanoFabric be reconfigurable why not exploit the reconfigurability to build application-specific processors?

Reconfigurable fabrics offer high performance and efficiency because they can implement hardware matched to each application. Further, the configurations are created at compile time, eliminating the need for complex control circuitry. Research has already shown that the ability of the compiler to examine the entire application gives a reconfigurable device efficiency advantages because it can:

- exploit all of an application's parallelism: MIMD, SIMD, instruction-level, pipeline, and bit-level.
- create customized function units.
- eliminate a significant amount of control circuitry.
- reduce memory bandwidth requirements.
- size function units to the application's natural word size.
- use partial evaluation and constant propagation to reduce the complexity of operations.

However, this extra performance comes at the cost of significant work by the compiler. A conservative estimate for the number of configurable switches in a 1cm^2 nanoFabric, including all the overhead for buffers, clock, power, etc. is on the order of 10^{11}. Even assuming that a compiler manipulates only standard cells, the complexity of mapping a circuit design to a nanoFabric will be huge, and this creates a compilation scalability problem. Traditional approaches to place-and-route in particular will not scale to devices with billions of wires and devices.

In order to exploit the advantages listed above, we propose a hierarchy of abstract machines that will hide complexity and provide an intellectual lever for compiler designers while preserving the advantages of reconfigurable fabrics. At the highest level is a split-phase abstract machines (SAM), which allows a program to be broken up into autonomous units. Each unit can be individually placed and routed and then the resulting netlist of pre-placed and routed units, can be placed. This hierarchical approach will allow the CAD tools to scale. In this paper we present simulations of programs that have been decomposed into SAM threads.

4.1 The Split-Phase Abstract Machine (SAM)

The compilation process starts by partitioning the application into a collection of threads. Each thread is a sequence of instructions ending in a split-phase operation. An operation is deemed to be a split-phase operation if it has an unpredictable latency. For example, memory references and procedure calls are all split-phase operations. Thus, each thread, similar in spirit to a Threaded Abstract Machine (TAM) thread [15], communicates with other threads asynchronously using split-phase operations. This partitioning allows the CAD tools to concentrate on mapping small isolated netlists and it has all the mechanisms required to support thread-based parallelism.

Unlike a traditional thread model, where a thread is associated with a processor when executing, each SAM thread will be a custom "processor." While it is possible for a thread to be complex and load "instructions" from its local store, the intention is that it remains fairly simple, implementing only a small piece of a procedure. This allows the threads to act either in parallel or as a series of sequential processes. It also reduces the number of timing constraints on the system, which is vital for increasing defect tolerance and decreasing compiler complexity.

The SAM model is a simplification of TAM. A SAM thread/processor is similar to a single-threaded codeblock in TAM, and memory operations in SAM are similar to memory operations in Split-C [14]. In a sense, SAM will implement (in reconfigurable hardware) Active Messages [38] for all interprocessor communications. While this model is powerful enough to support multi-threading, in this paper we use it only as a way of partitioning large sequential programs in space on a reconfigurable nanoFabric.

While SAM can support parallel computation, a parallelizing compiler is not necessary. The performance of this model rests on the ability to create custom processors. A compiler could (and in its first incarnations will) construct machines in which only one processor is active at a time. Later, as the compiler technology becomes more mature, the inherently parallel nature of the model can be exploited.

The SAM model explicitly hides many important details. For example, it neither addresses dynamic routing of messages nor allocation of stacks to the threads. Once an application has been turned into a set of cooperating SAM threads it is mapped to a more concrete architectural model which takes these issues into account. The mapping process will, when required, assign local stacks to threads, insert circuits to handle stack overflow, and create a network for routing messages with runtime computed addresses. For messages with addresses known at compile time it will route signals directly.

5 SAM Simulations

In this section we describe a limit study designed to determine the performance of applications mapped to the SAM model. Our goal was to determine how aggressive compiler technology needs to be for the nanoFabric to compete with a CMOS processor. We estimate the area and simulate the execution time of applications using the simplest SAM model, i.e. each thread executes sequentially, and we do not perform any reconfigurable computing optimizations, e.g., loops are not turned into pipelines.

In this study we analyze the behavior of programs from the SpecInt95 [35] and Mediabench [22] [6] benchmark suites. We assume no parallel execution or pipelining between independent SAM threads.

5.1 Area requirements

We define the unit of area as the area of the implementation of one memory word (4 bytes). We assume that each integer operation can be also implemented in 1 unit of area. Integer multiplication and division and floating-point operations have a substantially larger area. These assumptions are overly pessimistic for memory, which can probably be packed more densely. We make the very conservative assumption that 1 unit of area is a single cluster. In fact, a cluster will probably be able to map between 1 and 64 units.

For our benchmarks the total area was between 2,000 and 250,000 units (see Figure 7), which fits liberally in the available hardware budget. This area includes only the executed instructions and touched memory words. Dead code and unused memory is excluded from this count, but their total size is small.

5.2 Simulation Methodology

We compare the execution time of the program running native on the CPU (Alpha), and running on a nanoFabric. We include application and library running time. We ignore time spent in the operating system. The simulation consists of two phases: trace collection and analysis, and trace-based simulation.

5.3 Trace Collection and Analysis

Each application to be simulated is compiled using `gcc` and optimized with `-O4` on the Alpha. The ATOM tool [34] is used to instrument the program for trace collection and summarization. We instrument each basic block, procedure call and memory access.

From the trace data we create a weighted undirected graph (with weights both on nodes and edges). Each node represents either a SAM thread or a memory location and its weight is the estimation of the area it requires on the nanoFabric. The weight of an edge represents the total data traffic carried along that edge. Each basic block and each memory address becomes a node. We eliminate some of aliasing of the memory locations by assuming that the stack frame of each function is a different memory region.

An edge between two basic blocks is weighted with the number of data values transferred across that edge (i.e. the number of register values defined in the first block and used in the second one)[7]. Transfers of control resulting from procedure calls are weighted with the total size of the procedure arguments. An edge between a basic block node and a memory node is weighted with the number of accesses made from the block to that address.

This graph is next placed on a two-dimensional grid. We strive to place nodes connected by heavy edges together. Assuming that the signal propagation time between two nodes is proportional to their distance, such a placement will minimize signal latency. We do the placement in two stages: clustering and placement.

- In the first stage nodes are clustered together into supernodes of approximately the same weight (the weight of a super-node being the sum the weights of all the component nodes). Each supernode is 100 units of area. The clustering is done so to minimize the total edge weight between the super-nodes. For this purpose we use the METIS [20] graph partitioning tool.
- In the second stage we place the super-nodes on a two-dimensional grid. We assume that each super-node is a 10x10 square. We next repeatedly "glue" pairs of super-nodes (obtaining progressively larger rectangles and squares: 10x10, 10x20, 20x20, 20x40, 40x40, etc). The ends of the heavier edges are coalesced first. For each pair of nodes we choose the relative orientation that minimizes the total edge cost (i.e. the nodes can be flipped or rotated).

5.4 Trace-based simulation

The second phase of the simulation runs the ATOM-instrumented binary, generating a trace of all important events: control transfer between blocks and memory read operations. We use the same input data that we used for the first phase.

We use this trace to evaluate the running time of the program when implemented as a circuit whose layout is the one computed by our placement algorithm. The running time of a node contains the following components:

- The time to pass control between the source to the destination. This is proportional to the Manhattan distance between the two super-nodes where these nodes belong.
- The time to transfer data from the source node to the destination. The registers which are live at the start of the node and used within are assumed to come from the previous node. We assume that the data is pipelined after the message which transfers control from the source, and each additional register value takes one clock cycle.
- The time to execute the node itself. We assume full hardware ILP (i.e. all independent instructions are executed in parallel). The time to execute the node is thus the time to execute the critical path. The critical path may depend on

[6] The other programs from these suites did not work properly with our simulation infrastructure; we present all the applications we could successfully instrument and analyze.

[7] For a faster calculation, we approximate this value with the number of registers used in the second basic block.

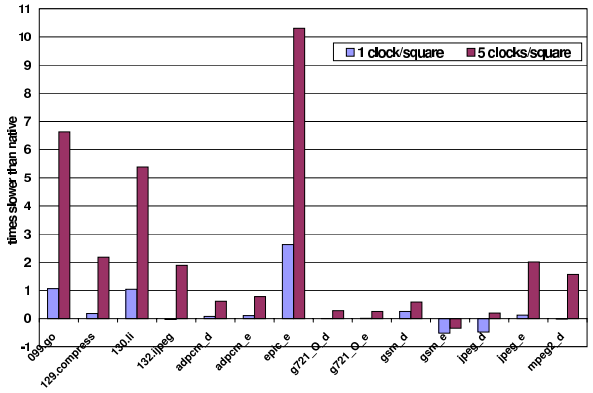


Figure 6. *Simulated slowdown (*simulated/base−1*).*

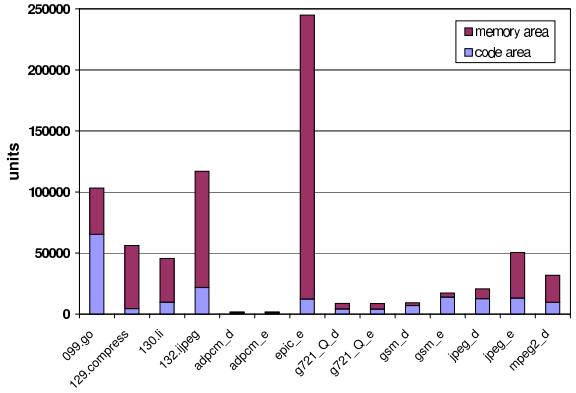


Figure 7. *Total area for instructions and memory.*

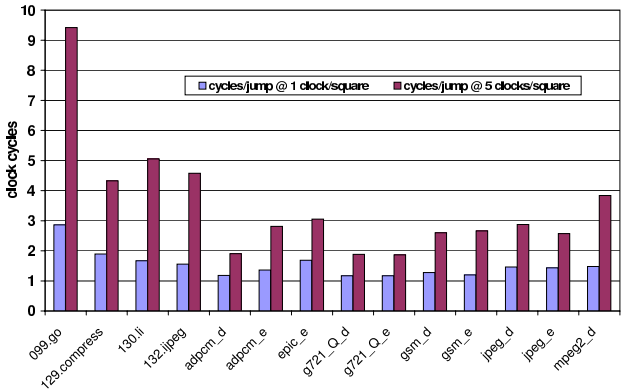


Figure 8. *Average clock cycles per control transfer.*

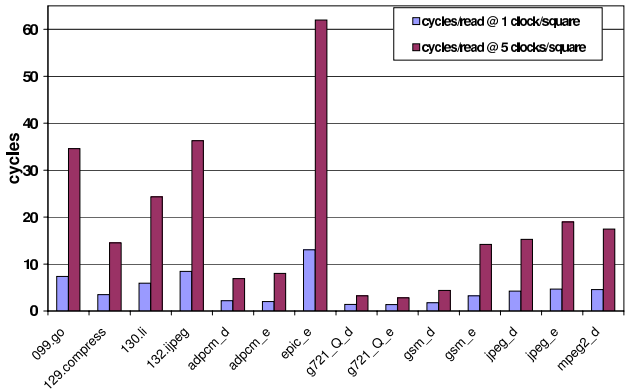


Figure 9. *Average clock cycles per memory read.*

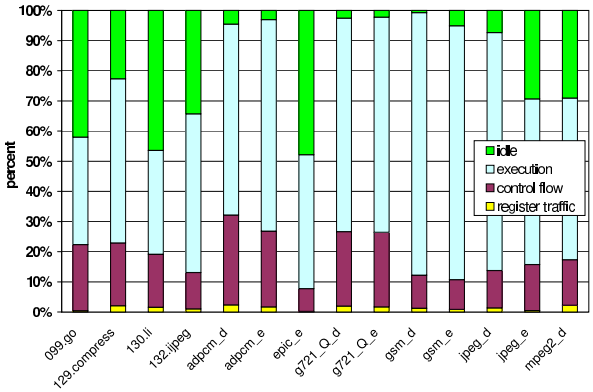


Figure 10. *Breakdown of execution time for 1 clock/square. Idle time is the time spent waiting for memory read operations to complete.*

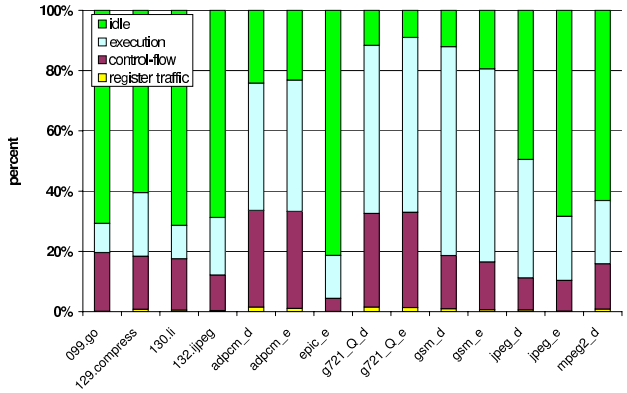


Figure 11. *Breakdown of execution time for 5 clocks/square.*

dynamic information, like the memory addresses that are accessed.

We use the following model for the execution of an instruction:

- Each elementary instruction takes one clock cycle.
- Each read operation takes time proportional to the Manhattan distance to the memory address which is accessed (the signal needs to propagate both ways: one direction with the read address, and the opposite direction with the read data).
- Each write instruction takes one cycle and is completed asynchronously. We assume ordered message delivery. All reads to the same address should be executed after the write completes, even if they occur in different nodes.
- For floating-point operations we use the double of the latency of the same instruction as executed on Alpha.

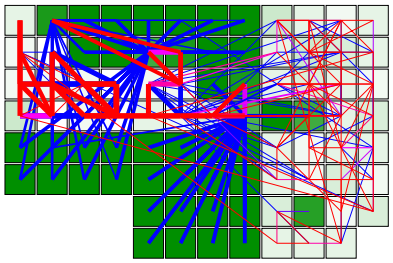


Figure 12. *The placed graph for* `g721_e`.

5.5 Comments

In some respects our simulation methodology is overly optimistic, in others it is pessimistic. Our goal here is to validate the assumption that we can map an application completely to hardware. In practice, more aggressive compilation and resource sharing would change the resulting configuration substantially. Here, we point out some of the assumptions which will need to be revisited in a more realistic study.

Optimistic assumptions:

- The placed graph depends on the input data. In general we cannot know what locations and nodes will be used.
- The "inputs" (registers live on entry) to a node do not necessarily come from its immediate predecessor.
- We assume that control and data transfer can be done directly between the two nodes involved.
- We assume that each procedure has a statically allocated stack frame (This cannot be true for recursive procedures).
- We underestimate the area by not disambiguating the aliasing caused by the re-allocation of de-allocated memory regions.
- We completely ignore routability issues between nodes. We assume sufficient wire bandwidth is available.
- We ignore the propagation delay introduced by the molecular latches.

Pessimistic assumptions:

- We "issue" a memory read operation only when the read instruction is executed. Reads could be initiated as soon as the address is known.
- No effort is made to customize the circuit for the application. For example, many of these applications have kernels which, when compiled properly, show speedup on the order of 100x over a conventional processor [18].
- We do not do any speculative or parallel execution.
- Nodes are strict, having to wait for their inputs before starting execution.
- By not disambiguating between the uses of a memory word which is freed and re-allocated we unnecessarily constrain the placement of the graph.
- The code we execute had its registers allocated for the Alpha, which has many fewer registers than would be available on the nanoFabric. Spilled registers become unnecessary costly memory operations.
- We allocate a lot of area for each memory cell.

5.6 Results

Figure 7 shows that the hardware area used for each program, including memory. The unit of memory is the area taken by a simple integer instruction. We notice that all the programs will fit within the hardware resources of a 1cm^2 nanoFabric.

Before discussing other aspects of the performance of our proposed implementation, we should notice some characteristics of the program graph: although it is sparse (the average node degree is less than 10), the node degree distribution is very skewed (each graph has a few large "stars"). Figure 12 displays one of the smallest graphs, where each square is a super-node, obtained after clustering (for readability). The shading of the squares indicates how many of the objects inside are instructions. A white square contains only code. The edges show communication patterns. The edge width is the logarithm of the number of messages sent across the edge. The edge color indicates the mix of types of messages: dark edge indicates memory reads only, while lighter edges indicates control transfers, with intermediate shadings for edges which carry mixed traffic. Despite the graph being very small, it exhibits some typical features for all our programs, like the big "stars": code regions which touch most of the memory of the program. "Stars" are bad, because there is no way to place all adjacent nodes close to the star's center node; some have to be remote. "Hot" memory locations, which are touched by a lot of basic blocks, are less common.

For example, the `memcpy` standard library function was the bottleneck in several programs, because it would access most of the memory used by the program. To reduce the size of the large "stars" we have inlined such functions at all their call sites, in effect duplicating the code. Each inlined copy of `memcpy` might service a different set of memory locations, to which it can be placed closer. This significantly improves in performance (our figures show the performance after inlining has been done). The area cost of inlining is negligible (less than 1%).

Inlining was ineffective or inapplicable for functions which were not leaves of the call graph, or which were called from a single place.

Figures 6, 8, and 9 explore the impact of the signal propagation speed on the application performance: the light bar is drawn assuming the signals take one clock cycle between two adjacent super-nodes, while the dark bar is for a five clock cycles/super-node latency.

Figure 6 shows the slowdown of our applications. Negative values indicate speed-ups. The media applications fare better than the SpecInt ones as they tend to have a smaller

memory footprint. Figures 8 and 9 show the impact of the propagation delay on the cost of the control flow transfer and of a memory read respectively. Because memory accesses can be executed in parallel to each other or to execution of the code, some of the memory latency is hidden. The cost of control-flow transfers scales better (slower) with increased propagation delay because the locality at the level of the code is much better. There are few "hot" nodes, and they can be placed very close to each other, often within the same super-node.

Figures 10 and 11 show how the running time of each application was spent for the two signal-propagation speeds. We notice that most often the dominant cost, especially for large signal propagation latencies, is in reading memory. The lack of caches makes remote memory operations costly.

6 Research directions

Some form of memory data caching is crucial for reducing the dominant runtime cost, memory access. Hardware and software solutions will have to be devised for implementing distributed caches.

Besides the simple inlining strategy which we used to reduce the size of the "stars" in the graph, we can imagine several other solutions, like making many copies of the function code and randomly calling one of them each time, or inlining non-leaf functions.

Further code restructuring may be necessary in order to better localize memory accesses. We could improve data placement by using separate memory pools for each type of object.

The `memcpy` function can be optimized by a special implementation, as a three-party transaction. Instead of copying the data from source to the code block and then to the destination, it could directly ship the data from the sources to the destination memory nodes.

Predicated execution, speculative execution and code duplication (for better placement locality) would reduce the cost of control flow transfers.

Some of the techniques we propose to better handle hot spots will introduce further complications: replicating memory and using speculative execution would invalidate our assumption that consecutive operations to memory reach the memory location in the order they were issued (the triangle inequality can no longer be assumed). Special distributed synchronization mechanisms will have to be devised to preserve the program semantics.

7 Conclusions

In this paper we propose a new architecture, the nanoFabric, based on chemically assembled electronic nanotechnology. The nanoFabric is designed to harness the smallness of electronic nanotechnology and to overcome some of the most onerous limitations introduced by chemical assembly. It eliminates the need for transistors, the neccesity of precise alignment and placement of wires, and provides for defect tolerance. In conjunction with CMOS support circuitry it will create a reconfigurable fabric with more than 10^{10} gate equivalents/cm^2.

The main computing element in the nanoFabric is a molecular-based reconfigurable switch. We exploit the reconfigurable nature of the nanoFabric to provide defect tolerance and to support reconfigurable computing. Reconfigurable computing offers the promise of not only increased performance, but ammortizes the cost of chip manufacturing across many users by allowing circuits to be configured post-fabrication. The molecular-based switch eliminates much of the overhead needed to support reconfiguration—the switch holds its own state and can be programmed without extra wires, making nanoFabrics ideal for reconfigurable computing. As the nanoFabric is a combination of two technologies, CMOS with CAEN, it suggests a hybrid architecture that may combine silicon-based custom circuits with CAEN-based reconfigurable ones.

To support defect tolerance, ease of placement and routing constraints, and enable faster compilation, we propose a new architectural model, the split-phase abstract machine (SAM). SAM ensures that all operations of potentially unknown latency use split-phase operations. The compilation strategy engendered by SAM is to partition an application into independent threads at all split-phase boundaries. This partitioning will allow compilers and CAD tools to handle the large designs than can fit on a nanoFabric. While the model is powerful enough to express parallel execution, we perform a study which limits the model to sequential execution, and uses only simple scalar compiler optimizations. Our simple compilation strategy produces results whose average performance is within a factor of 2.5 of the performance of an Alpha microprocessor, under the most pessimistic delay assumptions for our fabric. We uncover several performance bottlenecks which hint at future research avenues which will make the nanoFabric a viable substrate.

Acknowledgments

This work was sponsored in part by DARPA, under the Moletronics Program, by NSF, under a CAREER award, and Hewlett-Packard Corporation. The authors wish to thank the many reviewers for their helpful comments.

References

[1] I. Aleksander and R. Scarr. Tunnel devices as switching elements. *Journal British IRE*, 23(43):177–192, March 1962.

[2] Altera Corporation. Apex device family. `http://-www.altera.com/html/products/apex.html`.

[3] R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider. Teramac–configurable custom computing. In

D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 32–38, Napa, CA, Apr. 1995.

[4] A. Aviram and M. Ratner. Molecular rectifiers. *Chemical Physics Letters*, 29(2):277–283, Nov. 1974.

[5] V. Betz and J. Rose. Vpr: A new packing, placement and routing tool for fpga research. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*, Aug. 1997.

[6] F. Buot. Mesoscopic phyics and nanoelectronics: Nanoscience and nanotechnology. *Physics Reports*, pages 173–74, 1993.

[7] J. Chen, M. A. Reed, A. M. Rawlett, and J. M. Tour. Observation of a large on-off ratio and negative differential resistance in an electronic molecular switch. *Science*, 286:1550–2, 1999.

[8] J. Chen, W. Wang, M. A. Reed, M. Rawlett, D. W. Price, and J. M. Tour. Room-temperature negative differential resistance in nanoscale molecular junctions. *Appl. Phys. Lett.*, 77:1224, 2000.

[9] R. H. Chen, A. N. Korotov, and K. K. Likharev. Single-electron transistor logic. *Appl. Phys. Lett.*, 68:1954, 1996.

[10] C. Collier, E. W. Wong, M. Belohradsky, F. M. Raymo, J. F. Stoddart, P. J. Kuekes, R. S. Williams, and J. R. Heath. Electronically configurable molecular-based logic gates. *Science*, 285:391–3, July 1999.

[11] Y. Cui and C. Lieber. Functional nanoscale electronic devices assembled using silicon nanowire building blocks. *Science*, 291:851–853, 2001.

[12] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. The teramac custom computer: Extending the limits with defect. In *Proc. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 1996.

[13] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. Defect tolerance on the teramac custom computer. In *Proceedings of the 1997 IEEE Symposium on FPGAs for Custom Computing Machines*, pages 116–124, April 1997.

[14] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Proceedings of the Supercomputing '93 Conference*, Nov. 1993.

[15] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[16] A. N. et al. Room temperature operation of si single-electron memory with self-aligned floating dot gate. *Appl. Phys. Lett*, 70:1742, 1997.

[17] R. Feynman. *Lectures in Computation*. Addison-Wesley, 1996.

[18] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer. Piperench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, May 1999.

[19] S. Hauck, Z. Li, and E. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. In *IEEE Trans. on CAD of IC and Systems*, volume 18,8, pages 1107–13, August 1999.

[20] G. Karypis and V. Kumar. Multilevel graph partitioning and sparse matrix ordering. In *Proceedings of the 1995 Intl. Conference on Parallel Processing*, 1995.

[21] R. W. Keyes. Miniaturization of electronics and its limits. *IBM Journal of Research and Development*, 32(1):24–48, Jan 1988.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[23] C. Lent. A device architecture for computing with quantum dots. *Proceedings of the IEEE*, 85, April 1997.

[24] T. Mallouk. Nanomaterials: Synthesis and assembly. `http://research.chem.psu.edu/mallouk/-nano.pdf`, Nov. 2000. Foresight Conference Tutorial.

[25] B. Martin, D. Dermody, B. Reiss, M. Fang, L. Lyon, M. Natan, and T. Mallouk. Orthogonal self assembly on colloidal gold-platinum nanorods. *Advanced Materials*, 11:1021–25, 1999.

[26] R. Mathews, J. Sage, T. Sollner, S. Calawa, C. Chen, L. Mahoney, P. Maki, and K. Molvar. A new rtd-fet logic family. *Proceedings of the IEEE*, 87(4):596, 1999.

[27] J. Mbindyo, B. Reiss, B. Martin, B. Reiss, M. Keating, M. Natan, and T. Mallouk. Dna-directed assembly of gold nanowires on complementary surfaces. *Advanced Materials*, 2000.

[28] N. S. M.V. Martinez-Diaz and J. Stoddart. The self-assembly of a switchable [2]rotaxane. *Angewandte Chemie International Edition English*, 36:1904, 1997.

[29] H. Park, A. Lim, J. Park, A. Alivisatos, and P. McEuen. Fabrication of metallic electrodes with nanometer separation by electromigration. *www.physics.berkeley.edu/-research/mceuen/topics/nanocrystal/EMPaper.pdf*, 1999.

[30] M. A. Reed. Molecular-scale electronics. *Proceedings of the IEEE*, 87(4), April 1999.

[31] D. Rosewater and S. Goldstein. What makes a good molecular computing device? Technical Report CMU-CS-01-114, Carnegie Mellon University, April 2001.

[32] T. Rueckes, K. Kim., E. Joselevich., G. Tseng, C.-L. Cheung, and C. Lieber. Carbon nanotube-based nonvolatile random access memory for molecular computing. *Science*, 289:94–97, 2000.

[33] J. R. S. Brown, R. Francis and Z. Vranesic. *Field-Programmable Gate Arrays*. Kluwer, 1992.

[34] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. Technical report, Digital Equipment Corporation Western Research Laboratory, 1994.

[35] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[36] S. Tans and et al. Individual single-wall carbon nanotubes as quantum wires. *Nature*, 386(6624):474–7, April 1997.

[37] R. Turton. *The Quantum Dot: A journey into the Future of Microelectronics*. Oxford University Press, U.K., 1995.

[38] T. von Eicken, D. E. Culler, S. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[39] Xilinx Corporation. Virtex series fpgas. `http://www.xilinx.com/products/-virtex.htm`.

[40] A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.