Reconfigurable Computing and Electronic Nanotechnology

Seth Goldstein, Mihai Budiu, Mahim Mishra, and Girish Venkataramani
Carnegie Mellon University
{seth,mihaib,mahim,girish}@cs.cmu.edu

Abstract

In this paper we examine the opportunities brought about by recent progress in electronic nanotechnology and describe the methods needed to harness them for building a new computer architecture. In this process we decompose some traditional abstractions, such as the transistor, into fine-grain pieces, such as signal restoration and input-output isolation. We also show how we can forgo the extreme reliability of CMOS circuits for low-cost chemical self-assembly at the expense of large manufacturing defect densities. We discuss advanced testing methods which can be used to recover perfect functionality from unreliable parts. We proceed to show how the molecular switch, the regularity of the circuits created by self-assembly and the high defect densities logically require the use of reconfigurable hardware as a basic building block for hardware design. We then capitalize on the convergence of compilation and hardware synthesis (which takes place when programming reconfigurable hardware) to propose the complete elimination of the instruction-set architecture from the system architecture, and the synthesis of asynchronous dataflow machines directly from high-level programming languages, such as C. We discuss in some detail a scalable compilation system that perform this task.

1: Introduction

Moore's law has been equated with a guaranteed stream of good news, bringing ever higher clock speeds and more hardware resources in each new hardware generation. Microarchitects have been significant beneficiaries, building faster processors and using more resources to exploit program parallelism. However, significant difficulties loom for the traditional approach to processor design.

Traditional microarchitectures are monolithic in nature, as their component structures are tightly dependent on each other. Such designs are not scalable with the increasing amount of resources, and are already stretched to their limits. Approaches that aim at decoupling the components, but remain tied to previous architectural structures, introduce overhead and additional complexity.

Recent advances in molecular electronics, combined with increased challenges in semiconductor manufacturing, create a new opportunity for computer architects — the opportunity to recreate computer architecture from the ground up. New device characteristics require us to rethink the basic abstraction of the transistor. New fabrication methods require us to rethink the basic circuit abstraction. The scale of the devices and wires allows us to rethink our basic approach to designing computing systems. On the one hand, the scale enables huge computing systems with billions of components. On the other hand, the scale forces us to rethink the meaning of a working system; it must be a reliable system made from unreliable components.

Computer architecture builds computing systems as hierarchies of abstractions. Molecular computing may be a case where a reexamination of the layers of abstraction is required. In this paper we propose an alternative computer system architecture, based on dramatically different abstractions:

- Transistor $\rightarrow$ new molecular devices
- Custom hardware $\rightarrow$ generic reconfigurable hardware
- Yield control $\rightarrow$ defect tolerance through reconfiguration

- Synchronous circuits → asynchronous computation
- Microprocessors → hybrid CPU + compiler-synthesized application-specific hardware

Successfully harnessing the power of molecular computing requires us to rethink several key abstractions: the transistor, the circuit, and the ISA. The abstraction of the transistor should be broken down into more basic components, such as controlled signal switching, signal restoration and input/output isolation. The abstraction that a circuit is created at manufacturing time needs to be replaced by the ability to configure circuits at run-time. Finally, the abstraction of an ISA needs to be replaced with a more flexible hardware/software interface: instead of an ISA which hides the details of a processor implementation from the user, the new abstraction will expose more of the underlying computational structure to the compiler, allowing better use of the plentiful resources that become available through molecular computing.

In the rest of this paper we discuss briefly each of these new abstractions.

2: Nanotechnology and Molecular Circuits

Significant progress has been made in developing molecular scale devices. Molecular scale FETs, negative differential resistors, diodes, and non-volatile switches are among the many devices that have been demonstrated. Advances have also been made in assembling devices and wires into larger circuits. Estimated densities are between 10x and 1000x greater than those predicted by the ITRS 2001 road-map for silicon-based CMOS circuits [8]. In addition to the increases in density, molecular electronics also promises to introduce devices with characteristics not found in silicon-based systems. One example is the non-volatile programmable switch, which holds its own state without using a memory cell, and can be configured using the signal wires; such a switch has the same area as a wire-crossing.

A combination of the restrictions of self-assembly and the small feature size of the devices reduces the likelihood that near-ideal transistors will be available to circuit designers—in fact, three-terminal devices may be completely impractical. Thus, circuit designers will have to use devices that do not each by themselves provide all the features of an ideal transistor. Transistors simultaneously provide the capability of a switch, isolate inputs from outputs and and restore (amplify) logical signals. New devices may only provide some of these features. Thus, we propose the **SirM** device model, which explicitly separates the components performing **s**withing, **i**solation, **r**estoration, and state storage (**m**emory) [10]. The advantage of this model is that it will make explicit the fact that there are different kinds of devices which serve different roles in a circuit. It will be possible to design a switching network in which signals will degrade and possible interfere with each other. Then between switching networks the designer can insert the proper kinds of devices for signal restoration and input/output isolation.

Perhaps the biggest difference between molecular electronics and VLSI is in fabrication technology. Large-scale molecular electronics requires some form of *self-assembly*. The use of self-assembly requires not only a different manufacturing process, but requires rethinking every aspect of system design, including, but not limited to: circuit design, architecture, compilation, and testing. When using self-assembly, individual devices and wires are first manufactured, and only later assembled into a circuit. While self-assembly is a very economical process (compared with the cost of today's silicon fabrication equipment), it cannot build patterns of the practically arbitrary complexity that optical lithography creates. Only simple, crystal-like structures, can be created using self-assembly. Furthermore, defect densities of self-assembled circuits are orders of magnitude higher then in silicon-based devices. Thus, self-assembled circuits and architectures will have to be designed for defect tolerance.

In order to implement useful reliable functionality on top of crystal-like structures, post-fabrication customization is required; this customization will be used for two purposes (1) to implement the desired functionality and (2) to eliminate the deleterious effects of the defects. Thus, we expect that electronic nanotechnology architectures will be *highly reconfigurable*. A reconfigurable architecture allows one to configure a hardware system to implement a particular circuit. Reconfiguration can avoid using the defective components, thus creating a reliable system on an unreliable substrate.

One such reconfigurable architecture is the nanoFabric [11], which is designed to overcome the limitations of self-assembly of molecular scale components. The basic unit of the nanoFabric is the programmable cross-

point. These cross-points are organized into 2-D meshes which themselves are organized into clusters. The fabric is a fine-grained reconfigurable device which can implement any desired circuit by programming the cross-points.

3: Reconfigurable Hardware

In the last few years there has been a convergence in molecular-scale architectures towards reconfigurable platforms. In our proposed architecture, the nanoFabric, the main computing element is a molecular-based reconfigurable switch. The molecular-based switch eliminates much of the overhead needed to support reconfiguration in traditional CMOS circuits, since the switch holds its own state and can be programmed without extra wires. We exploit the reconfigurable nature of the nanoFabric to provide defect tolerance and to support reconfigurable computing. Reconfigurable computing not only offers the promise of increased performance but it also amortizes the cost of chip manufacturing across many users by allowing circuits to be configured after they are fabricated.

Reconfigurable Hardware shares features of both custom hardware and microprocessors. Its computational performance is close to custom hardware, yet, because it is programmable, its flexibility approaches that of a processor. We are not the first to recognize that RH devices have an enormous potential as computational elements and there has been much research into using RH for computing (see [12] for an overview.)

Several features of RH devices differentiate them fundamentally from processors, and are responsible for the extreme performance of some computing systems built using such technologies:

1. **Unbounded computational bandwidth:** a microprocessor is designed with a specific number of functional units. The *computational bandwidth* of a processor is thus bounded at the time of manufacturing. Moreover, it is unusual for a processor to reach its peak performance, because the parallelism available in the program rarely has the exact same profile as the available functional units.
   In contrast, RH can support a virtually unbounded number of functional units. Not only can highly parallel computational engines be built, they can exactly fit the application requirements, since the configuration is created post-fabrication.
2. **Unlimited register bandwidth:** another subtle but important difference between a processor and an RH device is in the way they handle intermediate computation results: processors have a predetermined number of registers. If the number of manipulated values exceeds the number of registers, then they have to be spilled into memory. Additionally, the fixed number of internal registers can throttle parallelism.
   Even more limiting is the number of register ports: in a processor the bandwidth in and out of the register file is bounded by a hard limit. In contrast, datapaths on RH directly connect the producers of a result to its consumers. If there is a register needed, it is inserted directly into the datapath. In other words, there is no monolithic register file, no register port limit, and no need to spill values to memory.
3. **Full out-of-order execution:** while superscalar processors allow instructions to execute in orders different from the one indicated by the program, the opportunity to do so is actually restricted by several factors, such as limited issue window, generic exception handling and structural hazards. None of these constraints exists in RH implementations.

Other often-cited advantages of RH are the abilities to:

4. exploit all of an application's parallelism: task-based, data, instruction-level, pipeline, and bit-level,
5. create customized function units and data-paths, matching the application's natural data size,
6. eliminate a significant amount of control circuitry.

3.1: Using nanoFabrics

There are two scenarios in which nanoFabrics can be used: (1) as factory-programmable devices configured by the manufacturer to emulate a processor or other computing device, and (2) as reconfigurable computing devices.

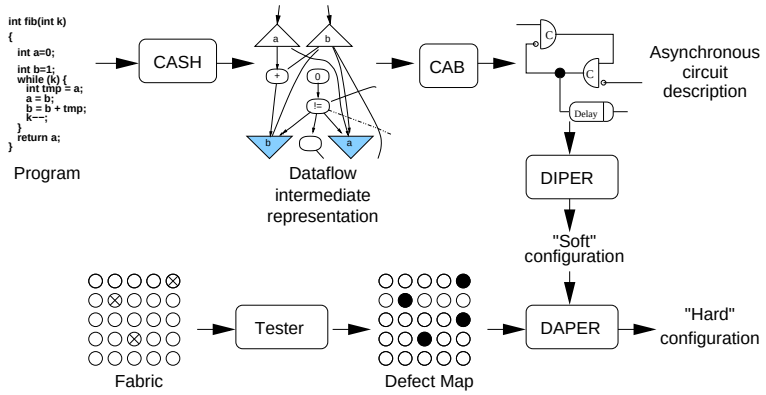

Figure 1. The tool-flow for using molecular reconfigurable fabrics for computation.

(1) 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 therefore a nanoFabric chip and a ROM containing the configuration for that chip. In this scenario, 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?

(2) Reconfigurable fabrics offer high performance and efficiency because they can implement hardware matched to each application. 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, creating 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 machine (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.

4: Configuring the nanoFabric

Using nanoFabrics requires a new manufacturing paradigm, one which trades-off complexity at manufacturing time with post-fabrication programming. The reduction in manufacturing time complexity makes reconfigurable fabrics a particularly attractive architecture for molecular circuits, since directed self-assembly will most easily result in highly regular, homogeneous structures.

Effectively utilizing the abundant fabric resources and dealing with high defect densities will require a new set of tools. This set includes fast testers to generate defect maps, compilers to convert high-level designs into low-level circuit descriptions, and place-and-route tools to convert circuit descriptions into fabric configurations taking into account the defect map for the fabric. Figure 1 depicts how these tools interoperate.

Fabric testers find and record defect locations. Defects can be tolerated by configuring circuits around them. A testing step first generates a defect map, much like the maps of defective blocks used for hard disk drives. This map may be a listing of all the defects in the fabric, or may be coarser-grained information such as a count of the number of defects in each portion of the fabric. Section 5 describes our approach to testing.

Compiler: The compiler takes a high-level design description and compiles it down to a circuit description. Since creating the functionality of the circuit has been moved from manufacturing time to configuration time, the compilation encompasses tasks traditionally assigned to software compilers and to CAD tools. The challenge of building a single tool spanning both domains presents us with the opportunity of reformulating the traditional interface between these two domains: the Instruction Set Architecture. Removing this rigid interface exposes a wealth of information to the compiler enabling novel classes of optimizations. The disadvantage of this approach is the complexity between the source and target of the compilation process.

We propose the use of traditional, high-level software programming languages for programming nanoFabrics (the traditional approach to programming RH uses hardware-description languages). Our compiler, *CASH*, further described in Section 6.2, compiles ANSI C programs into collections of dataflow machines. The resulting dataflow machines are highly *composable*: they have simple interfaces using simple, timing-independent protocols. The natural substrate for implementing these dataflow machines are asynchronous (or globally-asynchronous, locally-synchronous) circuits. CASH has an asynchronous circuit back-end, further described in Section 7.

Place-and-route: The place-and-route process generates fabric configurations that avoid using defective components. We propose a two-step process to deal with the fabric complexity and the one-of-a-kind nature of each fabric due to defects:

1. An initial, fabric- and defect-independent step (*DIPER*, or Defect-Unaware Place and Route) which includes technology mapping and some approximate placement and routing, and generates what we call a "soft" configuration. The "soft" configuration is guaranteed to be place-and-routable on a fabric that has a defect level below a certain threshold.
2. A final, defect-aware step (*DAPER*, or Defect Aware Place and Route) that transforms the "soft" configuration into a "hard" configuration, taking into account the target fabric's defect map. At this step the final place-and-route is performed, using only non-defective resources. (If the defect-map is imprecise, this step requires the use of an on-the-fly defect mapper to pin-point the precise location of defects). The hard configuration is specific to each fabric.

5: Testing to Locate Defects

Functional circuits require knowledge of the locations of all the defects in the reconfigurable fabric. The scale of the fabric and the very high defect densities require the development of a new testing methodology.

The main challenge in testing the nanoFabric is that it is not possible to test individual components in isolation, due to two reasons: first, the extremely small feature sizes make it impossible to probe individual or small groups of components with external probes, and any such access would require on-fabric wire and switch resources which may themselves be defective; and, second, sequentially testing each device would be extremely slow due to the sheer size of the fabric. Researchers on the Teramac project [1] faced similar issues. They devised a method that essentially allowed the Teramac to test itself [9]. This enabled the Teramac to work in spite of over 75% of its chips being defective.

The defect mapping algorithm configures a set of devices to act as tester circuits. These circuits (e.g., linear-feedback shift-registers) report a result which, if correct, indicates with high probability that all their constituent devices are defect-free. Since the reconfigurable resources are themselves configured into test-circuits, no extra on-fabric structures are required to facilitate testing and testing incurs no area and delay overhead in the fabric.

The testing method used for Teramac is not directly applicable for molecular devices, because of the significantly higher defect rates. To tolerate these higher defect rates, we propose a testing algorithm that is split into two phases [15]:

- A probabilistic *probability-assignment* (*p-a*) phase that assigns to each fabric component a probability of being defective. Those found defective with very high probability are removed from further consideration.
- A deterministic *defect-location* (*d-l*) phase, which tests the remaining components to determine with certitude which ones are defect-free.

This two-step process is conservative: no bad components are marked as good. The *p-a* phase makes use of two key ideas:

Powerful test circuits: the circuits used for testing the Teramac give yes/no answers — a correct circuit result implied the components used for that circuit are defect-free, while an incorrect result means there is one or more defects; we call such circuits *binary* circuits. Binary circuits rely on the fact that most of the test circuits will actually have no defects at all; this condition does not hold true when the defect rate is high. We propose using circuits that can give more than a binary qualification. In particular, we have explored using two kinds of circuits:

- *counter* circuits which can count the actual number of defects in their components. These circuits are hard to implement, but can be used to derive upper bounds on the testing effectiveness.
- *none-some-many* circuits, which can tell us if the circuit had none, some, or, many defects. In particular we have a linear-feedback shift register based circuit design that shows promising results in simulations.

Powerful analysis techniques: the outputs of the test circuits are analyzed to obtain defect probabilities for individual components. We have considered a *sorting* analysis and a *Bayesian* analysis. The former is easier to implement and quick, but produces results that are inferior (i.e., more false negatives).

Once the the *p-a* phase has eliminated components with high-defect-probability, the *d-l* phase uses binary circuits to pin-point defect-free components and eliminate errors introduced by the probabilistic *p-a* phase. These binary circuits can now be used because a significant proportion of the defective components are eliminated in the *p-a* phase. In particular, binary circuits give good results if the *p-a* phase eliminates enough defects that there is on average less than one defect per test circuit in the *d-l* phase.

Our defect-mapping algorithm shows promising results in simulations with defect rates of 1 to 15%. The quality metric we use is the fraction of defect-free components identified as defect-free by the algorithm. As expected, *counter* circuits outperform *none-some-many* circuits in general, and *Bayesian* analysis outperforms the simpler *sorting* analysis. All combinations of these techniques significantly outperform naive testing methods that only use binary circuits, or that do not include the probabilistic *p-a* phase. The number of required tests scales as the square root of the number of components being tested [15]. While the details of the testing method, and in particular the test-circuits used, will have to undergo significant evolution as information about devices, defect types and fabric architectures materializes, our initial work has shown that the defect-mapping problem can be tackled with good results, and with better-than-linear scaling in the amount of resources being tested.

6: Compilation

We expect future computing systems to be hybrid systems, composed of a general-purpose processor tightly coupled with a reconfigurable fabric. The processor provides excellent virtualization and will most likely be used for executing the operating system, device interactions and application code with limited parallelism. The RH will be used for compute-intensive kernels, which can take advantage of its wealth of resources.

6.1: Application-Specific Hardware (ASH)

There have been many proposals made for implementing hybrid CPU+RH computer systems. We take a fundamentally different approach based on the assumption that the RH will be a significant portion of the total resources. This change in assumptions leads to significant differences from previous research:

1. We assign an equally important role to both computing engines: the CPU and the reconfigurable fabric. Instead of a traditional master/slave relationship, they are peers; for example, the RH can invoke code residing on the CPU.

2. Our solution is completely automatic and compiler-driven. We present below an overview of a complete compilation framework for program analysis, partitioning and circuit synthesis. The compiler maps complex pointer-based applications onto the hybrid system and automatically exposes a significant amount of instruction, pipeline, and loop-level parallelism through the use of predicated execution.
3. Our execution model is heavily influenced by dataflow architectures. It is designed to tolerate unpredictable latencies through the use of dynamic scheduling. It scales for large fabrics because, by relying on an asynchronous model of computation, it avoids the need for low-skew clocks.

Our design and implementation goals are motivated by significant practical concerns. We strive to ensure that it is general, programmable, scalable, and efficient:

General The model supports a general model of computation. It can efficiently execute a wide variety of applications and does not impose unreasonable constraints on the programming model. In fact, the framework should easily accommodate any high-level programming languages available today, e.g. sequential (non-parallel), imperative, pointer-based languages. Our current compiler translates C to circuits.

Automatic Programming for an ASH architecture should be no different than programming a processor-based architecture. That is, the compiler should handle all peculiar architectural details, without programmer involvement. Ideally, the programming experience is exactly like traditional programming (and not, for example, like hardware design). We draw from the RISC experience and make our architecture compiler-centric: the hardware only offers features that can successfully be exploited by the compiler.

Scalable ASH is designed to scale with computational resources and clock frequencies. Furthermore, the compiler algorithms we present are scalable, i.e., they do not rely on excessively complex or expensive whole-program analysis methods.

Parallel ASH efficiently exploits the parallelism available in programs (pipeline, instruction level, data, bit-level, and loop-level parallelism).

6.2: Compiling for ASH

In this section we present an overview of the CASH compiler (Compiler for ASH), which translates C programs into hardware.

Hardware-Software Partitioning The compiler builds a *procedural interface* between the code running on the processor and the code implemented directly in hardware [6]. The application program is partitioned at procedure boundaries; each side invokes the services of the other side using a form of remote procedure call. The partitioner can use multiple sources of information: (1) a conservative estimation of the program call-graph, (2) the results of a width analysis [7], which estimates the required hardware resources for implementing each operation, (3) dynamic program profile information and (4) estimated instruction-level parallelism.

Dataflow Machine Construction After partitioning, the procedures selected for hardware implementation are processed by the CASH compiler. The output of CASH is a separate hardware circuit for each procedure. CASH essentially transforms the initial C program into a pure dataflow representation [4]. All C control-flow constructs are transformed into equivalent dataflow operations, as described below. The resulting dataflow representation has semantics very close to that of an asynchronous circuit: data producers handshake with data consumers on data availability and data consumption (see also Section 7).

Speculation and Predication In order to uncover more instruction-level parallelism, and to reduce the impact of control-flow dependences, aggressive predication and speculation are performed on the code. CASH leverages the techniques used in compilers for predicated execution machines [14]: it collects multiple basic

blocks into one hyperblock[1]; each hyperblock is transformed into straight-line code by using predication and static single-assignment (SSA). All instructions without side-effects are aggressively predicate-promoted, being speculatively executed. Thus control-flow dependences are reduced to a minimum.

Multiplexors When multiple definitions of a value reach a join point in the control-flow graph (for example, a variable assigned to in both branches of an `if-then-else` statement and used afterwards), they must be merged together. This is done by using multiplexors[2]. The multiplexor data inputs are the reaching definitions. We use *decoded multiplexors*, which select one of n reaching definitions of a variable; such a multiplexor has $2n$ inputs: one data and one predicate input for each definition. When a predicate evaluates to "true", the multiplexor selects the corresponding input. The hardware implementation of multiplexors and boolean operations is *lenient*: i.e., as soon as enough inputs have arrived to enable the computation of the output, the output value is immediately generated, without waiting for the remaining inputs. This implementation provides a solution to the problem of unbalanced control-flow created when speculating aggressively [2].

From Control-Flow to Data-Flow After hyperblock construction, the only remaining control-flow constructs are inter-hyperblocks transfers, including loops. In other words, each hyperblock is essentially an autonomous unit of SAM as described in Section 3.1. The next compilation phase stitches the hyperblocks together into a dataflow graph representing the whole procedure by creating dataflow edges connecting each hyperblock to its successors. Special "gateway" nodes guide the flow of data from a hyperblock to its successor [3]. This pure-dataflow representation is more detailed than the classical CFG-based representations because the control-flow for each variable is represented separately. This explicit representation naturally exposes loop-level parallelism, and gives rise to loop pipelining at run-time, as described below.

Tokens The previously described transformations remove all control-flow dependences from the code. The only constraints on the execution order are given by data dependences. However, not all data dependences are explicit in the original program: operations with side-effects could interfere with each other through memory. In order to maintain program order for operations whose side-effects do not commute we add edges between instructions that may depend on each other. While many other compilers use such dependence edges internally, in our architecture *they are part of the run-time system*. Such edges do not carry data values, but they carry an explicit synchronization *token*. The presence of a token at the input is required for these instructions to be executed; on completion, these instructions generate an output token to enable execution of their dependents. A memory operation issues a token to its successors as soon as it has initiated the memory access; it does not need to complete in order to issue the token. Thus, the tokens are used just to ensure that the memory operations are *issued* in the right order; dynamic memory disambiguation at the memory side is used to reorder memory accesses accessing different addresses; which naturally supports multiple memories. Token edges explicitly encode may-dependences between memory operations and allow the compiler optimizer to implement very efficiently a wealth of memory-related optimizations, such as redundant operation removal, synchronization removal and loop pipelining [5].

Registers Each operation in the dataflow graph has an associated register on its output. Once the operation has computed its result, it holds it in that register until all consumers are ready to accept it. Only after all consumers acknowledge the receipt of the result, can the operation start a new computation cycle. In this program representation there is no centralized register file, with multiple read/write ports. Values are stored only at each producer, waiting for the consumer to be ready. As many registers as necessary are synthesized, thus local values are never spilled to memory.

[1] A hyperblock is a contiguous portion of the program control-flow graph, having a single entry point and possibly multiple exits.

[2] The multiplexors correspond directly to the ϕ functions in the SSA representation.

Loop Pipelining Implementing the data-flow programs requires no control logic except the local circuitry controlling the handshake between data producers and consumers. The distributed nature of the control logic in a loop provides a natural pipelining effect of the computation. This phenomenon is well known from dataflow machine literature [17]. This effect stems from the fact that the original control-flow-based loop has been decoupled into a separate loop for each loop-carried value. In a processor a single backwards branch instruction signals the start of a new iteration; in our distributed implementation each loop-carried value has its own looping construct.

The pipeline effect is exploited, for example, by loop induction variables whose values can be quickly computed for several iterations ahead. Long-latency operations within a loop body are then pipelined, receiving in quick succession inputs from the multiple versions of the induction variables. The outcome is that operations from multiple loop iterations proceed simultaneously.

7: Translating C to Asynchronous Circuits

Asynchronous circuits eliminate the global clock, and are based on local communication and synchronization. The dataflow representation used by CASH is characterized by modularity, distributed control and localized communication. The data-triggered behavior of dataflow machines is a perfect match for asynchronous circuit implementation.

The use of asynchronous circuits brings other advantages: (1) their modularity and compositionality enables the compiler to scale to large programs and circuits by using a divide-and-conquer approach for program translation; (2) the distribution of controlled-skew clock signals to billion-devices circuits in the presence of potential defects in the distribution network is an insurmountable problem; (3) power consumption in asynchronous circuits occurs only in the computationally active parts; (4) placement and routing on synchronous RH must perform timing analysis of all signal paths with respect to the defect map so that the mapped design meets the global clock constraint.

The use of asynchronous designs however brings with it a wealth of research problems. For instance, the contemporary RH devices are all synchronous in nature; such RH platforms and their tool-chains, cannot be directly used to implement asynchronous circuits. Also, the CAD tools for asynchronous design lag far behind the state-of-the-art synchronous design tools. Hence, in order to implement asynchronous circuits on RH, we need to rethink the design of the reconfigurable platform architecture, and its tool-chain.

We have adopted the radical approach of synthesizing asynchronous circuits directly from high-level languages, such as C and FORTRAN. The CASH compiler has a back-end which translates the dataflow intermediate representation into Verilog descriptions of asynchronous circuits. These circuits can then be synthesized and technology mapped for implementation on RH. The compiler uses a library of modules as the building blocks of the asynchronous circuits. These modules closely correspond to the basic nodes of the intermediate representation, and they encapsulate both data-path and control components.

Every module consists of two parts: the node logic, and control circuitry. The goal of the control circuitry is to achieve flow-control in the absence of the global clock. When input data has arrived and has been processed, this circuit generates a control signal to latch the result in an output register. Flow-control is achieved by establishing a handshaking protocol with neighbors with whom data is exchanged. For this purpose, the *bundled data* protocol with four-phase handshaking is used. The bundled data protocol assumes that there exists a pair of wires, request (*req*) and acknowledgment (*ack*), associated with each bundle of data wires. The source node, N_s, sends a signal on the *req* wire when it has produced data to be consumed by destination, N_d. A property of this protocol is the *bundling constraint* that requires the data to arrive at N_d before *req*. After consuming the data, N_d sends a signal on *ack*. After having received all its inputs and successfully initiated the computation a node can immediately send an *ack* to its sources. The handshake naturally makes the circuits latency-tolerant when confronted by either long routes induced by defect tolerant place-and-route or by unpredictable-latency events, such as remote memory access or remote procedure invocations.

Modules representing ALU nodes are created by simply synthesizing the node logic to perform the corresponding arithmetic or logic function. In order to accommodate simultaneous memory accesses the compiler automatically synthesizes a circuit-specific memory channel access arbiter. While traditional processors have

one monolithic main memory, with RH we can take advantage of memory locality to create an application-specific memory architecture. The final, synthesized circuit is created by replacing the nodes in the intermediate representation with an instance of the equivalent module from the library. Value-flow edges in the graph are replaced by bundled wire connections.

Our approach to synthesizing these asynchronous circuits differs from prior work in several ways:

1. Control: Most work in high-level asynchronous circuit synthesis has concentrated on synthesizing asynchronous controllers and state machines. In contrast, the circuits generated by our compiler are dataflow-centric and application-specific, and are composed of arbitrary, non-uniform datapaths. Prior research in asynchronous datapath synthesis has been generally restricted to the design of either specific functional units (adders and multipliers), or the asynchronous processor pipeline.
2. Toolflow: In order to reliably build large-scale asynchronous circuits, a tool chain is required that can synthesize such circuits from some high-level language specification. The source language is generally a derivative of Hoare's communicating sequential processes [13], which describes a set of concurrent processes and their inter-communication. Our source language is C, which makes the tool chain accessible to a wider community. Our compiler uses powerful analysis techniques to identify parallelism and understand program behavior, and then transforms the program structure to exploit identified opportunities to improve the system performance.
3. We have devised and implemented a series of compiler optimizations particularly applicable to asynchronous dataflow machines. We will briefly mention here three such novel techniques: (1) the lenient multiplexor mentioned in Section 6.2, used to reduce the dynamic critical path of a circuit; (2) the use of explicit fine-grained synchronization between memory operations in the program through the use of token edges, described also in Section 6.2 and (3) the use of elastic pipelines [16] to allow loosely synchronized computations to slip with respect to each other, boosting the pipelining effect.

8: Conclusions

We believe that computer architecture is presently faced with an incredible opportunity: to harness the power of molecular computing to create computing systems of unprecedented performance per dollar. To realize this promise, we need to rethink the basic abstractions that comprise a computer system. Just as current systems benefited greatly from the underlying technology—the silicon-based transistor—new systems will benefit from the molecular reconfigurable switch. This can serve as the basis for a reconfigurable computing system that can construct customized circuits tailored on the fly for every application.

Acknowledgments

This research is funded in part by the National Science Foundation under Grants No. CCR-9876248 and CCR-0205523 by DARPA under contracts MDA972-01-03-0005 and N000140110659. This work could not have been done without the collaboration between our group and the Moletronics groups at Hewlett-Packard, UCLA, and Penn State.

References

[1] R. Amerson, R.J. Carter, W.B. Culbertson, P. Kuekes, and G. Snider. Teramac–confi gurable custom computing. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 32–38, Napa, CA, April 1995.

[2] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A framework for balancing control fbw and predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[3] Mihai Budiu and Seth Copen Goldstein. Compiling application-specifi c hardware. In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, Montpellier (La Grande-Motte), France, September 2002.

[4] Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[5] Mihai Budiu and Seth Copen Goldstein. Optimizing memory accesses for spatial computation. In *Proceedings of the 1st International ACM/IEEE Symposium on Code Generation and Optimization (CGO 03)*, San Francisco, CA, March 2003.

[6] Mihai Budiu, Mahim Mishra, Ashwin Bharambe, and Seth Copen Goldstein. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. In *Proceedings of 2002 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2002.

[7] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the 2000 Europar Conference*, volume 1900 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[8] M. Butts, A. DeHon, and S. Goldstein. Molecular electronics: Devices, systems and tools for gigagate, gigabit chips. In *ICCAD-2002*, Nov. 2002.

[9] W.B. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. The Teramac custom computer: Extending the limits with defect tolerance. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 2–10, 1996.

[10] Seth C Goldstein and Mihai Budiu. Molecules, Gates, Circuits, Computers in *Molecular Nanoelectronics*, edited by M. Reed and T. Lee American Scientific Publishers, 2003.

[11] Seth Copen Goldstein and Mihai Budiu. NanoFabrics: Spatial computing using molecular electronics. In *Proceedings of the 28th International Symposium on Computer Architecture 2001*, pages 178–189, 2001.

[12] Reiner Hartenstein. A decade of research on reconfigurable architectures - a visionary retrospective. In *Proc. International Conference on Design Automation and Testing in Europe 2001 (DATE 2001)*, Exhibit and Congress Center, Munich, Germany, March 2001.

[13] C. A. R. Hoare. Communicating sequential processes. In *C. A. R. Hoare and C. B. Jones (Ed.), Essays in Computing Science*. Prentice Hall, 1989.

[14] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, Dec 1992.

[15] Mahim Mishra and Seth C. Goldstein. Defect tolerance after the roadmap. In *Proceedings of the 10th International Test Synthesis Workshop (ITSW)*, Santa Barbara, CA, March 30–April 2 2003.

[16] Ivan Sutherland. Micropipelines: Turing award lecture. *Communications of the ACM*, 32 (6)(720–738), June 1989.

[17] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18 (4):365–396, 1986.