Parallel Bootstrapping of a Large Fine-Grain Self-Configurable System

Nicholas J. Macias Peter M. Athanas
Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia, USA
nmacias@vt.edu athanas@vt.edu

***Abstract*—The possibility of a sudden, significant increase in device complexity is proposed. Challenges in the management of such devices are described, and the potential for using self-configurability to meet these challenges is discussed. Basic operations on a self-configurable system are introduced, and their application to a parallel bootstrap mechanism is presented. A methodology for using this mechanism to more-efficiently bootstrap a large reconfigurable device is given, and its extension to efficient 3-D bootstrapping is explained. Plans for future work are presented.**

Keywords-configuration; compile time; bootstrap; self-configurable; 3-D

I. Introduction

As the feature size of transistors continues to shrink and new fabrication materials and processes emerge, the number of devices which can be fabricated on a single IC has grown, roughly doubling in number every 18-24 months [1], and presently, fabricating one billion transistors is commonplace. This continuing growth is in fact a result of both improvements in existing technologies as well as the development of new ones. Generally, a technology will be refined and improved over a period of years (with a corresponding increase in the number of devices which can be fabricated), but eventually, as it nears what is considered to be its limit in terms of device density, a new technology emerges to replace it. For example, a perceived hard limit in the resolution possible with optical lithography led to the development of electron-beam lithography [2].

This continuing increase in density is also reflected in our design emphasis:

- 3 decades ago, we were worried about minimizing transistors ;
- 2 decades ago, the emphasis was on minimizing gates ; and
- today, we are focused on miniaturizing cores .

Similarly, we are quickly approaching the point of a million "slices" in an FPGA; yet the tools and programming methodologies are seriously lagging.

This trend shows no sign of weakening, and, in fact, there are some who believe we may be approaching a fundamental leap in the density of fabrication techniques [3]. Such leaps could, perhaps, see the advent of so-called Avogadro machines: machines containing a trillion trillion (10^{24}) devices. At present, designing a fixed, heterogeneous circuit utilizing that many devices seems beyond human capabilities. More likely, such huge device counts would be used to build circuits with a very regular structure. One particularly attractive target would be a very large reconfigurable device such as an FPGA.

Current place-and-route times for modern reconfigurable devices can easily run into hours, even for designs of modest complexity [4]. Certainly, even if these times scale linearly with circuit- and device-size, these conventional techniques will be infeasible when component counts increase by several orders of magnitude. If there is some singularity-like event, after which we can manufacture devices containing, say, 10^{24} elements, there will be an even more basic impediment: the time required to load a configuration string. Supposing each element requires only 16 bits of configuration information, and a 1THz configuration clock is used, configuring 10^{24} elements would require 1.6×10^{13} seconds to configure, or approximately **half a million years**. Clearly, a different paradigm for bootstrapping such a high-component-count system is in order.

There are some applications that require a huge number of components, yet are, at some level, highly homogeneous. For example, simulation of heat transfer in a 2-D structure may require a large number of processing nodes – one node for each element in a finite element mesh – but those nodes are similar to one another, each comprised of identical digital circuitry for modeling heat flow across a small sub-region. In this case, it's conceivable one could design a configuration system which reads a small amount of information – enough to describe a single node – and uses it to identically-configure, in parallel, a large number of regions of the system.

Of course, such a configuration system would need to know the dimensions of each node, so that the proper stride could be used in configuring multiple elements in parallel; and these dimensions are highly dependent on the specific application being addressed. One could design a configuration system which allows specification of the height and width of each node, but there may be cases where, perhaps, the elements are homogeneous within each row, but change from one row to the next. And a configuration system might be augmented to support this as well. But in general,

there are many possible varieties of parallel configuration. Designing a configuration system of one particular type may be relatively easy, but designing a general-purpose configuration system is a more-formidable challenge.

In fact, the problem is simply that most reconfigurable systems allow you to modify the circuit being implemented on the system, but not any aspects (such as the configuration mechanism) of the reconfigurable system itself. In other words, while the reconfigurable elements can be modified, the rest of the device is still implemented with a fixed hardware structure.

II. Background

A sufficiently-simple self-configurable system can get around this limitation, by using the configurable elements to implement pieces of the reconfigurable system itself. The methodology described in this paper should be applicable to any-such reconfigurable system, provided each low-level element is able to configure some of the elements around it. An example of such an architecture is the Cell Matrix [5]. In this architecture, the lowest-level reconfigurable elements – the *cells* – are used to create low-level logic blocks, as well as for creating pathways between elements, including pathways used to configure a set of cells, i.e., to bootstrap the system. A major benefit of this is that, because these pieces are composed of cells, they are extremely flexible: their exact design can be chosen based on the given application being implemented. An additional potential benefit is that the inherent parallelism of hardware can be exploited, for example, to implement a parallel bootstrap.

The Cell Matrix is a fine-grained, self-configurable architecture which emphasizes autonomous, local control of the hardware, not only in the implementation of a target circuit, but in the management of the reconfigurable fabric itself, i.e., aspects such as its configuration, routing and fault handling [6]. The architecture is also highly scalable: the number of reconfigurable elements (cells) can be increased without any changes to the system architecture. This scalability, combined with the simplicity of the cells, the very small number of global signals, and the inherent locality of configuration operations within the system, makes the Cell Matrix an attractive architecture for organizing an extremely large number of switches into a reconfigurable substrate [7].

In the Cell Matrix architecture, cells are connected in a regular tiling, according to a system-wide topology which defines each cell's immediate neighbors. A typical organization for a 2-D matrix is a set of four-sided cells, each cell connected to its four ($r=1$) von Neumann neighbors. For a 3-D matrix, a typical organization is to have six-sided cells (cubes), packed in the expected way, with each cell connected to its six immediate neighbors.

A. Wires

When a circuit is implemented on the Cell Matrix, one or more cells are configured to implement the low-level components of the circuit: gates, multiplexers, flip flops, and so on. These components must then be connected to each other to realize the final circuit. In general, since each cell is only directly-connected to its immediate neighbors, a collection of cells must be used together to pass information from one area of the matrix to another. Such collections of cells are called "wires," and are comprised of one or more sets of cells operating in a bucket-brigade manner to transfer data (note that there are ways to avoid the compounded propagation delay which normally occurs when data is transferred in such a way).

B. Cell Configuration

Cell configuration is achieved primarily through the action of other cells (the exception being cells along the edge of a matrix, some of whose "neighbors" are thus located outside the matrix). The basic configuration mechanism works through the use of two inputs into each cell: a C input, which can be asserted to place the cell into *configuration mode*; and a D input, which transfers configuration information into the cell (if the cell is not in configuration mode, the D input is used to transfer binary data to be processed by the cell). Because each cell has only a small set of neighboring cells connected to it, a cell can be configured from only a small number of locations within the matrix. Again, this seeming limitation is handled by organizing collections of cells to form wires, which transfer not only data but also configuration information.

C. Simple Bootstrapping

By building wires in an appropriate order, a circuit can create a pathway to any given region of the matrix, and then sweep that region in a raster-scan style, configuring cells to one side of the scanning wire as it sweeps through the region. In this way, the cells within a 2-D region can be configured as desired (the technique can easily be extended to a 3-D region, but the analysis of a 2-D region is simpler). Disregarding the time required to build the initial wire to the target region, the bootstrapping of n^2 cells is an $O(n^2)$ operation, since, upon reaching the target area of the matrix, it requires approximately the same number of operations to subsequently configure each cell. This approach to bootstrapping will thus suffer from an increasingly-prohibitive time requirement as the region being bootstrapped grows larger.

III. Parallel Configuration

While the simple bootstrap algorithm described above does not take advantage of the potential for multiple cells to simultaneously configure multiple regions of the matrix, the circuit in Fig. 1 - a *1-D Parallel Configuration circuit* – does take advantage of this potential. The operation of this circuit is very straightforward: one set of cells forms a wire for transmitting configuration information ("D") from left to right; a second set, located below the first, transmits a single configure ("C") signal. By asserting C, each cell of the top wire will configure the cell directly above it, using the single set of configuration information sent down the D wire. In this way, an entire row of cells are configured in parallel: if the wire is n cells long, then a total of n cells will be configured in simultaneously.

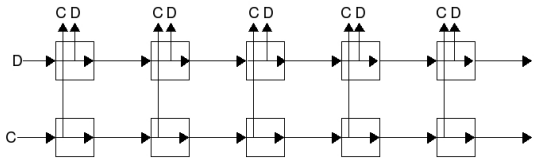

Figure 1. 1-D Parallel Configuration Circuit.
The C and D inputs on the left are sent to the C and D inputs of each cell directly above the circuit. This allows all such cells to be configured, identically, in parallel.

Such parallel configuration is one step closer to more-efficient bootstrapping, but a problem remains: the construction of such a parallel configuration circuit itself requires O(*n*) steps. Have we thus gained anything by using such a circuit? For configuring a single row of *n* cells, we have not: the total configuration time is still O(*n*). But once this configuration circuit is built, we can then configure *n* cells in O(1) operations, i.e., a fixed amount of time which is independent of *n*. **We can also use this capability to configure a second parallel configuration circuit in a fixed amount of time.** In other words, once we have built a single 1-D parallel configuration circuit (in O(*n*) timesteps), we can use it to build a second such circuit in O(1). We could then use that second circuit to construct a third circuit, and so on, with each subsequent circuit requiring only O(1) to configure. The only change here is that the 1-D parallel configuration circuit must be modified to allow configuration of cells below as well as above itself. The real trick here is that we are not only bootstrapping a target circuit in parallel, we are using the parallel bootstrap mechanism to assemble the parallel bootstrap mechanism!

Figs. 2a-2d illustrate how this process can be used to configure O(n^2) cells in only O(*n*) time steps. In Fig. 2a, a single, n-cell-wide 1-D parallel configuration circuit ("C1") has been built – this requires O(*n*) steps. In Fig. 2b, C1 is used to configure the top (possibly multiple-cell-high) row ("X1") of the desired target circuit. This may require multiple steps (for example, configuring two cells to the North requires three steps instead of one). However, the number of steps is still independent of n, since configuration of all *n* regions occurs in parallel; thus, this step is O(1). In Fig. 2c, configuration circuit C1 is used to build a new configuration circuit ("C2") to the South. Again, this is an O(1) operation. And finally, in Fig. 4d, circuit C2 is used to build a second piece of the desired target circuit ("X2") - also an O(1) operation.

Following the initial construction of C1, configuration of each pair of rows X_i/C_{i+1} is thus an O(1) operation. Therefore, configuring *n* rows of the target circuit (X_1 , X_2 , ..., X_n) requires O(*n*) steps. Combining with the original O(*n*) for building C1, the total time required for building the 2-D *n*x*n* region is thus only O(*n*). We have achieved a better-than-linear configuration time.

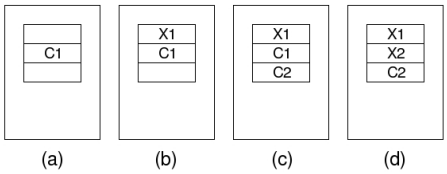

Figure 2. 2-D Parallel Configuration Methodology
(a) Initial 1-D parallel configuration circuit C1. (b) C1 has configured the first row (X1) of the target circuit. (c) C1 has configured a second configuration circuit (C2). (d) C2 has configured the second row (X2) of the target circuit. These last two steps are repeated until all rows (Xi) have been configured. Since each step requires O(1) operations, configuring n rows (X1, X2, ..., Xn) is an O(n) process.

IV. Medusa Wires

Wires such as those described above are referred to by the authors as "Medusa Wires." A Medusa Wire is a parallel configuration circuit which can be used to efficiently create a tiling of any desired circuit composed of homogeneous sub-circuits. Fig. 3 shows an example of such a wire, where each stage is built from 3 cells. This wire is designed for configuring, in parallel, a large number of identical sub-circuits, each of which is exactly 1 cell wide. If we wanted to configure a network of sub-circuits, each being, say, 16 cells wide, we would pad the circuit of Fig. 3 with 15 additional cells on the side, creating a stride of 16 cells in the replication pattern to the North. Fortunately, this would also be the exact stride necessary for building a new Medusa Wire to the South.

The wire has three inputs on the left:

- D is used to send data or configuration information to a set of cells;
- CN determines if the cell to the North treats D as data (CN=0) or as configuration information (CN=1); and
- CS determines if the cell to the South treats D as data (CS=0) or as configuration information (CS=1).

The circuit shown in Fig. 3 is actually a simplified Medusa Wire. It can be used to configure a 2-D region above each stage of the wire, but doing so is very time-consuming, because the circuit in Fig. 3 implements wires to the North and South which are only one cell wide, whereas a practical bootstrap circuit requires wires which are three cells wide [8]. A Medusa Wire employing such wires is easy to implement. However, the circuit in Fig. 3 is easier to describe, and the basic principle is exactly the same.

There is one additional detail to using a Medusa Wire: as new wires are built to the south, the connection points ("D," "CN," and "CS") to those wires also moves to the south, so the circuit which is driving those signals needs to build its own wires, extending them after each new Medusa Wire is built. Doing so is straightforward, using standard wire-building sequences [8].

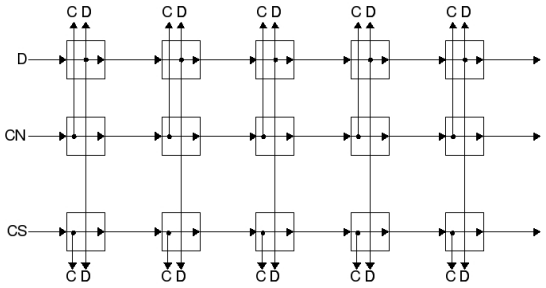

Figure 3. 2-D Parallel Configuration Circuit
D is used to pass either data or configuration information to the cells immediately above or below the circuit. CN causes D to be treated as configuration information by the cells to the North, and CS causes D to be treated as configuration information by the cells to the South.

By using the configuration pattern shown in Fig. 2, the circuit of Fig. 3 thus can configure $O(n^2)$ sub-circuits in $O(n)$ steps.

V. Extending to 3-D Bootstrapping

The circuit described above can be extended into a 3-D parallel replication circuit. This would simply be a 2-D tiling of identical blocks, each capable of configuring cells above and below (i.e., in the direction of the Z-axis within a 3-D reconfigurable substrate). The methodology of Fig. 2 would still be used, but instead of creating a single row of target circuits or new configuration circuits, each step would produce a 2-D sheet of such circuits. And since configuration of these new sheets of circuits takes a fixed (independent of n) number of time steps, configuring $O(n^3)$ circuits would require only $O(n)$ steps, **once the initial 2-D parallel configuration circuit has been built**. However, as we have seen above, we can configure a single 2-D sheet of n^2 circuits in only $O(n)$ time-steps. Therefore, we can configure the entire 3-D structure in only $O(n)$ time-steps.

In terms of our original analysis, this means we could configure 10^{24} elements in approximately 300 million time-steps:

- 100 million steps to build the first 1-D parallel configuration circuit (containing 100 million Medusa Wire elements);
- another 100 million steps to extend it to an initial 2-D parallel configuration circuit (100 million x 100 million); and
- a final 100 million steps to build 100 million sheets (each 100 million x 100 million) of our target circuit.

A prototype 2-D bootstrap circuit requires approximately $274+598*n$ cell configurations to build n^2 *6x3* sub-circuits, while a 3-D version requires approximately $300+2500*n$ cell configurations to build n^3 *6x3* sub-circuits. For a 2-D circuit of 1,000,000,000 such sub-circuits, this would thus require approximately 19,000,000 cell configurations to bootstrap 18,000,000,000 cells: a speedup of almost 1000x. For a 3-D circuit consisting of 10^{24} sub-circuits (i.e., n=100,000,000), the 3-D bootstrap would require 250,000,000,000 configurations. Going back to our original time estimate, still assuming 16 clock cycles for one cell configuration, and a 1THz configuration clock, this brings the system bootstrap time down from half a million years to a mere 4 seconds.

Figure 3. 2-D Parallel Configuration Circuit

VI. Status, Conclusions and Future Work

A large-scale reconfigurable system containing identical, simple reconfigurable elements is an ideal platform for implementing massively-parallel circuits with local connectivity, such as those used for finite element analysis. The challenge of efficiently configuring such parallel circuits can be met by utilizing local, parallel configuration. This is achieved by utilizing a system's self-configurability to, in effect, configure pieces of itself, such as its own system-level configuration mechanism. Similar uses of self-configurability can also be used to tackle challenges such as fault handling, autonomous synthesis and routing of heterogeneous systems, and run-time monitoring and adaption of dynamic circuitry [6].

A basic Medusa Wire has been designed and tested for the layout of 2-D networks of sub-circuits. Large-scale simulation is prohibitively-slow (since sequential simulation of a massively parallel system suffers an inherent slowdown on par with the degree of parallelism), but the simulated time-clock confirms the parallelism of the circuit. Fig. 4 shows a screenshot of the simulated 2-D bootstrap system. Clicking on this figure will show a movie of the simulation. Work on extending this to a 3-D circuit is ongoing.

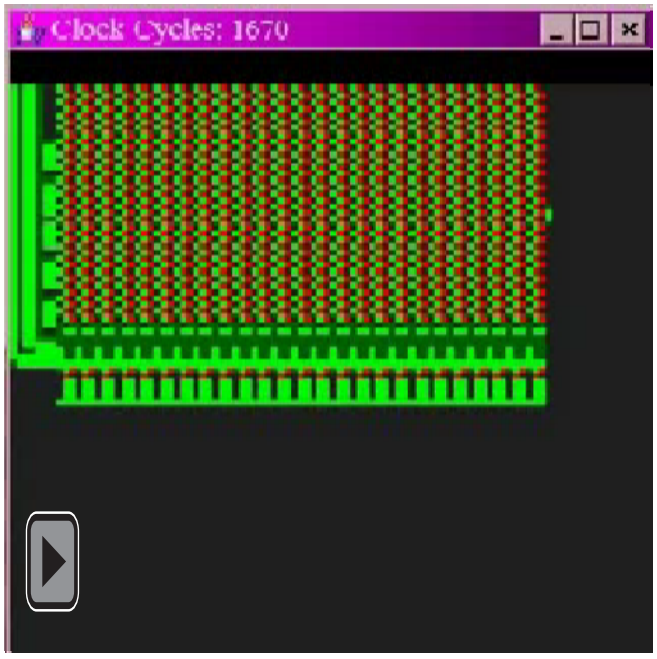

Figure 4. Screenshot of the Simulated Parallel Bootstrap Circuit.
Click on the above figure to watch the movie.

Future work includes extending the above methodology to allow for *local differentiation* during parallel configuration – an important capability for construction of non-homogeneous circuits/applications. This can be achieved, for example, by adding a 1-bit incrementer to each node of the Medusa Wire, and then using that bit as a mask to pass/block configuration information. In this way, identical configuration information would result in the configuration of two different sets of circuits.

Modifying the Medusa methodology to accommodate faults in the reconfigurable substrate is another challenge, involving a pre-build test phase, followed by instantiation of pathways for routing around defective regions.

Finally, combining this methodology with a self-replicating circuit would create an efficient, autonomous parallel replication system. Such a system would perhaps be suitable as an initial *seed* in an embryologically-inspired electronic stem cell system [9].

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," Electronics, Volume 38, Number 8, April 19, 1965.

[2] A. Tobey, "Wafer Stepper Steps up Yield and Resolution in IC Lithography", Electronics, pp. 109-112, Aug. 16, 1979.

[3] R. Kurzweil, "The Singularity is Near," Viking Penguin, 2005.

[4] R. Goering, "FPGA Tool Bottleneck Stalls HPC," EE Times, 5 Feb 2007.

[5] L. Durbeck and N. Macias, "The Cell Matrix: an architecture for nanocomputing," Nanotechnology vol 12 (Bristol, Philadelphia: Institute of Physics Publishing), pp. 217-230, 2001.

[6] N. Macias and L. Durbeck, "Self-Assembling Circuits with Autonomous Fault Handling," Proc. The 2002 NASA/DoD Conference on Evolvable Hardware, ed A. Stoica, J. Lohn, R. Katz, D. Keymeulen and R. Salem Zebulum, pp. 46-55, 2002.

[7] L. Sekanina, "Evolvable Components: From Theory to Hardware Implementations," Springer-Verlag (Natural Computing Series, ISBN 3-540-40377-9), pp. 20-22, 2003.

[8] N. Macias and L. Durbeck, "Advances in Applied Self-Organizing Systems," ed M. M. Prokopenko, London: Springer-Verlag ISBN: 978-1-84628-981-1, pp. 192-202, 2008.

[9] N. Macias and P. Athanas, "Application of Self-Configurability for Autonomous, Highly-Localized Self-Regulation," Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), pp. 397-404, 2007.