The PIG Paradigm: The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture

From Proceedings of the First NASA/DoD Workshop on Evolvable Hardware Copyright© 1999 IEEE

Nicholas J. Macias
P.O. Box 510485
Salt Lake City, UT 84151
nmacias@cellmatrix.com

Abstract

The requirements of a general purpose massively parallel processing system are outlined. The suitability of a fine-grained self-reconfigurable system to general massively parallel processing is shown. A new type of self-reconfigurable device called the PIG is introduced, and details of its design and operation are explained. The PIG's uniqueness compared to other reconfigurable systems is discussed. This uniqueness is further illustrated through specific examples of PIG circuits. An application of the PIG to evolvable hardware is described. Further potential applications are discussed. Plans for future work, including options for building a large-scale PIG are discussed.

1 Introduction

There is good cause to be interested in computing systems which can perform thousands or millions of operations in parallel. While traditional uniprocessors and specialized parallel processors are in no danger of becoming obsolete, there is a wide range of applications which are inherently massively parallel, and which can be executed most efficiently of a massively parallel processing system.

If a system is going to perform millions of independent operations in parallel, it clearly must contain millions of independent processors. What is less clear is exactly how this parallel hardware should be configured. The desirable capabilities of each processor depend on the nature of the problem being solved. For example, applications such as finite element analysis require high-precision floating point arithmetic, while pattern detection in DNA requires rapid comparison of memory blocks, and would benefit more from a hardware pattern matcher than from deep multiplicative pipelines. Moreover, the interconnections among these processors is critical. An application which performs multiple simultaneous operations on a fixed set of data may work best with a star topology, while a parallel circuit simulator is better suited to a nearest-neighbor interconnection scheme. This suggests that both the processors and their interconnection scheme must be extremely flexible.

One way to achieve this flexibility is to use a general purpose reconfigurable platform, composed of a large number of hardware elements which can be individually configured via software. The variation in the above examples suggests a very fine-grained architecture, where the individual configurable elements are relatively simple and are combined to perform more complex functions. Of course, such a reconfigurable platform needs to be quite large.

Since such a system is composed of general purpose hardware, there must be some way to specify the configuration of that hardware. Most current reconfigurable systems use an external control system (usually a PC or other sequential machine), with the reconfigurable platform (usually an FPGA) attached as a coprocessor [1]. The popularity of this setup extends even to single-chip solutions [2]. Such an arrangement is in fact sometimes stated as the **definition** of a reconfigurable system [3]. Unfortunately, this setup is not suitable for systems which require **massively parallel reconfiguration**, in which multiple parts of the system are simultaneously reconfigured, perhaps based on information within the reconfigurable elements themselves.

This suggests the following requirements for the configuration controller of a massively parallel reconfigurable system:

- Since it should be able to monitor, analyze and reconfigure multiple circuits simultaneously, the controller itself should be massively parallel

- To avoid communication bottlenecks, the configuration control should be distributed throughout the reconfigurable platform

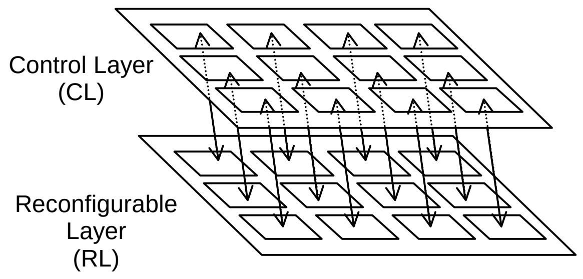


Figure 1. One possible way to distribute reconfiguration control throughout a grid of reconfigurable devices. The reconfigurable layer shown consists of a two-dimensional array of reconfigurable elements. Each element is controlled by a separate processing element in the control layer. While this allows parallel reconfiguration of the devices in the RL, the RL cannot directly configure itself, nor can the CL layer be configured without an external controller.

Figure 1 shows one way to satisfy these requirements. This setup consists of a reconfigurable layer (RL) of reconfigurable hardware, and a controller layer (CL) which monitors the circuits in the RL and can analyze and modify their configuration. Such a setup meets the above requirements, but still has two shortcomings:

1. The RL itself might need to configure other circuits, for example, in the evolution of circuits which themselves perform circuit synthesis.
2. Since the CL needs to be configured, there must be another controller to handle that.

In other words, the picture in figure 1 might need to be extended both above the CL and below the RL. One way around this is to utilize a reconfigurable platform which is **self-reconfigurable**, meaning the same circuits which are themselves configurable are also capable of configuring other circuits.

A massively-parallel, fine-grained, self-reconfigurable system embodies all of the above requirements. Obviously it is massively parallel and reconfigurable. Since it can modify its own circuits, the reconfiguration control (i.e., itself) is massively parallel, distributed throughout the system, and local to the circuits being configured.

Additionally, we introduce one more requirement: scalability. The system should have a regular internal structure, composed of identical atomic units in a simple, regular interconnection scheme. Certainly this simplifies the manufacturing process. Moreover, with the degree of parallelism we are discussing (millions or billions of devices), fault tolerance becomes a critical consideration, and a system with identical hardware and identical interconnections within is less likely to have critical failure points.

The next section describes a processing system which satisfies the above requirements.

2 The Processing Integrated Grid

The Processing Integrated Grid (US Patent #5,886,537), or PIG, is a massively parallel, fine grained, self-reconfigurable infinitely scalable system which satisfies the requirements presented in section 1. A few general comments about this system will help in the discussion which follows. First, the PIG is not intended as a replacement for traditional von Neumann processors, which are already extremely efficient at executing scalar algorithms. Neither is the PIG intended for scalar algorithms which have somehow been coerced into executing in parallel. The PIG is best suited to algorithms which are inherently massively parallel.

The PIG is not anything like a von Neumann machine. It is not programmed in the traditional sense of the word, it does not have an attached memory per se, there are no instructions, no internal buses, no registers. It is basically a platform of reconfigurable hardware which functions similar to a dataflow machine. However, the PIG is much more than a simple grid of blank hardware. While the PIG's circuits can process data, they can also interchangeably process configuration information. Therefore, unlike a Field Programmable Gate Array (FPGA) or other similar reconfigurable device, the PIG is capable of analyzing and modifying its own circuits.

The PIG is composed of a collection of simple reconfigurable elements called *cells*, connected in a regular array structure. Figure 2 shows one arrangement of cells as a regular two-dimensional array, with each cell connected to exactly four neighbors. This simple interconnection scheme among homogeneous cells, and the complete lack of non-adjacent connectivity, leads to an infinitely-scalable architecture. If you take two PIGs and connect them along their edges, the result is a larger PIG which functions identically to the originals.

Each individual cell can be configured to act as a simple combinatorial device, and as such, the entire PIG can be configured as a large digital circuit, with cells functioning together to implement, for example, logic gates, flip flops, and wires. While the PIG can thus be used to implement state machines, memories, or CPUs, it is capable of much more than simply implementing fixed digital circuits. The PIG is *self-reconfigurable*, meaning that it is capable of modifying its own circuits, without requiring external control. In fact, it can be rather difficult to control the PIG externally, as most of its cells are not directly accessible from outside the grid.

To understand this self-reconfiguration mechanism, which is the key to the PIG's power, we must take a closer look at a single PIG cell.

Each cell in figure 2 has two inputs (C_{in} and D_{in}) and two outputs (C_{out} and D_{out}) on each side. Additionally, each cell contains an internal 16-row by 8-column truth table, which governs the combinatorial behavior of the cell. Cells exchange information on their D lines, though the nature of this information depends on which of two *modes* the cell is currently operating in. In *Data* mode (or D mode), the cell is a pure combinatorial device, which reads its four D inputs and, using them as inputs to its truth table (to select one of 16 rows), determines a set of eight output values to present on the four D and four C outputs. In *Control* mode (or C mode), the D inputs are serially shifted into the cell's internal truth table, according to a system-wide clock. This allows one cell to write another cell's internal truth table, which subsequently affects that cell's behavior when it returns to D mode. Additionally, as the new truth table is shifted into the cell, the cell's prior truth table is shifted out on it's D outputs, and is available for reading. The current mode of a cell is determined by its C inputs. If any of a cell's C inputs is 1, the cell is in C mode, otherwise it is in D-mode. When a cell is in C mode, only the D inputs and outputs on sides where C_{in}=1 are relevant. Normally this would only be one side, but there are exceptions to this. Figure 3 illustrates these two modes of operation for a single cell.

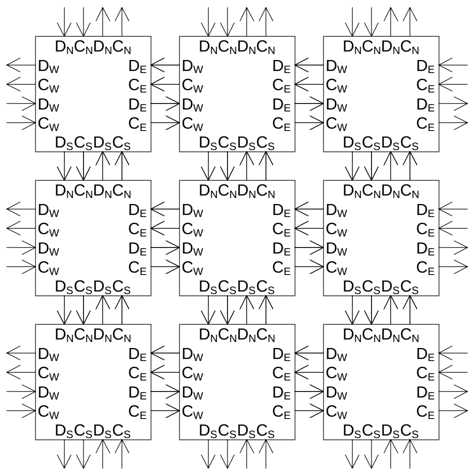


Figure 2. Sample two-dimensional 3x3 grid of PIG cells. Each cell is directly connected only to its immediate neighbors, and exchanges two inputs and two outputs with each neighbor. Sides are designated N, S, W and E for convenience.

The PIG's design has three immediate consequences:

1. Since a cell can control its C outputs (via its truth table), each of which is a neighboring cell's C input, **any cell can control the mode of any neighboring cell**.

2. By placing a neighboring cell in C mode and reading and writing that neighbor's D lines, a cell can read and write the truth table of any neighboring cell, and thereby configure it to subsequently perform any combinatorial function desired (after returning the neighbor to D mode).

3. Since the neighbor's new combinatorial function can produce any desired C and D outputs, that neighbor can be configured to itself configure any of **its** neighboring cells.

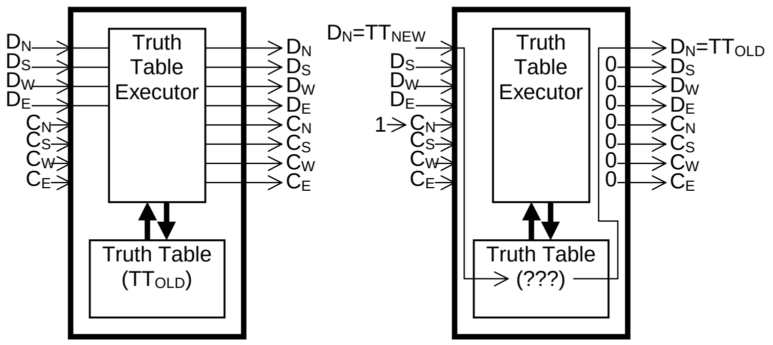
These characteristics are sufficient to allow any cell access to both the data and configuration information of any other cell within the PIG. Figure 4 shows a typical programming sequence. Cell X first configures cell Y to read data from X and pass it to Z, while asserting its C output to Z. Cell X then feeds data into Y, which passes it on as configuration information for Z. Hence cell X is able to reconfigure cell Z, even though there is no direct connection between cells X and Z. This is possible because cell Y is first the object of a configuration step, and then becomes the controller of Z's configuration, **all under the control of cell X**. This interchangeability of controllers and controlled-devices is sometimes refereed to as "Code/Data Duality." This duality is not merely an incidental consequence of the PIG's design. It derives directly from the original motivating problem behind the PIG, which was to write a software program which outputs its own source code. The software solution was reformulated in hardware, and the result was the first PIG cell.

Note that there is nothing special about cells X, Y or Z. They are identical to each other and to every other cell within the PIG.

Since any cell can configure any neighboring cell, as well as non-neighboring cells, the task of reconfiguring cells can be distributed throughout the grid, with multiple reconfigurations occurring simultaneously. Hence the PIG is quite capable of parallel self-reconfiguration. This internally-controlled reconfigurability, combined with the code/data duality, has been used to realize many interesting and powerful functions, including cell replication, dynamic path construction for control of non-adjacent cells, dynamic circuit analysis and synthesis, and the creation of self-replicating circuits.

While a small cMOS PIG has been built and successfully tested (see Section 5), most PIG work has been done on simulators. In addition to providing a higher cell count than currently possible with their physical counterparts, simulators offer direct access to interior PIG cells, which aids the development and debugging process.

Full details of the PIG's circuit design, timing diagrams for its programming, and other details can be found in the patent, which can be found online at [4].



Initial State of Cell. All C inputs are 0, so cell is in D-mode. The truth table executor uses the D inputs to look up output values in truth table TT_{OLD}.

Same Cell Being Reconfigured. Since C_N is 1, the cell is now in C-mode. The D_N input is fed directly into the cell's truth table, while the old truth table is output on the D_N output. All other outputs=0.

Final State of Cell. All C inputs are again 0, so cell returns to D-mode. The truth table executor uses D inputs to look up output values in the new truth table TT_{NEW}.

Figure 3. Reconfiguration of a single cell. In figure on left, the cell is processing D inputs based on the TT_{OLD} truth table. In the middle picture, the cell is being reprogrammed with TT_{NEW}, while its old truth table is being read. In the picture on the right, the cell has been reprogrammed to use TT_{NEW} for generating outputs. All C inputs are 0 unless shown otherwise. For simplicity, all inputs are shown on the left and all outputs are shown on the right.

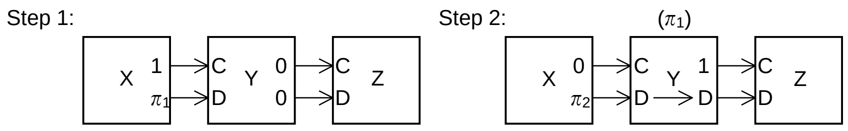


Figure 4. Cell X configures non-adjacent cell Z by first configuring cell Y to act as a router of X's configuration information (Step 1), and then passing Z's desired truth table (π_2) into Z via Y (Step 2). Cell Y is first an object of configuration, and then becomes a configuration controller itself. This is called "Code/Data Duality."

3 Sample PIG circuits

Most of the development work on the PIG has dealt with foundational tasks, such as how to analyze and configure circuits from within the grid. While these lower-level tasks do not represent complete applications in themselves, they deal with more PIG-specific features of the system, and as such, are illustrative of the unique nature of the PIG. Section 4 will describe a specific higher-order application involving evolvable hardware.

Some of the foundational circuits developed for the PIG include:

1. A cell replicator, which non-destructively reads the configuration of one adjacent cell, and uses it to configure another adjacent cell identically. This circuit requires only a single cell, and executes in 128 clock cycles.

2. A remote cell copier, which performs the above cell replication on non-adjacent cells. This circuit requires a single cell, plus pathways to the source and destination cells. These pathways are two cells wide.

3. A cell library, which contains an archive of fixed cells, any of which can be selected and replicated at some target location. A 2^n-cell library requires 2^nx(n+2) cells for storing and selecting the desired cell, one cell for performing the replication, plus a two-cell-wide pathway to the target cell.

4. A general wire building circuit, which combines the above circuits to build the structures necessary to access remote cells, **without having any pre-existing access paths to those remote cells**. The basic wire structure is still only two cells wide. The sequence of operations to extend the wire a single cell requires 512 clock cycles. One simple control circuit for this process can be built using 11x17 cells, though numerous configurations are possible.

This circuit is an example of a more general class of circuits called "sequence generators." Sequence generators create sequences of bits which modulate the C and D inputs of remote cells to ultimately control the configuration of other cells. These circuits are key to most of the interesting circuits developed so far for the PIG.

5. An expanding 12-bit counter, which responds to an impending overflow by synthesizing additional stages beyond the most-significant bit, thereby becoming a 13-bit counter, or a 14-bit counter, or as large a counter as necessary. The basic circuit for this requires 37x25 cells for the control system, plus n x 4 cells for an n-bit counter. The sequence of steps to extend the counter a single bit requires 6272 clock cycles.

6. A space filling circuit (SFC), which, from a single control circuit, can be replicated with time $O(\sqrt{n})$, i.e., it will fill a space of size n^2 in a time proportional to n. The space occupied by these SFCs expands outward in all directions along a wavefront of increasing diameter. At each step, the control circuit has simultaneous access to all cells along this wavefront. Additionally, the SFC is endowed with a dynamic locator circuit, which indicates where each copy of the SFC resides relative to the central controller. This positional information can be used to control configuration of individual cells along the expanding wavefront. In this way, the SFC can be used to dynamically construct large circuits. The basic SFC is an 11x11 circuit, and requires approximately 1,280,000 clock cycles to sweep outward one step. The control circuit for this system has not been implemented in PIG cells, but has been simulated in a higher-level “Virtual PIG” simulator.

7. An autonomous self-replicating circuit. This circuit, once loaded into the PIG, immediately begins making an exact copy of itself. As soon as that copy is finished being created, it (the copy) begins making a copy of **itself**, and so on. While this circuit is much larger and slower than the space-filling circuit, there is no need for a central control circuit, and as such the self-replicator may be more robust than an array of SFCs. This circuit requires 255x150 cells, and takes 19,381,504 clock cycles to execute a single replication sequence.

8. A guard wall, which surrounds a critical circuit and allows data to pass through the wall, but prohibits the cells inside from being reconfigured. If a cell tries to reconfigure one of the guard wall’s cells, that section of the guard wall is effectively shut down, and any exchange through that cell becomes impossible. A single piece of this wall is only 2x2 cells.

These are only some of the interesting circuits one can implement on the PIG. Again, in all these examples, there is no special hardware involved, other than a uniform array of homogeneous PIG cells which operate as described above.

The circuits described above are part of a growing set of building blocks which can be assembled into large self-reconfigurable circuits for executing parallel algorithms which require self-modification.

4 Application areas

While all of the above circuits have been thoroughly designed, simulated and studied on PIG simulators, they are really intended as building blocks for higher-level applications. One such application which has been studied is a highly parallel genetic algorithm for evolving digital circuits on the PIG. A new type of genetic algorithm called a Ringed GA (RGA) was developed to allow an O(1) evolutionary cycle, e.g., the time for each cycle of evaluation, selection and mating is independent of the population size. Execution of the RGA on a PIG was simulated for the evolution of a parity generator, 4-1 multiplexer and 3-bit counter. In each case, the algorithm was able to evolve a perfect circuit. Full details of the RGA and the experimental setup are available in [5].

While the RGA can not be implemented on a standard FPGA or other externally-controlled reconfigurable device (internal control is required to keep the mating time independent of population size), it uses only one of the building blocks described above (remote cell replication).

Other possible applications abound. For example, the expanding counter can also be made to contract after the higher-order bits are no longer needed. Expanding multipliers and dividers have been designed though not simulated. Together, these could be combined into a hardware management system, which dynamically adapts its circuits to changing problems. The space filling circuit is useful for fault handling and path finding, as well as bootstrapping a grid via external control. The self-replicating circuit is useful for autonomous bootstrapping, where a single seed is loaded into the grid, which then replicates and differentiates according to a global grid map. Guard walls have obvious potential for fault tolerant processing. It is hoped that work on specific applications, such as evolvable hardware, will lead to a better understanding of how to apply the PIG to other areas.

5 Conclusions and future work

While specialized massively parallel systems exist for solving specific types of problems, there is need for a truly general purpose massively parallel system. To be general purpose, such a system needs to be controllable at a very fine-grained level, suggesting the use of reconfigurable hardware. The need for massively parallel reconfiguration leads to the goal of having a self-reconfigurable system. Manufacturing and fault handling considerations impose the further requirement of an infinitely scalable architecture.

The PIG is a general purpose massively parallel system. While it is not intended as a replacement for traditional von Neumann systems, its capacity for self-reconfiguration, and its code/data duality make it well suited to a variety of tasks fundamentally different from those which a von Neumann machine does best.

While a PIG can be configured to solve a wide range of specific problems, one of its most important applications is for the general exploration of new concepts involving self-reconfigurability. Because of the PIG's distributed configuration control, it can be used to study not only parallel execution of algorithms in hardware, **but parallel reconfiguration of hardware**. Moreover, the PIG can implement circuits which create new circuits, which themselves create and modify other circuits. This is an extremely powerful capability which is not currently available to most algorithms. The RGA is one example of the use of this capability to execute an algorithm extremely efficiently.

A very small physical PIG has been built in cMOS and successfully tested. While fabrication errors rendered particular PIG cells unusable, the remaining cells and the overall grid generally remained fully functional, illustrating the inherent fault tolerance of the PIG's scalable architecture. A single-cell replication such as described above was successfully carried out on the physical system. Various other small circuits such as adders were also successfully tested on the system.

Most of the circuits which have been discussed are too large to run on this physical PIG, and as such, have been developed and tested on simulators. Unfortunately, such simulators are necessarily slower than a physical PIG, and as the amount of parallelism in a PIG circuit increases, this slowdown becomes greater. Hence the extremely parallel circuits which are best suited to the PIG are the hardest to study using simulators. This motivates the goal of constructing a large-scale physical PIG.

While the prototype PIG was built in silicon, it appears that building a very large (say one trillion cell) PIG would be prohibitively expensive, both in size and cost.

In contrast, there are emerging technologies such as nanotechnology [6] and molecular computing [7], which deal with densities many orders of magnitude beyond what can be achieved in silicon. Practitioners of these disciplines don't ask "How do we construct 10^{16} identical circuits in a small low-energy system?" but rather "What can we do with these kinds of circuit densities?" To utilize such densities, systems built with these technologies would probably be composed of large numbers of relatively simple circuits assembled into regular two- or three-dimensional arrays. This is precisely the kind of manufacturing required for the PIG.

If a large PIG can be built, it will be possible to work with larger circuits, and more ambitious applications can be explored. Some applications of interest include self-repairing systems, systems for evolving complex circuits, multi-modal systems which switch between multiple circuit configurations, and learning systems which analyze an algorithm's execution and synthesize hardware to capture its functionality.

Finally, we must note that the applications we've explored so far barely scratch the surface of the PIG's capabilities. The PIG represents a fundamentally different programming paradigm, and the types of problems it can efficiently solve are quite different from those we are used to thinking about in the von Neumann world. Until we have more experience working in the PIG paradigm, we cannot imagine most of the PIG's possible applications.

Acknowledgements

The author wishes to thank Larry Henry and Murali Raju, co-inventors of the PIG, for their inspiration, dedication and vision in the initial conception and design of the first PIG, as well as for many subsequent discussions. Thanks also to Jimmy Hargrove for many useful discussions on various aspects of the PIG. Finally, many thanks to Lisa Durbeck for numerous contributions to the PIG, including design and implementation of simulation tools, many helpful discussions, several profound insights, and her constant support of PIG research.

References

[1] S. Casselman, "Virtual Computing and The Virtual Computer," in *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, 1993.

[2] I. Kajitani et al, "A Gate-Level EHW Chip: Implementing GA Operations and Reconfigurable Hardware on a Single LSI," in M. Sipper, D. Mange and A. Pérez-Uribe editors, *Evolvable Systems: From Biology to Hardware*, pg. 1-12, Springer, 1998.

[3] H. Kitano, "Morphogenesis for Evolvable Systems," in E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware: The Evolutionary Engineering Approach*, pg. 99-117, Springer, 1996.

[4] http://www.patents.ibm.com/patlist?icnt=US&patent_number=5886537

[5] N. Macias, "Ring Around the PIG: A Parallel GA with Only Local Interactions Coupled with a Self-Reconfigurable Hardware Platform to Implement an O(1) Evolutionary Cycle for EHW," to appear in *The 1999 Congress on Evolutionary Computation.*

[6] E. Drexler, *Engines of Creation*, Anchor Press/ Doubleday, Garden City, NY, 1986.

[7] L. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," in *Science Vol. 266*, pg. 1021-1024, Nov 1994.