Appearing in *IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM 2004), April 20–23, 2004

Design Patterns for Reconfigurable Computing

André DeHon, Joshua Adams, Michael DeLorimier, Nachiket Kapre,
Yuki Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton
contact: <andre@cs.caltech.edu>
Department of Computer Science, 256-80
California Institute of Technology
Pasadena, CA 91125

Abstract

It is valuable to identify and catalog design patterns for reconfigurable computing. These design patterns are canonical solutions to common and recurring design challenges which arise in reconfigurable systems and applications. The catalog can form the basis for creating designs, for educating new designers, for understanding the needs of tools and languages, and for discussing reconfigurable design. Tying application and implementation lessons to the expansion and refinement of this catalog will make those lessons more relevant to the design community. In this paper, we articulate this role for design patterns in reconfigurable computing, provide a few example patterns, offer a starting point for the contents of the catalog, and discuss the potential benefits of this effort.

1 Introduction

As we have seen repeatedly in this conference, reconfigurable solutions can often be orders of magnitude faster or less expensive than conventional alternatives. Many designers in this community have become quite skilled at harnessing FPGA-based systems to solve hard computational problems. Good reconfigurable solutions are often quite different from the good sequential solutions familiar to most programmers. Building good reconfigurable designs requires an appreciation of the different costs and opportunities inherent in reconfigurable architectures. This leads us to a recurring question: *How do we teach programmers and designers to design good reconfigurable applications and systems?*

Certainly, we can show them what an FPGA is: what resources it has, how they are organized, what they cost in area, delay, and energy. We can point them to proceedings from 11 years of FCCM; there they will see over 200 papers explaining particular applications and systems—numerous examples of how people have done it before. From the basic components and the complete solutions, the best students will begin to generalize how to put together an application.

Is this the most efficient way to communicate the information? Is this approach the fastest path to achieving design competence or expertise? Does it reliably communicate the major lessons to all individuals?

Borrowing a popular *meme* from the object-oriented software [1] engineering community (who borrowed it from the architecture community [2]), we suggest that it is useful to identify and study *design patterns*. That is, it is useful to crystallize out common challenges which reconfigurable designs may encounter and the typical solutions which are used to address them. These solutions are design elements that suggest how we might organize parts of our overall solution. Rather than just studying the properties of FPGAs, this identifies techniques for dealing with the challenges they pose; rather than just studying whole applications, this focuses on the key elements of the solution which may be reusable in other settings. Studying and discussing design patterns gives us a way to talk directly about the design process and design elements, helping the whole community more readily share and assimilate design lessons.

In this paper, we advocate the study of design patterns. We identify a number of design patterns which have been used in this community and in the broader hardware and software design communities which may be relevant to reconfigurable systems. We provide an initial classification of these patterns into broad classes based on the problems they address. We make no claims of originality for the patterns presented here—we didn't invent them. Rather, we attempt to call them out and classify them so they are more readily apparent for the reconfigurable designer. We think of the list and classification presented here as a starting point—no doubt there are many patterns we have not seen or been able to crystallize into nameable entities. Even for the 89 patterns we have identified, there is no way we can treat them all in appropriate detail in this paper. Rather, we focus on introducing the ideas and providing references into the existing literature as a placeholder until there is time and space to treat them in the appropriate depth.

In the next section (Section 2), we further define design patterns and point to their progenitors. In Section 3 we include two design pattern descriptions to illustrate examples of reconfigurable design patterns. In Section 4, we introduce our classification of design patterns. We offer a few recommendations and lessons which come from an understanding of design patterns (Section 5), and close with thoughts about the next steps toward the development of a set of useful reconfigurable computing design patterns.

2 Background

2.1 What is a Design Pattern?

Simply put, a design pattern is a solution to a recurring problem. *e.g.* **[problem]** designs are often too big to fit onto the hardware available, so we **[solution]** time multiplex the large design onto the limited, available hardware. TIME MULTIPLEXING is a design pattern we use to address the problem of limited, fixed capacity for a particular reconfigurable platform.

Design patterns are elements of a design at an organizational level. That is, we certainly use leaf-cells and libraries as the building blocks for a design (*e.g.* adders, multipliers, multiplexers, CORDIC rotators, FIRs, FFTs), but this does not tell us how they should be put together, organized, and used, or even how they should be parameterized and selected for a design. Design patterns offer us organizing and structuring principles that help us understand how to put them together. Once we decide to use the TIME MULTIPLEXING pattern, then we can think about which leaf-cells should be instantiated and reused or which leaf-cells should be reconfigured in each time-cycle to realize the large graph.

2.2 Object-Oriented Software Engineering

The term "Design Patterns" has been popularized in recent years by the work of Gamma, Helm, Johnson, and Vlissides [1] [3] and others. They articulate the idea that writing in an object-oriented (OO) language is not sufficient, by itself, to produce good, reusable software. Rather, using the popular languages of the day (*e.g.* C++, Java, Smalltalk), there are many challenging requirements that arise during program design and modification, and it is not immediately obvious to the beginner how to achieve good modularity and reusability. Over time, experienced designers have learned common idioms and tricks that they use to address these challenges. Good designers tend to employ a few common patterns to address the recurring challenges. It is useful to identify these patterns and share them with others. By giving names to the solutions people have been re-inventing regularly, design patterns make it possible to talk at a higher level about object-oriented designs. Warned about the common hazards of OO design and armed with these higher level building blocks for avoiding them, the OO software engineer is in a better position to write more flexible and reusable code.

The design patterns which Gamma *et. al.* articulate [3] are somewhat narrowly focused for organizing OO software for flexibility and reuse. In particular, there are many useful idioms and patterns in the broader context of software design which they do not identify. We will be taking a broader view of design patterns here, attempting to identify patterns which solve more problems than simply code reuse and adaptability for change. In this sense, our design patterns are broadly in the theme of Christopher Alexander's work [2], which the OO design pattern community often cites as part of their inspiration.

2.3 Paradigms of Programming

Before the modern OO movement and before modern FPGAs, Robert Floyd articulated the value of identifying the *paradigms* of programming and listed "a pattern, exemplar, example" as the first definition for paradigm [4]. Floyd pointed out the value of identifying and studying the different paradigms (patterns) that allowed us to solve computing problems. Floyd, too, takes a broader view of patterns including, for example, STRUCTURED PROGRAMMING, DIVIDE-AND-CONQUER, and GENERATE-FILTER-ACCUMULATE in his list of programming paradigms. He advocates that the study of paradigms should be part of our systematic education of new programmers and that language design should strive to support the paradigms which are useful for programmers.

2.4 Pattern Description

One of the contributions of Gamma *et. al.* is to suggest a standardized format and set of contents for the description of a design pattern [3]. This stylization helps call attention to the key things one will typically need to know in order to understand and use a pattern. Their form includes:

- **Name** – a standard name for the pattern; by convention, we typeset pattern names in SMALL CAPS to distinguish them.
- **Intent** – What problem is this addressing? What is the pattern trying to do?
- **Motivation** – Why would you want to use this pattern?
- **Applicability** – When can this pattern be used? One distinction of patterns is that there are often different patterns for different, specialized contexts. What special property of the problem would make this pattern applicable or even preferred? What properties might make this pattern inappropriate?
- **Participants** – What are the component players in the pattern?
- **Collaborations** – How do the participants collaborate to solve the problem?
- **Consequences** – What are the trade-offs associated with using this pattern?
- **Implementation** – How would you implement this? What general lessons are there for implementing this pat-

tern? What pitfalls should be avoided? Are there hints for optimization?

- **Known Uses –** What are common examples of where this pattern has been used in a real system?
- **Related Patterns –** Which patterns are related to this one? How do they work together? When might we choose one pattern over the other?

3 Example Patterns

To be concrete about what design patterns are, we present design pattern descriptions for two common patterns. Again, we are not claiming to have invented any of these patterns. We are simply illustrating how these are described in a canonical manner so that they are readily available for a would be reconfigurable designer or as a reference for the experienced designer.

3.1 Coarse-Grained Time Multiplexing

3.1.1 Intent COARSE-GRAINED TIME-MULTIPLEXING allows a large design to be run on a smaller or fixed capacity platform, perhaps at the expense of design throughput or latency.

3.1.2 Motivation The capacity in any reconfigurable platform is fixed. When that fixed capacity is smaller than what would be required to implement a design fully spatially, then it is necessary to reuse the resources in time in order to realize the full design. Even when the fully spatial implementation of a design fits onto a platform, it may run too fast for an application, particular if the application has real-time interfacing requirements or limitations. Here, time multiplexing may allow the platform to implement more applications and features simultaneously or allow implementations with greater quality or precision.

3.1.3 Applicability The COARSE-GRAINED TIME-MULTIPLEXING pattern is intended for cases where the platform is not prepared to change configurations in just a few cycles. Rather, this pattern is intended for cases where it may take many thousands or millions of cycles to reconfigure the device.

Since reconfiguration is slow, this pattern is limited to cases where:

1. there are no feedback loops in the computational flow graph (ACYCLIC DATAFLOW GRAPH pattern),
2. the feedback loops can be contained within the capacity of the device, OR
3. the feedback loops are very slow, such that an upstream element need not be effected by downstream elements for thousands or millions of cycles

3.1.4 Participants We have a *computational graph* which we wish to implement on a *limited capacity platform*. The computational graph is divided into a number of *subgraphs*, each of which is capacity feasible for the platform.

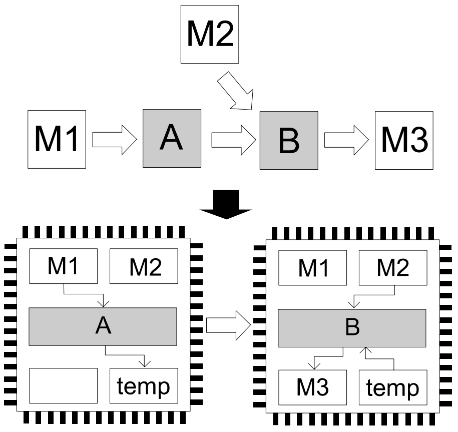


Figure 1: Time Multiplexing Pattern

During execution, a *control algorithm* coordinates the reconfiguration of the platform to implement the various subgraphs in time.

3.1.5 Collaborations The controller manages reconfiguration (SEQUENCER/CONTROLLER pattern), directing when the subgraphs are swapped onto the platform.

3.1.6 Consequences The slow reconfiguration of the platform is the most distinctive feature that must be addressed. Since this can take millions of cycles, one is usually forced to design applications so they can run for tens of millions of cycles in order to amortize out the fixed overhead associated with reconfiguration. Long run lengths in turn require that we employ large buffers to hold intermediate results between the computational subgraphs which are not running concurrently.

The fixed-size platform creates a bin into which we must pack the subgraphs. This can result in a certain amount of design fragmentation that prevents us from perfectly using device resources.

3.1.7 Known Uses Villasenor, Jones, and Schoner demonstrated that a video processing algorithm can be broken into suitable subgraphs and run on a small platform which is reconfigured several times during the computation of the entire video processing pipeline [5]. Villasenor *et. al.* also demonstrated a specialized target recognition system based on time multiplexing [6]. Eldredge and Hutchings demonstrate a neural network implementation which is broken into distinct phases and time multiplexed onto an FPGA platform [7]. Caspi *et. al.* [8] described a compute model based on COARSE-GRAINED TIME-MULTIPLEXING and several other patterns in order to automate design scaling.

3.1.8 Implementation The design must be broken into subgraphs which will fit onto the available platform. This has been done manually (*e.g.* [5], [7]). With the appropriate compute models, this can be automated [9].

Once the subgraphs exist, they must be sequenced onto the platform. Typically a conventional processor is used to issue commands to the platform to control reconfiguration (*e.g.* SEQUENCER/CONTROLLER pattern). In the simplest case, each subgraph is run for a fixed length of time before being swapped. However, for dynamic tasks, it is often useful to monitor buffers and dynamically identify events which motivate reconfiguration [9].

As noted above, reconfiguration time is a key issue. Steps which can reduce reconfiguration time are often very important. Parallel, on-chip memories for reconfiguration [8] [10] can reduce reconfiguration times to thousands of cycles. *Partial reconfiguration* can be used to reduce configuration time [11]. Configuration stream compression may also help reduce reconfiguration time (*e.g.* [12]).

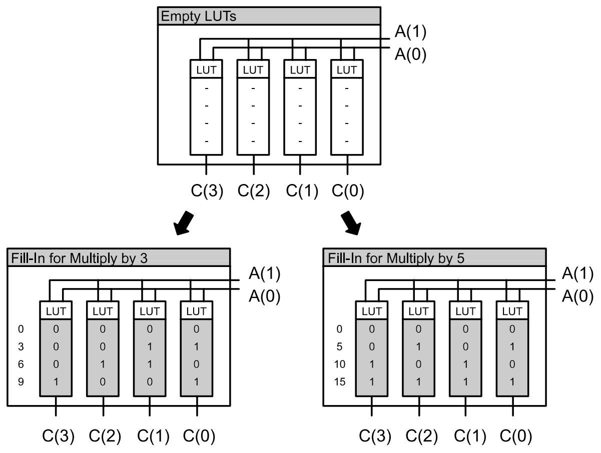
3.1.9 See Also Since configurations should be long running, COARSE-GRAINED TIME MULTIPLEXING often works with STREAMING DATA. QUEUES WITH BACK-PRESSURE are naturally employed between portions of the graph which are not co-resident on the platform, and they are often useful between co-resident portions to smooth out dynamic flow rate or delay effects. PARALLEL-PREFIX REDUCTIONS are natural for quickly determing when reconfiguration is appropriate [9].

3.2 Template Specialization

3.2.1 Intent Specialization patterns reduce space and/or time for a computation, minimizing the instantaneous computing requirements for the task. The TEMPLATE pattern is distinguished from other specialization patterns in that it minimizes the run-time work required to generate the specialized instance.

3.2.2 Motivation When you have early-bound data, that data can often be folded into the implementation in order to generate a more specific computation which requires less computation (fewer resources, shorter critical paths) than the generic problem. If the early-bound data remains constant for a sufficiently long time, we can implement the specialized computation instead of the generic computation and save area and/or time. This may allow us to place more computations into a fixed-capacity reconfigurable system or to fit the specialized problem when the generic problem does not fit onto the resources available.

For the TEMPLATE pattern specifically, we want to minimize the work required to set up the reconfigurable system for the specific instance. The more general specialization patterns may require that we invoke the full CAD flow (*e.g.* synthesis, placement, routing) in order to obtain the specialized mapping. Such invocations can be very time consuming, diminishing the performance benefits of specialization and forcing the techniques to only be viable when the early-bound data is known to remain constant for very long periods of time. The TEMPLATE pattern, in contrast, is much



(Shown with 2-LUTs for simplicity)

Figure 2: Template Pattern

lighter weight, making it viable for more modest epochs of bound data.

3.2.3 Applicability The TEMPLATE pattern is applicable when the specialization can be cast so that we only need to change the values of data in tables to adapt to the specific early-bound data; usually this means we simply program the LUTs which serve as programmable gates in the reconfigurable array. The interconnect between the LUTs must be able to remain the same across all instances.

3.2.4 Participants The basic design and connectivity is the *template cell*. A *template filler* will need to fill in the template (*i.e.* program the LUTs) with the values appropriate for each specialized instance.

3.2.5 Collaborations The CONSTRUCTOR pattern is a natural place for the template filler. That is, in a reconfigurable system with dynamic instance instantiation, an instance is created like an object in an object-oriented system (*e.g.* with a `new`) [8]. The constructor will take as arguments the specification of the behavior of this instance. Part of the constructor's function is to convert the specification into suitable LUT programming and apply that to the instantiated template cell.

Template specialization may also be used with slowly changing data. Here, the EXCEPTION pattern might be used to invoke an exception handler that would contain the template filler.

3.2.6 Consequences The optimization available is usually more limited than when a full specialization is performed. The template always has the same size. In many cases this size is larger than every specific instance. The inability to vary the interconnect is one reason the template may always be bigger than any particular instance.

The fact that the template is always the same size is good in that the specific instance specialized does not affect the capacity of the reconfigurable platform. It is bad in that the area and time could be smaller and faster for all instances and, perhaps, much smaller in favorable instances.

3.2.7 Known Uses

Multiply-by-Constant: Specialized multipliers are a widely-known example of the TEMPLATE pattern [13]. We can use $n+4$ 4-LUTs to multiply a 4-bit value by an n-bit constant simply using table lookup. We can multiply an m-bit value by an n-bit constant by performing $\frac{m}{4}$ such constant multiplies and adding up the resulting partial products. In contrast, it would take 4 4-LUTs to simply multiply together two non-constant 2-bit values.

The constant coefficient multiplier template requires $(n+4)\times\frac{m}{4}$ 4-LUTs for the table lookups, followed by an adder tree on $\frac{m}{4}$ inputs. The adder tree needs $\left(\frac{m}{4}-1\right)$ adds, each of which could be implemented with less than $n+4$ augmented LUTs with carry logic. Consequently, the table lookups and the adder tree each require roughly $\frac{n\cdot m}{4}$ 4-LUTs for a total of $\frac{n\cdot m}{2}$ 4-LUTs. In contrast, for a generic $m\times n$ multiplier the adder tree after generating partial products will require m, n-bit adders, for a total of $m\cdot n$ 4-LUTs. Partial product generation will add additional cells. This gives at least factor of two reduction in area and a factor of two reduction in time versus using a generalized multiplication routine.

To “reconfigure” the template cell for the a new constant, we simply need to fill in the values in the lookup tables (See Fig. 2). In FPGAs where the LUTs can be used as read/write memories (*e.g.* Xilinx 4000 [14] or Virtex [15] series), this can be configured at the user level and there is no need to have access to the device bitstream for reconfiguration and no need to reload the device configuration bits which are not part of the template and remain unchanged.

Note that we trade area for flexibility with this pattern. Multiplications by many specific constants can be performed in even less area than this; for example, multiplication by 0x48 requires a single addition. Even the worst-case constant can be implemented with a specific adder-tree which is smaller than the template multiplication.

String and Sequence Matching: For string and sequence matching problems, we can program a 4-LUT to match a specific 4-bit input. When one value is a constant we compare 4-bits per 4-LUT and need an $\frac{n}{4}$-bit product tree. In contrast, if we had to compare two variable n-bit sequences, we would only be able to compare 2-bits of both sequences with each 4-LUT, then we would need an $\frac{n}{2}$-bit product tree. Further, with the constant programmed into the 4-LUTs, no registers are needed to hold the match value. This can easily be exploited in string matching, when the maximum string length is fixed. Genetic sequence matching is essentially the same problem [16]. This can be extended to perform broader (*e.g.* wild-card, match one character of a set) matches as well.

Content Addressable Memories: In a similar manner, a sequence of 4-LUTs can implement the match signal for a content-addressable memory. By filling in the 4-LUT contents, we define the content match (*e.g.* [17]).

3.2.8 Implementation

As noted for the multiply, if the LUTs have a mode that allows the values to be both read and written, one can simply build the write path into the FPGA design allowing direct write into the memory.

For devices with direct write into configuration memory (*e.g.* XC6000 [18]), a processor or controller can use the configuration port directly to efficiently rewrite the tables.

If the full bitstream must be reloaded, including interconnect (even if it is a portion of the bitstream as in the Virtex/Virtex-II series [19]), and the location of the LUT programming in the bitstreams is known, then one can edit the bitstream, overwriting only the LUT locations participating in the template and leaving the interconnect intact.

3.2.9 See Also

As noted above, the CONSTRUCTOR or EXCEPTION patterns may be useful for containing the template filler.

The TEMPLATE pattern has a more restricted structure than the PARTIAL EVALUATION or CUSTOM INSTANCE GENERATION patterns. The TEMPLATE pattern requires minimum computation and handling to accommodate a new specialized instance, whereas the other patterns may require heavier CAD optimization. The TEMPLATE pattern is of fixed size, while other specialization optimizations may produce smaller designs at the expense of needing to perform run-time logic packing.

4 Design Pattern Classification Sketch

In this section we present a first pass at a catalog of important patterns and group them into broad classes based on the problems they address. This list is, no doubt, incomplete and we invite input on important patterns and classes which should further fill out the catalog of important reconfigurable computing patterns. In lieu of providing a pattern writeup for each pattern listed here, we reference sample papers where the pattern appears prominently. Patterns which require at least *load-time configuration* are marked with an icon of a loading CD ( ), and patterns which require *run-time reconfiguration* are marked with a running man icon ( ).

- *Patterns for Area-Time Tradeoffs* — one of the biggest problems we have to address with reconfigurable machines is fitting the logical design to the hardware; further, for designs to scale efficiently to larger reconfigurable platforms over time (*e.g.* larger FPGAs), we need to be able to change the area-time tradeoff without completely re-implementing the application. Consequently, patterns that helps us trade off area and time are essential to reusable reconfigurable designs.

1. Sequential vs. Parallel Implementation (hardware/software partitioning) [20]
2. Fine-Grained Time Multiplexing [21] [22] [23]
3. Coarse-Grained Time Multiplexing (Section 3.1)
4. Common Element Sharing for Regular Graphs (*e.g.* Cellular Automata) [24]
5. Common Operator Sharing for General Graphs (*e.g.* boolean net, neural net, VLIW) [25]
6. Synthesis Objective Function Tradeoffs [26]
7. Scheduled Operator Sharing [27]
8. Datapath Serialization [28] [29] [30]

- *Patterns for Expressing Parallelism* — performance in reconfigurable systems often comes from parallelism. Consequently, patterns that allow us to parallelize the computation are important. Parallelism patterns could be considered a focused subset of the area-time tradeoff patterns.

9. Extract Implicit Parallelism from Control Flow [20]
10. Dataflow [31]
11. Synchronous Dataflow [32]
12. Functional
13. Acyclic Dataflow Graph
14. Data Parallel [33] [34] [35]
15. Multithreaded [36] [37] [8]
16. Futures [38]

- *Patterns for Implementing Parallelism* — once we have captured or discovered the parallelism, we employ various patterns to implement it.

17. If-Conversion and Predicated Execution [20]
18. SIMD [39]
19. Vector [40] [41]
20. Parallel Prefix, Reductions, Scans [33] [42] [35]
21. Communicating FSMDs [27]
22. Datapath Duplication
23. Direct Implementation of Graph [43] [44] [45]

- *Patterns for Processor-FPGA Integration* — often we find it useful to use FPGAs and processors together. This too, may be a subset of area-time tradeoffs as we put the performance critical portion on the FPGA and implement the rest of the problem on the processor. Over the years we have seen several patterns for how we might integrate and coordinate the processors and the FPGAs.

24. Interfacing/IO [46] [47]
25. Co-processor [48]
26. Streaming Co-processor [48] [40]
27. Instruction Augmentation [49] [50] [51]
28. Sequencer/Controller [52] [8]

- *Patterns for Common-Case Optimization* — the general case required to handle every conceivable possibility is usually large and potentially slow. However, what happens most of the time is often simpler and can be done quickly with little hardware. A particularly useful set of hardware-software tradeoffs is to implement the common-case spatially in minimal hardware and have an escape mechanism to handle the less common cases which are needed for completeness. Often the uncommon cases are handled by a processor in software (see *processor-FPGA Integration* patterns above); sometimes they are simply handled by slower logic. We are familiar with many of these from conventional processor architecture (*e.g.* [36]), but the ideas are much more general and may be even more important in building efficient reconfigurable systems. Most of these optimizations are not suitable for real-time applications. These patterns are all most useful in a run-time reconfiguration setting, but there are uses of these patterns even when the FPGA never reconfigures.

29. Caching and Memoization (data, results, instructions, instances) [50] [20] [53]
30. Common/Simple Hardware with Escape (*e.g.* Translation Lookaside Buffer (TLB), Top-of-stack Cache, Overflowing Queue [8])
31. Exceptions
32. Trace Scheduling/Exceptional Exit [20]
33. Prediction (branch, value)
34. Speculative Execution [20]
35. Parallel Verifier [54]

- *Patterns for (Re)Using Hardware Efficiently* — hardware is efficient when it can be reused very rapidly; that is, we want to run it at a high clock rate rather than letting the resource sit idle.

36. Pipelining
37. Wave Pipelining [55]
38. Retiming [56] [57] [58]
39. C-slow (interleaved, data parallel, data independent multithreading) [56] [42] [59]
40. Software Pipelining [60] [61] [62]

- *Patterns for Specialization* — one of the advantages of reconfigurable machines is that they can be configured to solve exactly the problem at hand, rather than needing to be general enough to solve any problem. This allows us to fold constants into the FPGA configuration. This specialization can often go a long way toward closing the gap between programmable, reconfigurable designs and full-custom designs [63].

41. Template (Section 3.2)
42. Worst-Case Footprint
43. Constructive Instance Generator [64] [65]

44. Instance Generator [6] [43]
45. Partial Evaluation [66] [67]
46. Constructor [8] [68]

- *Patterns for Partial Reconfiguration* — full FPGA reconfiguration can be very slow. If we are going to change configuration during execution as part of time multiplexing or specialization, we can often save some time by reconfiguring only the minimum subset of the FPGA necessary to effect the change.

47. Isolate Varying Design from Fixed Portion [11]
48. Constant Fill-in [11] (related to TEMPLATE above)
49. Unify Datapath Space/Structure for Variants [11]
50. One-Dimensional Function Space [51]
51. Fixed-Size and Standardized-IO Pages [69] [8]
52. Bus Interface [70]

- *Patterns for Expressing Communications* – a key component of any parallel implementation will be data communication between portions of the computation.

53. Streaming Data [71] [8] [20]
54. Message Passing [72] [73] [74] [75]
55. Remote-Procedure Call [76] [77]
56. Shared Memory [36] [78]

- *Patterns for Synchronization* — coordinating the behavior of parallel operators is a recurring problem in all forms of parallelism; this is particularly important when operations can take variable time and when we hope to reuse the design in different technologies where the delays may vary.

57. Synchronous Clocking [79]
58. Asynchronous Handshaking (self-timed) [79] [80]
59. Tagged Data Presence [81] [82] [8]
60. Queues with Backpressure [8] (perhaps including windowed advertisement [83])
61. H-Tree

- *Patterns for Efficient Layout and Communications* — for spatial computations, interconnect is one of the biggest consumers of area, delay, and energy. Further, computing good placements is a computationally hard problem that modern EDA tools only approximate poorly. Regular, constructive layout strategies that allow us to keep wires short can be important to achieving high clock rates and efficient embeddings of designs onto reconfigurable arrays.

62. Cellular Automata [84] [85] [86] [24]
63. Systolic, Semi-Systolic [87] [42] [39]
64. Fixed-Radius Communication [88]
65. Folded/Interleaved Torus [89]
66. Tree-of-Meshes and Fold-and-Squash Layouts [90] [91]

- *Patterns for Implementing Communication* — interconnect is often the dominant area, delay, and energy consumer in a programmable system. Consequently, we want to make sure we use the interconnection wires and switches efficiently. Further, interconnect requirements can often be the biggest limitation determining when a design fits onto a system. Consequently, we have developed several patterns for using and reusing wires efficiently.

67. Shared Bus
68. Token Ring
69. Reconfigurable Interconnect
70. Pipelined Interconnect [57]
71. Serialized Communication [92]
72. Time-Switched Routing [93]
73. Circuit-Switched Routing [94]
74. Packet-Switched Routing [95]

- *Value-Added Memory Patterns* — memory bandwidth is a key performance limiter in many systems. This arises in part because conventional systems place a large amount of data behind a limited-bandwidth interconnect and serialize communication. We can often use a little bit of logic close to the memory to reduce the total memory bandwidth requirements and memory latency. CACHING (included under *Patterns for Common-Case Optimization*) might also fit in this group.

75. Address Generator (*e.g.* stride, zig-zag, FFT) [96]
76. Content-Addressable Memory [97]
77. Read-Modify-Write Operations [98]
78. Data Filter
79. Multiple Indirection/Redirection (forwarding) [99]
80. Scan-Select-Reorganize [100] [101]
81. Data Compression or Digest [102] [103]
82. Stack, Queue [104]
83. Data Structure [101] [105]

- *Number Representation Patterns* — fine-grained reconfigurable architectures allow us to use just as little or as much precision and representation as necessary for the problem; they further allow us to select the most efficient representation for the distribution of computations we will need. This allows us to save space when the design can use less precision than a fixed or coarse-grained architecture and allows us to efficiently allocate more precision when the application needs it.

84. Parameterize Datapath Operators by {bitwidth, decimal point, signedness, exponent} [64] [106] [107] [108]

85. Redundant Number Systems [109] [110]
86. Distributed Arithmetic [111]
87. Abstract Operator (implementation specialized to match representation to operation sequence) [68]
88. Stochastic Bit-Serial Computation [112]
89. Bit-Slice Datapath [113]

As we move forward, patterns for energy minimization will become important. Also, as we move to smaller features sizes, we will begin to elaborate patterns for defect and fault tolerance.

5 Lessons and Recommendations

For Instructors As stated at the outset, design patterns can be an import part of how we explicitly teach people to design reconfigurable systems. They draw attention to common problems and share known solutions. They are organized to present the key lessons from various applications.

For Developing Designers The pattern encyclopedia can serve as a valuable reference source for design solutions. The intent and motivation section of each pattern helps the developing designer quickly find relevant solutions. Has someone else encountered this problem before? How did they solve it?

For Application Developers and Authors As the proceedings of this conference attests, many application developers are motivated to share their experiences with others. Patterns can provide a useful organizing principle for communicating the lessons of the design. What patterns were employed in developing your solution? Are they all old patterns? Do you use a new design technique that might be broadly applicable to others—a new pattern? What problem did you need to solve? Where you have employed familiar patterns, does your experience suggest new ideas for implementations or warn of pitfalls not previously documented?

For Tools, Language, and Programming System Designers What patterns do you support? How does your language, tool, or system support patterns which are known? What patterns do you automate? Is there a non-obvious way to support some pattern with your system? If you are a tool or language developer looking for a place to contribute, are there patterns that are not well supported by existing tools or languages? Can you provide a system that provides better integrated support for the collaborative use of a large fraction of the patterns people are identifying as useful?

6 What's Next?

More ideas? The list we offer here is a starting point. We are certain there are many things already known which we have not been able to identify or articulate. We invite input and suggestions so that we can help create a more complete catalog to educate ourselves and the community.

Fill In Details In this forum, we were only able to catalog most patterns. Each of the patterns in Section 4 needs to be developed into descriptions at least as detailed as the ones in Section 3, and the examples in Section 3 could stand further elaboration.

We will be collecting ideas and references for patterns and summarizing them on our web site: <`http://www.cs.caltech.edu/research/ic/design_patterns/`>.

Acknowledgments

Primary support for this project has come from NSF grant number CCR-0205471 and a grant from the Okawa Foundation. Stephan Lavavej was involved in early discussions which helped catalyze our study of patterns. Randy Harr, Steve Trimberger, Kees Vissers, John Wawrzynek, and Nicholas Weaver provided valuable feedback on early versions of these ideas and suggested pattern ideas.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, July 1993, pp. 406–431.

[2] C. Alexander, *The Timeless Way of Building*. Oxford University Press, 1979.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison Wesley, 1995.

[4] R. W. Floyd, "The Paradigms of Progamming," *Communications of the ACM*, vol. 22, no. 8, pp. 455–460, 1979.

[5] J. Villasenor, C. Jones, and B. Schoner, "Video Communications using Rapidly Reconfigurable Hardware," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, pp. 565–567, December 1995.

[6] J. Villasenor, B. Schoner, K.-N. Chia, and C. Zapata, "Configurable Computer Solutions for Automatic Target Recognition," in *FCCM*. IEEE, April 1996, pp. 70–79.

[7] J. G. Eldredge and B. L. Hutchings, "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration," in *FCCM*, April 1994, pp. 180–188.

[8] E. Caspi, M. Chu, R. Huang, N. Weaver, J. Yeh, J. Wawrzynek, and A. DeHon, "Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial," <http://www.cs.berkeley.edu/projects/brass/documents/score_tutorial.html>, short version appears in FPL'2000 (LNCS 1896), 2000.

[9] Y. Markovskiy, E. Caspi, R. Huang, J. Yeh, M. Chu, J. Wawrzynek, and A. DeHon, "Analysis of QuasiStatic Scheduling Techniques in a Virtualized Reconfigurable Machine," in *FPGA*, February 2002, pp. 196–205.

[10] S. Perissakis, Y. Joo, J. Ahn, A. DeHon, and J. Wawrzynek, "Embedded DRAM for a Reconfigurable Array," in *Proceedings of the 1999 Symposium on VLSI Circuits*, June 1999.

[11] J. D. Hadley and B. Hutchings, "Design Methodologies for Partially Reconfigured Systems," in *FCCM*, April 1995, pp. 78–84.

[12] Z. Li and S. Hauck, "Configuration Compression for Virtex FPGAs," in *FCCM*, 2001.

[13] K. D. Chapman, "Fast Integer Multipliers fit in FPGAs," *EDN*, vol. 39, no. 10, p. 80, May 12 1993.

[14] *The Programmable Logic Data Book*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, 1993.

[15] *Xilinx Virtex 2.5V Field Programmable Gate Arrays*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, April 2001, dS003 <http://direct.xilinx.com/bvdocs/publications/ds003.pdf>.

[16] E. Lemoine and D. Merceron, "Run Time Reconfiguration of FPGA for Scanning Genomic Databases," in *FCCM*, April 1995, pp. 90–98.

[17] *Designing Flexible, Fast CAMs with Virtex Family FPGAs*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, September 1999, xAPP 203 <http://www.xilinx.com/bvdocs/appnotes/xapp203.pdf>.
[18] *XC6200 FPGA Advanced Product Specification*, Version 1.0 ed., Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 1996.
[19] *Xilinx Virtex-II Platform FPGAs Data Sheet* , Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, October 2003, dS031 <http://direct.xilinx.com/bvdocs/publications/ds031.pdf>.
[20] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," *IEEE Computer*, vol. 33, no. 4, pp. 62–69, April 2000.
[21] C. Iseli and E. Sanchez, "Spyder: A Reconfigurable VLIW Processor using FPGAs," in *FCCM*, April 1993, pp. 17–24.
[22] A. DeHon, "DPGA Utilization and Application," in *FPGA*, February 1996, pp. 115–121.
[23] S. Cadambi, J. Weener, S. Goldstein, H. Schmit, and D. Thomas, "Managing pipeline-reconfigurable FPGAs," in *FPGA*, 1998, pp. 55–64.
[24] T. Kobori, T. Maruyama, and T. Hoshino, "A Cellular Automata System with FPGA," in *FCCM*, 2001.
[25] M. Denneau, "The Yorktown Simulation Engine," in *19th Design Automation Conference*. IEEE, 1982, pp. 55–59.
[26] J. Cong and Y. Ding, "On Area/Depth Trade-Off in LUT-Based FPGA Technology Mapping," *IEEE Transactions on VLSI Design*, vol. 2, no. 2, pp. 137–148, June 1994.
[27] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.
[28] P. Denyser and D. Renshaw, *VLSI Signal Processing: A Bit-Serial Approach*. Addison-Wesley Publishing Company, 1985, pp. 1–28.
[29] T. Isshiki and W. W.-M. Dai, "High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systems," in *FPGA*. ACM, February 1995, pp. 167–173.
[30] A. F. Tenca and M. D. Ercegovac, "A Variable Long-Precision Arithmetic Unit Designed for Reconifgurable Coprocessor Architectures," in *FCCM*, 1998, pp. 216–225.
[31] R. Rinker, M. Carter, A. Patel, M. Chawathe, C. Ross, J. Hammes, W. A. Najjar, and W. Böhm, "An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware," *IEEE Transactions on VLSI Systems*, vol. 9, no. 1, pp. 130–139, 2001.
[32] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.
[33] W. D. Hillis and G. L. Steele, "Data Parallel Algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, December 1986.
[34] M. Gokhale and R. Minnich, "FPGA Computing in a Data Parallel C," in *FCCM*, April 1993, pp. 94–101.
[35] S. Guccione and M. Gonzalez, "A Data-Parallel Programming Model for Reconfigurable Architectures," in *FCCM*, April 1993, pp. 79–87.
[36] J. Hennessy and D. Patterson, *Computer Architecture a Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, Inc., 1996.
[37] C. A. R. Hoare, *Communicating Sequential Processes*, ser. International Series in Computer Science. Prentice-Hall, 1985.
[38] R. H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transaction on Programming Languages and Systems*, vol. 7, no. 4, pp. 501–538, 1985.
[39] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*. 10662 Los Vasqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1264: IEEE Computer Society Press, 1996.
[40] J. A. Jacob and P. Chow, "Memory Interfacing and Instruction Specification for Reconfigurable Processors," in *FPGA*, February 1999, pp. 145–154.
[41] M. Weinhardt and W. Luk, "Pipeline Vectorization for Reconfigurable Systems," in *FCCM*, 1999, pp. 52–62.
[42] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, Inc., 1992.
[43] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean Satisfiability with Configurable Hardware," in *FCCM*, April 1998, pp. 186–195.
[44] J. Babb, M. Frank, and A. Agarwal, "Solving Graph Problems with Dynamic Computational Structures," in *Proceedings of SPIE: High-Speed Computing, Digital Signal Processing, and Filtering Using reconfigurable Logic*, vol. 2914, November 1996, pp. 225–236.
[45] M. Abramovici and P. Menon, "Fault Simulation on Reconfigurable Hardware," in *FCCM*, April 1997, pp. 182–190.
[46] M. Shand, "Flexible Image Acquisition using Reconfigurable Hardware," in *FCCM*, 1995, pp. 125–134.
[47] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture," in *FCCM*, April 1998, pp. 28–37.
[48] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *FCCM*. IEEE, April 1997, pp. 12–21.
[49] P. Athanas and H. F. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11–18, March 1993.
[50] R. Razdan and M. D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*. IEEE Computer Society, November 1994, pp. 172–180.
[51] M. J. Wirthlin and B. L. Hutchings, "A Dynamic Instruction Set Computer," in *FCCM*, April 1995.
[52] W. Luk, N. Shirazi, and P. Y. K. Cheung, "Compilation Tools for Run-Time Reconfigurable Designs," in *FCCM*, April 1997, pp. 56–65.
[53] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera Reconfigurable Functional Unit," in *FCCM*, April 1997, pp. 87–96.
[54] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *MICRO*, 1999, pp. 196–207.
[55] E. Boemo, S. López-Buedo, and J. M. Meneses, "The Wave Pipeline Effect on LUT-based FPGA Architectures," in *FPGA*, February 1996, pp. 45–50.
[56] C. Leiserson, F. Rose, and J. Saxe, "Optimizing Synchronous Circuitry by Retiming," in *Third Caltech Conference On VLSI*, March 1983.
[57] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon, "HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array," in *FPGA*, February 1999, pp. 125–134.
[58] D. P. Singh and S. D. Brown, "The Case for Registered Routing Switches in Field-Programmable Gate Arrays," in *FPGA*, February 2001, pp. 161–169.
[59] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-Placement C-slow Retiming for the Xilinx Virtex FPGA," in *FPGA*, 2003, pp. 185–194.
[60] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software Pipelining," *ACM Computing Surveys*, vol. 27, no. 3, pp. 367–432, September 1995.
[61] T. J. Callahan and J. Wawrzynek, "Adapting Software Pipelining for Reconfigurable Computing," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000.
[62] G. Snider, "Performance-Constrained Pipelining of Software Loops onto Reconfigurable Hardware," in *FPGA*, 2003, pp. 177–186.
[63] A. DeHon, "The Density Advantage of Configurable Computing," *IEEE Computer*, vol. 33, no. 4, pp. 41–49, April 2000.
[64] M. Gokhale and E. Gomersall, "High Level Compilation for Fine Grained FPGAs," in *FCCM*, April 1997, pp. 165–173.
[65] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek, "Object Oriented Circuit-Generators in Java," in *FCCM*, April 1998, pp. 158–166.

[66] S. Singh, J. Hogg, and D. McAuley, "Expressing Dynamic Reconfiguration by Partial Evaluation," in *FCCM*, April 1996, pp. 188–194.

[67] Q. Wang and D. M. Lewis, "Automated Field-Programmable Compute Accelerator Design Using Partial Evaluation," in *FCCM*, April 1997, pp. 145–154.

[68] P. N. Pontus Äström, Stefan Johansson, "Design Patterns for Hardware Datapath Library Design," in *Proceedings of the Swedish System on a Chip Conference*, 2001.

[69] G. Brebner, "The Swapable Logic Unit:a Paradigm for Virtual Hardware," in *FCCM*, April 1997, pp. 77–86.

[70] D. Lim and M. Peattie, *Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, May 2002, xAPP 290 <http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf>.

[71] V. M. Bove, Jr. and J. A. Watlington, "Cheops: A Reconfigurable Data-Flow System for Video Processing," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 5, no. 2, pp. 140–149, April 1995.

[72] C. L. Seitz, "The Cosmic Cube," *CACM*, pp. 22–33, January 1985.

[73] T. v. Eicken *et al.*, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th Annual Symposium on Computer Architecture*, Queensland, Australia, May 1992.

[74] M. Snir and W. Gropp, *MPI: The Complete Reference*, 2nd ed. MIT Press, 1998.

[75] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, "Implementing an API for Distributed Adaptive Computing Systems," in *FCCM*, 2001.

[76] B. J. Nelson, "Remote Procedure Call," Xerox Palo Alto Research Center, CSL 81-9, 1981.

[77] M. Budiu, M. Mishra, A. Bharambe, and S. C. Goldstein, "Peer-to-Peer Hardware-Software Interfaces for Reconfigurable Fabrics," in *FCCM*, 2002.

[78] D. E. Culler and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaughmann, 1999.

[79] C. Seitz, *Introduction to VLSI Systems*. Addison-Wesley, 1980, ch. 7: System Timing.

[80] M. Budiu and S. C. Goldstein, "Compiling Application-Specific Hardware," in *FPL*, 2002, pp. 853–863.

[81] B. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," in *Proceedings fo the Symposium on Real-Time Signal Processing*, 1981, pp. 241–248.

[82] Arvind, R. S. Nikhil, and K. K. Pingali, "I-Structures: Data Structures for Parallel Computing," in *Procceedings of the Workshop on Graph Reduction (Springer-Verlag Lecture Notes in Computer Science 279)*, Sept. 1986.

[83] E. J. Postel, "Transmission Control Protocol – DARPA Internet Program Protocol Specification," USC/ISI, Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, California, 90291, RFC 793, September 1981.

[84] J. von Neumann, *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966, compiled by Arthur W. Burks.

[85] M. Gardner, "The Fantastic Combinations of John Conway's New Solitaire Game "life"," *Scientific American*, vol. 223, pp. 120–123, October 1970.

[86] G. Milne, P. Cockshott, G. McCaskill, and P. Barrie, "Realising Massively Concurrent Systems on the SPACE Machine," in *FCCM*, April 1993, pp. 26–32.

[87] H. T. Kung, "Why Systolic Architectures?" *IEEE Computer*, vol. 15, no. 1, pp. 37–46, January 1982.

[88] B. V. Herzen, "Signal Processing at 250 MHz using High-Performance FPGA's," in *FPGA*, February 1997, pp. 62–68.

[89] W. J. Dally and C. L. Sietz, *The Torus Routing Chip*, ser. Distributed Computing. Springer-Verlag, 1986, vol. 1, pp. 187–196.

[90] S. Bhatt and F. T. Leighton, "A Framework for Solving VLSI Graph Layout Problems," *Journal of Computer System Sciences*, vol. 28, pp. 300–343, 1984.

[91] R. I. Greenberg and C. E. Leiserson, "A Compact Layout for the Three-Dimensional Tree of Meshes," *Applied Math Letters*, vol. 1, no. 2, pp. 171–176, 1988.

[92] *RocketIO X Transceiver User Guide*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, December 2002, uG035 <http://www.xilinx.com/bvdocs/userguides/ug035.pdf>.

[93] J. Babb, R. Tessier, and A. Agarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators," in *FCCM*, April 1993, pp. 142–151.

[94] A. DeHon, F. Chong, M. Becker, E. Egozy, H. Minsky, S. Peretz, and T. F. Knight, Jr., "METRO: A Router Architecture for High-Performance, Short-Haul Routing Networks," in *ISCA*, May 1994, pp. 266–277.

[95] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks," in *Design Automation Conference*, 2001, pp. 684–689.

[96] V. M. Bove, Jr., M. Lee, C. McEniry, T. Nwodoh, and J. Watlington, "Media Processing with Field-Programmable Gate Arrays on a Microprocessor's Local Bus," in *Proceedings of SPIE Media Processors*, vol. 3655, 1999.

[97] S. A. Guccione, D. Levi, and D. Downs, "A Reconfigurable Content Addressable Memory," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2000, pp. 882–889.

[98] M. F. Deering, S. A. Schlapp, and M. G. Lavelle, "FBRAM: A new Form of Memory Optimized for 3D Graphics," in *Proceedings of SIGGRAPH*, 1994, pp. 167–174.

[99] D. A. Moon, "Architecture of the Symbolics 3600," in *ISCA*, 1985, pp. 76–83.

[100] P. Diniz and J. Park, "Data Search and Reorganization Using FPGAs: Application to Spatial Pointer-based Data Structures," in *FCCM*, 2003, pp. 207–217.

[101] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: a Model of Computation for Intelligent Memory," in *ISCA*, June 1998.

[102] F. Douglis, "The Compression Cache: Using On-line Compression to Extend Physical Memory," in *Proceedings of the Winter USENIX Conference*, 1993, pp. 519–529.

[103] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The Case for Compressed Caching in Virtual Memory Systems," in *Proceedings of the Summer USENIX Conference*, 1999, pp. 101–116.

[104] C. Leiserson, "Systolic Priority Queues," Carnegie-Mellon University, Pittsbugh, Pennsylvania 15213, CMU-CS-TR 115, April 1979.

[105] O. Mencer, Z. Huang, and L. Huelsbergen, "HAGAR: Efficient Multicontext Graph Processors," in *FPL*, 2002, pp. 915–924.

[106] J. Liang, R. Tessier, and O. Mencer, "Floating Point Unit Generation and Evaluation for FPGAs," in *FCCM*, 2003, pp. 185–194.

[107] R. Matoušek, M. Tichý, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman, "Logarithmic Number System and Floating Point Arithmetics on FPGA," in *FPL*, 2002, pp. 627–636.

[108] S. Coric, M. Lesser, E. Miller, and M. Trepanier, "Parallel-Beam Backprojection: An FPGA Implemenation Optimzied for Medical Imaging," in *FPGA*, 2003, pp. 217–226.

[109] A. R. Omondi, *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*. Prentice-Hall, 1994.

[110] Y. Li and W. Chu, "Implementation of Single Precision Floating Point Square Root on FPGAs," in *FCCM*, 1997, pp. 226–232.

[111] L. Mintzer, "FIR Filters with Field-Programmable Gate Arrays," *Journal of VLSI Signal Processing*, vol. 6, pp. 119–127, 1993.

[112] M. van Daalen, P. Jeavons, and J. Shawe-Taylor, "A Stochastic Neural Architecture that Exploits Dynamically Reconfigurable FPGAs," in *FCCM*, April 1993, pp. 202–211.

[113] A. Koch, "Structure Design and Implemenation — A Strategy for Implementing Regular Datapaths on FPGAs," in *FPGA*, February 1996, pp. 151–157.

		Purpose	
Class	**Subclass**	Expression	Implementation
Area-Time Tradeoffs	Basic		Sequential vs. Parallel
			Fine-grain Time-Multiplexing
			Coarse-grain Time-Multiplexing
			Element Share Regular Graphs
			Operator Share General Graphs
			Synthesis Objective
			Scheduled Operator Sharing
			Datapath Sizing and Serialization
	Parallel	Extract Control Flow	If-Conversion/Predication
		Dataflow	Parallel Prefix, Reduce, Scan
		Synchronous Dataflow	SIMD
		Acylic Dataflow Graph	Vector
		Functional	Datapath Duplication
		Data Parallel	Communicating FSMDs
		Multithreaded	Direct Implementation of Graph
		Futures	
	Processor-FPGA		Interfacing/IO
			Co-processor
			Streaming Co-processor
			Instruction Augmentation
			Sequencer/Controller
	Common-Case		Caching
			Simple Hardware with Escape
			Exception
			Trace-Schedule/Exceptional Exit
			Prediction
			Speculation
			Parallel Verifier
Reducing Area or Time	Reuse Hardware		Pipelining
			Wave Pipelining
			Retiming
			C-Slow
			Software Pipelining
	Specialization	Constructor	Template
			Worst-Case Footprint
			Constructive Instance Generator
			Instance Generator
			Partial Evaluation
	Partial Reconfiguration		Isolate Fixed/Varying
			Constant Fill-in
			Unify Datapath Variants
			1D Function Space
			Fixed-Size and Std. IO Page
			Bus Interface
Communications	Basic	Streaming Data	Shared Bus
		Message Passing	Token Ring
		Remote-Procedure Call	Reconfigurable Interconnect
		Shared Memory	Pipelined Interconnect
			Serialized Communcations
			Time-Switched Routing
			Circuit-Switched Routing
			Packet-Switched Routing
	Layout	Cellular Automata	
		Systolic	
		Semi-Systolic	Fixed-Radius Communication
			Folded/Interleaved Torus
			Tree-of-Meshes and Fold-and-Squash
	Synchronization		Synchronous Clock
			Asynchronous Handshaking
			Tagged Data Presence
			Queues with Back Pressure
			H-Tree
Memory	Value-Added		Address Generator
			Content-Addressable Memory
			Read-Modify-Write
			Data Filter
			Indirection/Redirection
			Scan-Select-Reorganize
			Data Compression/Digest
			Stack, Queue
			Data Structure
Numbers and Functions	Representation	Abstract Operators	Parameterize Datapath Operators
			Redundant Number System
			Distributed Arithmetic
			Stochastic Bit-Serial Computation
			Bit-Slice Datapath

Table 1: Pattern Role and Classification Summary

Web links for this document: <http://www.cs.caltech.edu/research/ic/abstracts/despat_fccm2004.html>