FPGA '96 -- ACM/SIGDA Fourth International Symposium on FPGAs
February 11-13, 1996, Monterey, CA

DPGA Utilization and Application

André DeHon
andre@mit.edu
MIT Artificial Intelligence Laboratory
NE43-791, 545 Technology Sq., Cambridge, MA 02139
Phone: (617) 253-5868 FAX: (617) 253-5060

Abstract

Dynamically Programmable Gate Arrays (DPGAs) are programmable arrays which allow the strategic reuse of limited resources. In so doing, DPGAs promise greater capacity, and in some cases higher performance, than conventional programmable device architectures where all array resources are dedicated to a single function for an entire operational epoch. This paper examines several usage patterns for DPGAs including temporal pipelining, utility functions, multiple function accommodation, and state-dependent logic. In the process, it offers insight into the application and technology space where DPGA-style reuse techniques are most beneficial.

1 Introduction

FPGA capacity is conventionally metered in terms of "gates" assigned to a problem. This notion of gate utilization is, however, a purely spatial metric which ignores the temporal aspect of gate usage. That is, it says nothing about how often each gate is actually used. A gate may only perform useful work for a small fraction of the time it is employed. Taking the temporal usage of a gate into account, we recognize that each gate has a capacity defined by its bandwidth. Exploiting this temporal aspect of capacity is necessary to extract the most performance out of reconfigurable devices.

To first order, conventional FPGAs maximally exploit their potential gate capacity only when fully pipelined to solve a single task with data processing rates that occupy the entire array at its maximum clock frequency (*e.g.* Figure 1). As task requirements move away from this extreme, gates are used at a fraction of their potential capacity and FPGA utilization efficiency drops. Away from this heavily pipelined extreme, multiple context devices, such as DPGAs, provide higher efficiency by allowing limited interconnect and logic element resources to be reused in time. These devices allow the temporal component of device capacity to be deployed to increase total device functionality rather than simply increasing throughput for fixed functionality.

In this paper we examine several, stylized usage patterns for DPGAs focusing on the resource reuse they enable. We start by identifying capacity and utilization metrics for programmable devices (Section 2). We examine a few characteristics of application and technology trends (Section 3) to understand why resource reuse is important. In Section 4 we briefly review DPGA characteristics and look at the costs for supporting resource reuse. The heart of the paper then focuses on four broad styles for resource reuse:

1 Multiple, Independent Functions (Section 5)
2 Utility Functions (Section 6)
3 Temporal Pipelining (Section 7)
4 Finite State Machines (Section 8)

Section 9 reviews the themes introduced and summarizes the domain of application where DPGAs are most efficient.

2 Capacity and Utilization

An FPGA is composed of a number of programmable gates interconnected by wires and programmable switches. Each gate, switch, and wire has a limited bandwidth (f_{max}) — or time between uses ($t_{min} = \frac{1}{f_{max}}$) necessary for correct operation. In a unit of time T, we could, theoretically, get a maximum of

$$F_{gate} = \left(\frac{T}{t_{min_{gate}}}\right) \times N_{gate}$$

gate evaluations. We can calculate F_{wire} and F_{switch} in a similar manner. Each gate, switch, and wire has an inherent propagation latency (t_{pd}) as well.

For simplicity, let us model an FPGA as having a single, minimum reuse time, t_{min_cycle}, which is the minimum reuse time for a gate evaluation and local interconnect. We will assume t_{min_cycle} captures both the bandwidth and the propagation delay limitations. The flip-flop toggle rate which was often quoted by vendors as a measure of device performance [13], provides a rough approximation of t_{min_cycle} for commercial devices (*i.e.* $t_{min\_cycle} \approx \frac{1}{F_{CLK}}$). Throughout, we assume $t_{ff\_setup}, t_{clk \rightarrow q} << t_{min\_cycle}$ to allow simple time discretization and comparison.

The peak operational capacity of our FPGA is thus:

$$C_{peak} = \frac{N_{resource}}{t_{min\_cycle}} \qquad (1)$$

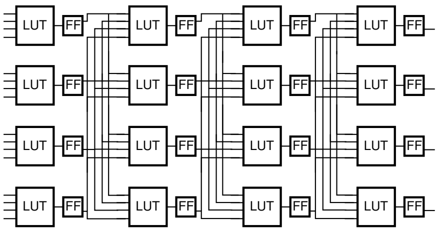


Figure 1: Full Pipelining to Achieve Peak FPGA Utilization

$N_{resource}$ may be any of the potentially limited device resources such as gates, wires, or switches. This peak capacity is achieved when the FPGA is clocked at $f_{max} = \frac{1}{t_{min\_cycle}}$ and every gate is utilized. In our conventional model, that means every gate registers its output, and there is at most one logic block between every pair of flip flops (See Figure 1). This is true even when the register overhead time is not small compared to t_{min_cycle}, but the computation latency may be increased notably in such a case.

When we run at a slower clock rate to handle deeper logic paths, or the device goes unused for a period of time, we get less capacity out of our FPGA. Running $N_{resources\_used}$ at a clock rate t_{clk}, we utilize a capacity:

$$U = \frac{N_{resoures\_used}}{t_{clk}} \tag{2}$$

As $t_{clk} > t_{min\_cycle}$ and $N_{resources\_used} < N_{resource}$, the utilization is below capacity. For example, if the path delay between registers is four gate-interconnect delays such that $t_{clk} = 4 \times t_{min\_cycle}$, even if all the device gates are in use, the gate utilization, U_{gate}, is one-quarter the peak capacity C_{gate_peak}.

3 Technology and Application Trends

Utilization of a resource is only important to the extent that one is capacity limited in that resource. For example, if an FPGA has a plethora of gates, but insufficient switching to use them, gate utilization is irrelevant while switch utilization is paramount. When one resource is limited, pacing the performance of the system, we say this resources is the *bottleneck*. To improve performance we deploy or redeploy our resources to utilize the bottleneck resource most efficiently. In this section we look at the effects of bottlenecks arising from application requirements and from technology-oriented resource costs.

Bottlenecks and Capacity One reason we do not fully pipeline every design and run it at the maximum clock frequency is that we often do not need that much of every computation. While heavy pipelining gets the throughput up for the pipelined design, it does not give us any more functionality, which is often the limiter at the application level.

Most designs are composed of several components, each performing a task necessary to complete the entire application (See Figure 2). The overall performance of the design is limited by the processing throughput of the slowest device. If the performance of the slowest device is fixed, there is no need for the other devices in the system to process at substantially higher throughputs. In these situations, reuse of the active silicon area on the non-bottleneck

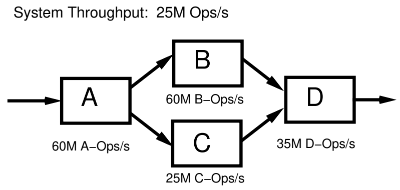


Figure 2: Typical Multicomponent System

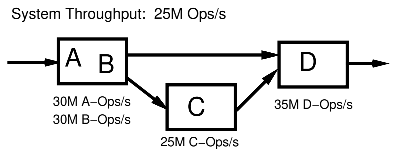


Figure 3: Multifunction Component in System

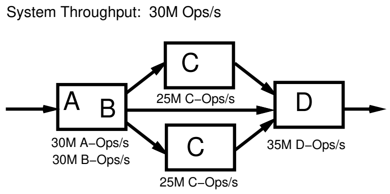


Figure 4: Function Distribution in System

components can improve performance (See Figure 4) or lower costs (See Figure 3).

Many applications, such as input processing on sensor data, display processing, or video processing, have fixed requirements. In these applications, processing faster than the sample or display rate is not necessary or useful. Once we achieve the desired rate, the rest of the "capacity" of the device is not required for the function. With reuse of active silicon, the residual processing capacity can be employed on other computations.

Technology and Bottleneck Resources Many bottlenecks arise from implementation technology costs. Resources which are relatively "expensive" in area, have inherently high latencies, or have inherently low bandwidths tend to create bottlenecks in designs. I/O and wiring resources often pose the biggest, technology-dictated bandwidth limitations in reconfigurable systems.

Device I/O bandwidth often limits the rate at which data can be delivered to a part. When data throughput is limited by I/O bandwidth, we can reuse the internal resources to provide a larger, effective, internal gate capacity. This reuse decrease the total number of devices required in the system. It may also help lower the I/O bandwidth requirements by localizing larger sets of interacting functions on each IC.

Internal routing resources tend to be one of the limiting factors in FPGA designs. Even though it is not uncommon for 75-80% of device area to go into interconnect, the amount of interconnect is often insufficient to handle designs which push gate usage near spatial capacity. This limitation is also technology based. Desirable switch and wire resources grow close to $O(N^2_{gate})$ rather than linearly in N_{gate}. To the extent we try to increase our spatial FPGA gate capacity linearly with silicon area, this makes the routing network the limiting resource. Reuse of network wires and switches can be one of the biggest benefits arising from temporal reuse.

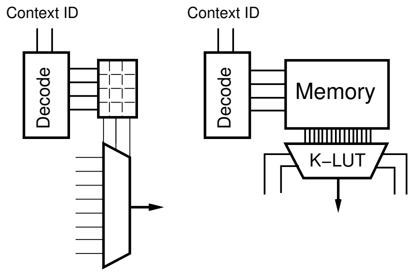


Figure 5: DPGA LUT and Interconnect Primitives

Latency Limited Designs Some designs are limited by latency not bandwidth. Here, high bandwidth may be completely irrelevant or, at least, irrelevant when it is higher than the reciprocal of the design latency. This is particularly true of applications which must be serialized for correctness (*e.g.* atomic actions, database updates, resource allocation/deallocation, adaptive feedback control) or have cyclic dependencies (*e.g.* FSMs). By reusing gates and wires, we can use device capacity to implement these latency limited operations with less resources than would be required without reuse.

4 DPGA Characteristics

In multicontext FPGAs, we increase utilization, U, by allocating space on chip to hold several configurations for each gate or switch. Rather than having a single configuration memory, each Look Up Table (LUT) or multiplexor, for instance, has a small local memory (See Figure 5). A broadcast context identifier tells each primitive which configuration to select at any given point in time.

By allocating additional memory to hold multiple configurations, we are reducing the potential C_{peak} for a fixed amount of silicon. At the same time, we are facilitating reuse which can increase utilization. Multiple contexts are beneficial to the extent that the additional space for memory creates a net utilization increase.

Let us consider our array as composed of context memory occupying a fraction of the die A_{cmem} and active area occupying A_{active}. For simplicity, we assume $A_{cmem} + A_{active} = 1$. We can relate the multicontext peak to the single-context FPGA peak:

$$C_{mcpeak} = C_{peak} \times A_{active} \quad (3)$$

We can calculate multicontext gate utilization, for example:

$$U_{mcgate} = \frac{N_{total\_gates\_evaluated}}{t_{clk}} = \sum_{i=0}^{c} \left(\frac{N_{gates\_evaluted}(i)}{t_{ctxt\_clk}} \right) \quad (4)$$

We can calculate an efficiency for the use of active multicontext resources:

$$E = \frac{U_{mc}}{C_{mcpeak}} \quad (5)$$

Alternately, we can compare total silicon utilization efficiency to the single context case:

$$E_{mc} = \frac{U_{mc}}{C_{peak}} = E \times A_{active} \quad (6)$$

Equation 6 computes the net utilization efficiency of the silicon. This is the quantity we need to maximize to get the most capacity out of our silicon resources.

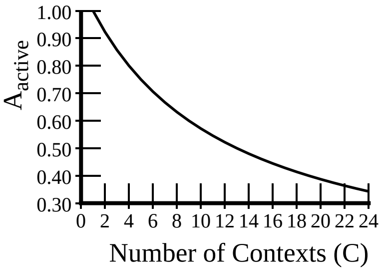


Figure 6: Active Area Percentage versus Number of Contexts based on DPGA Prototype Areas

Breaking up the area by number of contexts, c:

$$A_{cmem} = (c-1) \times A_{mem} \quad (7)$$

In Equation 7, we use $c-1$ instead of c since we assume base silicon usage (A_{active}) includes one configuration and any fixed overhead associated with it. A_{mem} is the incremental cost of one context configuration's worth of memory. To the extent the size of the context configuration is linear in the size of the device, we can approximate A_{mem} as proportional to A_{active}:

$$A_{mem} = C_{mem} \times A_{active} \quad (8)$$

This gives us:

$$A_{active} + (c-1) \times C_{mem} \times A_{active} = 1 \quad (9)$$

$$A_{active} = \frac{1}{1+(c-1)C_{mem}} \quad (10)$$

Our first generation DPGA prototype [11] was implemented in a 3-layer metal, 1.0μm CMOS process. The prototype used 4-LUTs for the basic logic blocks and supported 4 on-chip context memories. Each context fully specified both the interconnect and LUT functions. Roughly 40% of the die area consumed on the prototype went into memory. We estimate that half of that area is required for the first context, while the remaining area is consumed by the additional three contexts. For the prototype implementation, then, $A_{cmem} \approx$ 20% and $A_{active} \approx$ 80%. Based on the relative sizes in the prototypes and using the above relations, this gives $C_{mem} = \frac{1}{12}$. With these technology constants, Equation 10 becomes:

$$A_{active} = \frac{1}{1+(c-1)\frac{1}{12}} = \frac{12}{12+(c-1)} = \frac{12}{c+11} \quad (11)$$

Figure 6 plots the relationship shown in Equation 11.

Context switch overhead associated with reading new configuration data can further decrease multicontext capacity by increasing $t_{min\_ctx\_clk}$ over t_{min_clk}. Our experience suggests that unpipelined context changes yield a $t_{min\_ctx\_clk} < 1.5 t_{min\_clk}$. This effect can be minimized by pipelining the context read at the cost of the additional area for a bank of context registers (lowering A_{active}). We make the pedagogical assumption that $t_{min\_ctx\_clk} \approx t_{min\_clk}$ for this discourse.

5 Multiple Independent Functions

The easiest and most mundane way to increase the utilization on an FPGA is to use the FPGA for multiple, independent functions. At a very coarse granularity, conventional FPGAs exploit this kind of reuse. The essence of reconfigurable resources is that they do not have to be dedicated to a single function throughout their lifetime. Unfortunately, with multi-millisecond reconfiguration time scales, this kind of reconfiguration is only useful in pushing up utilization in the coarsest sense. Since conventional devices do not support background configuration loads, the active device resources are idle and unused during these long reconfiguration events.

With multiple, on-chip contexts, a device may be loaded with several different functions, any of which is immediately accessible with minimal overhead. A DPGA can thus act as a "multifunction peripheral," performing distinct tasks without idling for long reconfiguration intervals. In a system such as the one shown in Figure 3, a single device may perform several tasks. When used as a reconfigurable accelerator for a processor (*e.g.* [1] [3] [8]) or to implement a dynamic processor (*e.g.* [12]), the DPGA can support multiple loaded acceleration functions simultaneously.

Within a CAD application, such as `espresso` [9], one needs to perform several distinct operations at different times, each of which could be accelerated with reconfigurable logic. We could load the DPGA with assist functions, such as an ASCII decoder (*e.g.* [8]), bitvector manipulator, first one locator (*e.g.* [1]), or hamming distance calculator (*e.g.* [1]). Since these tasks are needed at distinct times, they can easily be stacked in separate contexts and selected as needed. To the extent that function usage is interleaved, the on-chip context configurations reduce the reload idle time which would be required to share a conventional device among as diverse a set of functions.

6 Utility Functions

Some classes of functionality are needed, occasionally but not continuously. In conventional systems, to get the functionality at all, we have to dedicate wire or gate capacity to it, even though it may be used very infrequently. A variety of data loading and unloading tasks fit into this "infrequent use" category, including:

- Data offload – *e.g.* debugging snapshot, testing observability, fault recovery snapshot, context data offload
- Data onload – *e.g.* configuration setting, value initialization, debugging value injection, testing accessibility, fault recovery, context data reload (after coarse-grain context switch)
- Operation idle/enable – *e.g.* conditional operation, exception handling, stall

In a multicontext device, the resources to handle these infrequent cases can be relegated to a separate context, or contexts, from the "normal" case code. The wires and control required to shift in (out) data and load it are allocated for use only when the respective utility context is selected. The operative circuitry then, does not contend with the utility circuitry for wiring channels or switches, and the utility functions do not complicate the operative logic. In this manner, the utility functions can exist without increasing critical path delay during operation.

A relaxation algorithm might operate as follows:

1 Load in starting point and boundary conditions
2 Calculate relaxation updates
3 Check for convergence, return to 2 if not converged
4 Offload result

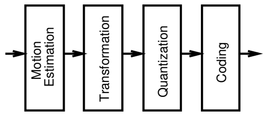


Figure 7: Canonical Video Coding Pipeline

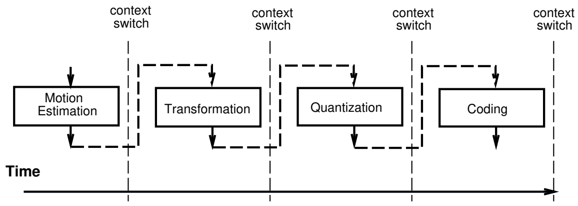


Figure 8: Temporally Systolic Video Coding Pipeline

Each of these operations may be separate contexts. The relaxation computation may even be spread over several contexts. This general operation style, where inputs and outputs are distinct and infrequent phases of operation, is common for many kinds of operations (*e.g.* multi-round encryption, hashing, searching, and many optimization problems).

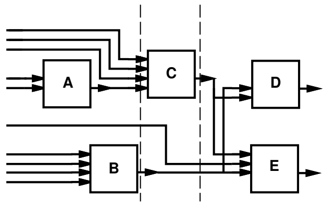
7 Temporal Pipelining

In the introduction we noted that we can extract the highest capacity from our FPGAs by fully pipelining every operation. When we need the highest throughput for the task, but limited functionality, this technique works well. However, we noted in Section 3 that application and technology bottlenecks often limit the rate at which we can provide new data for a design such that this maximum throughput is seldom necessary. Further, we noted that many applications are limited in the amount of distinct functionality they provide rather than the amount of throughput for a single function. *Temporal pipelining* is a stylized way of organizing designs for multi-context execution which uses available capacity to support a larger range of functions rather than providing more throughput for a single piece of functionality.

7.1 Temporally Systolic Pipelines

Figure 7 shows a typical video coding pipeline (*e.g.* [7]). In a conventional FPGA implementation, we would lay this pipeline out spatially, streaming data through the pipeline. If we needed the throughput capacity offered by the most heavily pipelined spatial implementation, that would be the design of choice. However, if we needed less throughput, the spatially pipelined version would require the same space while underutilizing the silicon. In this case, a DPGA implementation could stack the pipeline stages in time. The DPGA can execute a number of cycles on one pipeline function then switch to another context and execute a few cycles on the next pipeline function (See Figure 8). In this manner, the lower throughput requirement could be translated directly into lower device requirements.

This is a general schema with broad application. The pipeline design style is quite familiar and can be readily adapted for multicontext implementation. The amount of temporal pipelining can



Dashed lines show one full levelization of this circuit. With a context switch at each dashed line, the circuit can be evaluated on two physical gates.

Figure 9: Levelization Example

be varied as throughput requirements change or technology advances. As silicon feature sizes shrink, primitive device bandwidth increases. Operations with fixed bandwidth requirements can increasingly be compressed into more temporal and less spatial evaluation.

7.2 Levelized Logic

Levelized logic is a CAD technique for automatic temporal pipelining of existing circuit netlists. Bhat refers to this as *temporal partitioning* in the context of the Dharma architecture [2]. The basic idea is to assign an evaluation context to each gate so the gate's predecessors are evaluated in a context prior to the gate's context. With latency constraints, we may further require that the levelized network not take any more t_{min_clk} steps than necessary. With a full levelization scheme, the number of contexts used to evaluate a netlist is equal to the critical path in the netlist.

Figure 9 shows a simple netlist with five logic elements. The critical path (A→C→E) is three elements long. Spatially implemented, this netlist evaluates a 5 gate function in 3 cycles using 5 physical gates. In three cycles, these five gates could have provided $5 \times 3 = 15$ gate evaluations, so we realize a gate usage efficiency, E_{gate}, of $\frac{5}{15} = 0.33$. The circuit can be fully levelized as shown in Figure 9. The total gate evaluation capacity occupied is $2 \times 3 = 6$ ($E_{gate} = \frac{5}{6} = 0.83$). Alternately, the circuit can be levelized into 5 contexts, taking a delay of $5 \times t_{min\_cycle}$ and a capacity of 5 ($E_{gate} = \frac{5}{5} = 1.0$).

The preceding example illustrates the kind of options available with levelization.

- Slack in the network allows us some freedom in the context placement for components outside of the critical path. In general, this slack should be used to equalize context size, minimizing capacity usage.
- Up to a point, more contexts allow increased utilization.
- Achieving the highest utilization often requires increasing the evaluation delay.

Table 1 summarizes full levelization results for several MCNC benchmarks. `sis` [10] was used for technology independent optimization. `Chortle` [5] was used to map the circuits to 4-LUTs. For the purpose of comparison, circuits were mapped to minimize delay since this generally gave the highest, single context utilization efficiency. No modifications to the mapping and netlist generation were made for levelized computation. Gates were assigned to contexts using a list scheduling heuristic. Levelization results are not optimal, but demonstrate the basic opportunity for increased utilization efficiency offered by levelized logic evaluation.

Design	Single Context				Full Levelization		
	L	N_g	Cap	E_s	Cap	E_{mg}	$\frac{E_{mg}}{E_s}$
5xp1	6	55	330	0.16	72	0.76	4.58
9sym	5	155	775	0.20	510	0.30	1.51
9symml	5	130	650	0.20	420	0.30	1.54
C499	7	406	2842	0.14	588	0.69	4.83
C880	9	289	2601	0.11	342	0.84	7.60
alu2	10	323	3230	0.10	480	0.67	6.72
apex6	5	454	2270	0.20	460	0.98	4.93
apex7	5	158	790	0.20	165	0.95	4.78
b9	3	55	165	0.33	57	0.96	2.89
clip	6	162	972	0.16	324	0.50	3.00
cordic	8	529	4232	0.12	888	0.59	4.76
count	4	128	512	0.25	164	0.78	3.12
des	8	2749	21992	0.12	3168	0.86	6.94
e64	4	385	1540	0.25	456	0.84	3.37
f51m	7	152	1064	0.14	252	0.60	4.22
misex1	3	24	72	0.33	27	0.88	2.66
misex2	4	58	232	0.25	60	0.96	3.86
rd73	5	157	785	0.20	295	0.53	2.66
rd84	5	381	1905	0.20	935	0.40	2.03
rot	8	398	3184	0.12	400	0.99	7.96
sao2	5	98	490	0.20	150	0.65	3.26
vg2	5	92	460	0.20	135	0.68	3.40
z4ml	4	13	52	0.25	16	0.81	3.25
Mean				0.19		0.71	4.08

Table 1: Full Levelization for Benchmark Circuits

The following equations summarize the metrics used in Table 1:

$$E_s = \frac{1}{L} \quad (12)$$

$$\text{Cap} = N_g \times L_{total} \quad (13)$$

$$E_{mg} = \frac{N_g}{L \times N_{max\_level\_gates}} \quad (14)$$

For the purpose of pedagogical comparison, we normalize gate usage to the number of gates required, N_g in each implementation. Referring back to Equations 1 and 2, single context efficiency is $E_s = \frac{U}{C_{peak}}$. With the normalization, the single context efficiency is simply the reciprocal of the critical path delay, L (Equation 12). Capacity, Cap, is the time-space capacity occupied by an implementation. E_{mg} results from reducing Equation 5 normalized to $N_{max\_level\_gates}$, the number of gates required by the the largest context in the multicontext implementation.

We saw in our simple example above that utilization efficiency varies with the number of contexts actually used. Figure 10 plots the gate efficiencies achieved for the DES benchmark for various numbers of contexts. Recall from Section 4 and Figure 6 that context memory takes area away from active silicon. Figure 10 also combines gate efficiencies with Equation 11 according to Equation 6 to show the net silicon utilization efficiency for this design.

In this section we have focussed on cases where the entire circuit design is temporally pipelined. There are, of course, hybrids which involve some element of both spatial and temporal pipelining. Once we have determined the level of spatial pipelining necessary to provide the requisite throughput, we are free to temporally pipeline logic evaluation within each spatial pipeline stage.

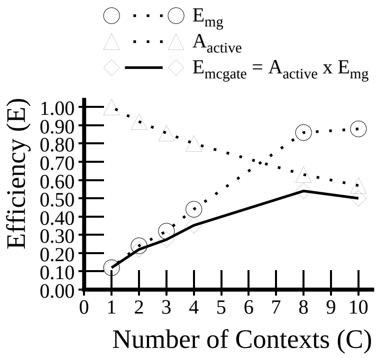


Figure 10: Overall Silicon Utilization for DES Benchmark

8 Finite State Machines

Since the next state calculation must complete and be fed back to the input of the FSM before the next state behavior can begin, there is no benefit to be gained from spatial pipelining within the FSM logic. Temporal pipelining can be used to increase gate and wire utilization. The middle section of Table 2 summarizes the full levelization of several MCNC benchmark FSMs in the same style as Table 1.

Finite state machines, however, happen to have additional structure over random logic which can be exploited. In particular, one never needs the full FSM logic at any point in time. During any cycle, the logic from only one state is active. In a traditional FPGA, we have to implement all of this logic at once in order to get the full FSM functionality. With multiple contexts, each context need only contain a portion of the state graph. When the state transitions to a state whose logic resides in another context, we can switch contexts making a different portion of the FSM active. National Semiconductor, for example, exploits this feature in their multicontext programmable logic array (PLA), MAPL [6].

In the most extreme case, each FSM state is assigned its own context. The next state computation simply selects the appropriate next context in which to operate. Table 2 shows the reduction in logic depth and increase in utilization efficiency which results from multiple context implementation. FSMs were mapped using `mustang` [4]. Logic minimization and LUT mapping were done with `espresso`, `sis`, and `Chortle`. All single context FSM implementations use one-hot state encodings since those uniformly offered the lowest latency and had the lowest capacity requirements. The multicontext FSM implementations use dense encodings so the state specification can directly serve as the context select. Delay and capacity are dictated by the logic required for the largest and slowest state. Comparing context per state partitioning in Table 2 to full levelization, we see that this state partitioning achieves higher efficiency gains and generally lower delays than levelization.

The capacity utilization and delay are often dictated by a few of the more complex states. It is often possible to reduce the number of contexts required without increasing the capacity required or increasing the delay. Figure 11 shows the CSE benchmark partitioned into various numbers of contexts. These partitions were obtained by partitioning along `mustang` assigned state bits starting with a four bit state encoding.

While demonstrated in the contexts of FSMs, the basic technique used here is also fairly general. When we can predict which portions of a netlist or circuit are needed at a given point in time, we can generate a more specialized design which only includes the required logic. The specialized design is often smaller and faster than the fully general design. With a multicontext component, we can use the contexts to hold many specialized variants of a design, selecting them as needed.

9 Conclusions

Programmable device capacity has both a spatial and a temporal aspect. Traditional FPGAs can only build up functionality in the spatial dimension. These FPGAs can only exploit the temporal aspect of capacity to deliver additional throughput for this spatially realized functionality. As a result, with heavy pipelining, FPGAs can provide very high throughput but only on a limited amount of functionality. In practice, however, we seldom want or need the fully pipelined throughput. Instead, we are often in need of more resources or can benefit from reducing total device count by consolidating more functionality onto fewer devices.

In contrast, DPGAs dedicate some on-chip area to hold multiple configurations. This allows resources such as gates, switches, and wires, to implement different functionality in time. Consequently, DPGAs can exploit both the temporal and the spatial aspects of capacity to provide increased functional capacity.

Fully exploiting the time-space capacity of these multicontext devices introduces new tradeoffs and raises new challenges for design and CAD. This paper reviewed several stylized models for exploiting the time-space capacity of devices. Multifunction devices, segregated utility functions, and temporally systolic pipelining are all design styles where the designer can exploit the fact that the device function can change in time. Levelized logic and FSM partitioning are CAD techniques for automatically exploiting the time-varying functionality of these devices. From the circuit benchmark suite, we see that 3-4x utilization improvements are regularly achievable. The FSM benchmarks show that even greater capacity improvements are possible when design behavior is naturally time varying. Techniques such as these make it moderately easy to exploit the capacity improvements enabled by DPGAs.

Acknowledgments

This research is supported by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

References

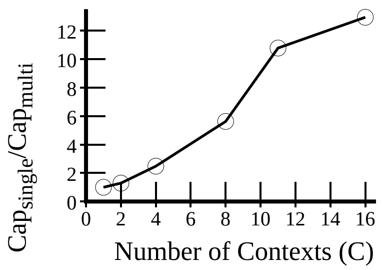
[1] Peter Athanas and Harvey F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[2] Narasimha B. Bhat. Novel Techniques for High Performance Field Programmable Logic Devices. UCB/ERL M93/80, University of California, Berkeley, November 1993.

[3] André DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

		Single Context				Full Levelization			Context per State				
FSM	States	L	N_g	Cap	E_s	Cap	E_{mg}	$\frac{E_{mg}}{E_s}$	L	N_g	Cap	E_{mg}	$\frac{Cap_{single}}{Cap_{mc}}$
bbara	10	3	40	120	0.33	45	0.88	2.66	1	6	6	1.00	20.00
bbsse	16	3	60	180	0.33	60	1.00	3.00	2	13	26	0.50	6.92
beecount	7	2	22	44	0.50	22	1.00	2.00	1	7	7	1.00	6.28
cse	16	4	97	388	0.25	104	0.93	3.73	2	15	30	0.50	12.93
dk14	7	3	67	201	0.33	84	0.79	2.39	1	8	8	1.00	25.12
dk16	27	3	83	249	0.33	87	0.95	2.86	1	8	8	1.00	31.12
dk512	15	2	20	40	0.50	20	1.00	2.00	1	7	7	1.00	5.71
ex1	20	4	151	604	0.25	164	0.92	3.68	2	26	52	1.00	11.61
ex6	8	3	62	186	0.33	69	0.89	2.69	1	11	11	1.00	16.90
mc	4	2	14	28	0.50	14	1.00	2.00	1	7	7	1.00	4.00
planet	48	4	172	688	0.25	172	1.00	4.00	1	25	25	1.00	27.52
pma	24	4	139	556	0.25	160	0.86	3.47	2	15	30	0.50	18.53
s1	20	4	195	780	0.25	228	0.85	3.42	3	31	93	0.33	8.38
s1488	48	4	183	732	0.25	184	0.99	3.97	2	27	54	0.50	13.55
s208	18	3	40	120	0.33	42	0.95	2.85	1	7	7	1.00	17.14
s386	13	4	54	216	0.25	56	0.96	3.85	2	12	24	0.50	9.00
s510	47	3	83	249	0.33	84	0.98	2.96	1	13	13	1.00	19.15
sand	32	4	224	896	0.25	260	0.86	3.44	4	38	152	0.25	5.89
styr	30	5	285	1425	0.20	335	0.85	4.25	2	24	48	0.50	29.68
tbk	32	5	510	2550	0.20	1065	0.47	2.39	4	46	184	0.25	13.85
Mean								3.08					15.16

Table 2: Full Levelization and State per Context FSM Partitioning



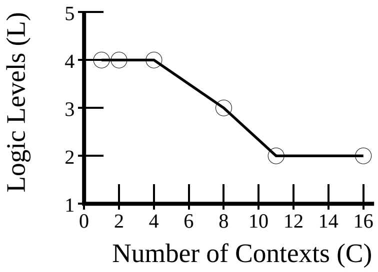


Figure 11: Capacity and Delay versus Number of Contexts for CSE Benchmark

[4] Srinivas Devadas, Hi-Keung Ma, , A.R. Newton, and Alberto Sangiovanni-Vincentelli. MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(12):1290–1300, December 1988.

[5] Robert Francis. *Technology Mapping for Lookup-Table Based Field-Programmable Gate Arrays*. PhD thesis, University of Toronto, 1992.

[6] David Hawley. Advanced PLD Architectures. In Will Moore and Wayne Luk, editors, *FPGAs*, pages 11–23. Abingdon EE&CS Books, 15 Harcourt Way, Abingdon, OX14 1NV, UK, 1991.

[7] Chris Jones, John Oswald, Brian Schoner, and John Villasenor. Issues in Wireless Video Coding using Run-time-reconfigurable FPGAs. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[8] Rahul Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard Univeristy, May 1994.

[9] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 6(5):727–751, September 1987.

[10] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. UCB/ERL M92/41, University of California, Berkeley, May 1992.

[11] Edward Tau, Ian Eslick, Derrick Chen, Jeremy Brown, and André DeHon. A First Generation DPGA Implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pages 138–143, May 1995.

[12] Michael J. Wirthlin and Brad L. Hutchings. A Dynamic Instruction Set Computer. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1995.

[13] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *The Programmable Logic Data Book*, 1989, 1994.