Multicontext Field-Programmable Gate Arrays

André DeHon

Abstract— **Traditional Field-Programmable Gate Arrays (FPGAs) store a single configuration context on-chip to control the device function. This organization yields a device which is efficient at performing exactly the same function at a high cycle rate. In practice, however, most FPGA resources sit idle most of the time since applications seldom need to perform precisely the same operations at the minimum cycle time possible with today's FPGAs. We generalize the FPGA structure by adding multiple, on-chip configuration contexts and data retiming support. As a result, the FPGA resources can be efficiently used to perform a number of different operations. This modification is interesting because the area required for the each additional configuration is small compared to the active area which it controls. Traditional FPGA netlists with long critical paths, low-throughput requirements, or time-varying logic can automatically be mapped into multi-context implementations which require only one-third of the area of a single-context implementation. Even greater economy may be possible when designs are crafted explicitly to exploit multicontext operation.**

Keywords— **reconfigurable-components, configurable, vlsi, FPGA, technology-mapping, instructions, area efficiency, Special-issue-fpga97**

I. Introduction

A Field Programmable Gate Array (FPGA) is a collection of bit processing elements (gates) whose operation and interconnect can be specified **after** the device has been fabricated. In most modern FPGAs the customization is non-permanent and reprogrammable, allowing the device to be reconfigured at will. To specify device behavior we load a set of bits, or *instructions*, which tell the device how to behave. The set of bits which specify the array-wide behavior of the device is commonly called a *configuration*.

In all conventional FPGAs, a single configuration is stored on the chip. That is, for each bit processing element and each interconnect switch, there is a single set of bits which tells the device how to behave. Every active resource, therefore, performs exactly the same function between load epochs when new instructions are loaded from off chip. This entire configuration is large. Typically 100–200 bits are required per bit processing element. Today's FPGAs contain thousands of processing elements and hence hundreds of thousands of configuration bits. Changing a configuration from an off-chip source requires many microseconds; more commonly, reload times are measured in milliseconds.

Each bit processing element can typically compute a results in a few nanoseconds and route it to a consuming processing element in 4–10 ns depending on physical separation on the die. Nonetheless, few modern FPGA applications run at even 100MHz. 30–50MHz operation is considered "high-speed", while most operations run at 10–25MHz. Operating at these speed, most of the device capacity sits idle most of the time. At 10MHz operation, any given bit processing element or interconnect switch is sitting idle during, at least, 90% of the clock cycle.

Author is currently with the University of California at Berkeley; this paper describes work performed at MIT. E-mail: andre@acm.org.

We observe that the configuration memory which holds a description of the behavior of the device is much smaller than the active resources required to actually perform the logic and switching functions on the device. This observation suggests that:

1. We can pack multiple configurations on chip without substantially increasing the area per bit processing element.
2. Once multiple configurations are on-chip, we can reuse each bit processing element during its "idle" time to perform different operations.

Reusing active processing elements allows us to implement a given task with *fewer* bit processing elements. The net effect is that we can build a device with multiple configurations, *multicontext*, which implements a given computation in less area than its single context counterpart. The resulting multicontext device can be viewed as Very Long Instruction Word (VLIW) processor with bit-level processing units and local, on-chip instruction memory.

There is, of course, a limit to how far we can take this technique. At some point, we will run out of spare capacity in active resources and the additional area required for extra, on-chip contexts will become greater than the savings allowed from resource sharing. Further, additional resources are required to retime intermediate data efficiently once we start reusing active interconnect. As a result, we find that each task has an "optimal" number of contexts associated with a given throughput requirement. If we want to engineer a single device to handle a variety of tasks, we must weigh the tradeoffs and determine the appropriate number of on-chip contexts to provide good results across the desired range of target workloads. Given typical areas for FPGA interconnect and SRAM configuration memory, we have seen that a modest number of contexts (*e.g.* 4–8) offers the smallest implementations for typical netlists and throughput requirements.

In Section II we build a simple architectural model and use it to underscore the basic tradeoffs involved. We then work through an illustrative example to make the design space more concrete in Section III. In Section IV we review our prototype multicontext device, and in Section V we touch on related work done by other. In Section VI we detail how existing FPGA netlists can be automatically retargetted for the multicontext device and use our model to estimate the area required to implement these circuits with varying numbers of contexts.

II. Architecture Modeling

A. Multicontext Organization

An FPGA is an array of post fabrication programmable bit processing elements. Most traditional FPGAs uses small lookup tables (LUTs) to serve as the programmable computational elements. These LUTs are wired together using programmable interconnect which actually accounts for most of the area in each FPGA cell (See Figure 1). Four input lookup tables (4-LUTs) are used for the programmable processing elements in many commercial devices due to their area-efficiency [1].

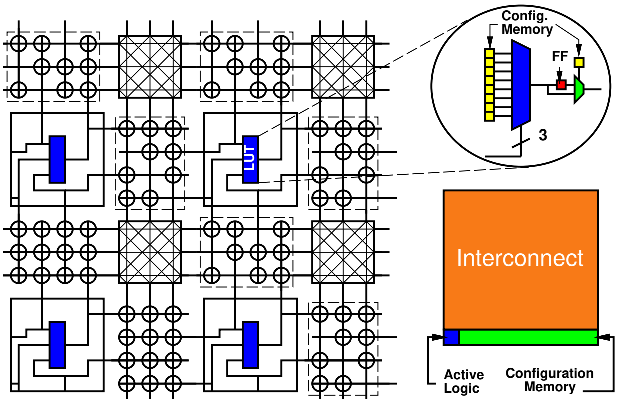

Fig. 1. LUT-based FPGA Caricature

A multicontext FPGA is a generalization of this FPGA model where we make c copies of the configuration memory (See Figure 2). This allows us to rapidly change the behavior of the device.

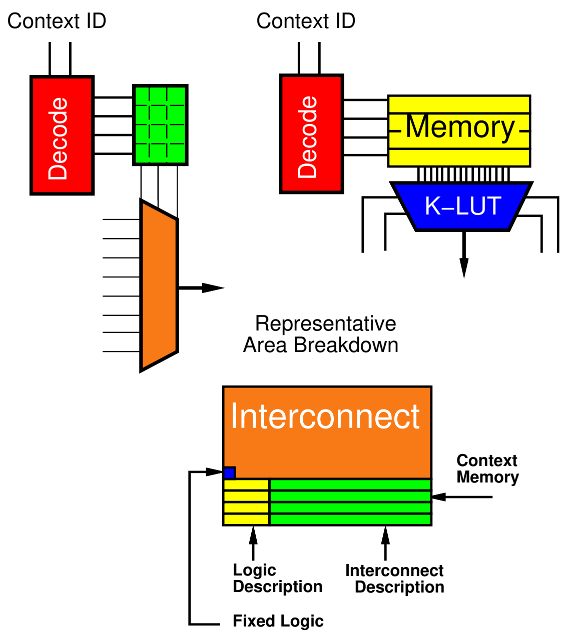

Fig. 2. Multicontext FPGA Caricature

As we will see later, when we evaluate a single logical computation in a series of stages, the data from previous stages must be presented to later stages. In order to reuse the LUT producing a datum during an earlier stage to perform a different computation at a later stage, the datum it produced must be stored until the time of consumption. Consequently, we will consider one other modification to the traditional FPGA architecture. Instead of optionally placing a flip-flop on the output of each LUT, we will place a series of optional flip-flops on the input of each LUT (See Figure 3). In this manner, data can be stored inexpensively on input flip-flops rather than consuming precious interconnect just so that it can be retimed forward in time to the point of consumption. The number of cycles of input retiming supported by each input is denoted by the parameter i.

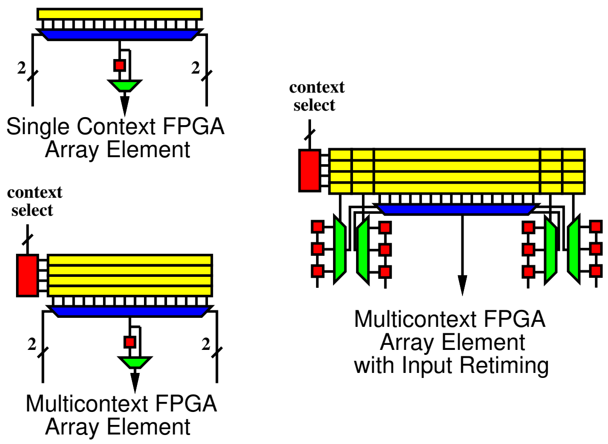

Fig. 3. Multicontext FPGA with Retiming Inputs

B. Area

The area of our FPGA bit processing element is thus composed of a base area for the LUT multiplexor and interconnect, an area proportional to c for the context memory, and an area proportional to i for input retiming:

$$A_{pe} = A_{base} + c \cdot A_{cmem} + i \cdot A_{input} \tag{1}$$

Strictly speaking, all of these areas are a function of k, the number of input to the LUT, however, we will assume $k = 4$ throughout this article. Using typical values based on commercial FPGAs and constituent area modeling, we consider:

$$\begin{aligned} A_{base} &\approx 800\mathrm{K}\lambda^2 \\ A_{cmem} &\approx 78\mathrm{K}\lambda^2 \\ A_{input} &\approx 26\mathrm{K}\lambda^2 \end{aligned}$$

λ, here, is half the minimum feature size in a CMOS process. A_{base} is consistent with arrays of $\approx$4096 4-LUTs. A_{cmem} assumes the context memory for a processing element is contained in 64 SRAM bits, each taking $1200\lambda^2$ of area. This entails more heavily encoded descriptions than conventional FPGAs [2], but this encoding makes sense

TABLE I
PARAMETERS AND VARIABLES

Symbol	Purpose
c	Number of configuration contexts
i	Depth of input retiming registers
k	LUT inputs (assumed 4)
L	Number of cycles available to perform a computation
N_a	Number of simultaneously active LUTs required to meet a target time bound
N_d	Number of LUTs in design description
N_p	Number of physical processing elements
A_{pe}	Area of each processing element
A_{base}	Base area of processing element
A_{cmem}	Area for a single configuration context memory for one processing element
A_{input}	Area per stage of input retiming

when $c > 1$. A_{input} assumes each input register is a $4\text{K}\lambda^2$ shift register with a $2.5\text{K}\lambda^2$ switch for selecting the desired input. For more details on these model values see [3]. The most important characteristic of these numbers is their ratios (*e.g.* $\frac{A_{base}}{A_{cmem}} \approx 10$). Table I summarizes the various parameters used in this article.

C. Basic Tradeoffs and Effects

We have two distinct components which combine to define the requisite area for a programmable computing device:

1. N_d – the total number of bit processing elements (LUTs) in the design—the descriptive complexity
2. N_a – the total number of these processing elements which must be evaluated simultaneously in order to achieve the desired task time or computational throughput—the parallelism required to achieve the temporal requirements

In an *ideal* packing, a computation requiring N_a active compute elements and N_d total 4-LUTs, can be implemented in area:

$$A_{compute} = N_a \cdot A_{base} + N_d \cdot A_{cmem} \quad (2)$$

If the throughput requirements are so high that each LUT in the description must be evaluated on every cycle, we have $N_a = N_d$ as in an FPGA. However, if the throughput requirements are low, we can often get away with substantially fewer active LUTs. In fact, ideally, if we have L cycles to evaluate a function, it should only require $N_a = \frac{N_d}{L}$ active LUTs. That is, we compute the first $\frac{N_d}{L}$ LUTs in the first cycle, then change configurations and compute the next $\frac{N_d}{L}$ LUTs in the second. After L such cycles, we have computed all N_d LUTs. Since $A_{cmem} \approx \frac{A_{base}}{10}$, an L-context implementation can be almost one tenth the size of a traditional FPGA when L is large (See Figure 4). As we see in later sections, packing and retiming limitations cause us to require $N_a > \frac{N_d}{L}$ active LUTs in practice.

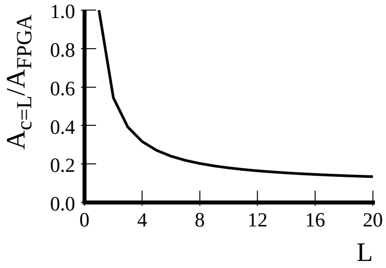

Fig. 4. Area v/s L given Ideal Packing

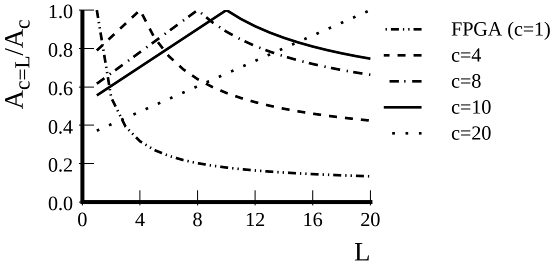

Fig. 5. Fixed c Area Efficiency v/s L given Ideal Packing

While the area tradeoff is not as ideal as shown in Figure 4, this tradeoff is this basic effect which makes multicontext devices interesting. L is often greater than one either because of external throughput requirements (*e.g.* data arriving from a 10Mbit/s interface) or because of cycles in the computation (*e.g.* a path composed of multiple LUTs which must be evaluated before the next input datum can be processed). In our multicontext model, we set $N_d = c \cdot N_a$ to keep all processing elements identical. Optimally, we would like to choose $c = L$. However L is an application characteristic, whereas c is an architecture parameter which we must choose when designing a particular device. Nonetheless, we can reasonably select values of $c > 1$ which are more robust across variations in L than $c = 1$. Figure 5 shows the size ratio between a configuration with optimally chosen L and various fixed choices for c. Notice that the FPGA case is a factor of 4–10 larger than the optimal $c = L$ for $L > 6$, while $c = 10$ is never more than a factor of 2 larger than optimal for any value of L.

III. EXAMPLE

To illustrate the opportunity and tradeoffs, consider the task of converting an ASCII Hex digit into binary. Figure 6 describes the basic computation required. Assuming we care about the latency of this operation, a mapping which minimizes the critical path length using SIS [4] and Chortle [5] has a path length of 3 4-LUTs and requires a total of 21 4-LUTs. Figure 7 shows the LUT mapping both in equations and circuit topology.

```
INORDER = C[7] C[6] C[5] C[4] C[3] C[2] C[1] C[0] ;
OUTORDER = O[3] O[2] O[1] O[0] ;
# stage 1 - 8 LUTs [C[3:0] pass through]
i0 = !C[1] * !C[2] ;
i1 = C[4] * C[5] * !C[6] * !C[7] ;
i3 = C[0] * C[1] * !C[2] ;
i4 = !C[3] * !C[4] * C[6] * !C[7] ;
i6 = !C[0] * C[2] ;
i7 = !C[0] * C[1] ;
i8 = C[0] * !C[1] ;
i11 = !C[7] * C[6] * !C[4] * !C[3] ;
# stage 2 - 9 LUTs [i1,C[3],C[1] pass through]
i5 = i0 * i1 + i3 * i4 ;
i9 = i6 * i4 + i7 * i4 + i8 * i4 ;
i10 = C[3] + i3 * i4 ;
i12 = i3 * i4 + i6 * i4;
i13 = i1 * !C[3] * C[2] ;
i14 = C[2] * !C[1] * i11 ;
i15 = i8 * i4 + i7 * i4 ;
i16 = i7 * i4 + i6 * i4 ;
i17 = i1 * !C[3] * C[0] + C[0] * i0 * i1 ;
# stage 3 - 4 LUTs
O[3] = (i10+i9)*(i5+i9);
O[2] = i12 + i13 + i14 ;
O[1] = i1 * !C[3] * C[1] + i15 ;
O[0] = i16 + i17 ;
```

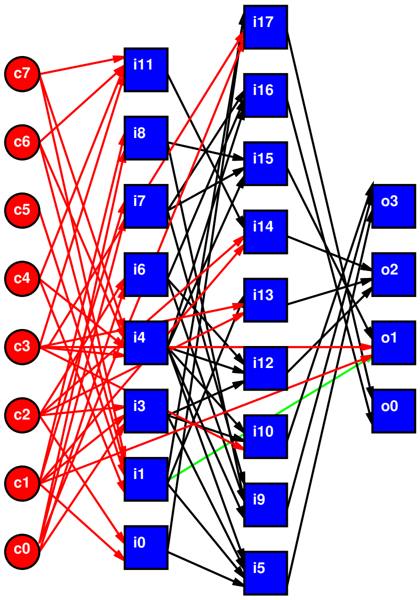

Fig. 7. 4-LUT Mapping of ASCII Hex→Binary

```
if (c >= 0x30 && c <= 0x39)
    res = c-0x30;
else if (c >= 0x40 && c <= 0x46)
    res = c - 0x40 + 10;
else if (c >= 0x60 && c <= 0x66)
    res = c - 0x60 + 10;
else
    res = 0;
```

Fig. 6. ASCII Hex→Binary Task Description

A. *Traditional Pipelining for Throughput*

If we cared only about achieving the highest throughput, we would fully pipeline this implementation such that it took in a new character on each cycle, producing its encoding three cycles later. Since flip-flops are only available on LUT outputs, this pipelining would require an additional 7 LUTs to pipeline data which is needed more than one pipeline stage after being generated (*i.e.* 4 to retime `c<3:0>` for presentation to the second stage and 3 to retime `c<3>`, `c<1>` and `i1` for presentation to the final stage—see Figure 8). Consequently, we effectively evaluate a design with $N_d = 21$ 4-LUTs with $N_p = 28$ physical 4-LUTs. Typical LUT delay for a modern FPGA, including a moderate amount of local interconnect traversal, is 7 ns. Assuming this is the only limit to cycle time, the implementation could achieve 140 MHz operation. Notice that the only reason we had to have any more LUTs or LUT descriptions than strictly required by the task description was in order to perform signal retiming based on the dependency structure of the computation. From our model, an FPGA LUT ($c = 1$) in a large array occupies $A_{pe} \approx 880\text{K}\lambda^2$. Consequently, this implementation requires:

$$A_{pipe} = 28 \cdot 880\text{K}\lambda^2 = 24.6\text{M}\lambda^2$$

B. *Multicontext Implementation – Temporal Pipelining*

If, instead, we cared about the latency, but did not need 140 MHz operation, we could use a multicontext device (see Figure 2) with 3 LUT descriptions per active element. To achieve the target latency of 3 LUT delays, we need to have enough active LUTs to implement the largest stage—the middle one. If the inputs are arriving from some other circuit which is also operating in multicontext fashion, we must retime them as before (Figure 8). Consequently, we require three extra LUTs in the largest stage, making for a total $N_p = 12$. Note that the 4 retiming LUTs added to stage 1 also bring its total LUT usage up to 12 LUTs. We end up implementing $N_d = 21$, with $N_p = 12$ and $c = 3$. If `c<7:0>` were inputs which did not change during these three cycles, we would only need one extra retiming LUT in stage 2 for `i1`, allowing us to use $N_p = 10$.

The multicontext LUT is slightly larger than the FPGA LUT due to the extract configuration memory. Two additional contexts add $160\text{K}\lambda^2$ to the processing element area, making for $A_{pe} \approx 1.04\text{M}\lambda^2$. The multicontext implementation requires:

$$A_{time-pipe} = 12 \cdot 1.04\text{M}\lambda^2 = 12.5\text{M}\lambda^2$$

In contrast, a non-pipelined, single-context implementation would require $N_p = N_d = 21$ LUTs, for an area of:

$$A_{nonpipe} = 21 \cdot 880\text{K}\lambda^2 = 18.5\text{M}\lambda^2$$

If we assume that we can pipeline the configuration read, the multicontext device can achieve comparable delay per

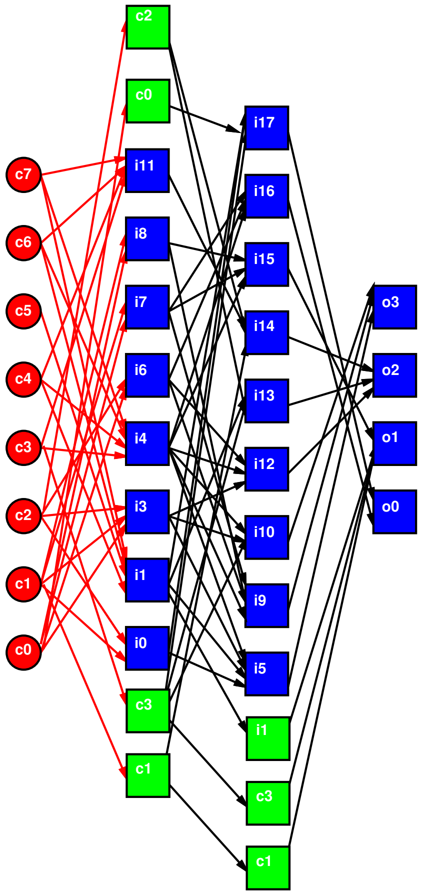

Fig. 8. ASCII→Hex Binary Circuit Retimed for Full Pipelining

LUT evaluation to the single context device. The total latency then is 21 ns, as before. The throughput at the 7 ns clock rate is 48 MHz. If we do not pipeline the configuration read, the configuration read may add another 2.5 ns to the LUT delay, making for a total latency of 28.5 ns and a throughput of 35 MHz.

C. Lower Throughput

In the previous example, we saw that we actually used $3 \cdot 12 = 36$ LUT configurations instead of the 21 in the actual design due to retiming and packing constraints. In fact, without input retiming support ($i = 0$), retiming requirements prevent us from implementing this task on any fewer than 12 active LUTs. This non-ideal saturation point arises from the one-to-one correspondence between LUTs and retiming registers. By adding input retiming registers as shown in Figure 3, we break this limitation. Figure 9 shows that each additional retiming stages allows a different, lower, point of saturation before additional contexts do not afford a decrease in active LUT requirements. Figure 9 also shows LUT area and total area for these implementations. At $c = 6$ and $i = 4$, the computation occupies only $5.5\mathrm{M}\lambda^2$, less than half the $12.5\mathrm{M}\lambda^2$ required for the 3 context implementation with no input retiming, and less than one third the $18.5\mathrm{M}\lambda^2$ required for the unpipelined FPGA version.

IV. A Prototype Multicontext FPGA

Early in our study of these multicontext devices, we built a prototype *Dynamically Programmable Gate Array* (See Figure 10) . We used the term "dynamically" to emphasize the device's ability to change its behavior on a cycle-by-cycle basis. The prototype was designed for 100 MHz operation allowing a context switch on every cycle. The context read itself was non-pipelined and contributed 2.5 ns to the cycle. If no context switch is required or if the context switch is pipelined, the device can run with a ≈7 ns cycle. Our prototype was based on 4-LUT computing elements and contained on-chip memory for four contexts ($c = 4$). The prototype was implemented in a 1μ, 3-metal CMOS process and contained 144 physical LUTs ($N_p = 144$). Each 4-LUT occupied an area of $0.16\mathrm{M}\mu^2$, or $640\mathrm{K}\lambda^2$, including interconnect and multicontext memory. The prototype used 3T DRAM cells for context memory, such that one LUT configuration context occupied only $24\mathrm{K}\lambda^2$; an SRAM context memory would have been roughly twice as large.

$$
\begin{aligned}
A_{pe} &= A_{base} + c \cdot A_{cmem} \\
A_{base} &\approx 544\mathrm{K}\lambda^2 \\
A_{cmem} &\approx 24\mathrm{K}\lambda^2
\end{aligned}
$$

Our base compute element is smaller than our model estimates from Section II due to the small array size and its limited interconnect. Our device needs only 40 bits to configure each 4-LUT and its associated interconnect. This is substantially smaller than the ≈ 200 bits required by conventional devices since we were careful to encode interconnect configurations as previously noted. Note that the ratio of configuration area to base LUT area is roughly 1:20. Had the prototype used SRAM instead of DRAM, this ratio would have been closer to 1:10 as in Section II.

More details on the prototype are available elsewhere [3] [6].

V. Related Devices

Recently several other devices have been proposed or built which can also be viewed as multicontext FPGAs. While these were originally motivated by different goals, we see that they fit basically within the architectural framework we describe here.

The Hydra processor which Arkos Design's developed for their Pegasus hardware emulator [7] can be considered a close cousin to our DPGA. They integrate 32, 16-context, bit processors on each Hydra chip ($N_p = 32$, $c = 16$). The logic function is an 8-input NAND with programmable input inversions.

VEGA [8] uses 1K–2K context memories to achieved a 7× logic description density improvement over single context FPGAs. At $c \geq 1024$, VEGA is optimized to be efficient primarily for very long cycles or low throughput and can be quite inefficient for regular, high-throughput tasks. VEGA uses a retiming scheme which looks more like a conventional, processor register file than the shift register scheme we use here. This allows a storage location to

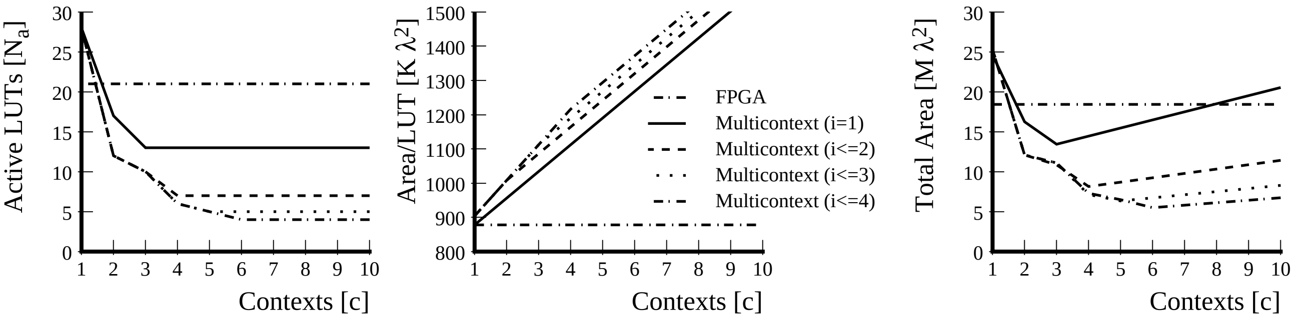

Fig. 9. Active LUT and Area Requirements v/s c and i for ASCII Hex→Binary Conversion

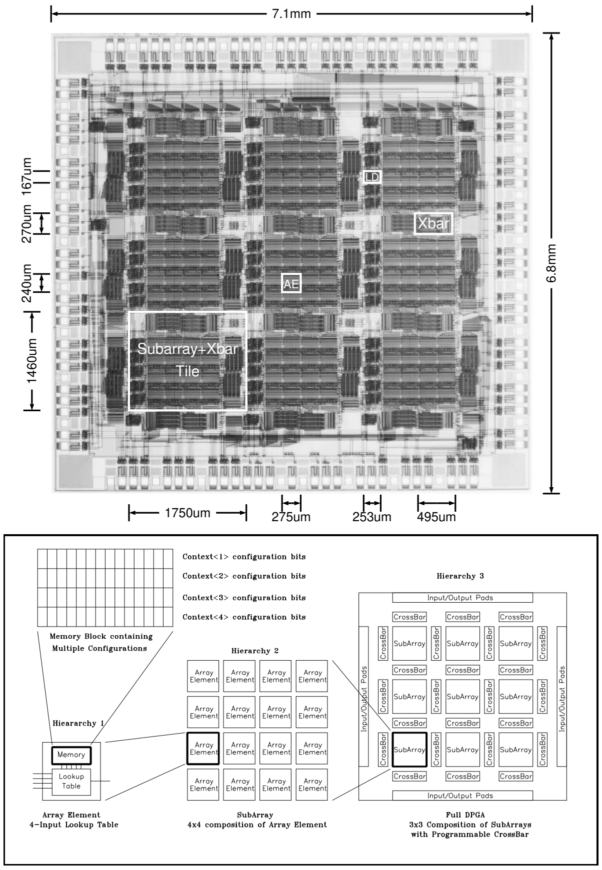

Fig. 10. DPGA Prototype

be used more than once, which has increasing importance when the retiming distances become large. At VEGA's design point, context memory is the dominant component of chip area, so it is not surprising that VEGA achieves a density improvement close to the 10× ratio between active logic and descriptions.

Dharma [9] was designed to solve the FPGA routing problem. Logic is evaluated in strict levels similar to the scheme used for circuit evaluation in Section VI with one gate-delay evaluation per cycle. Dharma is based on a few, monolithic crossbars which are reused at each level of logic. Once gates have been assigned to evaluation levels, the full crossbar makes placement and routing trivial. While this arrangement is quite beneficial for small arrays, the scaling rate of the full crossbar makes this scheme less attractive for large arrays.

DRFPGA [10] takes a different approach to relaxing retiming restrictions. They insert separate buffering resource which are dedicated to retiming data. These buffers can be smaller than LUT processing elements, but still require space for routing if they are to be flexibly deployed as needed in a circuit.

VI. Mapping Logic Netlists

In this section we describe the results of automatically mapping several MCNC benchmark circuits to multicontext FPGAs. The goal is three fold:

1. demonstrate CAD to automatically exploit multiple contexts FPGAs
2. show the benefits broadly across a benchmark set
3. identify the most promising region in the design space

As with our example in Section III, netlists were mapped using SIS [4] and Chortle [5], both in area mode. No modifications have been made to this technology mapping phase to tailor for multicontext evaluation. The mapped netlist characteristics for the benchmark set are shown in Table II.

As we have seen, the number of active LUTs needed depends on the throughput requirement. In order to compare implementations across throughput requirements, we pipeline each implementation to achieve a range of target throughputs. That is, if we wish to produce a new result every n LUT delays, we add pipelining registers every n LUTs in the critical path. For example, if the critical path on a circuit is 8 LUT delays long and the desired throughput is one result ever 2 LUT delays, we break the circuit into four pipeline stages, adding registers every 2 LUT delays in the original circuit. We use a simple annealing algorithm to assign non-critical path LUTs in order to minimize the number of retiming registers which must be added to the design. In hindsight, we could have performed the retiming more efficiently using Leiserson's retiming algorithm [11].

Similarly, we divide the multicontext cases into separate spatial pipeline stages such that the path length between pipeline registers is equal to the acceptable period between results. The LUTs within a phase are then evaluated in multicontext fashion using the available contexts. Again,

TABLE II
BENCHMARK SET CHARACTERISTICS

Circuit	levels	N_d
5xp1	10	46
9sym	7	123
9symml	8	108
C499	10	85
C880	21	176
alu2	19	169
apex6	9	248
apex7	7	77
b9	7	46
clip	9	121
cordic	13	367
count	16	46
des	13	1267
e64	9	230
f51m	17	45
misex1	6	20
misex2	8	38
rd73	10	105
rd84	9	150
rot	16	293
sao2	9	73
vg2	9	60
z4ml	7	8

if the critical path on a circuit is 8 LUT delays long and the desired throughput is one result every 2 LUT delays, we break the circuit into four spatial pipeline stages, adding registers ever 2 LUT delays in the original circuit. The spatial pipeline stage is further subdivided into two temporal pipeline stages which are evaluated using two contexts. This multicontext implementation switches contexts on a one LUT delay period. Similarly, if the desired throughput was only one result every 4 LUT delays, the design would be divided into 2 spatial pipeline stages and up to 4 temporal pipeline stages, depending on the number of contexts available on the target device. The same annealing algorithm is used to assign spatial and temporal pipeline stages to non-critical path LUTs in a manner which minimizes the number of total active LUTs required in the levelized circuit.

After all levelization has been done, a grouping pass is performed. The grouping pass attempts to group together c logical LUTs within a spatial partition to reside on one physical LUT. Here, no attempt is made to share physical LUTs among spatial pipelining stages; it may be possible to obtain tighter packings if such sharing is permitted. For a group of LUTs to be compatible, it must be possible to permute the LUTs' inputs such that no two LUTs require a different value to arrive on the same input on the same clock cycle. Rather than trying all $(4!)^{(c-1)}$ permutations, we use a randomized, greedy LUT input assignment heuristic to check compatibility.

Grouping is performed independently on each spatial partition. The grouping routine starts by packing all the logical LUTs in a spatial partition into the minimum number of physical LUTs—*i.e.* the number of physical LUTs required to implement the largest temporal stage. The attempt is made by first randomly assigning logical LUTs to physical LUTs, then greedily swapping logical LUTs in order to reduce incompatibility conflicts. Swapping continues until a compatible set of groupings is found or the swapping runs longer than a predetermined time limit which is proportional to the number of logical LUTs which can be described in the spatial partition. When packing fails, we increment the number of target physical LUTs and retry packing. The grouping results are generally within a few LUTs of the packing limits.

Figure 11 shows the relative area of various multicontext FPGA implementations as compared to a single context FPGA. The data shown is the average across the benchmark set (Table II). The first eight graphs show the effects of varying retiming depth for a particular choice for c. For these experiments, we never set i greater than c. The final, composite, graph mixes several (i,c) design points into a single graph for comparison. Without data retiming, support we see little net area improvement on average. However, with only four retiming stages ($i = 4$), we can get most of the area benefit achievable on this benchmark set.

With a LUT-cycle delay in the 7–9.5 ns range, even today's "high" throughput implementations in the 30–50 MHz range are producing new results only once ever 3–5 LUT delays. At these speeds 3–4 context devices are 40–50% smaller than the single context implementation. At lower throughputs, the multiple context implementations are almost one-third the size of the single-context implementation on average.

As processing continues to improve and the native LUT-cycle delay decreases, we will have three classes of applications:

1. **unlimited throughput requirements** – these applications will be able to directly exploit the decreased FPGA cycle time in the traditional way.
2. **latency limited throughput** – tasks whose throughput has been limited by the latency of internal cycles, may run faster, but their period in terms of LUT-cycle delays will remain the same. These applications will continue to fall at the same point on the normalized throughput curve, and hence continue to to extract the same benefit from multicontext implementation.
3. **limited throughput requirements** – applications with limited throughput requirements will see their permissible L increase as the native LUT-cycle delay decreases. Consequently, these applications will derive greater benefit from multicontext implementations with improved native device speeds.

VII. Summary

Standard FPGA applications typically run below the device's peak throughput due to cyclic paths in the logic or limited external throughput requirements. This leaves active device resources idle most of the time. Multicontext FPGAs allow us to reclaim this lost capacity and translate it into smaller area requirements.

These multicontext devices exploit the fact that the

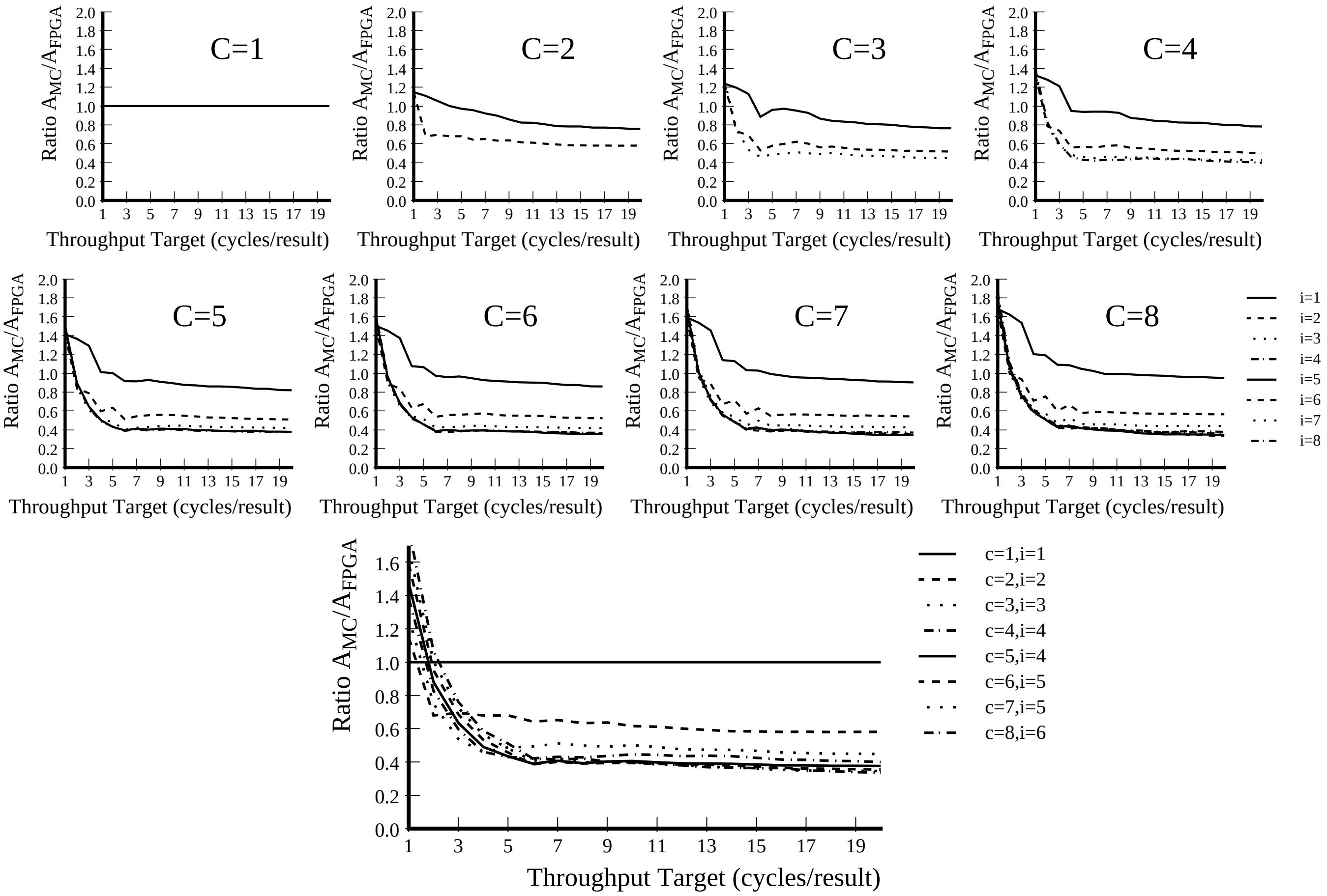

Fig. 11. Relative Area v/s Throughput for Various c and i

memory to configure FPGA resources is much smaller than the active resources themselves. By placing memory for several configurations next to each active resource, a small area increase per processing element translates into a large reduction in the number of requisite processing elements, and consequently, a net reduction in area.

Multiple configuration contexts alone, however, are not sufficient to fully realize this area reduction. We must also take care to transport intermediate data through the device in time. In single context devices, these intermediate values could reside on the outputs of idle processing elements, but as we reuse these processing elements it becomes necessary to find less expensive ways to bridge data between production and consumption. Our input retiming registers provide one such scheme, allowing us to inexpensively retime data and, consequently, reduce the net area required to implement logic functions.

As a result, multicontext FPGAs with several contexts (c=4–8) and retiming stages (i=4–6) are half to one-third the size of single context FPGAs for typical application characteristics. As device cycle time decreases along with processing improvements the advantages of multicontext evaluation are likely to stay the same (*e.g.* latency limited tasks) or increase (*e.g.* throughput limited tasks).

Acknowledgments

This work was performed as part of the Reinventing Computing Project at MIT under the direction of Dr. Thomas F. Knight, Jr. The research was supported by the Defense Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

The original idea for a "Dynamically Programmable Gate Array" was developed with Michael Bolotski. Jeremy Brown, Derrick Chen, Ian Eslick, Ethan Mirsky, and Edward Tau made the DPGA prototype implementation possible. Their efforts were invaluable in developing these ideas to their current level of maturity.

References

[1] Jonathan Rose, Robert Francis, David Lewis, and Paul Chow, "Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1217–1225, October 1990.

[2] André DeHon, "Entropy, counting, and programmable interconnect," in *Proceedings of the 1996 International Symposium on Field Programmable Gate Arrays*. ACM/SIGDA, February 1996.

[3] André DeHon, "Reconfigurable architectures for general-purpose computing," AI Technical Report 1586, MIT Artificial Intelligence Laboratory, 545 Technology Sq., Cambridge, MA 02139, October 1996.

[4] 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, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, May 1992.

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

[6] 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*, May 1995, pp. 138–143.

[7] Lisa Maliniak, "Hardware emulation draws speed from innovative 3d parallel processing based on custom ics," *Electronic Design*, pp. 38–41, May 30 1994.

[8] David Jones and David Lewis, "A time-multiplexed fpga architecture for logic emulation," in *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*. IEEE, May 1995, pp. 495–498.

[9] Narasimha B. Bhat, Kamal Chaudhary, and Ernest S. Kuh, "Performance-oriented fully routable dynamic architecture for a field programmable logic device," UCB/ERL M93/42, University of California, Berkeley, June 1993.

[10] Douglas Chang and Malgorzata Marek-Sadowska, "Buffer minimization and time-multiplexed i/o on dynammically reconfigurable fpgas," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, February 1997, pp. 142–148.

[11] Charles Leiserson, Flavio Rose, and James Saxe, "Optimizing synchronous circuitry by retiming," in *Third Caltech Conference On VLSI*, March 1993.