Choose-Your-Own-Adventure Routing: Lightweight Load-Time Defect Avoidance

Raphael Rubin
Computer and Information Science
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104
rafi@seas.upenn.edu

André DeHon
Electrical and Systems Engineering
University of Pennsylvania
200 S. 33rd Street
Philadelphia, PA 19104
andre@acm.org

ABSTRACT

Aggressive scaling increases the number of devices we can integrate per square millimeter but makes it increasingly difficult to guarantee that each device fabricated has the intended operational characteristics. Without careful mitigation, component yield rates will fall, potentially negating the economic benefits of scaling. The fine-grained reconfigurability inherent in FPGAs is a powerful tool that can allow us to drop the stringent requirement that every device be fabricated perfectly in order for a component to be useful. To exploit inherent FPGA reconfigurability while avoiding full CAD mapping, we propose lightweight techniques compatible with the current single bitstream model that can avoid defective devices, reducing yield loss at high defect rates. In particular, by embedding testing operations and alternative path configurations into the bitstream, each FPGA can avoid defects by making only simple, greedy decisions at bitstream load time. With 20% additional tracks above the minimum routable channel width, routes can tolerate 0.01% switch defect rates, raising yield from essentially 0% to near 100%.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design aids—*Placement and routing*; B.7.3 [**Integrated Circuits**]: Reliability and Testing; B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms

Algorithms, Design, Experimentation, Reliability

Keywords

alternatives, bitstream load, defect tolerance, in-field repair, programmable interconnect

FPGA'09, February 22–24, 2009, Monterey, California, USA.

1. INTRODUCTION

When we shrink feature sizes, the variation of component characteristics increases, as do the chances that a device is unusable. Coupled with increasing device count enabled by smaller feature sizes, this reduces the probability of yielding a chip with a perfect set of devices (Section 2.1). If this issue is left unmitigated, chip yields will decrease, eventually limiting chip capacity and feature scaling.

Since FPGAs provide a large pool of fine-grained interchangeable resources, FPGA architectures already have the structural organization necessary to tolerate defective devices (Section 2.2). If we can arrange to assign logic functions in such a way as to avoid the defective devices, an FPGA can accommodate relatively large defect rates. This allows (1) earlier access to advanced technologies, (2) larger chip sizes, and (3) beneficial scaling to smaller feature sizes even if the smaller features come with higher defect rates.

If defect locations are known, it is relatively easy for CAD mapping tools to avoid the defective devices. However, such component-specific mapping disrupts the current single bitstream model and is not considered viable. Potentially unacceptable costs are associated with:

- producing a defect map, rather than simply checking if all the devices are perfect
- performing an expensive CAD mapping per FPGA (requiring workstation-class processor and memory capacity)
- tracking a different bitstream for each component

These costs include time on potentially expensive testers, time and energy consumption on CAD workstations, and flow complications due to the additional handling needed to create and install the unique bitstream for each component.

To exploit the reconfigurability of FPGAs to avoid defects while preserving the current single bitstream model, we introduce the Choose-Your-own-Adventure (CYA) bitstream format and bitstream loader (Section 3). The CYA bitstream includes both tests and pre-computed alternative configurations for each logical function mapped to the FPGA. The associated bitstream loader uses the embedded test to choose among the set of alternatives during the load operation. Choices are made greedily in sequential bitstream order, so there is only a linear increase in bitstream load time. This scheme shows how a very simple bitstream remapping that can easily be embedded into the FPGA will allow FPGAs to use manufacturing processes with high defect rates. The CYA bitstream format does not require changes to the core of the FPGA, including the FPGA routing architecture.

Novel contributions of this work include:

- an architecture for integrating design-specific testing
- a bitstream model with alternate routing
- an inexpensive (architecture neutral) load-time path selection technique
- an algorithm for alternate route generation (Section 3)
- a quantitative characterization of the yield enhancement benefit (Sections 4 and 5) of this technique as a function of defect rates, alternatives, and extra and reserved channels
- estimates of bitstream size and load times (Section 6)

2. BACKGROUND

2.1 Scaling Challenges

Conventional semiconductor feature sizes will not scale below the width of an atom (0.5 nm for a silicon lattice). However, long before that point, the discrete nature and statistical behavior of individual atoms may pose challenges to scaling. Traditional semiconductor doping depends on the statistics of large numbers of dopants to create consistent devices, but variation increases as device size, and hence nominal dopant count, decreases. Small features are more susceptible to movement or displacement of a few atoms and local variations in processing, such as etching and reaction rates. As a result, we expect increasing parameter variation in devices (*e.g.* [1] (Table 18, Design chapter), [4,8]). These variation effects are in addition to the traditionally increasing challenge of avoiding catastrophic photolithographic defects (*e.g.* [9]). Some have gone so far as to suggest that 20% of the transistors or other devices fabricated on a component may be unusable by the time we reach processes with a half pitch of 11 nm [7].

2.2 FPGA Defect Tolerance

Architectures that tolerate defects, such as RAMs with row or column sparing, enjoy benefits over designs that must be defect free. These benefits have traditionally included (1) early access to advanced processes and (2) large capacity dies. In the future they may also include (3) the ability to economically exploit the most advanced process nodes where the device defect rate cannot be made trivially small.

FPGAs have a regular structure of largely interchangeable resources. Therefore it should be possible to exploit this regularity for sparing-based defect tolerance. However, the conventional model of a single bitstream that maps each function to a single physical resource prevents us from exploiting this capability.

Perfect Component Model One approach to defect tolerance is to perform the defect tolerance behind the scenes, always presenting the view of a perfect component to the end system. RAM, for example, undergoes row and column sparing at the factory, such that the shipped RAMs appear defect-free to the customers. Altera has employed this kind of behind the scenes sparing at the level of rows and columns to enhance yield and allow them to produce very high capacity devices [11, 19]. Yu and Lemieux show how to spare at the finer-granularity of individual wire tracks [36]. These techniques add additional reconfiguration mechanisms and redundancy beyond the reconfigurability already inherent in the FPGA. This results in additional area and delay overheads compared to simply configuring the base FPGA to avoid defective elements.

Component-Specific Model An alternate approach is to expose the defects in the FPGA and allow CAD software to map around them. HP's TERAMAC pioneered this technique, demonstrating the ability to tolerate element defect rates of 3–10% in their components [12]. Katsuki *et al.* map out the delay of regions of a chip and use that map during placement to keep critical paths away from the slowest resources on the component [14]. The long runtime for conventional FPGA mapping tools is seen as one of the weaknesses of this approach. To accelerate the mapping process, TERAMAC employed a richer and more regular interconnect architecture than conventional FPGAs [2], trading density for mapping speed.

Local substitution of resources can potentially reduce or avoid the high cost of complete FPGA remapping at the cost of less optimal mapping solutions. Lach introduced a design style for FPGAs that reserved one or more logic blocks in an $N \times N$ tile so that defects could be repaired with local sparing [15]; Lakamraju and Tessier note that reserving spare LUTs in an island-style cluster also allows local repair of logic [16]. These two solutions focus on logic defects rather than interconnect defects. In this paper we focus on interconnect defects and describe a general scheme that could use one of these techniques for logic defects.

Component-Specific Designs In its EasyPath program, Xilinx matches a component's defects to the needs of a particular design. Since no design uses all of the features and resources in an FPGA, defects in the unused resources are tolerable. This avoids additional CAD or resources for defect tolerance, but does require design-specific testing [29], and the components assigned to a design may not be functional when changes are needed in the design.

Bitstream Alternatives Since there is never a single way to map a design to an FPGA, we can exploit this freedom to avoid defective or undesirable devices. At a coarse granularity, one could place and route a design several times to produce multiple bitstreams; then we could test the bitstreams on the component. If any of the bitstreams avoids all the defective devices, we have a successful mapping [18, 23, 28]. This fully exploits the FPGA redundancy and avoids the need for per-component mapping. However, it does demand a discipline for bitstream testing, validation, and management. This monolithic scheme does not scale well to higher defect rates; the finer granularity of our alternatives gives us greater diversity impact, and hence yield improvement, for a given size of bitstream (*e.g.* number of alternatives stored).

Campergher *et al.* [10] and Trimberger [27] describe approaches most similar to our alternatives scheme where every path is *covered* by an alternate path. Campergher also detailed how to redesign switchboxes to increase the diversity of routing options. Both Campergher and Trimberger only suggest a single alternative to tolerate a few defects. We detail a general scheme that can provide a tunable level of defect tolerance and quantify the tradeoff space; we further detail how we can embed simple testing and minimize the complexity of the bitstream loader.

2.3 Defect Model

We model defects at the architectural level as stuck-open switches. That is, we abstract the host of potential defect causes (such as disconnected metal, or slow or leaky transistors caused by excessive voltage threshold variation) as

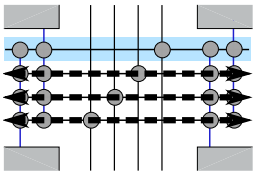

Figure 1: Channel with four tracks, one of which is reserved. Three nets saturate the three base tracks.

C-Box or S-Box switches that cannot be used to form the desired connection. This assumes that the SRAM controlling the switch can still be used to turn the switch off or that the SRAM fails into the "off" state for the switch.

The CYA model can tolerate wire defects. A configuration bit that fails in the stuck-on condition can be abstracted as a failure of the driven wire. However, the experiments in Sections 4 and 5 quantify only switch defects and not wire defects. The model can further tolerate bridging defects at the expense of more complicated test sequences.

One rationale for focusing initially on stuck-open switch defects is that configuration bits can be made more tolerant to variation than buffered switches without a power penalty. In particular, one way reduce dependence on threshold voltage (V_t) variation is to increase V_t and the total voltage swing. Increasing the voltage swing on active signal paths, such as buffers, would increase the energy per operation. With power consumption already a key limiter on computational density, there is pressure to reduce the signaling voltage swing as much as possible. However, since the configuration SRAMs do not switch, they can use a higher nominal V_t and voltage that reduces their sensitivity to variation.

3. CYA

The CYA bitstream bears some similarity to the eponymous Choose Your Own Adventure novels [21]. Unlike traditional FPGA routing, the CYA bitstream contains several alternatives for each path, and, like a reader seeking the best possible ending, the CYA loader tries each alternative one by one. Each time it encounters a defect (comparable to the novel's protagonist being tossed into a raging volcano), it tries the next alternative instead until it finds a path that successfully avoids all the defects on the chip (the protagonist finds a vast treasure and lives happily ever after).

3.1 Illustrative Example

Consider a simple FPGA channel (Figure 1). We have four tracks and need to route three nets through this channel. Labeling the fourth track as "off limits" (reserved), we route the channel, packing the nets into the first three tracks (base tracks). Then, for each consumer, we find an alternate path that uses the fourth track. When programming the FPGA, the loader tries the default route, and, if it is bad, the loader tries the alternate path. If the chip has at most one bad track in the channel, it will either be in the fourth track and will not disturb the default routes, or it will upset a single route forcing it to use the alternate path on the fourth track.

We can expand the notion of tracks to domains. By domains, we refer to architectures where the interconnect network is partitioned into independent sets of routing resources (*e.g.* [30]). With domains the initial track choice leaving a source block determines the track that will be used when entering the destination block. The independence property of domains means that when we reserve a domain, we do not affect the rest of the network. Resources that are part of the reserved domain are not available to the base domains and *vice versa*. Now we can look at a whole chip and say that default paths are routed in the base domains and alternatives are routed on the reserve domain and unused portions of the base domains. This allows us to guarantee that an FPGA with a single defect anywhere in the interconnect has an unused alternate route which will avoid the defect.

If we want to provide resilience to any two defects we can add a second domain reserved for alternatives and prepare alternative paths using each domain. We can add additional domains to tolerate more defects. However, we should not need a full alternate domain to tolerate each single defect. In normal FPGA routing, we are able to both find multiple paths between a source and sink in a single domain and route multiple two-point nets in a single domain. Consequently, each additional domain should allow us to tolerate multiple additional defects.

3.2 CYA Components

We now describe CYA bitstream composition, generation, and loading.

3.2.1 CYA Bitstream

Base Route The base route is a normal FPGA route which is prepared using only the base tracks.

Alternatives In addition to the traditional base route, the CYA bitstream includes a set of alternative paths that can be used for defect avoidance. An alternative path consists of a set of resources that connects a single source to a single sink and that differs by at least one resource from the corresponding path found in the base route. Additionally an alternative path may not use any resources which are part of the base paths of other nets. This guarantees that we are free to switch from a base path to any alternative path without interfering with any of the base routes. Alternative paths may use both the reserved spare resources and any non-reserved resources which were left unused by the base route.

Test Instructions To detect defective paths, the loader must check the functionality of each path. For a net, the test must simply check that a high-to-low and a low-to-high transition of the source are each correctly observed at the destination. The information needed for this test can be inferred from the path definitions. To generalize testing for more complex cases (*e.g.* LUT and Flip-Flop testing, testing for bridging) and to simplify the design of the loader, we embed testing instructions in the bitstream.

Composition Putting it all together, the bitstream is a list of nets each with (1) a set of testing instructions, (2) a base path, and (3) a list of alternatives. This means that bitstream programming is ordered by logical functions rather than by physical location on the FPGA. For fabrics where the configuration bits are not randomly addressable, such as Virtex frames [32,33,35], we assume the presence of a translator (see Section 3.3) that can add or remove the portion of the input path that applies to the configured frame.

3.2.2 Routing and Alternatives Generation

Base Route The base route can be generated with a standard FPGA router such as Pathfinder [20]. The only difference is that we may choose to reserve some resources

solely for use in the preparation of alternate paths. The router must therefore be capable of acknowledging these resources as being "off limits" to the base route.

Alternatives Generation As discussed earlier, the alternatives are just paths that must not conflict with the base route. We want a diverse set of alternatives for each base path in order to maximize the opportunity for defect avoidance. Much like in Pathfinder, this can be implemented with repeated calls to a shortest path search, updating the cost of each resource in the graph as it is used in order to promote diversity (See Algorithm 1). Pathfinder-based routers can easily be modified to add this functionality. During alternative generation, the cost for each resource is:

$$cost = alternatives\_using + 1 \quad (1)$$

Future work should explore more sophisticated cost functions (*e.g.* cost weightings which consider the likelihood the resource will be available when the alternate is considered).

```
foreach Resource R ∈ {base route} do Disable R;
foreach Net N ∈ {all nets} do
    foreach Resource R ∈ {N.base_path} do
        Enable R;
    end
    for i=1 to number of alternatives do
        Path P := FindShortestPath(N);
        foreach Resource R ∈ P do
            R.alternatives_using + +;
        end
        WriteAlternatePath(P);
    end
    foreach Resource R ∈ {all resources} do
        R.alternatives_using = 0;
    end
    foreach Resource R ∈ {N.base_path} do
        Disable R;
    end
end
```

Algorithm 1: Alternatives Generation

3.2.3 Bitstream Loader

We decompose the loader into four components, a programmer, a deprogrammer, a tester, and a controller.

Programmer By the term "programmer" we refer to that element of any FPGA bitstream loader that sets the configuration bits. If the architecture supports the ability to set configuration bits in any order, then we need not change the programmer from its standard design. Architectures which do not have this ability may need a translator. For example to support the Virtex series we need the ability to edit frame configurations (see Section 3.3).

Deprogrammer When a path fails, the loader needs to undo the configuration changes to free up resources for other paths. Functionally, the deprogrammer must roll back the configuration to its state before the last path was programmed. One way to accomplish this is to record changes made during programming so they can be reverted; this has the advantage of demanding no semantic understanding of the bitstream, but requires space to store the changes. An alternate version might use the same path specification for programming with the configuration sense reversed.

Tester The tester is responsible for testing each path and reporting the success or failure of the test. The bitstream loader only needs to know if the end-to-end path test fails. It does not necessarily understand anything about the resources which compose a path. The alternatives encoded in the bitstream directly tell the loader what to try next when a test fails. If the bitstream loader does not have random access into the bitstream, the loader will need adequate local space to store the current test specification to be used with the sequence of alternatives.

One simple way to support testing is to drive and recover data using the internal CLB flip flops. It may be possible to set up the tests using bitstream configuration, trigger timing tests using readback capture, then view the results using configuration and state readback [34, 35]. End-to-end connectivity tests check that we can see both driven zeros and driven ones at the destination. Timing tests can be performed using a variant of "launch-from-capture" transition fault testing (*e.g.* [22, 29]).

Controller At the top of the bitstream loader is a controller which applies the simple procedure shown in Algorithm 2 to the incoming bitstream. One key feature to note is that the controller is a very straightforward entity which makes no complex decisions and performs no complex actions. Notably, the controller does not need to understand the FPGA architecture or the semantics of the configuration bits. All the intelligence about the meaning of the bitstream is effectively compiled into bitstream. The controller only needs to mechanically follow the bitstream load program. With suitable test support, the embedded PowerPC on Virtex devices could be used to run this algorithm, using the ICAP [35] to perform the configuration.

```
Bitstream B ← read from file;
foreach Net N ∈ {B} do
    found ← FALSE;
    while (not(found)) do
        if N.outOfPaths then
            return failure;
        P ← N.nextPath;
        if P.isUsable /* interferes with no
            existing configurations */
        then
            Program(P);
            if Test(P) then
                found ← TRUE;
            else
                Deprogram(P);
            end
        end
    end
end
return success;
```

Algorithm 2: Bitstream Load

3.3 Configuration Architecture

There is a rich design space for providing the logical configuration architecture assumed by CYA. In the simplest case, we might have random access to the configuration bits (*c.f.* Xilinx 6200 [31]). This makes it fast and easy to set each bit but demands greater area overhead for configuration support than conventional configuration chains.

At the cost of more time and work loading the bitstream, we can exploit the frame schemes that exist in modern FPGA (*e.g.* [32, 33, 35]). Specifically, we could organize the bitstream to specify the frame address and the address of bits to change within the frame. Then, the loader can (1) read out the old frame, (2) change the specified bits in the frame, and (3) load the modified frame. This is the same kind of operation used by Xilinx J-bits to perform bitstream modification on Virtex series components [13]. Such a scheme would require no changes to the core of the FPGA. The cost is longer load times since we must spend an entire frame read/write sequence for every frame touched by an alternative (See Section 6.2 and Table 3).

4. EXPERIMENTS

We designed a series of experiments to determine:

- How much yield improvement can we get from CYA at each defect rate and over what defect rates is it effective?
- How many alternatives does CYA need to store? How does this impact bitstream size and load time?
- How does CYA yield improve with dedicated *reserved* tracks available only for alternatives?
- How does the presence of *extra* base tracks beyond the minimum number required for the design to be routable impact CYA yield?
- How should additional tracks be partitioned between extra and reserved tracks?

4.1 Experimental Framework

Defect Map We want to characterize the the likelihood that a given chip can be made to function correctly in the presence of a given level of initial fabrication defects (the yield). To do this we need to be able to model defects and vary the defect rate. If we simply varied the defect rate and generated independent sets of defects for each experiment, it would (a) make experiments non-repeatable, (b) prevent direct comparisons between techniques or options (*i.e.*, because experiments never see the same set of defects), (c) create occasional anomalies where an experiment at a higher defect rate outperformed an experiment at a lower defect rate (*e.g.* due to less favorable location of defects in the lower defect rate experiment). To provide clean experiments and avoid these pitfalls, we generate a collection of partially defective "chips" (defect maps) with a tunable defect rate. Specifically, we assign an independent, uniformly distributed, random value to each resource in the chip. Then we apply the target defect rate as a threshold to differentiate good resources from bad. This guarantees that defects monotonically increase with defect rates. That is, as we apply higher and higher defect rates, this process will simply add more and more defects to the existing set without removing any of the initial defects, making the results at each defect rate easily comparable to each other. By reusing the same collection of defect maps across experiments, we guarantee consistent comparisons across different techniques and defect rates.

Simulator VPR 4.3 [5] provided a scaffolding for a functional CYA simulator. We implemented each of the components in Section 3 using or modifying existing portions of the router. Some portions required more significant additions such as the ability to read route and bitstream files. Since the simulator has the advantage of global knowledge, the simulator does testing before programming. This does not alter the semantics at a functional level but means we did not need to simulate the deprogrammer.

4.2 Experimental Flow

Placement We prepare our placements with the unaltered placer in VPR with no extra options. A single placement is generated for each benchmark before all other work. That fixed placement is used for all subsequent operations.

Minimum Channel Width For experiments where channel width (the number of tracks per channel) is a controlled parameter, we needed to find the minimum routable channel width (W_{min}). For this purpose we used VPR with the "verify binary search" option. All other options remain at their default settings. Note that this means that the router uses a timing-driven alpha-beta path search.

Extra Channels Designs are seldom mapped to FPGAs at their absolute minimum channel width. Typically some "extra" channels are available, making it both easier to find a route (*e.g.* [26]) and possible to find faster, more direct routes (*e.g.* [17]). The number of extra channels we allocate is proportional to the number of minimum channels (*e.g.* $0.2W_{min}$), similar to prior work.

Base Route To generate the base route we use our modified version of VPR's fixed-width router with the option "–max_router_iterations 100". We route each design with a target channel width determined by the minimum channel width and the extra channels (*e.g.* $1.2W_{min}$ when we allocate 20% extra channels). We also specify the the number of additional reserved tracks to use only during alternative generation.

Generating Alternatives The base route is passed to the alternatives generator. Initially, all resources are available to the alternative paths, except those belonging to the base route, which are marked as "off limits". The block I/O pins belonging to each base path are also marked as available during the generation of its corresponding alternatives, but they remain off limits to the alternatives associated with any other base path.

We generate 40 alternatives for each path. We chose this number to be sufficiently large that this step need be performed only once for each base route. Loading a bitstream with these alternatives onto each defect map will then allow us to determine a sufficient number of alternatives needed to successfully load the design as a function of defect rate.

Loading the Bitstream Once we have generated a defect map, a base route, and a set of alternatives, we can simply feed them back into a bitstream loader simulating Algorithm 2 which we added to VPR. VPR then reports the success or failure of the load and, for successful loads, also reports the number of alternatives used.

4.3 Architecture Details

We used an architecture based on "4x4lut_sanitized.arch" that is distributed with the VPR source code. We kept the following key features:

- "subset" Switch boxes (S-Boxes)
- Uniform segment length of 4 ($L_{seg} = 4$)
- CLB contains 4 4-LUTs ($O = 4$)
- CLB has 10 input pins ($I = 10$)

To better fit the uniform defect rate model (Section 2.3), we use a single buffered switch for all S-Box switches. Also the addition of reserved tracks alters the staggering distribution of segments in the default staggering pattern. In our

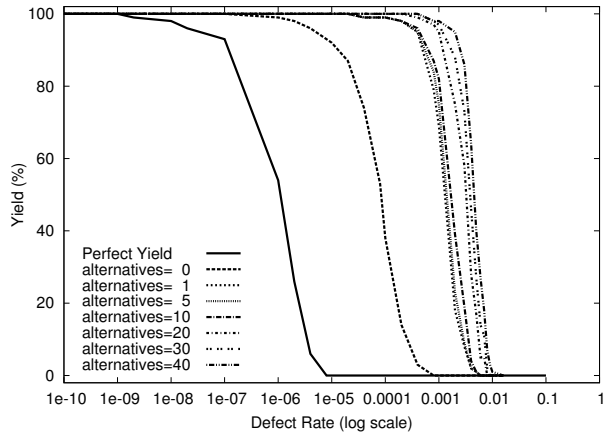

Figure 2: Yield *vs.* defect rate for DES with 20% extra base tracks and 20% additional reserved tracks

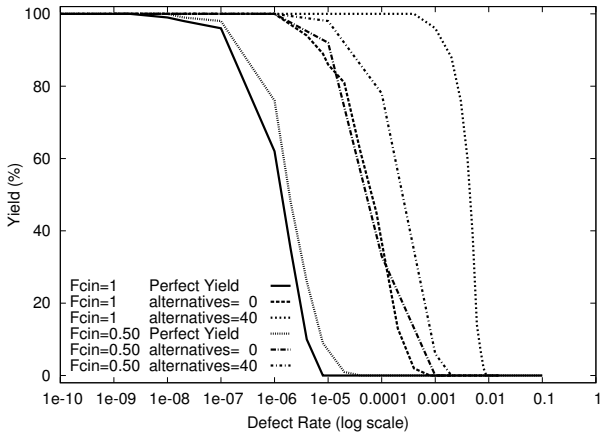

Figure 3: Yield *vs.* defect rate for DES with depopulated and fully populated C-Boxes (no extra base tracks, 20% reserved tracks, depopulated case sets W_{min})

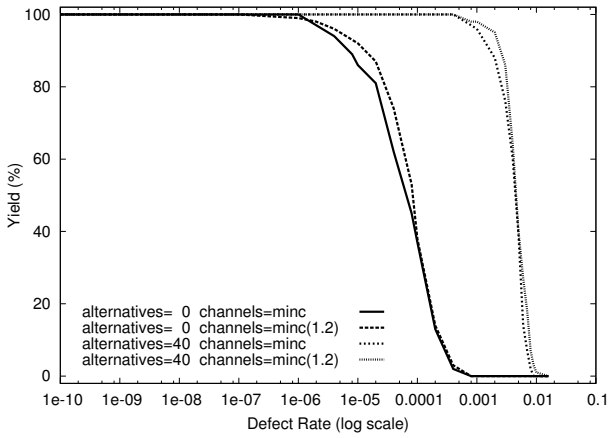

Figure 4: Effects of extra base tracks (DES with 20% reserved tracks)

architecture we therefore use a staggering pattern where increasing the number of tracks does not change the position of segments in the tracks that are already present.

Finally, for Connection Box (C-Box) population we present results with fully populated C-Boxes ($F_{cin} = 1$ and $F_{cout} = 1$) as well as $F_{cin} = 0.50$ and $F_{cout} = 0.25$ as found in the original architecture files distributed with VPR. We focus on the fully populated C-Boxes; C-Boxes should be assumed to be fully populated except when explicitly stated otherwise.

4.4 Experimental Design

We ran our experiments on the Toronto 20 benchmark set [6]. We collected two primary types of data from our simulations: the yield of functioning devices at each defect level and the number of alternatives needed to produce a functioning device. We also collected data on path lengths in order to estimate bitstream costs (Section 6).

Data was collected from 100 independently generated defect maps, each used for multiple defect rates. With 100 Bernoulli trials, the 90% confidence interval for the results reported as 100% yield is 97.5–100%; in the midrange (around 50% yield), the 90% confidence interval is ±8.1%.

5. RESULTS

Figure 2 illustrates the yield benefits of CYA. With 20% extra base tracks and an additional 20% reserved tracks, DES sustains essentially 100% yield at defect rates five orders of magnitude above the point where we begin to see defects in the FPGA (*i.e.* "Perfect Yield" curve). Even with only a single alternative, CYA achieves yield near 100% for defect rates two orders of magnitude higher than the point where one achieves high design-specific yield (*i.e.* alternatives=0 case). The figure further shows how the benefits vary with the number of available alternatives. Note that 40 alternatives provide only a slight improvement in yield over the case of 30 alternatives. This trend suggests that the addition of further alternatives beyond this point will offer little benefit.

5.1 C-Box Population

Our initial experiments with depopulated C-Boxes show more modest benefits from CYA (See Figure 3). While the yield is definitely higher with CYA than without, we do begin to see some yield-loss at much lower defect rates than with full population. It is not surprising that the lower connectivity of depopulation results in lower defect tolerance. However, we have anecdotal evidence that the depopulated cases should be more robust to defects than shown. This motivates future work to tune the CYA alternative generation or load strategy to achieve greater yield with the more limited connectivity associated with the C-Box depopulation typical of modern FPGAs.

5.2 Additional Tracks

Since a typical FPGA design is seldom routed at W_{min}, we wanted to understand the impact of larger channel widths. The larger channel width designs mean a smaller fraction of the routing resources are used, potentially leaving more resources available for alternatives. As such, designs with more *extra* channels might work even better with CYA. In fact, if we were fortunate, designs with many extra channels might not require that we *reserve* many channels exclusively for alternatives.

With our assumption of full C-Box population in place, we examined the effects of varying numbers of extra base tracks and reserved alternative tracks on CYA yields. In

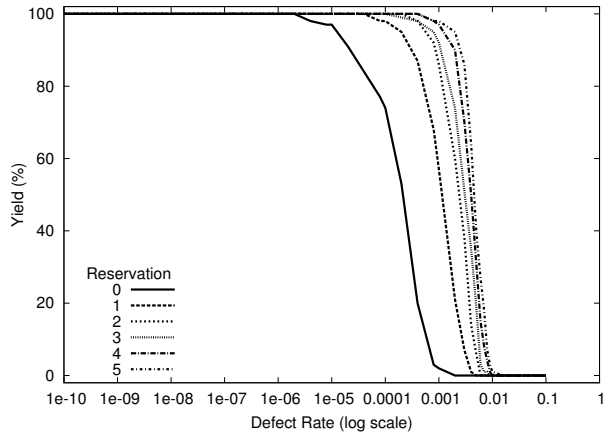

Figure 5: Effects of reserved tracks (DES with 20% extra base tracks)

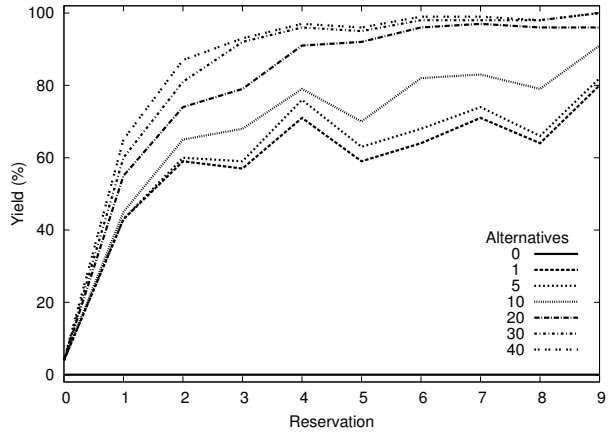

Figure 6: Effects of dividing extra tracks between the base route and alternatives (DES with 1.4W_{min} total tracks, defect rate 0.001)

Figure 4, we see that adding 20% extra base tracks to DES with 20% reserved tracks provides essentially no benefit as compared to a case with no extra base tracks. Conversely, Figure 5 shows that adding a single reserve track per channel to DES with 20% extra base tracks can provide two orders of magnitude improvement in the tolerable defect rate as compared to the same device with no reserve tracks. This demonstrates that there is value to reserving tracks exclusively for alternatives even when there are a number of extra channels. Paths in the non-reserved tracks get fragmented, preventing them from serving as effective alternatives.

Figure 6 makes this contrast between the usefulness of extra base tracks *vs.* reserved alternative tracks more explicit. In this graph, we fix at 9 (40%) the number of additional tracks (above W_{min}) while varying the division of these tracks between extra base tracks and reserved alternative tracks. These data show benefits for CYA from each additional track that is converted from an extra track to a reserved track; however, the most significant improvements come from the addition of the first four (roughly $0.2W_{min}$). In practice, the number of useful reserved tracks will be a function of defect rate.

reserved tracks →		+0%			+20%		
# alternatives →		0	1	40	0	1	40
name	LUTs	% yield					
tseng	1064	65	76	76	54	100	100
ex5p	1120	44	50	50	49	99	99
apex4	1336	48	51	51	48	100	100
dsip	1372	48	56	56	45	100	100
misex3	1448	48	49	49	49	98	100
diffeq	1516	47	56	56	48	100	100
alu4	1556	44	49	49	48	99	99
des	1660	41	60	60	37	100	100
bigkey	1708	47	56	56	41	99	100
seq	1792	41	51	51	37	100	100
apex2	1940	39	44	44	31	96	100
s298	1956	43	48	48	44	95	100
frisc	3572	8	14	14	15	97	100
elliptic	3624	15	24	24	17	89	100
spla	3820	15	20	20	5	94	99
pdc	4760	3	5	5	4	95	100
ex1010	4804	3	10	10	3	96	100
s38417	6440	3	19	19	4	97	100
s38584.1	6452	5	15	15	4	97	100
clma	8548	0	1	1	1	84	98
Geometric Mean		18	27	27	17	96	99

Table 1: CYA Yield Improvement for Toronto 20 designs, (no extra base tracks, defect rate 0.0001)

Symbol	**Definition**
s	Number of CLBs across the side of the FPGA; s^2 is the total number of CLBs in the FPGA
N_{2pt}	Number of 2-point nets in the design
T_{pl}	Total path length of a configuration: $\sum_{i=0}^{N_{2pt}} (net2pt[i].path\_length)$
T_{alt}	Total number of alternatives tried during a configuration (average across our 100 chips)
T_{plalt}	Total path length of all alternatives tried during a configuration: $\sum_{i=0}^{T_{alt}} (alt[i].path\_length)$ (average)
F_{tch}	Number of frames touched setting and clearing paths

Table 2: Bitstream Table Parameters

5.3 Summary

While our examples above all discuss results for DES only, the results for the other Toronto 20 designs were similar. Table 1 shows the results for simulations with a defect rate of 0.01% and no extra base tracks. The larger designs show dramatic yield improvements upon the addition of alternatives, improving from single-digit to near-100% yield as we go from 0 to 40 alternatives in the presence of reserve tracks.

6. BITSTREAM IMPACT

Storing and loading alternatives will make the bitstream larger and lengthen bitstream load time. Table 3 records design and experimental statistics and estimates bitstream sizes and load times. These estimates suggest that the CYA bitstream may be a factor of 2–50 larger than a conventional bitstream depending on the number of alternatives stored (Section 6.1) and take 2–200 times longer to load depending on the configuration architecture (Section 6.2).

Design					Bitstream Size (in Kbits)			Experiment Data		Load Time		
name	s	W	N_{2pt}	T_{pl}	**conv.**	**CYA-1**	**CYA-40**	T_{alt}	T_{plalt}	**conv.** (μs)	**Random Access** (μs)	**Frame Mod.** (ms)
tseng	17	29	2069	6231	131	333 (2.5)	4448 (34)	2075	6273	168	292 (1.7)	28 (164)
ex5p	17	48	2516	7564	217	409 (1.9)	5499 (26)	2532	7628	278	359 (1.3)	34 (121)
apex4	19	46	2908	11660	260	598 (2.3)	8926 (35)	2935	11794	333	497 (1.5)	44 (132)
dsip	27	28	3260	13076	319	686 (2.2)	10026 (32)	3270	13148	409	574 (1.4)	49 (119)
misex3	20	41	3215	12894	257	661 (2.6)	9871 (39)	3260	13131	328	553 (1.7)	49 (149)
diffeq	20	32	2987	8971	200	479 (2.4)	6402 (33)	2994	8996	256	420 (1.6)	40 (154)
alu4	20	38	3367	13531	238	674 (2.8)	9952 (42)	3384	13630	304	561 (1.8)	51 (166)
des	32	28	3612	14456	448	766 (1.7)	11227 (26)	3629	14484	574	639 (1.1)	54 (94)
bigkey	27	24	3661	14701	274	771 (2.8)	11272 (42)	3722	15076	350	657 (1.9)	56 (160)
seq	22	44	4000	16032	333	822 (2.5)	12272 (37)	4014	16096	426	679 (1.6)	60 (140)
apex2	23	44	4386	17592	364	923 (2.5)	13488 (38)	4463	17990	466	785 (1.7)	67 (144)
s298	23	32	4107	16479	265	865 (3.3)	12635 (48)	4173	16739	339	732 (2.2)	63 (184)
frisc	30	49	7427	29786	690	1621 (2.4)	24030 (35)	7445	29916	882	1344 (1.5)	111 (126)
elliptic	31	45	7153	35840	676	1855 (2.7)	29158 (44)	7244	36446	865	1501 (1.7)	121 (139)
spla	31	60	8757	43823	901	2285 (2.5)	36001 (40)	8796	44060	1154	1828 (1.6)	145 (126)
pdc	35	68	10968	54972	1302	3005 (2.3)	46977 (37)	11050	55702	1666	2430 (1.5)	184 (110)
ex1010	35	49	10772	43354	938	2431 (2.6)	35465 (38)	10843	43772	1201	2044 (1.7)	162 (135)
s38417	41	38	12284	36942	999	2259 (2.3)	29934 (30)	12314	37122	1278	1991 (1.6)	163 (127)
s38584.1	41	36	11185	44776	946	2514 (2.7)	36622 (39)	11248	45315	1211	2118 (1.7)	168 (139)
clma	47	58	18266	91805	2002	5140 (2.6)	79280 (40)	18422	93804	2563	4215 (1.6)	309 (121)

Parenthesized data at right of columns is ratio to conventional case.

Table 3: Bitstream Size and Load Time (defect rate 0.0001)

6.1 Bitstream Size

We estimate the number of routing configuration bits, B_{conv} for a conventional, unencoded FPGA bitstream as:

$$B_{conv} = s^2 \cdot W \cdot (F_{cin} \cdot I + F_{cout} \cdot O + 1 + 4/L_{seg}) \quad (2)$$

I, O, and L_{seg} are defined in Section 4.3. We assume 5 bits to configure switchpoints at the end of a segment (2 bits to specify which of the 4 sides is the source and 3 bits to specify drive into each of the other 3 directions) and 1 bit to configure mid-segment switchpoints. s is the side as defined in Table 2. Routing configuration make up 80–90% of a typical FPGA bitstream.

Assuming sparse storage where we must provide an address for each configuration bit in the CYA component or CLB flip-flop, we estimate the number of bits required to specify a CYA bitstream as follows:

$$\begin{aligned} B_{alt} &= N_{2pt} \cdot \left(\left\lceil \log_2\left(s^2 I \cdot W \cdot F_{cin}\right)\right\rceil + \right. \\ &\qquad \left. \left\lceil \log_2\left(s^2 O \cdot W \cdot F_{cout}\right)\right\rceil\right) \\ &\quad + (T_{pl} - 2N_{2pt}) \cdot \left(\left\lceil \log_2\left(s^2 W\right)\right\rceil + 5\right) \quad (3) \\ B_{tpath} &= N_{2pt} \cdot \left(\left\lceil \log_2\left(s^2 O\right)\right\rceil + 1\right) \cdot 5 \quad (4) \\ B_{cya} &= (N_{alt} + 1) \cdot B_{alt} + B_{tpath} \quad (5) \end{aligned}$$

N_{2pt} and T_{pl} are as defined in Table 2. B_{cya} is the total size of a bitstream with N_{alt} alternatives. B_{tpath} are the bits required to specify the test, while B_{alt} is the number of bits required for one alternative for every 2-point net. To test a two point net, we need to (1) set a zero into the driver, (2) set up a one for transition on the driver, (3) read out the result of the zero-one test, (4) set up a zero for transition on the driver, and (5) read out the result of the one-zero test; this sets the multiplier of 5 in Eq. 4. Each path starts and ends at a C-Box, so the first term in Eq. 3 is for specifying the start and end C-Box connections, while the second term is for specifying the S-Box switch settings. We again assume that an S-Box switchpoint requires 5 bits.

Beyond simply storing multiple configurations, the size overhead for the estimated CYA bitstreams comes from providing a complete address for every configuration or test resource. Exploiting locality and regularity, it should be possible to reduce this overhead cost significantly. Our approach here deliberately avoided making assumptions about the structure of the FPGA architecture and bitstream; exploiting architectural structure (*e.g.* domain) would allow more compact expression of paths and alternatives.

6.2 Bitstream Load Time

A conventional bitstream load is typically limited by load bandwidth:

$$L_{conv} = B_{conv}/BW_{load} \quad (6)$$

For concreteness, assume a system that can load 16-bit values at 50MHz (BW_{load} =16b/20ns) (*e.g.* Virtex-5 [35]).

Assuming random access into the stored bitstream, CYA can skip over alternatives that it does not need to load. The number of bits we need to read in this case is:

$$\begin{aligned} R_{cya} &= T_{alt} \cdot \left(\left\lceil \log_2\left(s^2 I \cdot W \cdot F_{cin}\right)\right\rceil + \right. \\ &\qquad \left. \left\lceil \log_2\left(s^2 O \cdot W \cdot F_{cout}\right)\right\rceil\right) \\ &\quad + (T_{plalt} - 2T_{alt}) \cdot \left(\left\lceil \log_2\left(s^2 W\right)\right\rceil + 5\right) \\ &\quad + T_{alt} \cdot \left(\left\lceil \log_2\left(s^2 O\right)\right\rceil + 1\right) \cdot 5 \quad (7) \end{aligned}$$

T_{alt} and T_{plalt} are as defined in Table 2. If we have random access to set configuration bits and set and read CLB flip-flops for testing, then the time to read the R_{cya} bits from the CYA bitstream will determine CYA load time.

Alternately, if we use a frame modification scheme (Section 3.3), time will be determined not by the number of bits changed, but by the number of frames touched and the time to shift and modify each frame. We assume the C-Box connections at the beginning and end of a path are each in one frame; for a conservative estimate, we assume that every S-Box switch in the path touches a separate frame. This means the path length is equal to the number of frames touched, so we estimate the frames touched as T_{plalt}. When a path is bad, we must unload it. So, all but T_{pl} of the frames must be touched twice.

$$F_{tch} = 2T_{plalt} - T_{pl} \quad (8)$$

For concreteness we assume 1312 bit frames similar to Virtex-5 [35] and define frame load time to match the previous bandwidth assumptions ($T_{frame} = 1312/BW_{load}$). Using Eq. 8, we estimate the CYA load time:

$$L_{frame} = F_{tch} \cdot T_{frame} + T_{alt} \cdot 5 \cdot T_{frame} \quad (9)$$

As Table 3 shows the frame scheme loads two orders of magnitude slower than a conventional bitstream load. This is the tradeoff it makes to guarantee that *no changes* are required to the core of the FPGA architecture. The random read case is a factor of 2–3 slower than the conventional case, and its slowdown is driven largely by the simplistic encoding assumed to estimate bitstream size. We expect both cases can be reduced significantly with minor modifications.

7. EXTENSIONS AND FUTURE WORK

Our work to date highlights the promise and viability of the CYA approach. It also raises many additional questions and suggests several directions for future work.

Tuning The results presented here are based on a simple diversity metric for the alternatives of a single net (Eq. 1). This leaves open the possibility that more sophisticated cost functions for alternate selection, alternate ordering, and net ordering may allow even higher robustness. As suggested in Section 5.1, we expect this may be particularly important when the C-Boxes are depopulated.

Defect Models and Chip Yield In this work we only characterized the impact of CYA on routing of stuck-open switch defects. Future work should characterize the effectiveness of CYA for stuck-closed switch defects, wire defects, bridging, and intra-cluster interconnect and LUT defects. Complete chip yield calculations will also need to account for the non-repairable portions of the FPGA die.

Scaling Since the benchmark designs in the Toronto 20 set are small compared to modern FPGAs, it is important to understand how effective CYA is as design and chip sizes scale up. For example, it will be important to characterize how the required percentage of spare tracks or alternatives scale compared to chip sizes.

Variation The CYA techniques may be adapted to deal with variations and timing. Rather than simply identifying a path as good or bad, we can identify whether or not a path is fast enough to be usable to achieve a particular timing goal. These extensions will demand time-sensitive tests and time targets for nets (*e.g.* delay budget distribution [24]).

Interaction with Architecture In this work, we have deliberately focused on characterizing the benefit of CYA on the most standard FPGA architectures. It will be useful to characterize the impact of common architectural options (*e.g.* alternate S-Box, C-Box, and LUT cluster designs) on CYA benefits, including exploring additional architectural options which may enhance CYA effectiveness.

Lifetime Extension CYA alternatives can also be used to compensate for lifetime wear. That is, at these small feature sizes, component characteristics change by large amounts during operation (*e.g.* NTBI, electromigration, hot-carrier effects) [4, 8, 25] and potentially fail completely. With the CYA bitstream loader performing tests and alternative selection as an integral part of the load operation, it is also capable of avoiding in-field failure of individual resources. Simply reloading the bitstream after a new in-field defect will reroute around new defects just as it does manufacturing-time defects. We can view the defect rates shown on the graphs in the previous section as estimating the total accumulated defect fractions the component can tolerate before it becomes unusable. A lifetime extension strategy might be to make sure shipped devices have a sufficiently lower defect rate and use the residual defect tolerance to support in-field repair. To fully realize this benefit, we will also need to develop techniques for online detection of timing violations (*e.g.* Razor latches [3]).

8. CONCLUSIONS

We have introduced the Choose-Your-own-Adventure router, a new, component-specific mapping algorithm for FPGAs, and shown that it can tolerate high defect rates. The use of a single bitstream means that we only pay for CAD once for all chips instead of once for each chip. By keeping the loader simple and adding at most 20% additional tracks, we limit the hardware cost of CYA. In exchange for this comparatively small cost, we reap major improvements in defect tolerance and substantial yield recovery. With 0.01% switch defects we show an improvement from 17% yield without CYA to near 100% with CYA; even a single alternative raises yield to 96%.

9. ACKNOWLEDGMENTS

This research was funded in part by National Science Foundation grants CCF-0403674 and CCF-0726602. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Anne Hanna provided significant review and editing assistance. Benjamin Gojman and Nikil Mehta provided valuable feedback on several drafts.

10. REFERENCES

[1] International technology roadmap for semiconductors. <http://www.itrs.net/Links/2005ITRS/Home2005.htm>, 2005.

[2] R. Amerson, R. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Plasma: An FPGA for million gate systems. In *FPGA*, pages 10–16, February 1996.

[3] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with Razor. *IEEE Computer*, 37(3):57–65, March 2004.

[4] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance CMOS variability in the 65-nm regime and beyond. *IBM J. Res. and Dev.*, 50(4/5):433–449, July/September 2006.

[5] V. Betz. VPR and T-VPack: Versatile Packing, Placement and Routing for FPGAs. <http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html>, March 27 1999. Version 4.30.

[6] V. Betz and J. Rose. FPGA Place-and-Route Challenge. <http://www.eecg.toronto.edu/~vaughn/challenge/challenge.html>, 1999.

[7] S. Borkar. Microarchitecture and design challenges for gigascale integration. <http://www.microarch.org/micro37/presentations/MICRO37f>, December 2004. Keynote Talk at the 37th Annual IEEE/ACM International Symposium on Microarchitecture.

[8] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November–December 2005.

[9] N. Campregher, P. Y. K. Cheung, G. A. Constantinides, and M. Vasilko. Yield modelling and yield enhancement for FPGAs using fault tolerance schemes. In *FPL*, 2005.

[10] N. Campregher, P. Y. K. Cheung, G. A. Constantinides, and M. Vasilko. Reconfiguration and fine-grained redundancy for fault tolerance in FPGAs. In *FPL*, 2006.

[11] R. G. Cliff, R. Raman, and S. T. Reddy. Programmable logic devices with spare circuits for replacement of defects. United States Patent Number: 5,434,514, July 18 1995.

[12] W. B. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. Defect tolerance on the TERAMAC custom computer. In *FCCM*, pages 116–123, April 1997.

[13] S. Guccione, D. Levi, and P. Sundararajan. JBits: Java based interface for reconfigurable computing. In *Proc. MAPLD*, 1999.

[14] K. Katsuki, M. Kotani, K. Kobayashi, and H. Onodera. A yield and speed enhancement scheme under within-die variations on 90nm LUT array. In *CICC*, pages 601–604, 2005.

[15] J. Lach, W. H. Mangione-Smith, and M. Potkonjak. Efficiently Supporting Fault-Tolerance in FPGAs. In *FPGA*, pages 105–115, February 1998.

[16] V. Lakamraju and R. Tessier. Tolerating operational faults in cluster-based FPGAs. In *FPGA*, pages 187–194, 2000.

[17] A. Marquardt, V. Betz, and J. Rose. Timing-driven placement for FPGAs. In *FPGA*, pages 203–213, 2000.

[18] Y. Matsumoto, M. Hioki, T. K. H. Koike, T. Tsutsumi, T. Nakagawa, and T. Sekigawa. Suppression of intrinsic delay variation in FPGAs using multiple configurations. *ACM Tr. Reconfig. Tech. and Sys.*, 1(1), March 2008.

[19] C. McClintock, A. L. Lee, and R. G. Cliff. Redundancy circuitry for logic circuits. United States Patent Number: 6,034,536, March 7 2000.

[20] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *FPGA*, pages 111–117. ACM, February 1995.

[21] E. Packard. *The Cave of Time*. Bantam Books, 1979.

[22] J. Saxena, K. M. Butler, J. Gatt, R. Raghuraman, S. P. Kumar, S. Basu, D. J. Campbell, and J. Berech. Scan-based transition fault testing - implementation and low cost test challenges. *Proc. Intl. Test Conf.*, pages 1120–1129, 2002.

[23] P. Sedcole and P. Y. K. Cheung. Parametric yield modeling and simulations of FPGA circuits considering within-die delay variations. *ACM Tr. Reconfig. Tech. and Sys.*, 1(2), June 2008.

[24] K. So. Enforcing long-path timing closure for FPGA routing with path searches on clamped lexicographic spirals. In *FPGA*, pages 24–33, 2008.

[25] S. Srinivasan, R. Krishnan, P. Mangalagiri, Y. Xie, V. Narayanan, M. J. Irwin, and K. Sarpatwari. Toward increasing FPGA lifetime. *IEEE Trans. on Dep. and Secure Comput.*, 5(2):115–127, 2008.

[26] J. S. Swarz, V. Betz, and J. Rose. A Fast Routability-Driven Router for FPGAs. In *FPGA*, pages 140–149. ACM/SIGDA, February 1998.

[27] S. M. Trimberger. Structures and methods of overcoming localized defects in programmable integrated circuits by routing during the programming thereof. United States Patent Number: 7,251,804, July 31 2007.

[28] S. M. Trimberger. Utilizing multiple test bitstreams to avoid localized defects in partially defective programmable integrated circuits. United States Patent Number: 7,424,655, September 9 2008.

[29] R. W. Wells, Z.-M. Ling, R. D. Patrie, V. L. Tong, J. Cho, and S. Toutounchi. Application-specific testing methods for programmable logic devices. United States Patent Number: 6,817,006, November 9 2004.

[30] Y.-L. Wu, S. Tsukiyama, and M. Marek-Sadowska. Graph based analysis of 2-D FPGA routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(1):33–44, January 1996.

[31] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *XC6200 FPGA Advanced Product Specification*, version 1.0 edition, June 1996.

[32] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Xilinx Virtex-II 1.5V Platform FPGAs Data Sheet* , July 2002. DS031 <http://www.xilinx.com/partinfo/ds031.pdf>.

[33] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Xilinx Virtex-4 Family Overview*, June 2005. DS112 <http://direct.xilinx.com/bvdocs/publications/ds112.pdf>.

[34] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Virtex FPGA Series Configuration and Readback*, March 2005. XAPP 138 <http://www.xilinx.com/bvdocs/appnotes/xapp138.pdf>.

[35] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Virtex-5 FPGA Configuration User Guide*, September 2008. UG191 <http://www.xilinx.com/bvdocs/userguides/ug191.pdf>.

[36] A. J. Yu and G. G. Lemieux. Defect-tolerant FPGA switch block and connection block with fine-grain redundancy for yield enhancement. In *FPL*, pages 255–262, 2005.