Efficient Error Detection, Localization, and Correction for FPGA-Based Debugging

John Lach
UCLA EE Department
56-125B Engineering IV
Los Angeles, CA 90095
310-794-1630
jlach@icsl.ucla.edu

William H. Mangione-Smith
UCLA EE Department
56-125B Engineering IV
Los Angeles, CA 90095
310-206-4195
billms@ee.ucla.edu

Miodrag Potkonjak
UCLA CS Department
3532G Boelter Hall
Los Angeles, CA 90095
310-825-0790
miodrag@cs.ucla.edu

Abstract

Simulations for modern designs are often performed on Field Programmable Gate Array technology in a functional test and debugging process known as emulation, allowing for more complex simulations than possible in software. One drawback to emulation is the lengthy time spent in the back-end CAD tools for each debugging iteration, including debugging changes and the introduction of control and observation logic. We have developed a technique that confines the re-place-and-route area to only the portions of the design affected by the introduction of the test logic and by the debugging changes. Therefore, the back-end CAD effort for error detection, localization, and correction is reduced. This benefit is achieved by partitioning the design at the physical level into independent blocks, and the test logic and design changes are localized to the affected blocks. The result is a shortened time between debugging iterations, and thus a shortened time-to-market for the design.

1 Introduction

Debugging is an inherently cyclic process. At each cycle, four steps are essential: test pattern generation, error detection, error localization, and error correction. These steps are repeated until sufficient design confidence is established. Software simulation has been, and continues to be, a common tool for hardware debugging, and the introduction of control and observation logic for error detection and localization as well as the process of error correction are straightforward in software.

However, these software tools are often unable to simulate designs at high enough speeds to enable complex testing. Therefore, recent hardware design testing strategies have emerged that involve mapping designs to Field Programmable Gate Arrays (FPGAs) for more efficient functional testing through logic emulation. For example, emulation was used in designing the UltraSPARC-I and provided "advanced debug facilities to rapidly isolate any functional problems [3]." Pre-tapeout confidence also was increased due to the more complex simulations made possible by the enhanced testing environment. Therefore, the number of silicon iterations, a costly and lengthy process, was reduced. Such advantages helped to shorten the time-to-market of the design [3].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2000, Los Angeles, California
©2000 ACM 1-58113-187-9/00/0006..$5.00

1.1 Motivation

For emulation, the four step debugging sequence is done in hardware. Test pattern generation remains the same as in software simulation, but error detection, localization, and correction must all be implemented in the emulation hardware. Detection and localization require the insertion of control and observation logic in the areas to be examined. Error correction requires the changing of the FPGA physical layout. This insertion and correction requires extensive use of the back-end CAD tools for the physical layout alterations. For emulation to remain an effective alternative to software simulation, the added CAD tool effort to implement these last three debugging steps in hardware must not be excessive. To generate an entire UltraSPARC-I configuration for emulation consumed approximately 36 hours using five computers for the main work and 70 computers during logic mapping [3]. Spending 36 hours between each debugging iteration would be unreasonably inefficient.

We have developed an FPGA physical design partitioning technique that shortens the time between debugging iterations by reducing the back-end CAD tool effort required to implement these three debugging steps. The physical design is partitioned into independent blocks, and the design alterations resulting from the introduction of the test logic and the debugging changes are localized to these blocks. Therefore, the modified area is confined to only the affected portions of the design.

Currently, design partitioning techniques for emulation are only as fine-grained as the synthesis tools allow, and rarely is the structure that exists at higher levels in the design hierarchy carried through to the physical level. Therefore, each change to the design (in the hardware description language (HDL) or in a design representation at a different level (e.g. the design netlist)) for any debugging step likely is not localized in the physical layout further than the often large functional blocks as defined by high level partitioning. This flattening of the design requires back-end CAD tools to re-place-and-route large blocks for every debugging cycle, regardless of how small the logic introduced and debugging changes are. A significant amount of time is lost between each debugging iteration, needlessly extending the debug cycle.

While recent advances in incremental place-and-route have helped to reduce the time spent in the back-end tools, the process is still lengthy. Adding logic (either for error detection and localization or as an error correction) to high density areas may require an

incremental tool to re-place-and-route a much larger portion of the design to make sufficient room for the new logic. Consequently, portions of a design originally unaltered during these debugging steps are often re-placed-and-routed by incremental tools due to necessary surrounding physical design alterations.

To minimize this re-place-and-route effort, the proposed physical design partitioning technique increases the granularity and localizes the necessary physical design alterations for each debugging step. Because only the affected portions of the design need be re-placed-and-routed, the back-end CAD tool effort is reduced to a fraction of that required by existing approaches (e.g. incremental place-and-route, Quick_ECO [2], etc.). Therefore, the time between emulation and debugging iterations is reduced, and emulation's advantages over software simulation for complex testing is enhanced.

1.2 Motivational Example

Our scheme centers around partitioning the physical design into independent blocks (tiles) with fixed interfaces, much in the same way that design partitioning is done during synthesis. As a result, logic addition and debugging changes can be made within each tile without affecting the rest of the design.

Consider the Boolean function Y=(A∧B)∧(C∨D), which might be implemented in a tile containing four logic blocks as shown in Figure 1.I. The same function can be implemented in a number of different ways (e.g. Figures 1.II-IV), but the fixed interface leaves the rest of the design unchanged with the possible exception of the performance of the circuit due to changes in routing, which is unimportant for emulation[1].

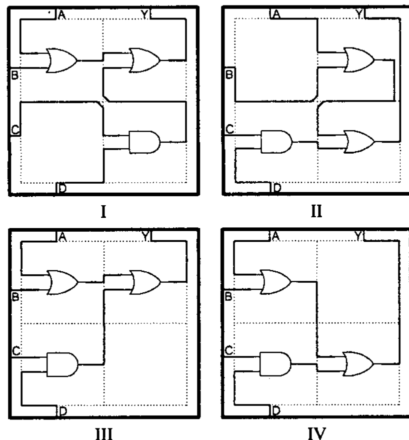

Figure 1. Four Functionally Equivalent Tile Implementations

Similarly, the functionality of the tile can be altered without affecting the rest of the physical design. For example, a debugging change may alter the tile's functionality to Y=(A∧B)∨(C∨D). Figure 2 shows how the change could be locally implemented using tiling. Again, the fixed interface allows the back-end CAD tool to re-place-and-route only the affected tile. This not only shortens the time spent in the back-end CAD tool, but keeping the rest of the design fixed insures that no errors will be introduced in the unchanged portions of the design during re-place-and-route.

[1] Emulation is primarily concerned with functional testing. Therefore, circuit performance is not a paramount issue.

Finally, given enough unused resources in a tile, logic can be introduced. The amount of logic required for controllability and observability varies, as does that introduced by a debugging change. Therefore, if there are enough unused resources in a single tile, logic can be introduced into the tile without affecting the rest of the design. If more resources are needed, neighboring tiles can also be re-placed-and-routed as they may contribute some of their unused resources for the addition. Tile boundaries are then re-established, and the interfaces are re-locked.

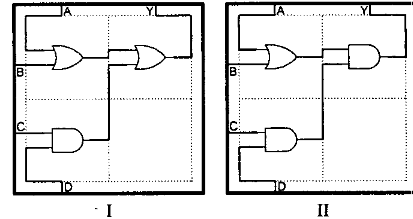

Figure 2. Example of Localized Functional Change

The CAD tool effort to re-place-and-route these small tiles is much less than is required for the coarser grained partitions created by higher levels in the design process. As a result, the turnaround time for debugging iterations is reduced.

2 Related Work

Emulation has been used for functional verification for several years. Recently, advancements have been made improving debugging through emulation that reduce the time it takes to implement debugging changes by shortening the previously long trips through back-end CAD tools. In the past, entire designs would be re-synthesized, remapped, repartitioned, and re-placed-and-routed. The Quick_ECO approach, developed by Fang, Wu, and Yen, tracks the partitioning through the design process in order to isolate design changes [2]. The system traces debugging changes through partitioning down to the netlist level, reducing the amount of the design that needs to run through the time consuming CAD tool process. Therefore, the level of granularity was the functional bock, as partitioned at the netlist level. Instead of having to re-place-and-route the entire design for each debugging change, only the affected functional blocks (linked by Quick_ECO) were processed.

Tiling reduces the time between iterations further, as it enables a more fine-grained approach than the Quick_ECO system. Quick_ECO stops its tracing at the netlist level, while tiling continues the tracing down to the physical level. This allows the designer to control the size of the physical design portions that must be re-placed-and-routed instead of entire functional blocks, which are often quite large and may span multiple FPGAs. Tiling also enables the efficient introduction of control and observation logic with this fine granularity. Therefore, back-end CAD tool effort is minimized for debugging changes.

Other techniques have been developed to minimize logic re-synthesis [12] and restrict synthesis to modified portions of the design after an engineering change [1], reducing the front-end CAD effort. Similarly, lower level design perturbation due to high level engineering changes has been limited by another methodology [4], thus helping to minimize the number of affected

tiles at the physical level. Employing these time and effort reduction front-end techniques, as opposed to synthesis tools that focus on power, speed, or area [7, 10, 11], along with tiling at the back-end reduces the total time between debugging iterations.

3 Tiling and Emulation

FPGA physical design partitioning introduces simplicity and fine granularity to FPGA designs, capitalizing on the same benefits as higher level synthesis and place-and-route partitioning do. The IC design process is not completely linear or terminal; it can often be cyclical or tangential. Therefore, partitioning at the physical level may provide some needed simplicity and fine granularity as it does at other levels in the design hierarchy.[2]

Emulation-based debugging is a perfect example of the often cyclical nature of the IC design process. The physical design is not a terminal point, but rather a level that is iteratively revisited for each debugging iteration. Therefore, simplicity and fine granularity at this level are valuable.

3.1 Global Flow

Tiling provides this simplicity and granularity by partitioning the FPGA physical design into independent blocks, as discussed in Section 1.2. Upon each return to the physical design for each debugging iteration, the independent nature of the blocks allows the back-end CAD tool to focus only on the affected portions of the design. The pseudo-code below reveals the cyclical nature of emulation-based debugging and how tiling fits within it.

```
1.  synthesize original HDL file(s);
2.  partition, map, place-and-route original netlist(s);
3.  emulate;
4.  if design error {
5.      re-place-and-route with resource slack;
6.      draw tile boundaries;
7.      lock tile interfaces;
8.  }
9.  while design error {
10.     generate test patterns;
11.     amend HDL file(s) {
12.         incorporate changes from previous iteration;
13.     }
14.     synthesize amended HDL file(s);
15.     map new netlist(s);
16.     determine test points;
17.     identify and clear affected tiles;
18.     introduce control logic;
19.     introduce observation logic;
20.     place-and-route affected tiles;
21.     emulate;
22. }
```

The tiling performed in steps 4-8 sets up the links to the physical level and creates independent blocks that can be altered based on test logic introduction or debugging changes. Step 5 leaves enough resources (a user-controlled parameter) in each tile unused for future logic introduction. If an error is discovered during emulation, the iterative process of steps 9-22 begins. Step 10 does not affect tiling, as test patterns are determined by software. Steps 11-15 are performed just as they would be with current emulation systems, using the synthesis discussed in Section 2 to minimize front-end CAD effort. The test points (desired control and observation logic locations) are determined in step 16. Tiling becomes a factor at step 17 when the physical design is linked to the amended files (with back annotation as discussed in Section 5) and test points. The affected tiles are cleared for re-place-and-route and the introduction of the test logic. The test logic is introduced in steps 18 and 19 and is placed-and-routed with the tile's functional logic in step 20. The tiles affected by debugging changes are also re-placed-and-routed at this time. Tiles unaffected by a debugging change or a test point remain untouched. If newly introduced logic requires more area than the amount of unused resources in the affected tile, neighboring tiles are included in the re-place-and-route, as their unused resources may be used for logic introduction. After, the tile interfaces are re-locked. The process repeats until no design errors are found during emulation. Depending on the changes made and test logic introduced, tiling boundaries can be kept the same or reestablished for each debugging iteration.

3.2 Tiling

Tiling is achieved through physical design constraints imposed on the place-and-route tool. Tiling boundaries are transparent in the layout, as they are simply conceptual boundaries of constraints. The default is that all resources are locked, and when a specific tile must be changed, the resources within that tile are unlocked. Therefore, the layout of that tile may be altered while the rest of the design remains unchanged and unaffected due to the locked interface between the tile and its surroundings. If one side of an interface is locked, the interface itself is locked.

Tile boundaries are determined by a number of factors. First, inter-tile interconnect is minimized. Locked interfaces are a hindrance to circuit performance and place-and-route flexibility. Therefore, interfaces must be as simple as possible.

Tiling should also be performed to user specifications. Parameters include acceptable area overhead, acceptable performance degradation (not relevant for pure functional test), and the type of test logic and debugging changes to be introduced (i.e. size of control and observation logic for error detection and localization, large block changes or small functional and implementation alterations). These variables are not entirely independent (e.g. the need for large test logic requires a large area overhead), but tiling remains flexible to user specifications.

The type, size, and number of points of test logic likely to be inserted are also concerns for tiling. If large pieces of logic for controllability and observability must be inserted (e.g. a large counter), the amount of unused resources in each tile should be large. If the area overhead must be low, the tiles must be larger to accommodate the insertion of large blocs of logic. Similarly, if a large number of test points may be inserted into the design, area overhead must again be increased to accommodate the dispersion of a large amount of logic. However, if a tile cannot support the introduction of a large amount of logic, neighboring tiles may also be used, and tile boundaries may be redrawn.

The types of debugging changes that may be made must also be taken into account during tiling. Large block changes are often

[2] Designs are at their highest level of complexity at the physical level, and the complexity will continue to increase with deeper sub-micron technologies [9].

made during the early stages of debugging, while small functional and implementation alterations are more likely to be performed during the final stages[3]. Large block changes may require the addition of considerable extra logic and/or extreme malleability, while small alterations add little, if any, logic and create fewer routing concerns. Larger tiles allow for greater malleability and a larger amount of unused resources per tile without added area overhead. Therefore, tile size can be altered based on the current debugging needs and the various expected design changes.[4]

During actual implementation, most users will likely desire a reasonable tradeoff among these features. Extremely small tiles require more area and timing overhead and have less malleability, but extremely large tiles reduce the granularity and therefore lengthen the amount of time spent in the back-end CAD tools. The average tile size will likely be between 20 and 50 Xilinx 4000 Configurable Logic Blocks [13], much smaller than the functional blocks created by CAD tool partitioning that are the minimum affected unit for existing emulation debugging techniques. [5]

4 Error Detection and Localization

Effective debugging begins with a formalized process for testing to detect, isolate, and identify errors. Software simulation has tools to perform such tasks, but emulation requires that they be performed in hardware. Therefore, control and observation logic must be introduced into the physical layout.

4.1 Controllability and Observability

For an error to be detected, it must actually occur. Therefore, the design must be put in a state at which the error arises. This can be done by operating the circuit normally until the proper state arises and the error occurs, but this may take a long time, if it occurs at all. Therefore, control logic is introduced into the circuit to induce certain states artificially. That is, the logic inputs specific state to suspected design error areas to the run exhaustive tests that are necessary for maximum design confidence.

A design error must also be detected once it occurs. Observation can be performed manually with a user reading the outputs of suspected design areas waiting for an error to occur. However, logic may be inserted which automatically detects an error upon its occurrence. This observation logic is designed to raise a flag once an erroneous output state is achieved. Therefore, the control logic creates artificially and potentially problematic input states, and the observation logic detects any error that results. If the observation raises a flag, the error can be localized and diagnosed based on the control inputs.

4.2 Logic Introduction

The combination of controllability and observability can detect and localize errors, but the necessary test logic must be embedded in the design. Tiling ensures that the smallest possible amount of the design is re-placed-routed to incorporate the test logic, but enough resources in the layout must be free to accommodate the test logic, the size of which may vary depending on the control and observation that are being inserted. If the affected tile does not have enough free resources, neighboring tiles can also be labeled "affected" and may contribute their unused resources. (Section 6.1 examines the number of tiles that are affected for different sizes of introduced test logic given a certain area overhead and tile size.) Once the affected tiles are identified, they are cleared (i.e. all of the logic and routing must be removed), and the logic is re-placed-and-routed along with the test logic. Error detection and localization may then be performed as the control and observation logic test for errors.

5 Error Correction

Upon error detection and localization, debugging changes can be made to the design at some level in the design hierarchy for the next emulation iteration. Assuming that most changes will be made in the HDL and the RTL level, synthesis must be performed. Modern techniques noted in Section 2 can be used to minimize the impact at the physical level. Those changes must then be linked to the physical level and specific tiles before they are updated in the layout.

5.1 Linking Debugging Changes

One of the most important areas of future development involves allowing synthesis and front-end tools to account for physical resource allocation in a cooperative manner with back-end tools. This move toward rationalizing multiple design hierarchies throughout the design process has its origin in existing approaches for back annotation, but it involves the maintenance of significantly more consistency information. The goal is to improve the synthesis results for a given amount of optimization time by capitalizing on designer knowledge regarding the physical device and overall design structure. Preliminary tools moving in this direction have been announced and released by a number of CAD companies (e.g. Synplicity).

Currently, we manually exploit the one-to-one linkage between the high-level and the low-level using back annotation to localize the change requirements to a set of tiles. Partitioning done throughout the design process creates a tree structure with children being dependent on their parents. Therefore, using back annotation, we trace the debugging changes made at any level in the design process through the sub-trees of all the altered nodes down to the affected tiles. Current forms of this type of tracing stop (e.g. Quick_ECO [2]) before they reach the physical level and are, therefore, not as fine-grained as made possible by tiling. This minimal physical design perturbation is necessary for full exploitation of the fine granularity that tiling provides.

5.2 Updating Debugging Changes

With the links from the altered files to the physical level made, it becomes possible to update the changes so the revised design bitstream can be created. Any tile that contains a design portion affected by the debugging change must be cleared, while still maintaining the locked interface to its surrounding tiles. If an interface is affected, the tiles comprising the interface must be cleared. Similarly, if two adjacent tiles are both affected, the fixed interface between the tiles can be removed. Once all of the

[3] Using the synthesis techniques discussed in Section 2, changes made at a higher level in the design hierarchy perturb the physical design as little as possible.

[4] Tile sizes need not be uniform across a design. Large tiles can be used in areas of expected large block changes, and smaller tiles can be used for small alteration areas.

[5] Tiling algorithmic details can be found in [5][6].

affected tiles are cleared, the remainder of the design is locked to its location. The affected portions are then re-placed-and-routed in the cleared tiles, any removed interfaces are re-locked, and the links from the higher levels to the physical level are reestablished via back annotation and the tree structures discussed in Section 5.1. Not only is the time spent in the back-end CAD tool shortened, but no errors can be introduced in the locked portions of the design as may occur when everything is re-placed-and-routed. This allows for more efficient error localization.

This approach is very different from incremental place-and-route, which re-places-and-routes a much greater portion of the design. Tiling imposes many more constraints on the back-end CAD tools that disallow changes that an incremental tool would make, and incremental tools do not account for logic to be added into the design. Ultimately, incremental place-and-route achieves better design performance but with a large added design effort as reflected in Section 6.1.

6 Experimental Results

The proposed approach was executed on nine designs, including seven MCNC designs (three combinatorial and four sequential) of various size and two larger, real world designs. The two large designs were a MIPS R2000 processor core designed for FPGAs developed at Brigham Young University and a digital encryption standard (DES) design [8]. All experiments were performed on the Xilinx XC4000 family with configurable logic blocks (CLBs) each containing two 16-bit lookup tables (LUTs) [13].[6]

Although the designs in question are small enough to be implemented on a single FPGA, the results can be accurately applied to larger designs that span multiple FPGA devices, as the UltraSPARC-I emulation required. Tiling breaks such designs into smaller portions, just as the non-tiled approaches do, though at the physical level for more fine-grained results. Therefore, experiments compare tiling results to approaches looking only at blocks as small as CAD defined functional blocks and incremental place-and-route tools. It is then irrelevant that a larger design may contain multiple functional blocks across multiple FPGA devices. For experimental purposes, each design will be considered the size of one functional block. The designs chosen for experimentation are a wide range of sizes and fit into the size variation of functional blocks, thus allowing the accurate analogy.

6.1 Results

The overhead of the proposed approach comes in the form of area (physical resources) and timing. However, as discussed in Section 3.2, this overhead is variable depending on user specifications. Area overhead can be as little as 10% (less would not allow enough room for additional logic or malleability for easy placement and routing) or as large as the FPGA allows. Timing overhead depends on user specifications and the design changes made. Small design changes may have no, or even a positive, impact on timing, while large changes that negatively affect or create a new critical path may severely affect timing. Whatever the timing degradation becomes during debugging, the entire design may be re-placed-and-routed for better performance if timing critical tests must be executed during emulation.

[6] Xilinx claims that XC4000 CLBs can roughly implement an average of 20 logic gates [13].

Table 1 shows the physical layout statistics for the designs after they were tiled. An approximately 20% area overhead was introduced in each case for future logic introduction and flexibility. As the table shows, tiling actually increases design performance in some situations. This is due to the dramatically different placement and corresponding timing that often result from relatively small design changes. The timing impact of tiling appears to be below the characteristic variance associated with such small changes.

design	# CLBs	area overhead	timing overhead
9sym	56	0.217	-0.045
styr	98	0.210	0.074
sand	100	0.220	0.129
c499	115	0.223	0.000
planet1	115	0.211	0.137
c880	135	0.227	-0.055
s9234	235	0.205	-0.014
MIPS R2000	900	0.190	0.047
DES	1050	0.200	0.036

Table 1. Tiled Physical Layout Statistics

Given the 20% area overhead for each design, tile size affects the number of tiles that will be affected when control and observation logic is introduced. For example, if s9234 were partitioned into ten tiles that average 23.5 CLBs, each tile would have approximately 4.7 CLBs to implement test logic. If this is sufficient, only one tile is affected. However, if large blocks are introduced, neighboring tiles must be re-placed-and-routed to provide the necessary resources. This technique ensures fine-granularity for small changes but remains functional when large changes must be made, all while maintaining low area overhead. Figure 3 shows the percentage of tiles that would be affected for various sizes of test logic introduction. The same results hold if the logic introduced is a debugging change rather than logic for error detection and localization.

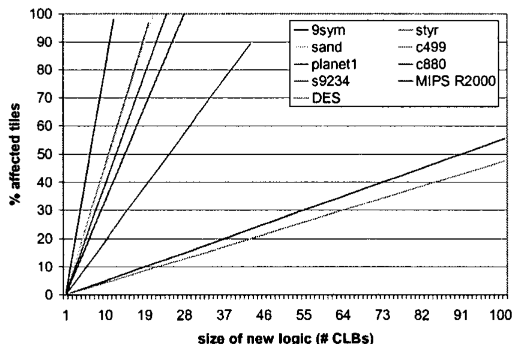

Figure 3. Number of Tiles Affected by Logic Introduction

Similarly, the number of points of controllability and observability impacts the number of tiles affected. Figure 4 shows the maximum size of the test logic for a variable number of test points for the same design and overhead assumptions as were used for Figure 3. The percentage of tiles affected by a variable number of test points depends on the distribution of the points. If the test points are clustered, then the number of test points may be multiplied by the size of the test logic for Figure 3 to be used for the percentage of affected tiles. If the many test points are randomly distributed, then all of the tiles may be affected, regardless of the size of the test logic.

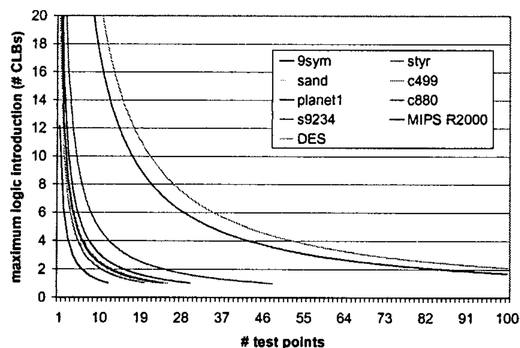

Figure 4. Maximum Test Logic Size

The most relevant result is the amount of time that is saved between emulation and debugging iterations due to the finer granularity tiling (as opposed to function block partitioning and incremental place-and-route) provides. Since granularity (i.e. tile size) is variable, the amount of time saved varies. Also, the introduction of large pieces of test logic and some significant design changes may affect a large number of tiles, thereby increasing the granularity to a non-tiled level. However, the resulting CAD tool effort will never exceed that required by a non-tiled approach.

Figure 5 shows the place-and-route speedup provided by tiling with different tile sizes (in percentage of total design) compared to the incremental and Quick_ECO techniques. These results assume that only one tile was affected by each physical design change, but the effort can be scaled as if the affected tiles equal one larger tile. For example, a change affecting three adjacent tiles that are 5% of the total design effectively affects one tile that is 15% of the design.

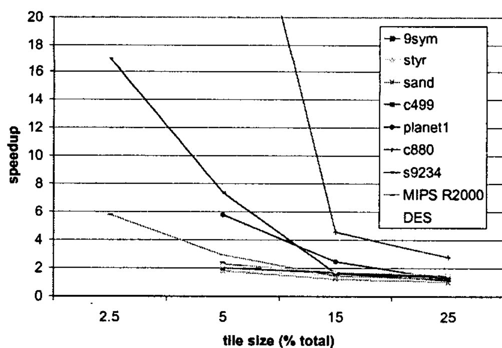

Figure 5. Place-and-Route Speedup

The three largest designs (DES, MIPS R2000, and s9234) can be partitioned into tiles 2.5% the size of the total design. In those cases, speedup was 2.8, 5.6, and 17.0 respectively. As tile size grows to 5% and 15% of the total design, the average (median) speedup reduces to 7.6 (2.6) and 2.1 (1.7). Finally, as the tile size reaches 1/4 of the total design size (effectively eliminating the purpose of tiling), the average (median) speedup falls to 1.5 (1.3).

7 Conclusion

Design debugging is growing increasingly laborious due to the increase in design and simulation complexity. Therefore, emulation for functional verification has become a more widespread technique, enabling the execution of more complex simulations. However, design complexity has been a barrier to efficient emulation as a large design space must be re-placed-and-routed for each debugging iteration.

Tiling uses partitioning to introduce simplicity at the physical level. Fine-grained, independent partitioned blocks enable the localization of physical design changes due to test logic introduction or debugging changes. Therefore, the back-end CAD tool must re-place-and-route only those blocks affected by the alterations, resulting in a shorter time between debugging iterations and, therefore, time-to-tapeout. Experiments show that re-place-and-route CAD tool effort can be reduced significantly for the finest-grained physical design partitioning.

Acknowledgements

This work was supported by the Air Force Research Laboratory of the United States of America, under contract F30602-96-C-0350 and subcontract QS5200 from Sanders, a Lockheed Martin company.

References

[1] Brand, D. et al., "Incremental synthesis," *International Conference on Computer-Aided Design*, 1994, 14-18.

[2] Fang, W.-J., A. Wu, and T.-Y. Yen, "A real-time RTL engineering-change method supporting on-line debugging for logic-emulation applications," *Design Automation Conference*, 1997, 101-106.

[3] Gateley, J. et al., "UltraSPARC-I emulation," *Design Automation Conference*, 1995, 13-18.

[4] Kirovski, D. and M. Potkonjak, "Engineering change: methodology and applications to behavioral and system synthesis," *Design Automation Conference*, 1999.

[5] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Low overhead fault-tolerant FPGA systems," *IEEE Transactions on VLSI*, vol. 6, no.2, June 1998, 212-221.

[6] Lach, J., W. H. Mangione-Smith, and M. Potkonjak, "Fingerprinting digital circuits on programmable hardware," *International Workshop on Information Hiding*, 1998, 16-31.

[7] Lakshminarayana, G. et al., "Power management in high-level synthesis," *IEEE Transactions on Very Large Scale Integration*, vol.7, no.1, March 1999, 7-15.

[8] Leonard, J. and W.H. Mangione-Smith, "A case study of partially evaluated hardware circuits: key-specific DES," *Field Programmable Logic*, 1997, 151-160.

[9] Mitsuhashi, T. et al., "Physical design CAD in deep sub-micron era," *European Design Automation Conference*, 1996, 350-355.

[10] Pan, K.-R.R. and M. Pedram, "FPGA synthesis for minimum area, delay and power," *European Design and Test Conference*, 1996, 603.

[11] Schaumont, P. et al., "Synthesis of multi-rate and variable rate circuits for high speed telecommunications applications," *European Design and Test Conference*, 1997, 542-546.

[12] Swamy, G. et al., "Minimal logic re-synthesis for engineering change," *International Symposium on Circuits and Systems*, 1997, 1596-1599.

[13] Xilinx, The Programmable Logic Data Book, San Jose, CA, 1996.