

OPEN ARCHITECTURE HIERARCHICAL PLACEMENT FOR FPGA DATAPATH DESIGNS

Dong Kwan Kim, Cameron Patterson, and Peter Athanas
*Bradley Department of Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061 USA
{ikek70, cdp, athanas}@vt.edu*

ABSTRACT

The study of circuit placement in VLSI physical design has been conducted for several decades. As circuit complexity increases, it is non-trivial to place all cells of the circuit within a reasonable time. Many researchers have presented new placement algorithms and tools to address placement issues such as minimization of wire length, routability, and run-time. Our approach is different in that it focuses not on a specific placement algorithm, but on open architecture that provides a general placement API permitting FPGA users to create their own placement algorithms. The placement API is based on the hierarchical grouping of cells and provides common classes and methods for placement. By placing structured datapath circuits in a hierarchical manner, run-time can be significantly decreased while improving placement quality. The circuit's hierarchical netlist is physically realized as nested bounding boxes, and hierarchical placement, so called H-placement, is recursively applied to the bounding boxes at each level in a top-down manner, with backtracking used to align ports. We describe design and implementation issues, and the use of H-placement in run-time reconfigurable applications. This research is part of the JHDLBits project combining JBits and JHDL.

KEYWORDS

Hierarchical Placement, JHDLBits, JHDL, and JBits

1. INTRODUCTION

A typical VLSI circuit design cycle consists of specification, architectural design, logic design, circuit design, physical design, and test/fabrication. The physical design consists of partitioning, placement, and routing. Placement is a key step since a poor placement consumes larger area and results in performance degradation [1]. Our emphasis is on FPGA circuit placement, which is the process of arranging the circuit components on an FPGA surface [2]. The input to the placement stage is a cell library and netlist, and the output is a set of locations for all of the cells on the chip. As circuits include up to several million gates, the placement phase consumes a significant amount of time. Placement algorithms should consider both optimization and run-time, and are typically divided into two major classes: constructive placement and iterative improvement. In constructive placement, a method is used to create a placement from scratch, while iterative improvement algorithms start with an initial placement and repeatedly modify it to minimize a cost function [3]. Min-cut partitioning and numerical optimization are examples of constructive placement, and simulated annealing and force-directed are iterative placement examples.

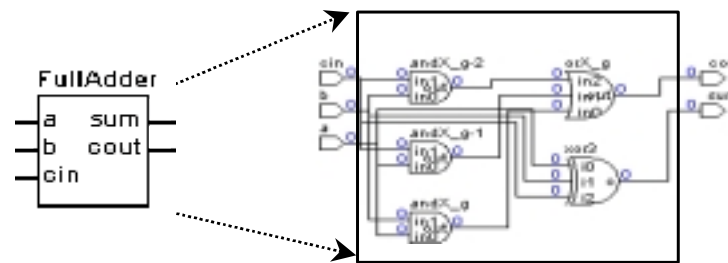
A variety of FPGA placement algorithms have been studied. Sankar and Rose [4] present an ultra-fast FPGA placement technique that combines multiple-level, bottom-up clustering and hierarchical simulated annealing. It can generate a placement for a 100,000-gate circuit in 10 seconds on a 300 MHz UltraSPARC workstation that is only 33% worse than a high-quality placement that takes 524 seconds using a pure simulated annealing implementation. In addition, it can provide an accurate estimate of the wire length achievable with good quality placement. Marquardt et al. [5] introduce a new simulated annealing-based timing-driven placement algorithm for FPGAs that is based on VPR's algorithm [6]. They show a method of determining source-sink connection delays during placement and introduce a new cost function that accurately balances wire-usage and circuit speed in any ratio. The method also combines connection-based

and path-based timing-analysis to obtain an algorithm that has the low time-complexity of connection-based timing-driven placement, while obtaining the quality of path-based timing-driven placement. Singh and Brown [7] present an algorithm to update the placement of logic elements when given an incremental netlist change. Their basic idea is to incrementally place logic elements created by layout-driven circuit restructuring techniques. The incremental placement engine assumes that the restructuring algorithms provide a list of new logic elements along with preferred locations for each of these new elements. The cost function considers cluster legality, timing, and wire length. Each cluster is penalized if it contains an architectural violation, and the timing cost is used to ensure that critical logic elements are not moved into locations that would drastically increase the critical path delay. Wire length estimation checks that a circuit is easily routable after the logic element moves. According to [8], existing placement algorithms are not stable, and their effectiveness varies depending on the characteristics of the benchmarks. New hybrid techniques may be needed to make future placement engines more scalable and stable. In summary, circuit placement still has significant room for improvement.

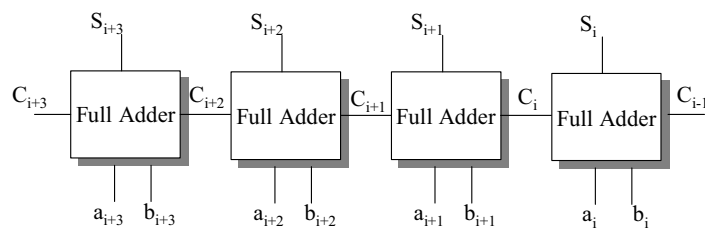
There is a trade-off between placement quality and speed, and our approach focuses on fast placement as would be required in a run-time reconfiguration environment. The H-placement described here is suited to hierarchical datapath circuits that exhibit a high degree of regularity and reuse to perform the same operation on multiple bits. Figure 1 gives an example of the hierarchy and regularity present in a simple datapath. While most placement algorithms use a flat netlist, we consider a circuit as a hierarchical structure of blocks to be placed in a top-down manner. In Figure 1 (a), a full adder is treated as single block even if it consists of several gates. Each operation corresponds to a dedicated functional block, such as an adder, register, buffer, multiplexer, or multiplier [9]. A full adder is used four times in Figure 1 (b). The H-placement algorithm will exploit this reuse by performing the internal placement of the full adder only once.

Our work is a part of the JHDLBits project that provides an integrated development environment for a digital circuit designer [10]. A JHDLBits user can place and route their circuit design entirely within the JHDLBits development environment and without using Xilinx's conventional MAP, PAR and Bitgen flow. Furthermore, the open architecture of JHDLBits allows any part of the implementation flow to be modified.

This paper is organized as follows: Section 2 provides background information on JHDL [11], JBits [12], and JHDLBits. Section 3 describes our H-placement concepts, flow, and support for run-time reconfiguration. In Section 4, we present our conclusions and future work.



(a) Hierarchy of a full adder



(b) Regularity

Figure 1. Hierarchy and regularity.

2. BACKGROUND

2.1 JHDL AND JBits

JHDL is a Java-based [13] hardware design language for FPGAs developed at Brigham Young University. Circuits are described by structurally instantiating lower-level building blocks. In JHDL, two basic Java classes form the basis for all circuits: `Logic` and `Wire`. Designers create a new logic cell in their design by extending `Logic`, defining a cell interface, and then defining the architecture of the cell. A JHDL design is a standard Java source code file – no extensions to the language are required. After compilation by a standard Java compiler, a class file is generated that can then be executed. A key feature of JHDL is its ability to operate in either simulation or hardware mode. When in simulation mode, the values of all elements in the circuit data structure are computed by the JHDL simulator [10, 14].

JBits is a collection of Java classes with an API that provides access to every configurable resource in a Xilinx FPGA. The JBits3 SDK is the latest release from Xilinx with support for the Virtex-II FPGA family. Device resources may be programmed and probed individually at run-time with the JBits API, even with the FPGA active in a working system. The interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams read back from actual hardware. Although JBits3 is a complete API for examining and modifying device configuration, it does not include a device simulator or router as in the previous JBits2.8 release [10, 15].

2.2 JHDLBits

JHDLBits is an open-source endeavor striving to merge the salient features of two prominent FPGA research environments: JHDL and JBits. There exist philosophical differences between the JHDL design flow and the JBits design flow. JHDL, while being highly dynamic, has been primarily designed to be a front-end to the traditional mainstream FPGA design flow (Xilinx is emphasized here, where the Xilinx Alliance or Foundation tools would ultimately be responsible for bitstream generation). JBits, on the other hand, has traditionally focused on rapid bitstream generation and bitstream modification, independent of the mainstream flow. Users can take advantage of the run-time and partial reconfiguration features of JBits without having to work entirely at the bitstream level. Furthermore, full control of placement and routing is still possible. The JHDLBits project consists of a collection of tightly integrated components that together provide an end-to-end pathway for creating, manipulating, and debugging FPGA bitstreams. More importantly, because most of the components of this project are open-source, researchers investigating architecture-specific placers/planners/routers/compilers have the advantage of replacing the stock JHDLBits components at will [10].

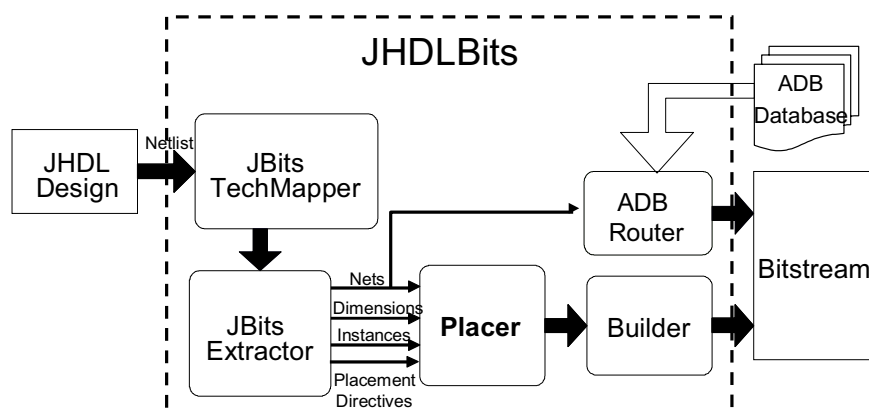


Figure 2. JHDLBits design flow.

Figure 2 shows a circuit design flow in JHDLBits. The main components of JHDLBits are the `JBitsTechMapper`, `JBitsExtractor`, `Placer`, `Builder`, and `ADB Router`. The `Placer` module in Figure 2 lies in between `JBitsExtractor` and `Builder` in JHDLBits. The `JBitsExtractor` module extracts information about nets, dimensions, instances, and placement directives and then passes to the `Placer` module. The information is used to place the primitive instances. The `Placer` provides a physical location to each primitive instance and then passes placed instances to the `Builder` module.

The default placer provided in JHDLBits sacrifices some fidelity in the quality of placement in order to emphasize speed. Generally speaking, placement is the process for determining the location for primitive cells on the FPGA device, and its goal is to achieve routability, higher speed and/or lower power. The placement problem in JHDLBits focuses on hierarchical information in the given circuit. In JHDL, the designer can express a circuit structurally without any concern for cell placement, yet provisions exist to annotate a design to help guide placement. When a logic circuit is designed, placement directives for some cells may be provided while other cells are left undetermined. The JHDLBits project is specifically designed to allow researchers and FPGA developers to modify, enhance, or totally replace the default placer present within the tool suite.

3. HIERARCHICAL PLACEMENT (H-PLACEMENT)

3.1 Approach Overview

H-placement converts a datapath's hierarchical netlist into a placed set of nested bounding boxes for the modules in the netlist. Each level of a hierarchical netlist consists of modules and/or primitive cells. A bounding box contains other bounding boxes or primitive cells and is placed, moved or swapped by the placer. Figure 3 shows the relationship between a hierarchical netlist and bounding boxes. Figure 3 (a) illustrates a hierarchy of three sub-netlists: `root`, `A`, `B`, and `C`. H-placement is a framework for circuit placement and can use a variety of placement algorithms such as simulated annealing, analytical, constructive, guided or user-defined placement algorithms. In Figure 3 (a), submodules at each level can use a different placement algorithm that depends on the module's internal structure. While a simulated annealing algorithm may suit control logic, a constructive algorithm may be preferred for a datapath circuit. H-placement determines which placement algorithm is appropriate for a submodule by analyzing its circuit structure. Assume that submodules `A` and `B` in Figure 3 (a) have a hierarchical and flat structure respectively. The H-placement may choose the constructive placement for submodule `A` and simulated annealing for submodule `B`.

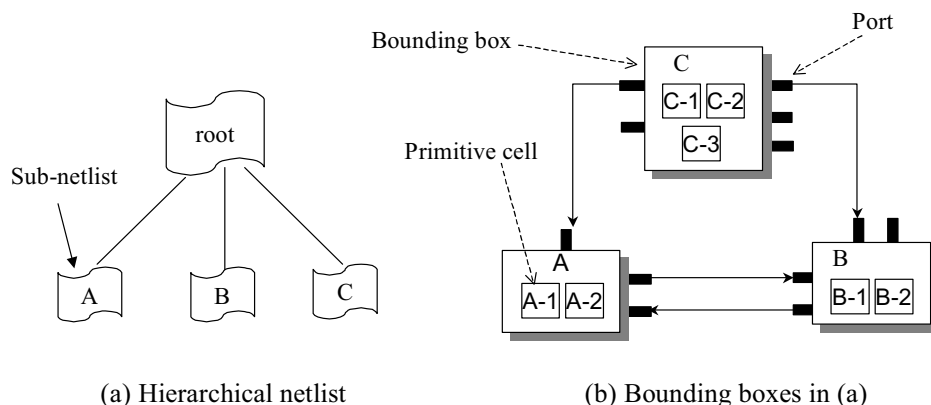


Figure 3. Hierarchical netlist and bounding box.

The bounding boxes as shown in Figure 3 (b) are identified from the hierarchical netlist; bounding boxes A, B, and C are created from sub-netlists A, B, and C respectively. The bounding box can have one or more ports around its periphery, where a port is a connection to other modules or primitive cells. In Figure 3 (b), the black rectangles indicate the ports. The number of ports varies according to the module and the port positions can change during placement. Modules or cells within each bounding box have positions relative to the encompassing bounding box. The bottom-left corner of the bounding box is a handle and internal coordinates for interior cells are based on the coordinate of the handle. For example, primitive cells A-1 and A-2 in Figure 3 (b) have positions relative to the bounding box A.

3.2 The H-placement Procedure

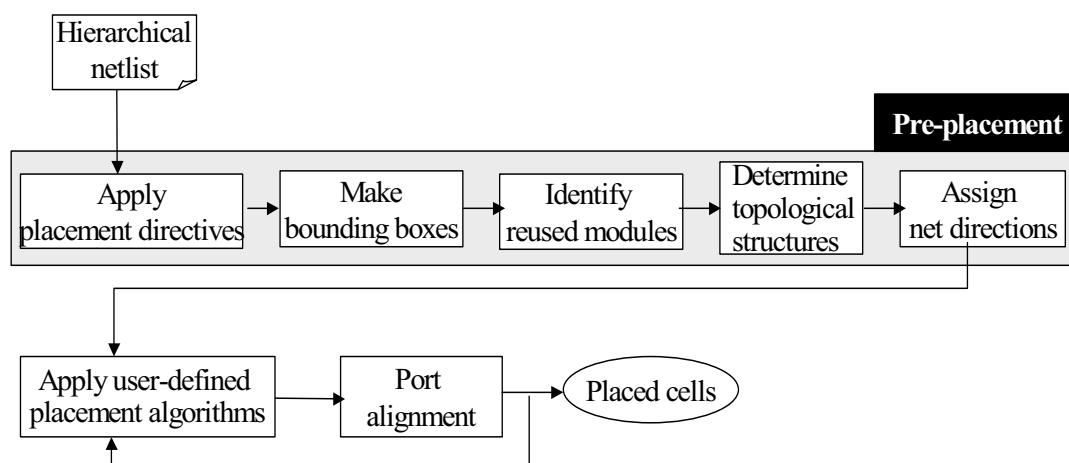


Figure 4. H-placement flow.

Figure 4 describes the main steps involved in H-placement. Pre-placement consists of five steps: applying placement directives, making bounding boxes, identifying reused modules, determining topological structures, and assigning directions to these topologies. The first step is to check whether the input netlist has user-defined placement directives specifying the juxtaposition of bounding boxes. Examples of placement directives are ABOVE, BELOW, LEFT, RIGHT, and ON. These do not specify the exact coordinates of a bounding box, but rather give topological constraints for module positions in each dimension. JBits2.8 [12] required that all modules have placement directives, but they are optional in JHDLBits.

Consider an n-bit counter example in order to illustrate the pre-placement procedure. Figure 5 shows a schematic diagram of an n-bit counter in JHDL. The n-bit counter consists of four full adders and each full adder is composed of three AND gates, one OR gate, and one XOR gate. We can make four bounding boxes (FullAdder, FullAdder-1, FullAdder-2, and FullAdder-3) with the hierarchical structure present in the netlist. These are four instances of the same module, and the same internal placement will be used for each instance. This reuse can drastically decrease placement execution time.

The next pre-placement step is to perform a topological sort on pertinent connections between modules in order to determine a bounding box order in each dimension. Topological sorts effectively create placement directives. Note that the topological sort indicates an ordering and not a direction. The netlist-inferred order in this example is FullAdder, FullAdder-1, FullAdder-2, and FullAdder-3. The direction may be northbound, southbound, eastbound, or westbound, and is chosen to minimize the length of nets between connected ports. At the top level, IOB positions also influence the direction. If an input and output pad are located leftmost and rightmost respectively, their positions would select the eastbound direction. Direction information may also be determined from the design's user constraint file (UCF), or FPGA architectural constraints such as carry chains.

A default or user-defined placement algorithm is now applied to all remaining components for which topologies could not be inferred. Clusters identified during the topological analysis are also placed at this point. The placement process can include an initial placement that usually uses a random or constructive

algorithm. During the initial placement bounding boxes are located onto FPGA physical elements, for example CLBs, slices, and LUTs. The bounding boxes may change their positions by optionally applying an iterative improvement algorithm.

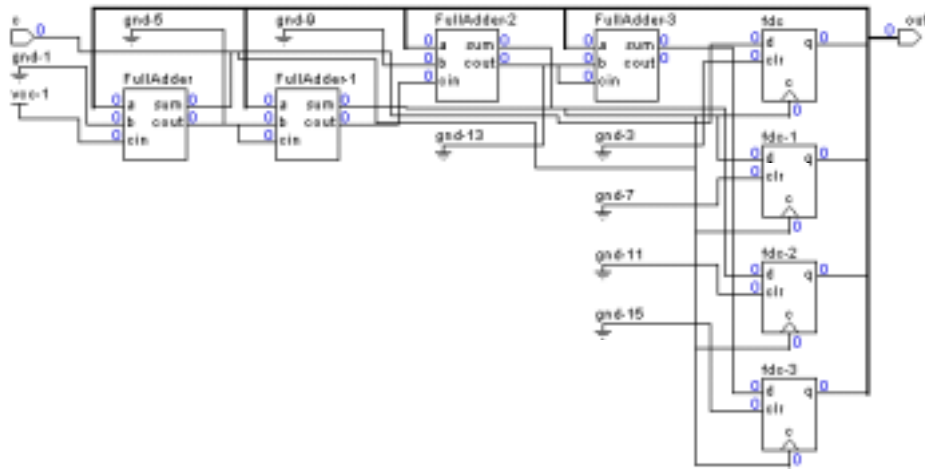
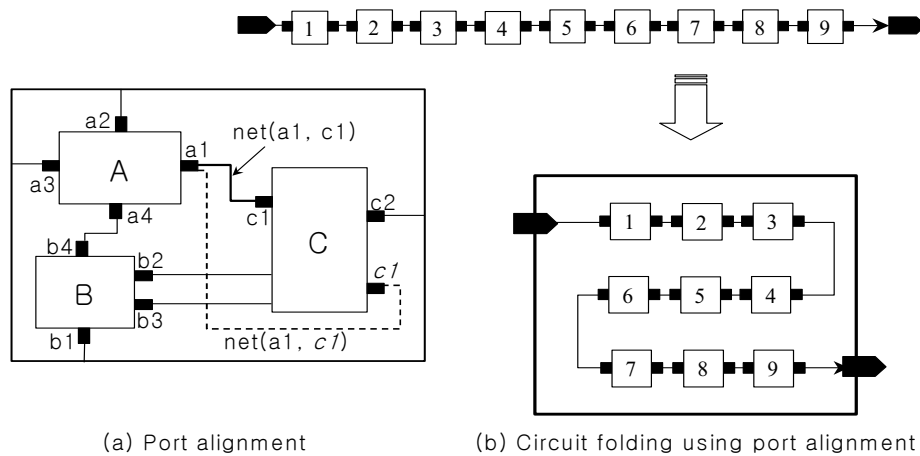


Figure 5. A schematic view of an n-bit counter.

Port alignment is the last step in H-placement and its objective is to optimize connections between placed cells. The position assigned at the previous step may change at this step due to placement constraints. Port alignment may result in backtracking to the previous step of applying user-defined placement algorithms. Both steps are applied iteratively and recursively.

Figure 6 illustrates port alignment and a circuit-folding example. In Figure 6 (a), bounding box A has ports a1, a2, a3 and a4, and bounding box C has ports c1 and c2. The dashed line, net (a1, c1) assigned at the initial step indicates a connection between port a1 and port c1. After port alignment, the position of port c1 moves closer to port a1 in order to minimize the wire length between bounding boxes A and C. Port c1 and net (a1, c1) move to port c1 and net (a1, c1) respectively. At the top level, the port alignment considers only bounding boxes A, B, and C; it ignores inside connections of each bounding box. After that, the ports at the next level are aligned recursively. Figure 6 (b) gives another example of port alignment, in which a chain of elements are too long to be placed onto an FPGA in a single direction. Folding is required, as shown in the bottom of Figure 6 (b). This causes the input and output ports of bounding boxes 4, 5, and 6 to be switched.

Port alignment is also used to meet architectural constraints and to reduce empty space between bounding boxes. For example, carry logic components should be positioned northbound since the input port is on the bottom and the output port is on the top.



(a) Port alignment

(b) Circuit folding using port alignment

Figure 6. Port alignment.

3.3 Support for Run-time Reconfiguration

Run-time reconfiguration (RTR) allows portions of the hardware to be modified in response to the demands placed on the system while it is running. The goal is to context-switch hardware in the same way that CPU's context-switch software [16]. In Figure 7 (a), modules A, A' and A'' implement different functions but have identical interfaces. These modules would be compiled as partial bitstreams that use the same physical connections to the rest of the system.

The relationship between hierarchical designs and bounding boxes has been previously described. H-placement can support run-time swappable modules by projecting the hierarchy in the time domain. In Figure 7 (a), modules A, A' and A'' will overlay each other at run-time, as will modules B and B'. Figure 7 (b) shows the footprint (i.e. bounding box dimensions and port positions) allocated for modules A, A' and A''. These modules have different internal structures and resource requirements: A has two submodules A-1 and A-2, and A' has four submodules, A'-1 through A'-4. For each set of run-time swappable modules, the H-placement procedure determines the largest module and places it first. Other modules in the set are constrained to use the same bounding box dimensions and port positions. The dashed rectangle in Figure 7(b) indicates for the overall bounding box assigned to modules A, A' and A''.

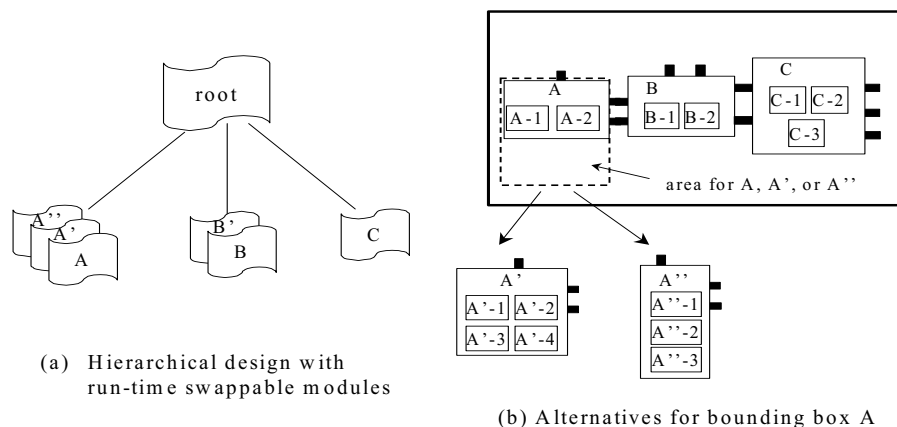


Figure 7. Creating compatible bounding boxes.

4. CONCLUSIONS AND STATUS

In this paper, we have presented a hierarchical placement algorithm for structured datapaths. The advantages of our H-placement approach are run-time performance and extensibility, and direct support for module swapping in an RTR context. Performance is achieved through a divide-and-conquer strategy and by reusing layouts for repeated modules that typically occur in structured datapaths. A reusable framework provides extensibility for placement algorithm developers. Researchers can implement their own placers without having to implement other parts of the implementation flow.

Implementation of the H-placement framework is in progress, and we expect to provide performance results in the near future. An effort level parameter will allow users to select the desired trade-off between CPU time and quality of results.

Currently, the APIs provided do not support timing-driven placement. Timing analysis and cost functions considering timing issues will be added. Sample placers built upon the API described in this paper are under development, and will serve to illustrate how JHDLBits users can develop their own placement

algorithms. In addition, the sample placers will be tested with datapath-oriented benchmark circuits to evaluate performance and quality.

ACKNOWLEDGEMENTS

This project is supported by a grant from Xilinx, Inc.; JBits (Xilinx and Virginia Tech) and JHDL (Brigham Young University) were both funded under the DARPA ACS program.

REFERENCES

- [1] Naveed Sherwani, 1995, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, Massachusetts, USA.
- [2] Sadiq M. Sait and Habib Youssef, 1995, *VLSI Physical Design Automation*, IEEE Press, New York, USA.
- [3] K. Shahookar and P. Mazumder, 1991, VLSI Cell Placement Techniques, *In ACM Computing Surveys*, Vol. 23, No. 2, pp 143-220.
- [4] Yaska Sankar and Jonathan Rose, 1999, Trading Quality for Compile Time : Ultra-Fast Placement for FPGAs, *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on FPGAs*, CA, USA, pp. 157-166.
- [5] Alexander Marquardt et al., 2000, Timing-Driven Placement for FPGAs, *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on FPGAs*, CA, USA, pp. 203-213.
- [6] V. Betz et al., 1999, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, USA.
- [7] Deshanand P.Singh and Stephen D. Brown, 2002, Incremental Placement for Layout-Driven Optimizations on FPGAs, *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, CA, USA, pp. 752-759.
- [8] Jason Cong et al., 2003, Optimality, Scalability and Stability Study of Partitioning and Placement Algorithms, *Proceeding of the 2003 International Symposium on Physical Design*, CA, USA, pp. 88-94.
- [9] Sabyasachi Das and Sunil Khatri, 2002, An Efficient and Regular Routing Methodology for Datapath Designs Using Net Regularity Extraction, *In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 21, No. 1. pp 93-101.
- [10] Alexandra Poetter et al., 2004, JHDLBits : The Merging of Two Worlds, *Proceedings of the 14th International Workshop on Field-Programmable Logic and Applications, FPL 2004*, Antwerp, Belgium.
- [11] The JHDL Home page, Brigham Young University, <http://www.jhdl.org/>
- [12] The JBits SDK, Xilinx Inc., <http://www.xilinx.com/products/jbits/>
- [13] Java website, <http://java.sun.com>.
- [14] Alexandra Poetter, 2004, *JHDLBits: An Open-Source Model for FPGA Design Automation*, Thesis, Virginia Tech, USA.
- [15] Jesse Hunter et al, 2004, VTSim: A Virtex-II Device Simulator, *Proceedings of the 2004 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, NV, USA.
- [16] Anup Kumar Raghavan and Peter Sutton, 2002, JPG-A Partial Bitstream Generation Tool to Support Partial Reconfiguration in Virtex FPGAs, *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, Florida, USA.