Layout Driven Data Communication Optimization for High Level Synthesis

Ryan Kastner[§], Wenrui Gong[§], Xin Hao[§], Forrest Brewer[§], Adam Kaplan[†], Philip Brisk[†] and Majid Sarrafzadeh[†]

[§]Department of Electrical & Computer Engineering
University of California, Santa Barbara
{kastner, gong, hao, forrest}@ece.ucsb.edu

[†]Computer Science Department
University of California, Los Angeles
{kaplan, philip, majid}@cs.ucla.edu

ABSTRACT

High level synthesis transformations play a major part in shaping the properties of the final circuit. However, most optimizations are performed without much knowledge of the final circuit layout. In this paper, we present a physically aware design flow for mapping high level application specifications to a synthesizable register transfer level hardware description. We study the problem of optimizing the data communication of the variables in the application specification. Our algorithm uses floorplan information that guides the optimization. We develop a simple, yet effective, incremental floorplanner to handle the perturbations caused by the data communication optimization. We show that the proposed techniques can reduce the wirelength of the final design, while maintaining a legal floorplan with the same area as the initial floorplan.

1. Introduction

Optimizations performed early in a design flow provide the greatest opportunity to optimize the final circuit; however, they have very little information about the remainder of the synthesis process. For instance, the physical layout of the circuit is determined in the final stages of synthesis (physical synthesis). Most often, high-level optimizations simply transform the specification to reduce the computing complexity of the application. If we could somehow take into account physical information during these optimizations, we would have the ability to greatly affect the layout of the circuit.

Previously, we showed that static single assignment (SSA) can reduce redundant data communication between modules in the hardware description, leading to a smaller hardware design via interconnect reduction [1]. We also demonstrated that data communication could be further reduced via intelligent placement of the SSA-generated Φ-functions. However, that Φ-function placement heuristic was limited by lack of knowledge about the communication cost between blocks. In other words, without knowing where blocks will be placed relative to each other in the final design, it is hard to model the cost of moving data between these blocks.

This work improves upon our previous work by augmenting the communication cost model with knowledge about placement information. We present a complete design flow encompassing behavioral and logic synthesis, and floorplanning. Our design flow contains a feedback loop from the floorplanner back to the system compiler. This helps the hardware compiler determine the actual physical cost of communicating data. We use this information to enhance the previously proposed Φ-function placement heuristic, and explore more of the design space of the system. We demonstrate that small changes during the high-level redesign of a circuit can drastically affect the subsequent physical re-layout, invalidating the original physical information upon which the optimization is based. We show that a physically aware synthesis approach based on incremental floorplanning can reduce the wirelength associated with data communication while maintaining the same floorplan area.

The remainder of the paper is organized as follows: Section 2 present our design flow from application specification to synthesizable register transfer level hardware description. Section 3 motivates the importance of the data communication problem and presents a layout driven solution to this problem. Section 4 shows experimental results. We conclude in Section 5.

2. Design Flow

The framework that we are investigating accepts an application specification and generates a synthesizable register transfer level (RTL) hardware description. The application specification undergoes syntactic and semantic analysis and is transformed into a control data flow graph using static single assignment (SSA CDFG). We provide techniques to translate the SSA CDFG form into an RTL hardware description. Finally, the RTL description is synthesized into a physical realization of the application, e.g. as a bitstream to program reconfigurable logic or a mask to fabricate an application specific integrated circuit.

2.1 Hardware Synthesis

The input to our tool is an *application specification*. Our project focuses on the use of C as a design language. We start by using the SUIF front end to perform syntactic/semantic analysis. Then, we use the Machine SUIF backend to generate a control data flow graph using static single assignment (SSA CDFG) form. *Static single assignment (SSA)* [2] transforms a CDFG such that each variable is defined in exactly one static code location. SSA is a safe transformation for hardware design because lone side effects of the transformation (Φ-functions) are easily implemented in hardware as multiplexers. Although SSA was originally intended to enable software-directed optimizations, it has been used in several projects where the final output is a hardware description language [3-5].

The next step in our framework requires that we transform the SSA CDFG into an RTL hardware description. The architecture body of a basic block entity is described using a behavioral representation. Each basic block entity is then synthesized to yield a structural representation of the basic block. This allows resource sharing within a basic block.

3-9810801-0-6/DATE06 © 2006 EDAA

After every basic block entity is synthesized, we generate a CFG entity. Each of the basic block entities is a block in the CFG entity. Then, we determine the control and data communications between the basic blocks. Distributed controllers within each basic block entity use handshaking to transfer control from one basic block to another. The data communication is determined using SSA. Section 3 describes exactly how this is done and presents a layout driven algorithm that optimizes the SSA form by moving and replicating Φ-functions to minimize the wirelength.

The entire design is finally realized as a two level hierarchical structural representation. We feed the design to a high level synthesis engine to get to a logic level structural representation. Then, we can hand off the design to any physical design tool to convert the RTL code into physical realization of the application.

2.2 Layout Driven High-level Transformations

High level transformations play a large role in determining the final properties of the application mapping. Unfortunately, most transformations are performed without much knowledge of the final circuit layout. However, it is possible to make an initial decision, continue on with synthesis, glean information from the final design, and then reevaluate the initial transformation. The reanalysis will have actual cost characteristics of the physical hardware layout, allowing a more informed decision making process. The drawback of this iterative approach is the large amount of time that is required to perform physical synthesis of the application. However, we can determine an approximate physical layout through the use of a floorplan, which is extremely fast when compared to full physical synthesis.

There are a number of previous works that consider floorplanning when solving various high level synthesis problems. Prabhakaran and Banerjee [6] developed a simulated annealing algorithm for simultaneous scheduling, binding and floorplanning. Fang and Wong [7] also use simulated annealing to solve the problem of functional unit binding while considering floorplan information. Dougherty and Thomas [8] use placement information while performing scheduling, allocation and binding. Fasolt [9] optimizes bus topology while considering layout information. Tarafdar and Leeser [10] incorporate floorplanning information with their synthesis flow.

We use a design feedback loop in our design flow. The feedback loop works as follows: First, we perform original optimization (without considering layout information) and generate an initial floorplan. Based on the layout obtained from the floorplan, we perform physically aware optimizations. The optimizations often alter the area and dimensions of the floorplan modules. This creates two potential problems. First and foremost, the floorplan may no longer be legal due to overlapping modules. Secondly, the floorplan may no longer be optimal with respect to the set of modules given. Therefore, the initial floorplan must either be legalized or re-optimized. In either case, a new floorplan is necessary. The initial floorplan can be obtained using any general floorplanner; however, the subsequent floorplans should use an incremental floorplanner. Our layout driven optimization step assumes that the initial floorplan is given, and optimizes data communication with respect to the locations of the modules given in that floorplan.

Consider the data communication problem (described in more detail in Section 3). The movement of Φ-functions corresponds to changing the physical location of the multiplexers, which in turn modifies the dimensions of the modules. Therefore, the initial floorplan becomes either illegal and/or sub-optimal. If a general floorplanner is used to generate the new legalized floorplan, the placement of modules within the new floorplan may be radically different from the initial floorplan. The data communication may no longer be optimized with respect to the current floorplan. If data communication is re-optimized, then another floorplan is required, and so on. From a conceptual standpoint, this feedback loop is neither likely to converge rapidly nor is it likely to produce ideal results. Our initial experiments attempted to perform layout driven data communication without an incremental floorplanner, i.e. we performed a full re-floorplanning. These results were mostly negative (see Section 4 for more details); we obtained worse wirelength due to the mismatch between the initial and subsequent floorplans.

On the other hand, if incremental floorplanning is used, then the newly generated floorplan will not deviate significantly from the initial floorplan. The new floorplan may be sub-optimal; moreover data communication may not be minimized for this floorplan. Nevertheless, the incremental floorplan largely mimics the initial floorplan, therefore the data assumed in the initial optimization remains mostly valid. Conceptually, this feedback loop is more likely to converge quickly and produce a well-optimized design than performing a full re-floorplanning as described above.

2.3 Incremental Floorplanning

We developed a simple incremental slicing floorplanner. We have not performed a detailed comparison with other incremental floorplanning algorithms, nor do we claim that our incremental algorithm gives better results. Our framework could easily incorporate other incremental floorplanners, e.g. ones that consider non-slicing floorplans or ones that minimize other metrics such as wirelength, area, congestion, etc.

We use a binary slicing tree and use the following definitions. A *basic module* is a block (corresponding to a basic block entity from Section 2) that cannot be divided, i.e. the leaf nodes of slicing tree. A *super module* is a module that contains two or more basic modules. Every internal node of the slicing tree corresponds to a super module. The *area* of a module is the summation of areas of all the basic modules in the module. The *room* of a module is the total space taken by the module, which includes the area plus the white space. The area and room of a super module are the sum of area and room of two children in the slicing tree.

The input of our incremental floorplanner is the initial floorplan and a list of perturbations to the floorplan. The first step of our algorithm calculates the area and the room

for each module in the modified floorplan. If the area of one module is greater than its room, we mark it with "+", otherwise we mark it with "-". FIGURE 1 shows an example. "32/36-" means the area is 32, the room is 36, and it is marked as "-".

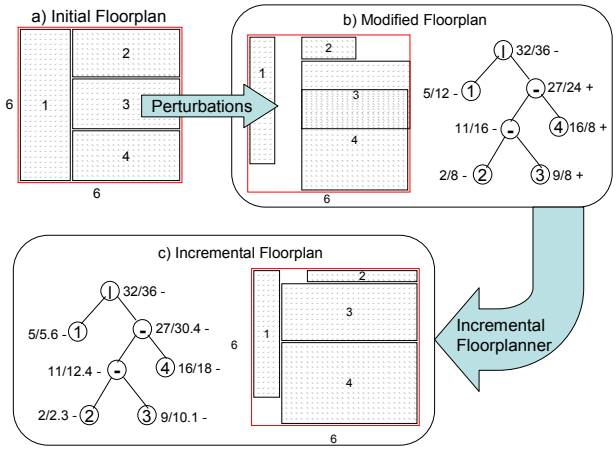

FIGURE 1 The initial floorplan is perturbed, which results in a modified floorplan. The incremental floorplanner generates a legal floorplan while attempting to maintain the structure of the initial floorplan.

The next step of the algorithm redistributes the area of the modules to account for the perturbations to the initial floorplan. The algorithm works in a top down fashion to redistribute the area. If one child of a module is marked "+", and the other is marked "-", we reallocate the room of the module by giving some of the room from the "-" module to the "+" module. The room is assigned in proportion to the areas of two sides. For example, in FIGURE 1 b), module 1 has an area of 5 and room of 12 while the super module corresponding to modules 2, 3 and 4 has area 27 but only has room of 24. The algorithm redistributes the room in the incremental floorplan (see FIGURE 1 c) by giving 5.6 units of area for module 1 and the remaining 30.4 for all the other modules (modules 2, 3 and 4). The algorithm continues to traverse the slicing tree and redistributes the room over all of the modules of the floorplan.

If the root super module is positive, i.e. has more area than room, then it is impossible to find a feasible solution. In this case we have to enlarge the room in order to create a feasible solution. We can guarantee all modules will become negative yielding a legal floorplan. In FIGURE 1 b), there are three positive modules in the slicing tree. After traversal, all of the modules become negative (FIGURE 1 c). That means each module can fit its room and we have a legal floorplan.

3. Data Communication

In order to determine the data exchange between basic blocks in a CDFG, we must establish the relationship between where data is generated and where data is used for calculation. We can determine this relationship through static single assignment. Static single assignment (SSA) renames variables with multiple definitions into distinct variables – one for each definition point. Thus, in SSA each variable name represents a value that is generated by exactly one definition. The variable name represents that value for the entire program.

In order to maintain proper program functionality, we must add Φ-functions into the CDFG. Φ-functions are needed when a particular use of a name is defined at multiple points. A Φ-function takes a set of possible names and outputs the correct one depending on the path of execution. A Φ-function can be viewed as an operation of the control node. Its selection of the proper name, which corresponds to the actual path taken through the program, can be implemented using a control-driven multiplexer. SSA is valuable as a hardware implementation model because it allows us to minimize the data communication by restricting data movement between basic blocks. As each variable in SSA corresponds to a specific value that is never reassigned, a variable is only shared between two basic blocks if one block produces a data value used in the second block.

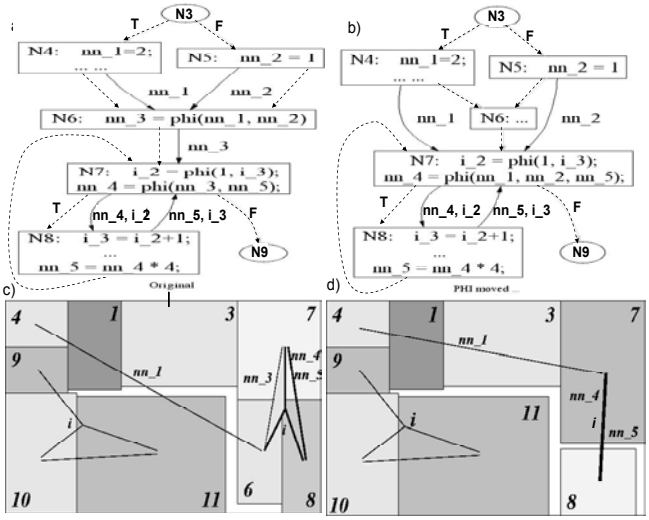

FIGURE 2 Synthesized FAST function. Parts a) and b) are a subset of basic blocks from the function's control flow graph along with the Φ-function related code. Part a) gives the initial locations of the variables and Φ-functions. Part b) shows the layout driven Φ-function locations. Part c) is the floorplan corresponding to the initial locations (part a). Part d) shows the incremental floorplan for the moved Φ-functions from Part b).

The standard placement of Φ-functions is at the earliest temporal join points of the set of basic blocks defining the values to be selected between. This location is known as the *iterated dominance frontier* (IDF). However, other legal placements of Φ-functions exist, and the selection of a placement can have a very large effect on the amount of data communication in the final hardware implementation.

We illustrate our point with FAST function from the MediaBench test suite [11]. FIGURE 2 exhibits SSA form with Φ-function placement at the IDF (FIGURE 2a). It also shows the corresponding initial floorplan (FIGURE 2c). The lines in the floorplan correspond to the data communication required by the three Φ-functions. The Φ-functions are placed in blocks 6 and 7. Of the three Φ-functions, only the

Φ-function corresponding to nn_3 (the one initially in basic block 6), has the opportunity to be moved. The other two Φ-functions only have one correct location, which is their current location at the IDF. By moving Φ-function nn_3 to block 7 (FIGURE 2b), we eliminate block 6 as it no longer has any computation and increase the area of block 7. Using our incremental floorplanner, we get a new floorplan (FIGURE 2d). The new floorplan has reduced wirelength due to the Φ-function movement. From this example, we can see that traditional IDF Φ-function placement does not always produce optimal data communication. In the following, we show how it is possible to decrease wirelength by changing the position of the Φ-functions.

3.1 Problem Definition

The Φ-placement problem is formalized as follows: Given a CFG $G_{cfg}(V_{cfg}, E_{cfg})$ and a fully connected, weighted data cost graph $G_{cost}(V_{cost}, E_{cost})$ defining the data communication cost between every basic block in the CDFG. V_{cost} is the exact set of CDFG basic blocks, i.e. $V_{cost} = V_{cfg}$, and

$$E_{\text{cost}} = \left\{ e_{ij} \mid head(e_{ij}) = v_i \wedge tail(e_{ij}) = v_j \wedge v_i, v_j \in V_{\text{cost}} \wedge i \neq j \right\}$$

The *Phi Placement Problem* finds the set $place(\phi_a)$ for each ϕ-function ϕ_a, such that the total communication cost C for $place(\phi_a)$ is minimized, where data cost $C(place(\phi_a))$ of a Φ-function ϕ_a is represented by

$$C(place(\varphi_a)) = \sum_{p \in place(\phi_a)} \sum_{s \in S(\phi_a)} weight(e_{sp}) \; + \sum_{p \in place(\phi_a)} \sum_{d \in D(\phi_a)} \begin{cases} 0 & \text{if no path } p \rightarrow d \text{ exists} \\ weight(e_{dp}) & \text{otherwise} \end{cases}$$

In the above equation, each element p of set $place(\phi_a)$ is a basic block in the CFG at which that Φ-function may be placed (i.e. a candidate Φ-node). The set S is the set of blocks that contain source values for the Φ-function. The set D is the set of blocks at which the name defined by the Φ-function is used. The edges e_{sp} and e_{dp} are edges of the graph G_{cost}. The solution to this problem, i.e. the sets of blocks of $place(\phi_a)$, is subject to the following constraints, which maintain the program's correctness in SSA form:

$\forall s \in S(\phi_a)\ \exists$ *a control flow path* s -+-> p $\quad \forall p \in place(\phi_a)$

$\forall d \in D(\phi_a)\ \exists$ *at least one* $p \in place(\phi_a)$ *such that* p-+->d

The notation x -+-> y is the iterated non-inclusive edge, which represents the set of edges in the directed path from x to y. The first constraint maintains that the definition of each source variable of a Φ-function is live at each block defining that function. This enables proper propagation of source values to a Φ-function that it may select between them. The second constraint maintains that every destination (or use) of a Φ-function's value can be reached by at least one basic block which defines the Φ-function. In other words, the variable of a Φ-function will be live at any given block that uses it. Each of these constraints is trivially required for program correctness, as we must ensure that a variable's definition may reach its use point.

3.2 Φ-Placement Algorithm

The solution to this problem requires finding and measuring the cost of various possibilities of $place(\phi)$ for a given Φ-function. The permuted list of possible Φ placements has been proven to be exponential, as Φ-functions may be moved and/or replicated from the original iterated dominance frontier position, and these replicas themselves may be moved and/or replicated, etc. When considering placement options, the number of times a Φ-function may be duplicated in a placement is constrained by a constant k. This constant limits the number of Φ-functions allowed in a given placement $place(\phi)$. By constraining $|place(\phi)| < k$, we do limit our design space, but we are also placing a pragmatic limitation on the amount of data communication required. For this work, k was chosen to be 3. This intuition is empirically backed by the applications in the benchmark suite. We found that most Φ-functions have only two sources, and almost all of them have less than four destinations. Thus, we feel that our restriction of $|place(\phi)|$ is economic in terms of number of wires and multiplexers used.

Φ-Placement Algorithm

```
1.  Given a CFG G_cfg(V_cfg, E_cfg)
2.  perform_ssa(G_cfg)
3.  calculate_def_use_chains(G_cfg)
4.  remove_back_edges(G_cfg)
5.  topological_sort(G_cfg)
6.  foreach vertex v ∈ V_cfg
7.    foreach ϕ-node ϕ ∈ v
8.      s ← ϕ.sources
9.      d ← |def_use_chain(ϕ.dest)|
10.     IDF ← iterated_dominance_fronter(s)
11.     PossiblePlacements ← findPlacementOptions(IDF)
12.     place(ϕ) ← selectBest(PossiblePlacements)
13.     distribute/duplicate ϕ to place(ϕ)
```

FindPlacementOptions Algorithm

```
1.  Given a set of CFG Nodes R
2.  ϕ-options ← ∅
3.  insert(R) into ϕ-options
4.  foreach instruction i ∈ R
5.    if( i is a destination of ϕ-function f )
6.      return ϕ-options
7.    temp_ϕ-options ← ∅
8.    foreach non-dominated child c of R
9.      temp_ϕ-options ← crossProductJoin(temp_ϕ_options, findPlacementOptions(c))
10. return ϕ-options ∪ temp_ϕ-options
```

FIGURE 3 Φ-Placement Algorithm and the FindPlacementOptions Algorithm, which recursively builds candidate sets for place(ϕ)

The Φ-Placement Algorithm (FIGURE 3) builds a set of placement options for each Φ-function. These placement options are limited to k CFG basic blocks per placement. The placement options are found using a search function (lines 10-11), which starts execution at the iterated dominance frontier of the Φ-function's sources. After the set of possible placement options are discovered by the function *findPlacementOptions*, each of the placement options is evaluated via the cost function, which will be described briefly. The best Φ placement option is used, and the Φ-function is distributed and duplicated to this final placement. The algorithm's internal representation of each placement option is a set of numbers, where each number represents a CFG block where the Φ-function must be placed.

The function *findPlacementOptions* is a recursive algorithm which generates the possible placements for each Φ-function. To generate the set of sets which lists the

placement options, *findPlacementOptions* uses dominance information to traverse the graph in top-down fashion. Line 10 of the *findPlacementOptions* algorithm makes use of a binary function named *crossProductJoin*, which takes two sets of sets and combines them, removing duplicates and sets that are supersets of other sets. This removal of supersets ensures that the Φ-function is only placed at as many blocks as necessary. Additionally, *crossProductJoin* immediately drops any sets from the generated set of sets which are larger than *k*.

The cost between a given pair of basic blocks has two components: the distance between the blocks on the floorplan, and the amount of data communicated between these two blocks using wires. The distance between a pair of blocks on the floorplan is taken using the Euclidean (center-to-center) distance between the modules. The amount of data communicated between two blocks is the sum of the total bits of communication passed between these blocks. The formula representing the communication cost between any two basic blocks *i* and *j* (with Euclidean distance *Euclidean(i,j)* and total communication amount *bits(i,j)*) is *Cost[i,j] = Euclidean (i,j)*bits(i,j)*. Of course, other cost functions are possible. Our framework allows us to simply replace the above cost function with any other cost function of interest during Φ-function evaluation.

4. Experimental Results

In order to verify the purposed approach, ten functions from MediaBench [11] were synthesized using the design flow specified in Section 2. This section presents these experimental results. Table 1 shows some statistical information of those benchmarks. *Block* gives the number of basic blocks, *Φ* gives the number of Φ-function, *links* gives the number of variables between blocks, *weight* is the total number of links times the bit width of the links and *initial WL* is the wirelength from the initial floorplan.

Table 1: **Statistical Information of all benchmarks used**

	benchmark	blocks	Φ	links	weight	Initial WL
1	adpcm coder	33	31	54	2688	35568
2	adpcm decoder	26	23	44	1952	21588
3	internal filter	10	143	60	17088	411637
4	Internal expand	101	94	257	14336	317031
5	compress output	34	17	60	2368	29114
6	mpeg2dec block	62	13	66	2272	34510
7	mpeg2dec vector	16	4	26	1024	4366
8	FAST	14	4	15	704	3714
9	FR4TR	77	87	155	704	340697
10	det	12	5	13	7936	3772

To measure the effectiveness of introducing feedback into the design flow, we compiled functions from MediaBench into CDFG form. We converted these CDFGs into SSA form, and placed the Φ-functions according to the algorithm in [1]. Finally, we converted these SSA-form CDFGs into behavioral and structural VHDL, and synthesized the designs in Synopsys Design Compiler. We took the individual areas of each design module (corresponding to basic blocks from the CDFG) from Synopsys. We created a net file using information from our behavioral VHDL conversion pass. The net file listed the connections and connection-widths (in bits) between blocks. These files were supplied to a floorplanner based on simulated annealing [12], which was then executed to create a working floorplan. The output of the floorplanner was converted into a cost matrix as described in Section 3. This is the *initial iteration*.

Next the MachSUIF compiler pass performing SSA was re-executed upon each benchmark, this time with the benchmark's cost matrix from the first synthesis iteration. Φ-functions were replicated and distributed as described in Section 3. Then the conversion to VHDL and subsequent synthesis with Synopsys Design Compiler was repeated with the new Φ placement. New data and net files were created from this synthesis, and the floorplanner was re-executed. This creation of the final floorplan concluded the *incremental iteration*.

Our initial experiments did not use an incremental floorplanner in the second synthesis iteration. Rather we performed a full floorplanning [12]. The results were almost all negative and in most cases there was a drastic increase in wirelength. Although the layout driven transformations made an informed Φ placement decision using the design's physical characteristics, the decisions were based upon the characteristics of the initial iteration. If the second synthesis iteration is sufficiently different than the first iteration, then these decisions may not lead to data communication reduction. In other words, a large difference between the first and second synthesized designs may lead to unexpected results in feedback-driven optimization. We found that even a small change in some module areas can result in an entirely different floorplan for the second design iteration. In the end, feedback-driven optimization of data communication fails for these experiments, largely because the feedback of the first design's floorplan has no bearing on the subsequent redesign. The randomness of simulated annealing and module size changes due to Φ-function movement results in an unpredictable final floorplan. Thus it becomes very difficult to estimate physical characteristics of the final circuit at the compiler level.

If we were able to send the original floorplan back to the floorplanner for redesign, and constrain the floorplanner to make only essential moves on the new modules such that all pieces fit, we might obtain better results. In this case, the floorplanner would make much fewer random decisions during the second design iteration, and the redesign might approximate the characteristics of the original design more closely. This motivates the need for incremental floorplanning in the feedback loop.

Let us first consider the extreme case when the data communication optimizations do not have any affect on the floorplan. This is an unrealistic assumption, however it can

give us a bound on what sort of improvements we could hope to achieve assuming this "optimal" incremental floorplan. These optimal results are estimated as follows. Suppose the hardware needed to implement the Φ-functions has zero area, therefore no matter how we move the Φ-functions, the floorplan modules will not change. Hence the incremental floorplan will be identical to the initial floorplan. Since the floorplans are identical, the Φ-placement with the minimum cost in the initial floorplan will remain unchanged in the incremental floorplan. FIGURE 4 presents our experimental results after incremental floorplanning. The first bar represents the normalized initial wirelength of the floorplan. The remaining bars show the wirelength normalized with respect to the initial wirelength. "Overall" is the total wirelength of the design, while "Phi" is just the wirelength associated with the Φ-functions. The data communication optimizations can only affect the wires associated with the Φ-functions. "Optimal" is the wirelength assuming Φ-functions have zero area while "incremental" are the results using realistic areas for the Φ-functions and our incremental floorplanner.

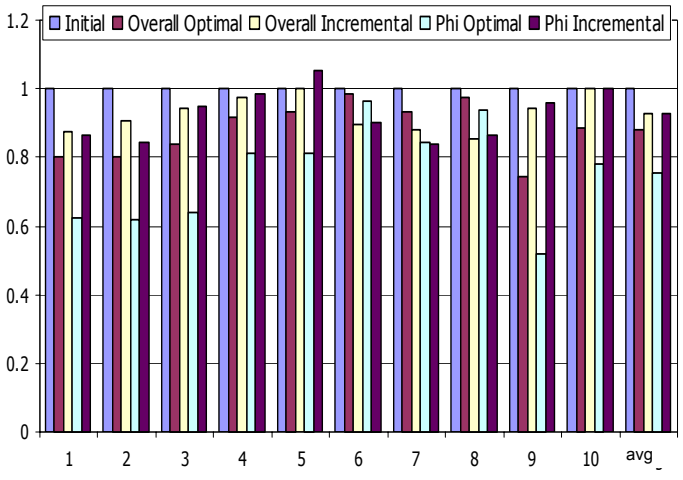

FIGURE 4 Wirelength results compared with initial floorplan wirelength and the "optimal" wirelength.

As indicated in our results, the "optimal" solutions are far better than the incremental results in most cases. The "optimal" approach achieves an average of 12% reduction on overall communication costs, and an average of 25% when we only consider the wirelength corresponding to the Φ-functions. Our approach using incremental floorplanning achieves an average of 6% reduction on overall communication costs, and an average of 8% when we only consider the communication costs of the Φ-function. These results point to the fact that the legalization of the floorplan does indeed play a role in the overall wirelength of the floorplan. In block_mpeg2dec, mpege2dec_vector and FAST (benchmarks 6, 7 and 8, respectively), our incremental costs are even smaller than the "optimal" solutions. In these cases, our new incremental floorplan causes additional wirelength reduction. In the benchmark 10 (corresponding to the function det) none of those Φ-functions are moved, hence there is no difference between the initial results and the incremental results. For compress_output (benchmark 5), worse results on Φ-related communications are obtained, which is mainly due to effects of actual cost of hardware implementation of the Φ-functions.

5. Conclusion

We presented a physically aware framework for compiling high level applications specifications to an RTL hardware description. We showed that an incremental floorplanner is a necessary component for layout driven optimizations, and we developed a fast, yet effective incremental floorplanning algorithm. We studied the data communication problem and developed a layout driven algorithm for this problem. Our results show our transformations have the ability to reduce the overall wirelength by an average of 12% using an "optimal" floorplan. We show that the proposed techniques can still reduce the overall wirelength of the final design by 6%, while maintaining a legal floorplan with the same area as the initial floorplan.

6. References

[1] A. Kaplan, P. Brisk, and R. Kastner, "Data communication estimation and reduction for reconfigurable systems," *Design Automation Conference,* 2003.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeick, "An efficient method of computing static single assignment form," *Symposium on Principles of Programming Languages*, 1989.

[3] M. Budiu and S. C. Goldstein, "Compiling application-specific hardware," *International Conference on Field-Programmable Logic and Application, 2002*.

[4] J. L. Tripp, P. A. Jackson, and B. L. Hutchings, "Sea cucumber: a synthesizing compiler for FPGAs," *International Conference on Field-Programmable Logic and Applications*, 2002.

[5] W. Gong, G. Wang, and R. Kastner, "A High Performance Intermediate Representation for Reconfigurable Systems," *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2004.

[6] P. Prabhakaran and P. Banerjee, "Simultaneous scheduling, binding and floorplanning in high-level synthesis," *International Conference on VLSI Design, 1997*.

[7] F. Yung-Ming and D. F. Wong, "Simultaneous functional-unit binding and floorplanning," *International Conference on Computer-Aided Design.1994*.

[8] W. E. Dougherty and D. E. Thomas, "Unifying behavioral synthesis and physical design," *Design Automation Conference, 2000*.

[9] D. W. Knapp, "Fasolt: a program for feedback-driven data-path optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems*, vol. 11, pp. 677-95, 1992.

[10] S. Tarafdar and M. Leeser, "A data-centric approach to high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits & Systems*, vol. 19, pp. 1251-67, 2000.

[11] L. Chunho, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," *International Symposium on Microarchitecture, 1997*.

[12] A. Ranjan, and M. Sarrafzadeh, "Floorplanner 1000 Times Faster: A Good Predictor and Constructor," *Workshop on System-Level Interconnection Prediction (SLIP)*, 1999.