Physically Aware Data Communication Optimization for Hardware 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, CA 93106
{kastner, gong, hao, forrest}@ece.ucsb.edu

[†]Computer Science Department
University of California
Los Angeles, CA 90095
{kaplan, philip, majid}@cs.ucla.edu

ABSTRACT

As the number of transistors in digital systems rises, we must rely on system level design techniques to manage the complexity and increase the efficiency of the designer. 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

Over the past five decades, semiconductor technology has experienced an unprecedented amount of growth and improvement. Computing systems have changed from simple circuits with hundreds of transistors to complex systems on a chip with more than one billion transistors. As the number of transistors has increased, so has the cost and complexity of designing and fabricating a computing system.

In order to manage the complexity of such complex digital systems, we advocate the use of system level design techniques. System level design methodologies and optimizations map an application specification on a computing platform. They increase the productivity of the designers by allowing them to reason about the underlying system in more abstract models of computation.

System level optimizations are performed early in the design flow and therefore 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, system 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.

In this paper, we describe a framework for physically aware transformations. Section 2 gives details on the design flow for our system compiler. We present a design flow from application specification to synthesizable register transfer level hardware description. We discuss the data communication problem, which determines the relative locations of application data. We argue that an incremental floorplanner is a necessary component for physically aware transformations, and we describe our incremental floorplanning algorithm. Section 3 motivates the importance of the data communication problem and presents a physically aware 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. FIGURE 1 shows our design flow.

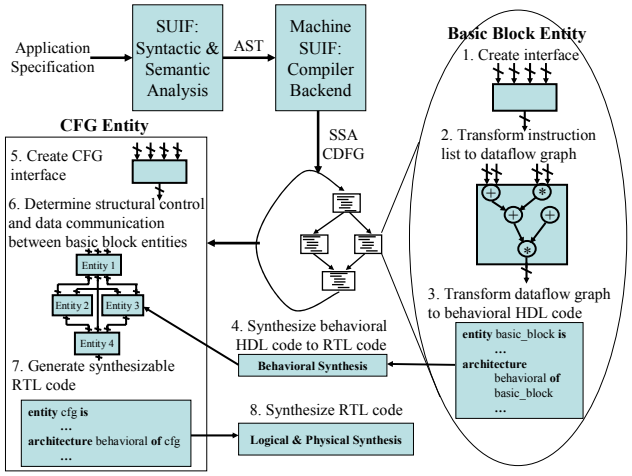

FIGURE 1 The flow from application specification to synthesizable register transfer level (RTL) hardware description.

An application specification undergoes syntactic and semantic analysis and is transformed into a control data flow graph using static single assignment (SSA CDFG) form. We provide techniques to translate the SSA CDFG form into a 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 (ASIC).

2.1 Hardware Synthesis

The input to our tool is an *application specification*. Languages for the specification of embedded applications are an active area of research ([1] provides an excellent survey). Our project focuses on the use of C as a design language.

We choose C-based design language for several reasons. First off, most programmers are familiar with the syntax and semantics of C. Additionally, embedded applications tend to use a lot of legacy code, much of which is implemented in C. Furthermore, it is easy to compile C to a wide variety of microprocessors. This allows the framework to use existing highly optimized backend code generation for simulation. Of course, C is far from the ideal language for hardware synthesis [2]. It should be noted that the design techniques described in this paper can be applied across a variety of systems design languages. Our transformations can be applied on any language that can use the SSA CFG form as an intermediate representation.

We start by using the SUIF front end [3] to perform syntactic/semantic analysis, which changes the specification into an abstract syntax tree (AST) program intermediate representation (IR). Then, we use the Machine SUIF backend to transform the AST into a control data flow graph using static single assignment (SSA CDFG) form.

Machine SUIF starts by transforming the AST from SUIF to a medium-level IR (MIR). The MIR is in the form of instruction lists and *control flow graphs (CFG)*. A CFG $G_{cfg}(V_{cfg}, E_{cfg})$ represents the control relationships between the set of basic blocks. For each basic block there is a unique vertex $v \in V_{cfg}$. An edge $e(u,v) \in E_{cfg}$ means that control can be transferred from basic block u to basic block v. A set of instructions is associated with every basic block. The instruction list of each basic block can be modeled by a *data flow graph (DFG)* $G_{dfg}(V_{dfg},E_{dfg})$ A *control data flow graph (CDFG)* is a CFG with the instructions of the basic blocks expressed as a DFG.

Static single assignment (SSA) [4] 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 it was originally intended to enable software-directed optimizations, it has been used in several projects where the final output is a hardware description language [5-7].

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

After every basic block entity is synthesized, we must 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 physically aware algorithm that optimizes the SSA form by moving and replicating Φ-functions to minimize the wirelength of the data communication.

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 (gate level) 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 Physically Aware 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 system compiler's 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 [8] developed a simulated annealing algorithm for simultaneous scheduling, binding and floorplanning. Fang and Wong [9] also use simulated annealing to solve the problem of functional unit binding while considering floorplan information. Dougherty and Thomas [10] use placement information while performing scheduling, allocation and binding. Fasolt [11] optimizes bus topology while considering layout information. Tarafdar and Leeser [12] incorporate floorplanning information with their synthesis flow.

We use a design feedback loop for physically aware transformations. The feedback loop works as follows: First, we perform original optimization(s) (without considering layout information) and generate an initial floorplan. Based on the layout obtained from the floorplan, we then perform physically aware optimizations.

The optimizations often alter the area and dimensions of the floorplan modules. Consider transformations to optimize data communication. These optimizations move multiplexers, equivalently Φ-functions, from one basic block to another; this is discussed in further detail in Section 3. This will likely change the dimensions of the individual modules comprising the design and will add or subtract from the areas of the modules depending on whether a Φ-functions is being moved to or from the module. 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. The physically aware 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 correspond 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, etc. 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 physically aware 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.

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

2.3 Incremental Floorplanning

Floorplanning is a fundamental physical design problem of allocating space to a set of modules in a plane while minimizing the area of bounding rectangle containing all the modules and/or total wirelength among modules.

Coudert et al [13] gave two definitions of incremental placement, which are equally applicable to incremental floorplanning: 1) Given an existing placement optimized with respect to a given metric, e.g. wirelength, modify the placement to improve it with respect to other metrics, e.g. congestion, timing, or routability. 2) Given an optimized placement and a set of changes to the netlist (e.g., due to technology remapping) modify the placement to improve it.

Somewhat surprisingly there is not much work on the problem of incremental floorplanning. Crenshaw et al [14] presented an incremental floorplanner using slicing tree. They mark any modules that changed greater than a specified threshold as critical, and put their parent nodes in to the critical queue. A full floorplanning algorithm is reapplied on each node in the critical queue. In the worst case this method does twice as much work as a full floorplan.

We developed a simple incremental floorplanner using slicing structure. The advantages of our methods are: 1) full floorplanning is not required (except for initial layout), 2) the initial slicing-tree is maintained, 3) changes to unperturbed modules are minimized, and 4) it is relatively simple to implement. We have not done a detailed comparison with Crenshaw et al [14], 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 tree to represent the slicing structure 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. Each internal nodes 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.

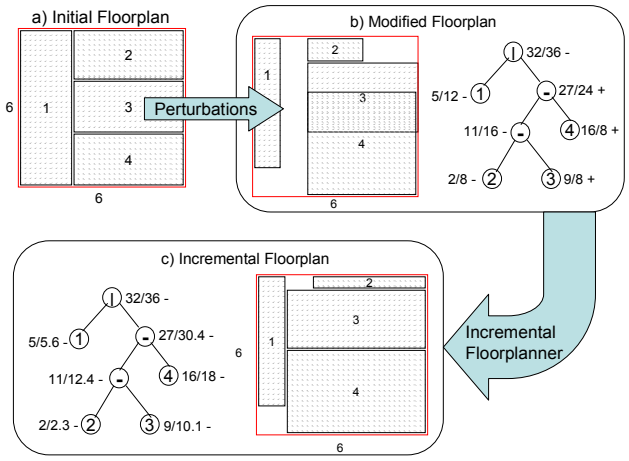

FIGURE 2 The initial floorplan is perturbed, which results in a modified floorplan. The incremental floorplanner generates a legal floorplan while attempting to maintain the structure of the initial floorplan.

The input of our incremental floorplanner is the initial floorplan and a list of perturbations to the floorplan. The perturbations denote the changes that have been made to the floorplanner that we must account for in the incremental floorplan. These include the modules with increased/reduced area and changes in wire connections between the modules. 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 2 shows an example. "32/36-" means the area is 32, the room is 36, and it is marked as "-".

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 2 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 2 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 2 b), there are three positive modules in the slicing tree. After traversal, all of the modules become negative (FIGURE 2 c). That means each module can fit its room and we have a legal floorplan.

During the top-down reallocation, we try to avoid changing the shape of unmodified blocks in an attempt to maintain the structure of the original floorplan. But it is still possible to move them slightly to satisfy room requests of neighboring blocks. Furthermore, we try to avoid allocating modules with extremely large/small aspect ratios. In such cases, we will allocate additional room to the module such that the aspect ratio can be reduced/increased, i.e. made more "squarish".

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. The specific place where data is generated is called its *definition point*, and the place where that data is used in computation is called a *use point*. The data generated at a particular definition point may be used in multiple places. Likewise, a particular use point may correspond to a number of different definition points; the control flow dictates the actual definition point used in later computation. In our design flow, the floorplan modules correspond to the basic block entities in the CDFG. Thus, we model data communication as a set of bits moved from one basic block to another.

We can determine the relationship between the use and definition points 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. A naïve yet standard hardware model would connect all the basic blocks reading/writing a variable of the same name. By contrast, SSA form only connects definition points and use points of data values – the minimal amount of communication required between basic blocks.

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.

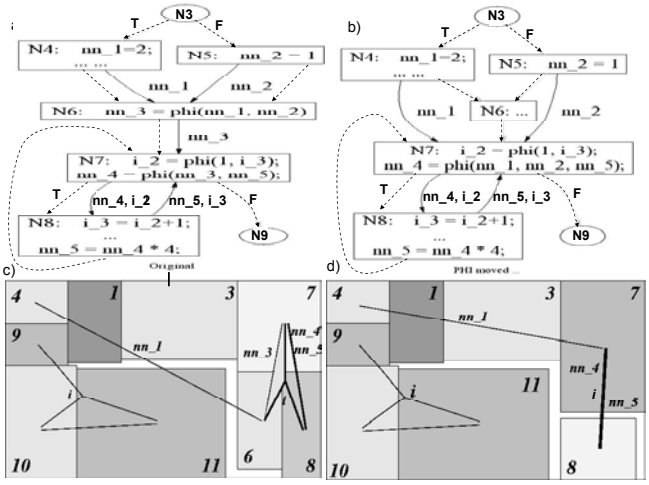

FIGURE 3 Synthesized FAST function. Parts a) and b) are a subset of basic blocks from the function's control flow graph along with the Φ-function related code. The dotted lines show the control flow while the solid lines show the data flow. Part a) gives the initial locations of the variables and Φ-functions. Part b) shows the physically aware 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).

We illustrate our point with FAST function from the MediaBench test suite [15]. FIGURE 3 exhibits SSA form with Φ-function placement at the IDF (FIGURE 3a). It also shows the corresponding initial floorplan (FIGURE 3c). 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 3b), 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 3d). 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 decrease wirelength by changing the position of the Φ-functions.

In our previous work [16], we showed that a hardware implementation of a CDFG in SSA form can reduce redundant data communication between modules in the hardware description, leading to a smaller hardware design via interconnect reduction. Also, we 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 basic blocks in the final hardware. In other words, without knowing where basic blocks will be placed relevant 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 system design, behavioral and logic synthesis, and floorplanning. Additionally, our design flow contains a feedback loop from the floorplanner back to the system compiler. This helps the system compiler determine the actual physical cost of communicating between any two basic blocks within a CDFG. 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.

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 $\forall p \in place(\phi_a)$

$\forall d \in D(\phi_a)$ $\exists$ *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 that define 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. These possibilities must adhere to the correctness constraints mentioned in the previous section. 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. Due to the necessity of building communication wires from all sources to all Φ-functions and from these Φ-functions to a set of destinations, replication of Φ-functions may hurt more than help in the general case. Additionally, the range of possible placements is not very large for a given Φ-function, as Φ live

ranges tend to be short, and Φ-functions tend not to have many sources. When considering placement options, the number of times a Φ-function may be duplicated in a placement should be constrained by a constant k. This constant will practically limit the number of Φ-functions allowed in a given placement *place*(ϕ). By constraining |*place*(ϕ)| < *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 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. For these cases, which represent the majority, it would be wasteful to consider Φ-placements with many duplicates of the Φ-function. Thus, we feel that our restriction of |*place*(ϕ)| 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 $v \in V_{cfg}$
7. **foreach** ϕ-node $\phi \in v$
8. $s \leftarrow \phi.sources$
9. $d \leftarrow$ |def_use_chain($\phi.dest$)|
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 $i \in R$
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 4 Φ-Placement Algorithm and the FindPlacementOptions Algorithm, which recursively builds candidate sets for place(ϕ)

The Φ-Placement Algorithm in FIGURE 4 builds a set of placement options for each Φ-function. These placement options are limited to *k* CFG basic blocks per placement. For instance, for a Φ-function whose placement options are {{1, 2}, {5}, {3,6,8}}, the Φ-function can either be placed at basic block 5, replicated and placed at blocks 1 and 2, or replicated and placed at blocks 3, 6, and 8. In this case, if *k*=2, then the {3,6,8} option would not be considered, and would be dropped from the list of placement options. As stated in lines 10-11, the placement options are found using a search function, 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 then 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*, i.e. would correspond to a placement of greater than *k* Φ-functions.

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. For instance, if two blocks share variables *i* and *j* (each 32-bit integers), and one ASCII character *c* (8 bits), then the amount of communication passed between these blocks is 72 bits.

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 as follows:

$$Cost\,[i,j] = \begin{cases} Euclidean\ (i,j) * bits(i,j) & if\ bits(i,j) > 0 \\ Euclidean\ (i,j) * 32 & otherwise \end{cases}$$

The second case, which substitutes the number of bits communicated for the constant 32, derives from the following anomaly: how can we determine the communication cost between two basic blocks if they are not yet sharing any data? In this case, bits(i,j) = 0. However, sharing data between these blocks still incurs a design cost. This cost must still depend on the distance between the two blocks on the final floorplan. For this cost to appear significant in the face of the other costs, a multiplier is needed which will make this cost commensurate with costs between blocks which share signals. The constant 32 was chosen because it is the most often exhibited size of bits shared between any two blocks (i.e. most blocks share data the size of a single integer variable). 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 examples from the MediaBench [15] 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 with the SUIF compiler [17] and MachSUIF compiler backend. We converted these CDFGs into SSA form, and placed the Φ-functions according to the algorithm in [16]. Finally, we converted these SSA-form CDFGs into behavioral and structural VHDL, and then synthesized the designs in the Synopsys Design Compiler. We then took the individual areas of each design module (corresponding to basic blocks from the CDFG) from Synopsys. We then 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 [18], 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 concluded the *first synthesis 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 behavioral/logic synthesis in Synopsys 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 *second synthesis iteration*.

Our initial experiments did not use an incremental floorplanner in the second synthesis iteration. Rather we performed a full floorplanning. The results were almost all negative and in most cases there was a drastic increase the wirelength. Although the physically aware transformations made an informed Φ placement decision with knowledge about the design's physical characteristics, the decisions were based upon the characteristics of the first synthesis 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 (via the removal/addition of multiplexers for Φ-functions) 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 further 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 quite an unrealistic assumption, however it can give us a good bound on what sort of improvements that 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. We need to note this "optimal" solution is ideal and can only be used as a reference.

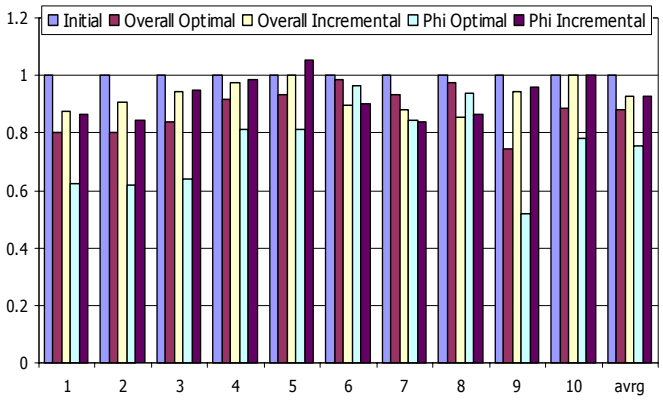

FIGURE 5 Wirelength results compared with initial floorplan wirelength and the "optimal" wirelength.

FIGURE 5 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 (described above) while "incremental" are the results using realistic areas for the Φ-functions and our incremental floorplanner.

As indicated in our results, the "optimal" solutions are far better than what 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 reduction wirelength. 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 a RTL hardware description. We showed that an incremental floorplanner is a necessary component for physically aware transformations and we developed a fast, yet effective incremental floorplanning algorithm. We studied the data communication problem and developed a physically aware 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] E. A. Lee, "Embedded Software," in *Advances in Computers*, vol. 56, M. Zelkowitz, Ed. London: Academic Press, 2002.

[2] S. A. Edwards, "The Challenges of Hardware Synthesis from C-like Languages," *International Workshop on Logic and Synthesis (IWLS)*, 2004.

[3] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, L. Shih-Wei, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, vol. 29, pp. 84-9, 1996.

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

[5] M. Budiu and S. C. Goldstein, "Compiling application-specific hardware," *International Conference on Field-Programmable Logic and Applications (FPL). 2002*, pp. 853-63.

[6] 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.

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

[8] P. Prabhakaran and P. Banerjee, "Simultaneous scheduling, binding and floorplanning in high-level synthesis," *International Conference on VLSI Design. 1997*, pp. 428-34.

[9] F. Yung-Ming and D. F. Wong, "Simultaneous functional-unit binding and floorplanning," *IEEE/ACM International Conference on Computer-Aided Design1994*, pp. 317-21.

[10] W. E. Dougherty and D. E. Thomas, "Unifying behavioral synthesis and physical design," *Design Automation Conference. 2000*, pp. 756-61.

[11] 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.

[12] 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, 2000.

[13] O. Coudert, J. Cong, S. Malik, and M. Sarrafzadeh, "Incremental CAD," *IEEE/ACM International Conference on Computer Aided Design. 2000*, pp. 236-43.

[14] J. Crenshaw, M. Sarrafzadeh, P. Banerjee, and P. Prabhakaran, "An incremental floorplanner," *Great Lakes Symposium on VLSI. 1999*, pp. 248-51.

[15] 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*, pp 330-5.

[16] A. Kaplan, P. Brisk, and R. Kastner, "Data communication estimation and reduction for reconfigurable systems," *Design Automation Conference. 2003*, pp. 616-21.

[17] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, 1996.

[18] K. B. Abhi Ranjan, and Majid Sarrafzadeh, "Floorplanner 1000 Times Faster: A Good Predictor and Constructor," presented at Workshop on System-Level Interconnection Prediction (SLIP), Monterey, CA, 1999.