Physically Aware Data Communication Optimization for Reconfigurable System Synthesis

ABSTRACT
Modern reconfigurable systems contain 100,000+ programmable logic elements. In order to program these devices, 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 system level design flow for mapping high level application specifications to programmable logic. In particular, we study the problem optimizing data communication of the variables in the application specification. The framework uses floorplan information to guide compiler optimizations. Our results show physically aware compiler transformations have the ability to reduce the wirelength by an average of 12%. However, compiler optimizations often modify the design which leads to an illegal floorplans. We develop a simple, yet effective, incremental floorplanner to handle the perturbations caused by compiler optimizations. We show that the proposed techniques can still reduce the wirelength of the final design by 6%, 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 close to one billion transistors. As the number of transistors has increased, so has the cost and complexity of designing and fabricating a computing system. Design costs are reaching the point where only the “big players” can afford to manufacture customized chips. Smaller technology nodes bring about an increasingly larger number of system transistors, but also brings along a host of nonscale issues. Both of these increase the complexity of designing a computing system. To further exacerbate the problem, designers are facing increasing demands from their consumers. We can be sure that users of embedded computing systems will continue to demand devices that are cheaper, smaller and have more functionality. In order to keep up with these demands, designers must decrease time to market, which furthers the need for fast, effective design tools and methodologies.

In order to overcome these challenges, we advocate the use of system level design techniques on reconfigurable computing systems. System level design techniques increase the productivity of the designers by allowing them to reason about the underlying system in more abstract models of computation. Reconfigurable systems are programmable and regular, which increases their flexibility, reduces the system complexity, decreases the time to market and amortizes the design costs over a large number of users.

Reconfigurable computing systems have emerged as an important organizational structure for implementing computations. Reconfigurable systems are a computing device that allows different tradeoffs between flexibility and performance. They combine the post-fabrication programmability of software running on a general purpose processor with the spatial computational style most commonly employed in hardware designs. Reconfigurable systems use programmability and regularity to create a flexible computing platform that can lower design costs, reduce system complexity and decrease time to market.

The computing power of modern, high performance reconfigurable systems allows us to use them to design extremely complex applications. The complexity of both the application and the underlying reconfigurable platform requires advanced design tools to increase productivity and exploit degrees of freedom. System level design is the methodologies and optimizations need to map an application specification on a computing platform. A specification contains only the necessary requirement, constraints and functionality for the application.

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. In particular, they have minute information about the physical layout of the circuit, which is often determined in the final stages of synthesis (physical synthesis). Most often, compiler optimizations simply transform the specification to reduce the computing complexity of the application. If we could somehow take into account physical information during compiler optimizations, we would have the ability to greatly affect the layout of the circuit.

In this paper, we describe a framework for physically aware compiler transformations. Section 2 presents the design flow for our system techniques for reconfigurable systems. We show that an incremental floorplanner is a necessary component for physically aware compiler transformations and we describe our incremental floorplanning algorithm in Section 3. To motivate our framework, we consider the data communication problem, which determines the relative locations of application data. Section 4 motivates the importance of the data communication problem, presents a physically aware solution to this problem and shows experimental results. We conclude in Section 5.

2. Design Flow
The framework that we are investigating accepts a high level application specification and generates a bitstream for programming reconfigurable logic. FIGURE 1 shows our design flow.

FIGURE 1 The flow from application specification to a bitstream for programming reconfigurable logic.

2.1 Application Specification
The input to our tool is an application specification. Languages for the specification of embedded applications are a rich and extremely active area of research [1]. Instead of creating yet another embedded system design language, the project looks at design techniques that can be applied across a variety of systems design languages. Our project focuses on the
use of C as a design language because of its many benefits in embedded systems design.

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. However, C lacks the ability to provide the amount of task level parallelism needed in embedded systems. Additionally, it has no methods for specifying timing constraints that are required by most embedded applications. Many projects use a C syntax enhanced with language features to describe things like concurrency, timing and variable bit widths.

An number of industrial tools use a variant of C/C++ as a design specification language [2]: Cynlib, Cyber, Scenic/Centric from Synopsys, OCAPI, System Compiler from C Level Design (acquired by Synopsys), C2Verilog, Celoxica’s Handel-C compiler, and Cadence’s ECL – a C/Estrel language. Additionally, there are several academic research projects that use a C-like language including SpC [3], PACT [4], CASH [5] and SPARK [6]. Furthermore, a number of projects have focused on transforming a high level language for implementation on a reconfigurable architecture. The notable projects are Chimaira [7], DEFACTO [8], DISC [9], Garp [10], OneChip [11], PRISC [12], RaPD [13], SA-C [14] and StreamsC [15].

2.2 Generating Synthesizable Register Transfer Level (RTL) Code

We start by using the SUIF front end 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 CFG) 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 \( G_{dfg}(V_{dfg}, E_{dfg}) \) represents the control relationships between the set of basic blocks. For each basic block there is a unique vertex \( v \in V_{dfg} \). An edge \( e(u,v) \in E_{dfg} \) means that a 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). 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) [16] is a compiler transformation that converts a CDFG into another CDFG in which each variable is defined in exactly one static code location. This new CDFG is said to be in SSA form. SSA is a safe transformation for hardware design because lone side effects of the transformation, \( \phi \)-functions, are easily implemented in hardware as multiplexers. Although it was originally intended to enable software-directed optimizations, it has been used in many projects where the final output is an HDL [5, 17, 18].

The next step in our framework requires that we transform the SSA CDFG into a bitstream used to program the reconfigurable logic. In this section, we describe the translation of the SSA CDFG into a synthesizable RTL code. We can then use commercial logical and physical design tools to convert the RTL code into a bitstream.

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. After every basic block entity is synthesized, the entire design is realized as a two level hierarchical structural representation. We can then 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 realize the final implementation of the application.

2.3 Physically Aware Compiler Transformations

High level compiler transformations play a large role in determining the final properties of the application mapping. Unfortunately, compiler transformations are performed without much knowledge of the final circuit layout. However, it is possible to make an initial decision at the compiler stage, continue on with synthesis, glean information from the final design, and then return to the compiler for reevaluation of the initial transformation. The 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 back end synthesis of the application.

We use of a design feedback loop for physically aware compiler transformation. The feedback loop works as follows: First, we perform original compiler optimizations (without layout information) and generate an initial floorplan. Based on the layout obtained from the floorplan, perform physically aware compiler optimizations. Since the optimizations will likely change our initial floorplan, we must now generate another floorplan. You can iterate on this process as necessary.

The first floorplan can be obtained using any general floorplanner. However, the subsequent floorplans must use an incremental floorplanner. The reason is that physically aware compiler transformations use layout information from the floorplan in order to make decisions. If the floorplan before the transformations and the floorplan after the transformations are vastly different, then the compiler transformation can perform detrimental optimizations since it assumed the information from the first floorplan would still hold even after the optimizations are performed. Ideally, the floorplan would not change at all, however, since the compiler may change the number, size and shape of the floorplan modules, this is usually not the case.

3. Incremental Floorplanning

Floorplanning is the 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. Modules can be classified as rigid (hard) or soft. If a module has variable shape and dimensions, it is said to be soft, otherwise, it is said to be rigid. Floorplans can be divided in to two categories, slicing and the non-slicing floorplans. A slicing floorplans can be obtained by recursively cutting rectangles horizontally or vertically in to smaller rectangles; otherwise it is non-slicing floorplan.

Coudert et al. [19] 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. Crenshaw et al. [20] presented an incremental floorplanner using slicing tree. He marks any modules that changed greater than a specified threshold as critical, puts 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. Another problem is that a module must be re-floorplanned once its “brother” node becomes critical, even if it is unchanged. This introduces unnecessary disturbance.

Now, we present an algorithm to build an incremental floorplanner with 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, and 3) changes to unperturbed modules are minimized.
We will use a binary tree to represent the slicing structure and use the following definitions. A basic module is a block (corresponding to an 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.

The input of our incremental floorplanner is the initial floorplan and a list of permutations to the floorplan. The permutations 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.

### 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.

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 have 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 “squareish”).

### 4. 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 may dictate the actual definition point used in later computation. In our design flow, the modules are referred to as control nodes, and roughly correspond to nodes on a CDFG. Thus, we model data communication at the compiler level as a set of bits moved from one CDFG node to another.

We can determine the relationship between the use and definition points through a classic compiler intermediate form known as Static Single Assignment [16]. 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 \( \Phi \)-functions into the CDFG. \( \Phi \)-functions are needed when a particular use of a name is defined at multiple points. A \( \Phi \)-function takes a set of possible names and outputs the correct one depending on the path of execution. A \( \Phi \)-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 inter-node communication by restricting data movement between control nodes. As each variable in SSA corresponds to a specific value that is never reassigned, a variable is only shared between two control nodes if one node produces a data value used in the second node. A naïve yet standard hardware model would connect all 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 control nodes.

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

We illustrate our point with FAST function from the MediaBench test suite. FIGURE 3 exhibits SSA form with \( \Phi \)-function placement at the IDF (FIGURE 3b). It also shows the corresponding initial floorplan (FIGURE 3d). The lines in the floorplan correspond to the data communication required by the three \( \Phi \)-functions. The \( \Phi \)-functions are placed in nodes 6 and 7. Of the three \( \Phi \)-functions, only the \( \Phi \)-function corresponding to \( \Phi_{a,b} \) (the one initially in cfg node 6), has the opportunity to be moved. The other two nodes only have one correct location, which is their current location at the IDF. By moving \( \Phi \)-function \( \Phi_{a,b} \) to cfg node 7 (FIGURE 3c), we eliminate cfg node 6 as it no longer has any computation and...
increase the area of_cfg node 7. Using our incremental floorplanner described in Section 3, we get a new floorplan (FIGURE 3e). The new floorplan has reduced wirelength due to the \( \Phi \)-function movement.

![FIGURE 3](image-url)

**FIGURE 3** Synthesized FAST function. Part a) shows the CFG of the function. Parts b) and c) are an enlargement of CFG nodes 4-8 from Part a). Part b) gives the initial locations of the variables and \( \Phi \)-nodes. Part c) shows the physically aware \( \Phi \)-node locations. Part d) is the floorplan corresponding to the initial locations (part b). Part e) shows the incremental floorplan for the moved \( \Phi \)-nodes from Part c).

From this example, we can see that traditional IDF \( \Phi \)-function placement does not always produce optimal data communication. The IDF is the best location to place a \( \Phi \)-function in software compilation, as it makes live ranges as short as possible and reduces register contention. However, the IDF is not the only place where a \( \Phi \)-node can exist, and in the case of hardware, it may not be the best location. In the following section, we show how it is possible to achieve better hardware by changing the position of the \( \Phi \)-functions.

Kaplan et al. [21] have shown 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. In their work, they demonstrated that data communication could be further reduced via intelligent placement of the SSA-generated \( \Phi \)-nodes. However, their proposed \( \Phi \)-node 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 the work of Kaplan et al. by augmenting their communication cost model with knowledge about module placement on the synthesized hardware. We present a complete design flow encompassing compilation, behavioral and logic synthesis, and floorplanning. Additionally, our design flow contains a feedback loop from the floorplanner back to the compiler. This helps the 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 \( \Phi \)-node placement heuristic, and explore more of the design space of the system. Ultimately, 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.

### 4.1 Problem Definition

The \( \Phi \)-placement problem is formally described as follows:

Assume that we are given 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 control nodes, and

\[
E_{cost} = \{ e \mid \text{head}(e) = v_i \land \text{tail}(e) = v_j \land v_i \lor v_j \in V_{cost} \land i \neq j \}
\]

The Phi Placement Problem is the following. Find the set \( \text{place}(\phi_i) \) (for each \( \phi \)-function \( \phi_i \)) such that the total communication cost \( C(\text{place}(\phi_i)) \) is minimized, where data cost \( C(\text{place}(\phi_i)) \) is represented by

\[
C(\text{place}(\phi_i)) = \sum_{e \in E_{cost}} \sum_{p \in \text{place}(\phi_i)} \text{weight}(e) \times \text{distance}(p, e)
\]

In the above equation, each element \( p \) of set \( \text{place}(\phi_i) \) is a node in the CDFG at which that \( \phi \)-function may be placed (i.e. a candidate \( \Phi \)-node). The set \( S \) is the set of nodes that contain source values for the \( \Phi \)-function. The set \( D \) is the set of nodes at which the name defined by the \( \phi \)-function is used. The edges \( e_{cap} \) and \( e_{dop} \) are edges of the graph \( G_{cost} \). The solution to this problem (i.e. the sets of nodes of \( \text{place}(\phi_i) \)) is subject to the following constraints:

1. \( \forall e \in E(\phi_i) \exists a \text{ control flow path } s \rightarrow p \rightarrow \text{place}(\phi_i) \) such that \( p \rightarrow d \)
2. \( \forall d \in D(\phi_i) \exists \text{ at least one } p \text{ place}(\phi_i) \text{ such that } p \rightarrow d \)

The first constraint maintains that the definition of each source variable of a \( \Phi \)-function is live at each \( \Phi \)-node defining that function. This enables proper propagation of source values to a \( \Phi \)-node that it may select between them. The second constraint maintains that every destination (or use) of a \( \Phi \)-function’s value can be reached by at least one \( \Phi \)-node defining that \( \Phi \)-function. In other words, the variable of a \( \Phi \)-function will be live at any given node 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.

### 4.2 \( \Phi \)-Placement Algorithm

The solution to this problem requires finding and measuring the cost of various possibilities of \( \text{place}(\phi_i) \) for a given \( \phi \)-function. These possibilities must adhere to the correctness constraints mentioned in the previous section. The permuted list of possible \( \Phi \) placements has been proven to be exponential (as \( \Phi \)s 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 \( \Phi \)-nodes (and from these \( \Phi \)-nodes to a set of destinations), replication of \( \Phi \)-functions (i.e. placement at many \( \Phi \)-nodes) may hurt more than help in the general case. Additionally, the range of possible placements is not very large for a given \( \Phi \)-function, as \( \Phi \) live ranges tend to be short, and \( \Phi \)-functions tend not to have many sources. When considering placement options, the number of times a \( \Phi \)-function may be duplicated in a placement should be constrained by a constant \( k \). This constant will practically limit the number of \( \Phi \)-nodes allowed in a given placement \( \text{place}(\phi) \). By constraining \( |\text{place}(\phi)| < k \) (for a chosen \( k \)), 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 \( \Phi \)-nodes have only two sources, and almost all

---

1 The notation \( x \leftrightarrow y \) is the iterated non-inclusive edge, which represents the set of edges in the directed path from \( x \) to \( y \).
of them have less than four destinations. For these cases, which represent the majority, it would be wasteful to consider \( \Phi \)-placements with many duplicates of the \( \Phi \)-function. Thus, we feel that our restriction of \( \text{place}(\phi) \) is economic in terms of number of wires and multiplexers used.

The \( \Phi \)-Placement Algorithm in FIGURE 4 builds a set of placement options for each \( \Phi \)-function. These placement options are limited to \( k \) CDFG nodes per placement. For instance, for a \( \Phi \)-function whose placement options are \( \{\{1, 2\}, \{5\}\} \), the \( \Phi \)-function can either be placed at CFG node 5, replicated and placed at CFG nodes 1 and 2, or replicated and placed at CFG nodes 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 \( \Phi \)-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 \( \Phi \) placement option is then used, and the \( \Phi \)-node 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 CDFG node where the \( \Phi \)-function must be placed.

**\( \Phi \)-Placement Algorithm**

1. Given a CDFG \( G \)
2. perform_ssa(\( G \))
3. calculate_def_use_chains(\( G \))
4. remove_back_edges(\( G \))
5. topological_sort(\( G \))
6. foreach node \( n \in N_{C F G} \) do:
   a. \( s \leftarrow \phi \)-sources
   b. \( d \leftarrow \text{def}_u \)-chain(\( \phi \)-dest)
   c. \( I F S \leftarrow \text{iterated_dominance_frontier}(s) \)
   d. \( \text{PossiblePlacements} \leftarrow \text{findPlacementOptions}(I F S) \)
7. place(\( \phi \)-node \( n \) \) in \( \text{PossiblePlacements} \)
8. \( s \leftarrow \phi \)-dests
9. foreach \( i \in \phi \)-dests do:
   a. \( d \leftarrow \text{def}_u \)-chain(\( \phi \)-dest)
   b. \( c \leftarrow \text{non-dominated} \left( \phi \right) \) \( c \) \( \leftarrow \text{dominated} \left( \phi \right) \)
10. \( \text{temp}_1 \leftarrow \text{crossProductJoin}(\text{temp}_2, \text{options}) \)
11. \( \text{return} \ \text{temp}_1 \)

**FindPlacementOptions Algorithm**

1. Given a region \( R \)
2. \( \phi \)-options \( \leftarrow R \)
3. insert(\( R \)) into \( \phi \)-options
4. foreach instruction \( i \in R \) do:
   a. \( R \leftarrow \text{dominated} \left( \phi \right) \)
5. if \( i \) is a destination of \( \phi \)-function \( f \) then:
6. \( \text{return} \ \phi \)-options
7. \( \text{temp}_1 \leftarrow \phi \)-options
8. \( \text{temp}_2 \leftarrow \text{crossProductJoin}(\text{temp}_1, \phi \)-options) \)
9. \( \text{return} \ \text{temp}_2 \)

**FIGURE 4 \( \Phi \)-Placement Algorithm and the FindPlacementOptions Algorithm, which recursively builds candidate sets for place(\( \phi \))**

The function findPlacementOptions is a recursive algorithm which generates the possible placements for each \( \Phi \)-function. To generate the set of sets which lists the placement options, findPlacementOptions uses dominance information to traverse the graph in top-down fashion. (It is assumed for this algorithm’s purposes that there are no unnatural loops in the code. A modification to the algorithm would need to be made otherwise.). 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 which are supersets of other sets. This removal of supersets ensures that the \( \Phi \)-function is only placed at as many nodes 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 \( \Phi \)-nodes).

The cost between a given pair of CDFG nodes has two components: the distance between the nodes on the floorplan, and the amount of data communicated between these two nodes using wires. The distance between a pair of nodes on the floorplan is taken using the Euclidean (center-to-center) distance between the modules. This can easily be obtained using simple geometry on the floorplanner’s output file (which specifies the size and coordinates of each module in the floorplan). The amount of data communicated between two nodes is the sum of the total bits of communication passed between these nodes. For instance, if CDFG nodes share variables \( i \) and \( j \) (each 32-bit integers), and one ASCII character \( c \) (8 bits), then the amount of communication passed between these nodes is 72 bits.

The formula representing the communication cost between any two basic blocks \( i \) and \( j \) (with Euclidean distance \( \text{Euclidean}(i,j) \) and total communication amount \( \text{bits}(i,j) \)) is as follows:

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

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 CDFG nodes if they are not yet sharing any data? In this case, \( \text{bits}(i,j) = 0 \). However, sharing data between these CDFG nodes still incurs a design cost. This cost must still depend on the distance between the two CDFG nodes 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 nodes which share signals. The constant 32 was chosen because it is the most often exhibited size of bits shared between any two CDFG nodes (i.e. most nodes 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 the compiler’s \( \Phi \)-function evaluation.

**4.3 Experimental Results**

In order to verify the proposed approach, ten examples from the MediaBench were synthesized using the design flow specified in Section 2. This section presents these experimental results.

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

<table>
<thead>
<tr>
<th>benchmark</th>
<th>nodes</th>
<th>( \Phi )</th>
<th>links</th>
<th>weighted</th>
<th>Initial WL</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm_coder</td>
<td>33</td>
<td>31</td>
<td>54</td>
<td>2688</td>
<td>35568</td>
</tr>
<tr>
<td>adpcm_decoder</td>
<td>26</td>
<td>23</td>
<td>44</td>
<td>1952</td>
<td>21588</td>
</tr>
<tr>
<td>internal_filter</td>
<td>10</td>
<td>143</td>
<td>60</td>
<td>17088</td>
<td>411637</td>
</tr>
<tr>
<td>internal_expand</td>
<td>101</td>
<td>94</td>
<td>257</td>
<td>14336</td>
<td>317031</td>
</tr>
<tr>
<td>compress_output</td>
<td>34</td>
<td>17</td>
<td>60</td>
<td>2368</td>
<td>29114</td>
</tr>
<tr>
<td>mpeg2dec_block</td>
<td>62</td>
<td>13</td>
<td>66</td>
<td>2272</td>
<td>34510</td>
</tr>
<tr>
<td>mpeg2dec_vector</td>
<td>16</td>
<td>4</td>
<td>26</td>
<td>1024</td>
<td>4366</td>
</tr>
<tr>
<td>FAST</td>
<td>14</td>
<td>4</td>
<td>15</td>
<td>704</td>
<td>3714</td>
</tr>
<tr>
<td>FR4TR</td>
<td>77</td>
<td>87</td>
<td>155</td>
<td>704</td>
<td>340697</td>
</tr>
<tr>
<td>det</td>
<td>12</td>
<td>5</td>
<td>13</td>
<td>7936</td>
<td>3772</td>
</tr>
</tbody>
</table>

Table 1 shows some statistical information of those benchmarks. \( \Phi \) gives the number of basic blocks (control nodes), \( \Phi \)-related costs are slightly better in most cases. \( \Phi \)-related costs are slightly better in most cases. \( \Phi \)-function communications are fixed. As discussed before, the communications among other nodes also affected the incremental communication cost. Since our approach benefits the \( \Phi \)-function communications during incremental synthesis and floorplanning, it is reasonable that \( \Phi \)-related costs are slightly better in most cases.

Results show that, in det, none of those \( \Phi \)-functions are moved, hence there is no difference between the initial results and the incremental one.
For compress-output, worse results on $\Phi$-related communications are obtained, which is mainly due to affects of actual cost of hardware implementation of $\Phi$-functions.

In order to further evaluate our approach, our incremental results are compared with the estimated pseudo-optimal solution of the initial floorplan. Those overall optimal and $\Phi$-optimal results are estimated as follows. Suppose the hardware of occupied by implementations of $\Phi$-functions have zero area, therefore no matter how we move the $\Phi$-functions, floorplan modules will not change. Hence the incremental floorplan will be identical to the initial floorplan. Since the floorplans are identical, the $\Phi$-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. As indicated in our results, the optimal solutions are far better than what we obtained in most cases. However, in two examples, our new costs are even smaller than the optimal solutions regarding the initial floorplan.

The "pseudo-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 $\Phi$-functions. These results point to the fact that the legalization of the floorplan plays a major role in the overall wirelength of the floorplan.

5. Conclusion
We presented a physically aware framework for compiling high level applications specifications to programmable logic. We showed that an incremental floorplanner is a necessary component for physically aware compiler 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 algorithm transformations has the ability to reduce the wirelength by an average of 12% using a "pseudo-optimal" floorplan. We show that the proposed techniques can still reduce the wirelength of the final design by 6%, while maintaining a legal floorplan with the same area as the initial floorplan.

6. References