Fast Module Mapping and Placement for Datapaths in FPGAs

Timothy J. Callahan*, Philip Chong, André DeHon, and John Wawrzynek
University of California at Berkeley

Abstract

By tailoring a compiler tree-parsing tool for datapath module mapping, we produce good quality results for datapath synthesis in very fast run time. Rather than flattening the design to gates, we preserve the datapath structure; this allows exploitation of specialized datapath features in FPGAs, retains regularity, and also results in a smaller problem size. To further achieve high mapping speed, we formulate the problem as tree covering and solve it efficiently with a linear-time dynamic programming algorithm. In a novel extension to the tree-covering algorithm, we perform module placement simultaneously with the mapping, still in linear time. Integrating placement has the potential to increase the quality of the result since we can optimize total delay including routing delays.

To our knowledge this is the first effort to leverage a grammar-based tree covering tool for datapath module mapping. Further, it is the first work to integrate simultaneous placement with module mapping in a way that preserves linear time complexity.

1 Background

Field programmable gate arrays (FPGAs) consist of configurable logic blocks (CLBs), usually arranged in a 2-dimensional grid, connected by a programmable interconnection network. Each CLB contains some amount of combinational logic and typically contains one or more storage elements. A desired digital circuit can be realized by setting the configuration of the CLBs and the interconnection network.

Computer-aided design (CAD) tools are indispensable in the realization of large circuits using FPGAs. Because FPGAs were originally developed with primarily random logic applications in mind, these tools typically perform their tasks at the level of individual gates or Boolean equations. These CAD tools perform at least three steps: *technology mapping*, which partitions the gates into groups that can be implemented in a single CLB; *placement*, which assigns each group to a specific CLB in the array; and *routing*, which assigns specific routing resources to form the appropriate nets between CLBs. The traditional CAD flow performs these tasks separately in the order described.

As the capacity of FPGAs increases, they are being used more and more in datapath-intensive applications consisting primarily of multibit logical and arithmetic operations. Unfortunately, the traditional gate/CLB-level CAD flow performs poorly with datapath designs. To illustrate, assume we have as input a dataflow graph (DFG) in which the nodes are multibit operators such as addition. There are several problems with the direct approach of first implementing each node with a datapath component, then flattening the datapath components to gates (discarding information about regularity) and feeding the resulting netlist to the gate-level design flow. Because the placement step often utilizes simulated annealing, it is unlikely that an efficient bit-slice layout will be rediscovered (Figure 1a). The generated irregular layout leads to a difficult routing problem, resulting in long compile times and/or poor results. Also, flattening to gates leads to a much larger problem size—there are many times more gates than there are nodes in the DFG. Since many popular CAD algorithms have greater than linear complexity, this can lead to a dramatic increase in compilation time. Finally, once the circuit is flattened to gates, it is usually not possible to rediscover uses of specialized features of the CLB such as fast carry chain circuitry.

A better approach for datapath circuits is to map each node to a prefabricated module (also called a hard macro). This approach can be fast, and it leads to a regular bit-slice layout. However, assuming modules with fixed layouts, no optimization across module boundaries is performed. As pointed out by Koch [10], this approach can lead to the underutilization of FPGA computation resources, especially with coarse-grained architectures (Figure 1b).

Koch's Structured Design Implementation [10] addresses this underutilization problem by performing a module compaction step after module selection and placement, using standard tools to optimize one bit slice of the datapath and then tiling the re-

*Correspondence e-mail address is timothyc@cs.berkeley.edu.
This work is supported in part by DARPA grant DABT63-96-C-0048, ONR grant N00014-92-J-1617, NSF grant CDA 94-01156, and NSERC Canada.

Appearing in Proc. FPGA'98 Monterey, CA

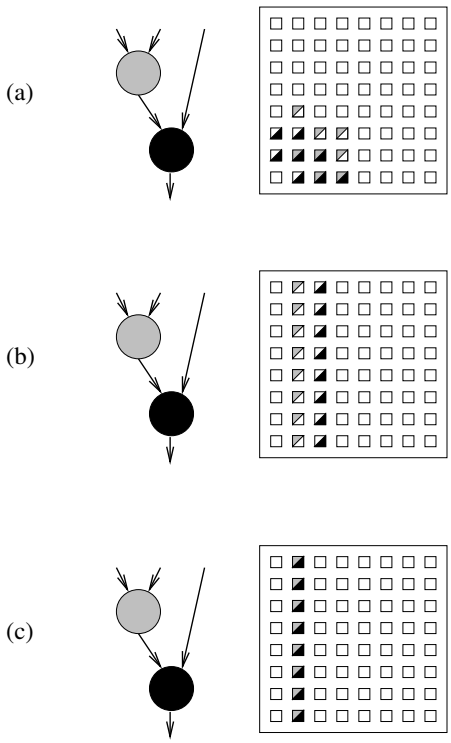


Figure 1: *Different approaches to implementing datapaths. (a) Implementing each operation as basic gates and then feeding to traditional flow. Regularity is lost. (b) Implementing each operation as a hard macro. Computation resources are underutilized. (c) Ideal approach of merging operations while maintaining regularity. Merged module is ready to tile with other modules in bit-slice datapath.*

sults together. One limitation of this approach is that the module compaction step cannot handle specialized CLB features such as a fast carry chain, and thus does not attempt to merge modules that utilize such features. Another limitation is that only physically adjacent modules in the previously determined floorplan are considered for compaction.

In the FAST system [13], groups of nodes in the graph of datapath operations that can be merged are identified; groups are then greedily chosen to be mapped to optimized modules. Because FAST neither performs nor considers placement, when it optimizes for delay it can only consider the computational delay in the CLBs but not the routing delay between CLBs in different modules. With large FPGAs, routing delay forms a significant contribution to overall delay.

The approach presented in this paper and implemented in our datapath mapping tool GAMA has a number of advantages over these approaches. GAMA's main feature is that it is extremely fast; it does not flatten the modules to gates and so has a small problem size; furthermore, it utilizes a mapping algorithm that is linear in the number of nodes in the datapath operation graph. Because it operates directly at the module level, it can intelligently utilize datapath-level features of the FPGA such as fast carry logic, an important advantage as more such features become common. In a novel extension to module mapping, GAMA simultaneously considers linear module placement in a bit-slice datapath in a way that preserves the linear time complexity of the algorithm. Knowing the relative placement of modules allows GAMA to accurately estimate the routing contribution to the overall delay along different paths and thus make better mapping decisions. Finally, this approach is flexible; it has been utilized in mapping to two FPGA architectures: Xilinx 4000 series FPGAs and the Garp chip's reconfigurable array being developed in our group [8]. Although both are based on 4-input lookup tables (4-LUTs), these two arrays present very different mapping problems.

In order to perform module placement in linear time, GAMA examines only a subset of all possible linear orderings of the optimized modules, and so is not guaranteed to find the optimal placement. Therefore it is a good candidate for "low effort" situations, such as getting quick cost estimates, obtaining a good starting point for iterative search techniques, or simply in situations where a high premium is placed on fast synthesis time. Despite the non-optimality, experiments have shown GAMA to achieve results of similar or better quality as those obtained by vendor tools.

2 GAMA Overview

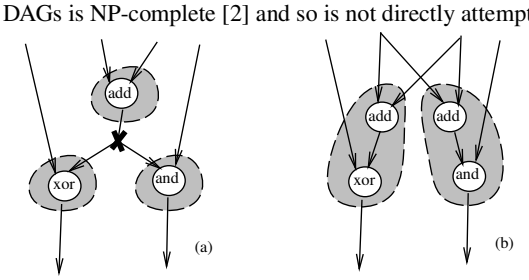
The input to GAMA is a dataflow graph in which each node is a multibit operation, such as an addition or a bit-wise AND. This dataflow graph may in general contain cycles as well as nodes with fanout. GAMA's job is to implement in the CLB array the computation expressed in the dataflow graph. Goals may be minimization of the number of CLBs required, minimization of the critical path delay through the dataflow graph, or minimization of the number of CLBs while meeting a given timing constraint.

GAMA does not attempt to perform sequential optimization (rearranging computing and storage elements to improve opportunities for optimization). When mapping to Xilinx FPGAs, it generates combinational circuits with registers at the primary inputs, primary outputs, and on feedback edges that form cycles. When mapping to the Garp chip, it generates a heavily registered circuit in which no combinational path delay (including both logic and interconnect delays) exceeds one clock cycle. GAMA inserts registers as necessary to break a long interconnect/logic path into shorter paths. This is possible with Garp because GAMA performs placement simultaneously with mapping and can use the Garp array's simplified interconnect delay model to get accurate upper bounds on routing delays. These registers contribute no additional delay in the Garp timing model and make subsequent pipelining easier.

The main operations performed by GAMA are outlined below.

- **Splitting into trees** Since GAMA utilizes a tree-covering algorithm that cannot directly handle cycles or graphs containing nodes with fanout, the input dataflow graph must be split into a forest of trees. Each of these will be fed to the tree-covering algorithm, and the results ultimately connected together. Cycles are broken at appropriate places, usually storage elements demarking iteration boundaries. This produces a directed acyclic graph (DAG), which must be further split into trees. The simplest approach, used in DAGON [9], is to split the DAG at the output of each mul-

tiple fanout node (see (a) below). GAMA goes further and considers duplicating a shared subtree if it is small, since duplication can lead to faster and smaller mappings in some cases (see (b) below). The size threshold for duplicating vs. splitting is a run-time option. Note that even if each tree covering is optimal, the overall solution is not necessarily optimal using this approach. However, optimal covering of DAGs is NP-complete [2] and so is not directly attempted.



– **Tree covering** Because of the hardware resources present in typical CLBs, it is often possible to implement multiple nodes from the DFG together in a *compound module* that is much smaller and/or faster than if they were implemented separately. Typically a compound module consumes a single column of CLBs, but it could be of any size. When such compound modules exist, there may be many different ways that the DFG can be *covered* with module patterns from the library of possible modules. Although in the worst case the number of possible coverings of a tree is exponential in the number of nodes in the tree, we can use dynamic programming to find the best cover in linear time. This algorithm is the heart of GAMA and will be described further in the next section.

  Each tree is passed to the tree-covering algorithm separately. The trees are covered in topological order: a tree that produces a certain value must be covered before a tree that uses that value as an input at one of its leaves. The delay as calculated by the covering of the producing tree is used as the arrival time for the input to the consuming tree.

– **Post-covering optimizations** A pass over the nodes after covering can be used to perform some localized optimizations. Opportunities for these optimizations often arise at boundaries between different trees when they are reconnected after the covering. Also, this phase may consider rearranging the modules after they have been placed by the tree-covering algorithm. This allows layout possibilities that are not considered by the tree-covering algorithm, such as intermingling the modules from different trees. However, there may be parts of the tree-covering placement that cannot be altered by this step because the module mapping relies on that relative placement for correctness.

– **Module generation** Finally, each specified module must actually be generated. A rich variety of functions can be implemented using a column of 4-input LUTs augmented with fast carry chain circuitry, which is the general architecture currently targeted by GAMA. It is therefore not feasible to simply instantiate each module by copying it from a static library, as the necessary library would contain tens of thousands of possible modules. Thus all modules are generated on demand. The generator, given a pattern of DFG nodes, values of constant inputs, datapath width in bits, etc., creates the module. All modules are generated with the same pitch. Currently all modules are generated with the same width in bits as well, although work in progress will generate modules only as wide as necessary.

3 Tree Covering

GAMA uses a linear-time tree-covering algorithm for finding the optimal mapping of the DFG nodes to simple and compound modules. The algorithm and underlying theory was originally developed for code generation in compilers [1], and was first used for the analogous problem of technology binding by Keutzer in DAGON. GAMA utilizes `lburg`, a tool developed for the task of code generation in the `lcc` compiler [7]. We found some modifications to `lburg` necessary as described in Subsection 3.3. This modified version of `lburg` translates a target-specific grammar into the actual tree-covering code that gets compiled into GAMA.

3.1 Basic Algorithm

For review, this subsection describes the basic tree-covering algorithm. The algorithm uses dynamic programming, labeling the nodes in topological order from leaves to the root, combining previously calculated solutions to create new solutions at each node. The algorithm is given in pseudocode in Figure 3. The following definitions, illustrated in Figure 2, are useful in understanding the algorithm:

– The *pattern library* contains the patterns with which the input tree is to be covered. Each pattern is a graph of one or more nodes corresponding to a module that can implement that computation graph. Because of this correspondence, we sometimes use *module* and *pattern* interchangeably.

– A pattern P from the pattern library *matches* at node N in the input tree if, when P is overlayed on the input tree with the root of P aligned with N, the type of each node in P is the same as that of the corresponding node in the input tree. That is, the root node of P matches N, the left child of the root of P (if present) matches the left child of N, etc.

– The *fanin nodes* for a pattern P that matches at node N are those nodes that are immediate predecessors of a node covered by P but are not covered by P themselves. We also define the *fanin tree* rooted at each fanin node.

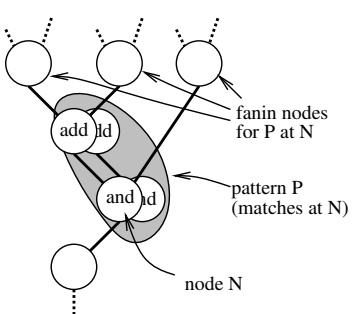


Figure 2: *Definitions for tree covering.*

```
function coverTree(T) {
    foreach node N in T in topological order {
        curBestCost = ∞;
        curBestMatch = null;
        foreach pattern P that matches at N {
            fanins[] = fanin nodes for P at N;
            forall i, faninCosts[i] = fanins[i].bestCost;
            C = costFunction(P, faninCosts[]);
            if (BETTER(C,curBestCost)) {
                curBestCost = C;
                curBestMatch = P;
            }
        }
        N.bestCost = curBestCost;
        N.bestMatch = curBestMatch;
    }
    /* information has been stored on T */
    return;
}


function costFunction(P, faninCosts[]) {
    cost.area = P.area + ∑_i faninCosts[i].area;
    cost.delay =
        max_i (P.inputToRootLatency[i] +
                faninCosts[i].delay);
    return(cost);
}
```

Figure 3: *Basic tree-covering algorithm*

The best cover at node N is calculated as follows. Every pattern P in the library is compared at node N to see if it matches. If so, the cost of the resulting cover is calculated by combining the cost of pattern P with the costs of the best covers at each fanin node of P at N. The way the costs are combined can be unique for each pattern and is specified in the library. In general there will be multiple matches at node N and thus multiple covers. Only the best cover—that with the least cost—is retained, and the rest are discarded. Note that the cost contains separate fields for area and delay; Subsection 3.3 will discuss costs further.

When the root of the tree is finally labeled with its best cover, the best global covering has been found, although the information is distributed throughout the nodes in the tree. The patterns making up the best cover can be found by noting the pattern P recorded as the best match at the root, finding each fanin node for P, and then recursively descending, finding the pattern that is the best match at each fanin node, etc.

Complexity

While conceptually every pattern is checked to see if it matches at every node, the code produced by `lburg` is optimized so that in practice many fewer checks are actually made. Specifically, the covering routine first looks at the type of the node being covered, and then branches to a section of code that only checks for those patterns that have that same node type at their root. Typically the number of patterns that apply at any specific node type is a small fraction of the total. However, this is dependent upon the library itself, and in the worst case, all patterns need to be checked at a node. Thus, assuming that the pattern matching and cost evaluations are all of constant complexity, the execution time of the tree-covering algorithm is *O(NR)*, where *N* is the number of nodes in the graph (after any duplication from the DAG splitting), and *R* is the number of patterns in the library.

Techniques used to reduce the number of patterns in the library will be described in Subsection 3.5. Numerical data regarding execution time will be presented in Section 4.

3.2 Placement by Tree Covering

Since the modules will form a bit-slice datapath layout, a linear ordering of the modules in the datapath must be determined. This placement ordering could be determined in a separate step, following the module mapping. However, if this approach were used, the module mapping step would not have the benefit of knowing the routing delays when making mapping decisions. Furthermore, sometimes placement and especially adjacency impact the consumption of CLB resources, particularly input resources, in ways that affect how much computation can be mapped to a CLB/module. In the absence of any placement information, conservative mappings must be made in order to guarantee that resources are not oversubscribed. This conservative behavior leads to reduced exploitation of CLB resources.

In GAMA, relative module placement in the linear datapath occurs simultaneously with module mapping. The best cover at each node also specifies the linear order of the modules in that cover. In the base case at the leaves of the tree, there is just a single module in the layout. In the inductive step, when creating a cover using a module M matching at a node, a linear layout is constructed by abutting module M with the layouts of the best covers of the fanin trees in some order. The upper bound on the number of fanins for a module bounds the number of permutations that must be considered. This means that the time to calculate the best cover and layout at each node is independent of the size of the tree, and thus the total time to cover the tree when considering placement is still linear in the size of the tree.

In forming the new linear layout, the fanin tree layouts can be placed in any order, but the module covering the current node is always placed at the "rootward" edge of the layout (Figure 4).

The following properties hold for module layouts using the placement policy described:

- The modules within the same subtree are placed contiguously.
- The output of a subtree is always available at the rootward edge of the layout.
- The distance from a module to each of its fanins is a function of only the sizes of the fanin trees and the order in which the fanin trees are placed.

The last property, which follows from the first two, simplifies distance calculations and therefore routing delay calculations.

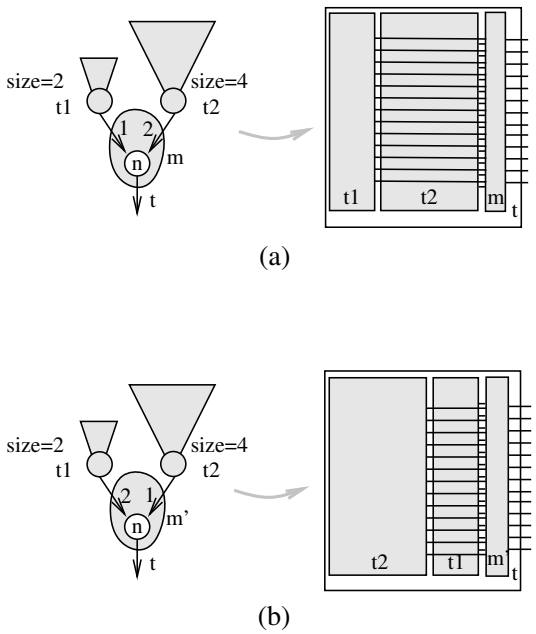


Figure 4: *Two alternative coverings of node n by two modules, m and m', that differ only by their fanin ordering. The different layouts that result are shown to the right. Note the different routing lengths and thus delays.*

For some particular ordering, the distance between a fanin tree F and the pattern P being matched is simply the sum of the sizes of the other fanin trees placed between F and P. This distance is used to estimate the routing delay in a modified cost function for use in evaluating different covers; the pseudocode is in Figure 5. This cost function replaces the costFunction() routine called from the coverTree() algorithm in Figure 3.

This layout strategy makes it straightforward to integrate placement with module mapping. The basic idea is to replace each module in the library with multiple copies having different fanin orderings. For example, a simple two input addition module would now have two versions in the library, one in which the fanin tree producing the first operand is placed first, and another in which the fanin tree producing the second operand is placed first. For a module with n inputs, there could be up to $n!$ different versions, one for each possible fanin ordering. In our current implementations of GAMA, n is typically three and never more than four. See Figure 4 for an example of the two versions for a two-input module.

This method only handles placement within a tree. The ordering of the different trees in the array is determined by a greedy algorithm that tries to place trees along the estimated critical path adjacent to each other. This ordering of trees is done before tree covering so that the tree covering algorithm can take into account which input(s) are available in the adjacent tree.

Considering only layouts that abut rather than intermingle the modules from each fanin layout seems very limiting. In Section 4 we present data regarding improvement resulting from global rearrangement of modules after the initial mapping and placement. The experimental results show that this leads to a slight improvement in result quality.

```
function placementCostFunction(P, faninCosts[]) {
    cost.area = P.area + ∑_i faninCosts[i].area;
    for i {
        dist[i] = sum of sizes of fanin trees
            placed between fanin[i] and P,
            according to P's fanin ordering;
    }
    cost.delay =
        max_i (P.inputToRootLatency[i] +
                routingDelayEstimate(dist[i]) +
                faninCosts[i].delay);
    return(cost);
}
```

Figure 5: *Cost function used for evaluating the cover and layout resulting from matching pattern P.*

3.3 Costs

The dynamic programming algorithm only remembers the "best" covering for each partial match at each node in the input graph. In the instruction selection task for which `lburg` was designed, the only cost that mattered was cycle count, whereas in this hardware mapping problem there are two important metrics: delay (critical path delay from any primary input to that node) and size (number of columns of CLBs utilized). Even if we only cared about critical path delay, we still need to know the sizes of the coverings of subtrees in order to calculate routing delays and make appropriate placement decisions. Thus, the required cost information is actually an aggregate of multiple scalar values. To support this, we extended `lburg` so that the costs are represented by a user-defined structure (where the 'user' here is the person coding GAMA). DAGON also used both area and delay costs. The tree matcher generator tool that it used, Twig [15], already had support for aggregate costs and so did not need modification.

Associated with each pattern in the library is a small C code fragment, supplied by the user, to calculate the cost of the resulting cover if that pattern is matched. The costs of the covers at the fanins to the pattern are supplied for use in the calculation. Typically the code fragment is just a call to a subroutine such as costFunction() in Figure 3 or placementCostFunction() in Figure 5, but it can be customized arbitrarily for each pattern. A pointer to the node itself is also available, so that any information stored on the node can also be used in the cost calculation.

There must be a way of comparing the costs in order to determine which candidate is "best". In the basic `lburg`, the lesser of the two scalar costs is selected. In order to handle costs that are arbitrary structures, `lburg` was extended to allow a user-defined macro `BETTER()` that takes two cost structures as arguments and returns true if and only if the first cost argument is better than the second.

There are currently two versions of `BETTER()` implemented. The area version favors the cost with smaller area, with delay used as a secondary key in the case of identical sizes. The delay version favors the cost with less delay, with size used as

the tie breaker. In its basic mode, GAMA uses the same version of `BETTER()` when covering the entire graph. Unfortunately, while minimizing just area or just delay is straightforward, minimizing both simultaneously or trading off between the two is not.

3.4 Size-Delay Tradeoffs

The tree-covering algorithm is optimal if the goal is minimum area. Picking the smallest solutions to the subproblems will always lead to the smallest solution for the entire tree.

Trying to optimize area and delay simultaneously is not straightforward. In this case it is impossible to pick the single best solution for each subproblem without some global information. For example, if a node is on the critical path, the best cover is probably the fastest one, but if a node is off of the critical path, the best cover is probably the smallest one. But at the time a node is being covered, it is not known whether or not it is on the critical path.

Our approach is to first cover the entire tree to minimize delay. This gives us an estimate of the ASAP (*as soon as possible*) value at each node. The ASAP values in turn can be used to estimate the operation delay at each node. The time constraint at the output of the tree (which must be greater than or equal to the ASAP value at the root node) is then used to calculate the ALAP (*as late as possible*) value at each node. This ALAP value represents a target delay. If it is exceeded at a node, the delay of the overall circuit will not meet the specified timing constraint.

After this estimation, the tree is covered again, as usual, from leaves to root. As each node is reached for covering, it is first covered using the area minimization version of `BETTER()`. If, however, the ASAP value for this cover is found to exceed the previously calculated ALAP goal value (i.e., all of the slack has been used up), the node is covered again, but using the delay minimization version of `BETTER()`. Thus, along a non-critical path, nodes are covered to minimize area until all of the slack is used up; then for the rest of the path, delay must be minimized. This method is not guaranteed to result in the smallest global solution since the slack is absorbed greedily by nodes nearest the leaves; it could be that a bigger area reduction would result if some of the slack were used by a node nearer the root that had a slower but much smaller alternative mapping.

The idea of optimizing area on non-critical paths was also used in Chortle [6], although the details of the implementation are slightly different.

3.5 Large Module Library

As mentioned earlier, there are an extremely large number of different compound modules that can be implemented by a row/column of LUT-based CLBs. GAMA uses two techniques to keep the library size manageable.

First, there are many DFG node types that, while computationally different, are equivalent in regards to mapping—they can be packed in exactly the same way into compound modules.

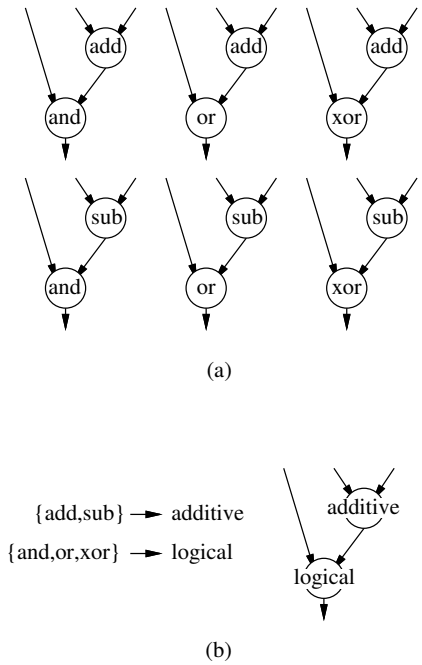


Figure 6: *Example of library size reduction via equivalence. In (a), not exploiting mapping equivalence, there must be a pattern for each combination of opcodes. In (b), the equivalence between addition and subtraction, and between bitwise-logical operations, is exploited so that this single pattern can replace the six patterns above.*

When performing the mapping, there is no need to differentiate those node types (opcodes) that are in the same *mapping equivalence class*. The nodes in the DFG as well as those in the patterns in the module library are therefore named by their mapping equivalence class rather than their opcode (Figure 6). The actual opcode is stored on each DFG node but is not used until module generation.

The second technique to reduce the size of the library is factoring out common subpatterns. This technique is commonplace, but is reemphasized here because it is even more important with the large library of patterns with which GAMA must contend.

3.6 Reconvergent Fanout

The basic tree-covering algorithm does not check for reconvergent fanout since the DAG has been split into trees beforehand, and there is no reconvergence in a tree. The different inputs at the leaves of a tree, however, may be from the same source. If there is a case in which recognizing repeated inputs to a module would be beneficial, the inputs can be explicitly checked in the cost calculation; the pattern will only produce a cover when both inputs are from the same source.

3.7 Limitations

There are several reasons why the mapping and placement solution given by GAMA is not optimal. The initial splitting of the input DAG into trees means that nodes in different trees cannot be combined into a single compound module, which could prevent the optimal solution for the DAG from being found. The restricted module placement similarly limits the number of potential solutions, possibly excluding the best one from consideration. As described earlier, when the optimization goal considers both area and delay, GAMA cannot guarantee finding the optimal solution when only the single best solution is kept at each node. Finally, GAMA can only do as well as is possible given the modules in the library, which is very different than the best solution possible given the FPGA architecture; the solution can only be as good as the module library.

4 Results

4.1 Targeting Xilinx XC4000 Series

We have implemented a simple grammar for mapping DFGs to Xilinx XC4000 series parts [16], for which GAMA shows compilation time and performance benefits over a number of alternative approaches.

Three simple C code fragments were used as benchmarks. Each operation in these fragments was mapped directly to a node in the dataflow graph fed to GAMA. In the case of `if/then/else` statements, the computation along both paths is performed unconditionally; multiplexors controlled by the `if` condition select the correct value—the one from the taken branch—for use later in the computation. Furthermore, values that are obviously Boolean—the results of comparisons and logical combinations thereof—are flagged as such and are moved outside of the datapath section of the circuit. Each benchmark was mapped to an 8-bit and a 32-bit wide datapath.

GAMA output is an XNF file containing mapped LUTs and carry logic components, each annotated with a LOC location constraint, in this way specifying both the merging and placement of modules. All cases were fed through the Xilinx XACT 5.2 tools. The resulting execution times reported are in some sense unfair to GAMA. Even though GAMA has completely specified the partitioning and placement of all components in the XNF file, `ppr` still consumes a significant amount of time attempting to re-perform these tasks. In a real CAD flow utilizing GAMA this time should be greatly reduced or eliminated. Routing time would also be reduced if the Xilinx tools could exploit the fact that all rows contain the same routing problem, although the amount of potential savings here is hard to estimate. However, the results from the CAD flow targeting Garp, presented in the next section, give evidence that substantial reductions are possible.

We compare the following design flows:

- **gama**: Mapping and placement performed by GAMA, optimizing for delay. The result is fed to Xilinx `ppr`, with placer effort set to the minimum.
- **gates**: Start with the file produced by GAMA, but flatten it to gates and then feed it back through Xilinx tools. LUT mapping and use of fast carry chain are lost.
- **hard**: GAMA's grammar is modified to emulate the hard macro approach by removing rules that merge nodes into compound modules. Relative location constraints (RLOCs) are used within each module to keep it rigid, but the XACT placement is free to position each module freely in the array—the modules are not constrained to line up in a bit-slice layout.

Initially we also tried a design flow that used GAMA's mapping to LUTs and carry chain components but placed no LOC or RLOC constraints on any of them. We were surprised to find that the individual carry chain components (each two bits) were not placed in vertical alignment by the placer as we expected. Thus the carry path would go through two bits worth of fast carry logic but then have to travel through the general interconnection network to get to the next two bits. This extra pressure on the general interconnect resources coupled with the irregular placement resulted in a very difficult routing problem, one that the Xilinx tools could not complete.

Results for CLB utilization and estimated delay along the critical path are shown below (Tables 1 and 2).

"caps" and "pp" make heavy use of Xilinx's fast carry logic. The "flat" implementations of these are understandably much larger since they must utilize extra CLBs to perform the carry computation. Also, the flattened carry path goes through 8 or 32 CLBs, drastically increasing the latency.

The "hash" benchmark is primarily shifts and exclusive-ors, although it also has an addition, a subtraction, and a comparison. In this circuit, there is greater opportunity for bit-level optimization. The results from flattening the design to gates demonstrate this with a reduction in the number of occupied CLBs. However, even with this benchmark the delay is significantly increased when flattening is performed.

Benchmark	gama	gates	change	hard	change
caps, 8 bits	48	64	+33%	59	+23%
caps, 32 bits	167	256	+53%	195	+17%
pp, 8 bits	57	81	+42%	67	+18%
pp, 32 bits	213	324	+52%	258	+21%
hash, 8 bits	49	30	−39%	64	+31%
hash, 32 bits	181	144	−20%	204	+13%

Table 1: *CLB Utilization – Xilinx*

Benchmark	gama	gates	change	hard	change
caps, 8 bits	30.2	47.9	+59%	29.9	−1%
caps, 32 bits	50.3	179.3	+256%	50.1	−0%
pp, 8 bits	32.3	49.4	+53%	32.5	+1%
pp, 32 bits	52.1	190.4	+265%	53.6	+3%
hash, 8 bits	37.5	48.6	+30%	35.8	−5%
hash, 32 bits	56.3	185.9	+230%	51.2	−9%

Table 2: *Critical Path Delay (ns) – Xilinx*

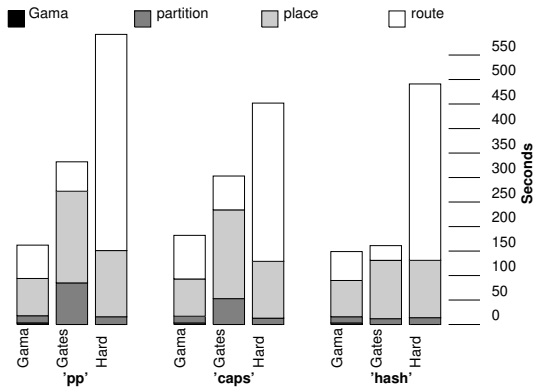


Figure 7: *Tool execution times (seconds) for different design flows, 8 bit wide datapath.*

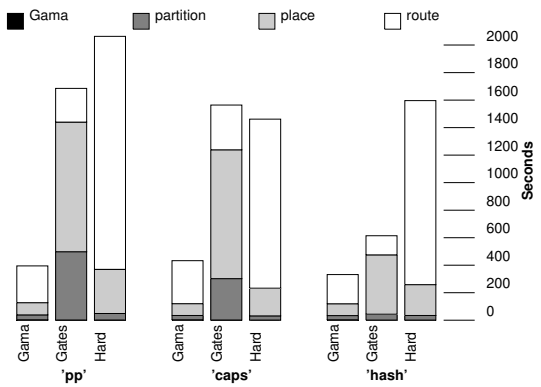


Figure 8: *Tool execution times (seconds) for different design flows, 32 bit wide datapath.*

The times for the Xilinx tools to complete the design are indicated (Figures 7 and 8). Execution times were measured on HP 9000/755 machines, operating at 99 MHz with 128 MB RAM. Execution time for GAMA is less than three seconds and so does not even show up in the graphs.

Note that in all cases, the processing takes more time to complete when the placement or both mapping and placement information from GAMA is ignored, even though no significant performance improvement results from doing so. Flattening to gates gives an average 224% increase in run times for the 32-bit designs, and an execution time increase of 60% on average for the 8-bit designs. The corresponding figures for the hard macro approach are 347% and 214%, respectively. In other words, by using GAMA we expect a 3.24 times increase in compilation speed over using a flattened netlist for the 32-bit designs, and a compilation speedup of 1.60 times for the 8-bit designs. The expected benefits of using GAMA over hard macros are compilation speedups of 4.47 times and 3.14 times for 32-bit and 8-bit designs, respectively.

As expected, the design flow starting from gates took the longest time to partition. The hard macro flow, however, took the most time to route. The flow that used GAMA's mapping and placement was reasonably fast in all tasks. The next section, however, shows that a design flow that from start to finish is optimized for datapath-intense circuits can perform the complete synthesis task orders of magnitude faster.

4.2 Targeting Garp

The majority of our experience with GAMA to this point is targeting the Garp chip being developed in our group [8]. Garp consists of a standard MIPS processor augmented with a reconfigurable coprocessor on the same chip. The Garp array contains a rich variety of computation, routing, and I/O resources especially designed to support datapath computation. It is likely that FPGAs with similar features will become commonplace in the future as their use as reconfigurable coprocessors become more widespread [5]. A commercial embedded processor with a reconfigurable coprocessor, the National Semiconductor NAPA1000 [14], has already been announced.

While the Garp array's specialized resources ultimately enable better datapath performance, they complicate the mapping task. Many different resource interactions must be handled. We have found that GAMA's grammar handles these interactions in a very natural way.

We are using GAMA as part of automatic compilation from ANSI C. This work is based on the SUIF compiler system [3] from Stanford. Kernels from the C program are automatically extracted and fed to GAMA. The remainder of the program is compiled to execute on Garp's MIPS processor. The array is dynamically reconfigured as needed throughout the execution of the program. Since the Garp programmable array can be dynamically reconfigured very rapidly, many different kernels can be mapped to the array by the compiler. This means that GAMA may be called upon to synthesize a large number of configurations. This, combined with the desire that compilation time for the Garp chip be comparable to typical software compilation, places a very tight constraint on allowable execution time for GAMA.

Making efficient use of all of the resources available in the Garp chip presents a challenging mapping problem. Even with grammar reduction strategies described in Subsection 3.5, the grammar is quite large—currently around 1000 patterns. This number would be reduced by a factor of about five if placement determination were not handled by the grammar.

4.2.1 Module Rearrangement Postpass

There are no other tools against which to compare the quality of GAMA's output for the Garp array, so our quantitative results are limited to comparing Garp against itself. Here we investigate the benefit of a post-mapping module rearrangement pass that can move modules globally, intermingling modules from different trees. This pass cannot alter the module mapping. The rearrangement algorithm is based on greedy clustering.

Twenty-five kernels extracted from a C preprocessor program were used as benchmarks. Of the twenty-five, only six benefited from the rearrangement postpass. All of the improvements resulted from eliminating the need to insert a register in the routing between modules in two different trees. If the eliminated register was the only functionality in a row, the entire

row could be eliminated. The postpass caused a reduction in the mean of the delays over all twenty-five kernels from 11.28 cycles to 10.92 cycles, or 3.2%. The reduction in the mean of the areas was from 9.72 rows to 9.56 rows, or 1.6%. The effect of the postpass on GAMA's execution time was negligible.

The fact that freely rearranging modules within a tree led to no improvements is evidence that GAMA's restricted placement within trees most likely does not hurt result quality very much.

4.2.2 Execution Times for C to Garp

Table 3 breaks down the execution time for the complete compilation from C. GAMA accounts for an extremely low fraction of the overall compilation time, approximately one percent. Profiling of GAMA execution shows that more time is spent parsing the ASCII input file than is spent mapping, placing, and generating the modules. These results show that GAMA is extremely fast even with a realistic, complicated grammar. As part of a CAD flow that understands and exploits datapath regularity from start to finish, GAMA makes possible hardware compilation times that compare favorably to those for software compilation.

Task	Time (seconds)	Percentage
`garpcc` (driver)	0.31	17.9%
Kernel identification	0.26	15.0%
Write out kernel DFG	0.11	6.3%
GAMA	**0.02**	**1.2%**
`gatoconfig`	0.10	5.8%
Kernel replace	0.03	1.7%
`gcc -O2`	0.90	52.0%
Total	1.73	100%

Table 3: *Breakdown of compilation time from ANSI C to the Garp architecture. Input routine* `caps.c` *capitalizes a memory-resident, null-terminated ASCII string. Times measured on a 200 MHz Ultrasparc.*

These results show that even though GAMA utilizes a very large grammar for mapping to Garp, it performs its mapping task very quickly. Our experience is that maintenance difficulties become a limit on the size of the grammar long before performance becomes an issue.

5 Related Work

Koch's Structured Design Implementation (SDI) [10, 11] also optimizes across datapath module boundaries while retaining and exploiting the regularity present. It first records the regularity present in a design. Then it basically feeds a single bit slice of the design to the standard random logic tools to perform logic optimization. The resulting slice layout is then replicated as appropriate to achieve the complete optimized datapath (geometry and port constraints are added to make sure the tiling works). Unlike GAMA, this approach attempts to compact only modules that are already adjacent in a previously determined performance-optimized floorplan. This restriction is tolerated because compacting physically non-adjacent modules often disrupted the floorplan so that the increased routing delays more than offset any savings in CLB delays. Furthermore, this approach does not attempt to compact modules that utilize fast carry chain circuitry, because of the difficulty in recognizing uses of this circuitry after general logic optimization has been performed. The use of general logic optimization, place, and route tools also means SDI is stuck with the often slow run times of these tools, although SDI does run much faster than using these tools directly. One benefit that SDI gains by using the general tools is that it can effectively create an unbounded variety of very large compound modules. The variety of modules created by GAMA is limited by library size considerations.

The approach used in the FAST compiler [13] is similar to GAMA in that it directly attempts to find in the dataflow graph "feasible cones" (groups of nodes that can be implemented in a compound module), rather than using standard logic optimization techniques on the gate-level representation. Basically, all feasible cones rooted at all nodes in the DFG are found. For each feasible cone, a "cost benefit" is calculated, which is the difference between the cost of the compound module implementing the nodes in the cone versus the cost of implementing each node separately. The cost can be either area or the delay along the critical path. Then the covering is performed in a greedy fashion, repeatedly selecting from the set of all feasible cones the cone that had the largest cost benefit and does not overlap with any previously selected feasible cone. It is clear that this approach cannot guarantee an optimal covering for any cost metric. It is not clear whether the computational complexity of this approach is linear in the number of nodes in the DFG as is GAMA, or whether it is more expensive. This approach does not integrate floorplanning with the module optimization, and thus its optimization goal can only consider CLB delays, but not the increasingly important routing delays. FAST does use a module delay model that is more sophisticated than GAMA's in that it models delays internal to a module from low bits to high bits or vice versa (sometimes called a ripple delay model).

Finally, another method using dynamic programming to obtain a mapping and linear placement of logic modules is SEMPA, as described by Lou et al. [12]. SEMPA solely addresses area minimization, however, and utilizes a super-linear-time algorithm that considers all possible module orderings. In contrast, GAMA can perform delay optimization and runs in time linear with the number of nodes in the tree, but only considers a restricted subset of module orderings. Moreover, because SEMPA targets standard-cell ASIC technology, its formulation explicitly includes the area consumed by feedthrough wiring as part of the cost for the placement. In the case of FPGAs, the wiring cost does not fit this model. All wires in an FPGA are prefabricated, so there is no cost associated with using additional wires, as long as a sufficient number are available.

6 Future Work

We are in the process of improving the way GAMA handles area and delay. We will implement and experiment with an algorithm similar to that described by Chaudhary and Pedram [4]. The dynamic programming algorithm will be modified to keep not just a single "best" cover at each node, but all non-

inferior points along the area-time tradeoff curve. This will involve further modification of `lburg` to generate code that handles lists of covers rather than single covers.

We are also looking at keeping track of a max-cut cost—that is, the maximum number of inter-module buses at any point in the layout. As was the case with delay, calculation of max-cut routing costs is simplified by the layout used by GAMA.

7 Summary

In this paper we have described a novel approach to FPGA datapath mapping and placement, implemented in GAMA. By casting the problems of module mapping and placement as a unified tree covering problem, GAMA generates compact datapaths as quickly and easily as compilers generate code—in fact, using the same algorithm and tool, `lburg`. The results of running GAMA on benchmark kernels from C programs targeting both Xilinx and our own Garp FPGA architectures show that this is a fast, flexible approach to datapath synthesis.

For 32-bit datapath designs mapped to the Xilinx 4000 architecture, GAMA gives compilation speeds 3.24 times faster than compiling flattened netlists, and 4.47 times faster than using a hard macro approach. Designs generated using GAMA are roughly of the same quality or even better than their flattened or hard macro equivalents in terms of both CLB usage and critical path delay.

Acknowledgements

GAMA originally started as a class project in CS265, Advanced Programming Language Implementation, under the direction of Prof. Susan Graham. Support for the Xilinx XC4000 architecture was added as a class project in CS294-7, Reconfigurable Computing, under the direction of André DeHon.

We benefited greatly from additional feedback and suggestions from Krste Asanović, Andreas Koch, and John Hauser. The comments from the anonymous referees were also extremely valuable.

References

[1] AHO, A., AND GANAPATHI, M. Efficient Tree Pattern Matching: An Aid to Code Generation. In *Conf. Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1985), pp. 334–340.

[2] AHO, A., JOHNSON, S., AND ULLMAN, J. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM 24*, 1 (Jan. 1977), 146–60.

[3] AMARASINGHE, S. P., ANDERSON, J. M., WILSON, C. S., LIAO, S.-W., MURPHY, B. M., FRENCH, R. S., LAM, M. S., AND HALL, M. W. Multiprocessors from a Software Perspective. *IEEE Micro 16*, 3 (June 1996), 52–61. See also http://suif.stanford.edu/.

[4] CHAUDHARY, K., AND PEDRAM, M. A Near Optimal Algorithm for Technology Mapping Minimizing Area Under Delay Constraints. In *Proc. 29th ACM/IEEE Design Automation Conference DAC '92* (June 1992), IEEE Computer Society Press, pp. 492–498.

[5] DEHON, A. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (Cat. No.94TH0611-4)* (1994), IEEE Comput. Soc. Press, pp. 31–9. AN4754544.

[6] FRANCIS, R. *Technology Mapping for Lookup-Table Based Field-Programmable Ga te Arrays*. PhD thesis, University of Toronto, 1992.

[7] FRASER, C., AND HANSON, D. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.

[8] HAUSER, J., AND WAWRZYNEK, J. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1997), K. L. Pocek and J. M. Arnold, Eds.

[9] KEUTZER, K. DAGON: Technology Binding and Local Optimization by DAG Matching. In *Proc. 24th ACM/IEEE Design Automation Conference* (1987), ACM, pp. 341–347.

[10] KOCH, A. Module Compaction in FPGA-based Regular Datapaths. In *Proc. 33rd ACM/IEEE Design Automation Conference* (1996), ACM.

[11] KOCH, A. Structured Design Implementation – A Strategy for Implementing Regular Datapaths on FPGAs. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey CA USA, 1996), ACM, pp. 151–157.

[12] LOU, J., SALEK, A. H., AND PEDRAM, M. An Exact Solution to Simultaneous Technology Mapping and Linear Placement Problem for Trees. In *Proc. International Workshop on Logic Synthesis* (May 1997), pp. 1–4.

[13] NASEER, A., BALAKRISHNAN, M., AND KUMAR, A. An Efficient Technique for Mapping RTL Structures onto FPGAs. In *Proc. 4th International Workshop on Field-Programmable Logic and Applications, FPL '94* (Prague, Czech Republic, Sept. 1994), Springer-Verlag, pp. 99–110.

[14] NATIONAL SEMICONDUCTOR CORPORATION. NAPA1000 Data Sheet, 1996.

[15] TJIANG, S. Twig Reference Manual. Comp. Sci. Tech. Rep. 120, AT&T Bell Laboratories, January 1986.

[16] XILINX. *The Programmable Logic Data Book*. 1994.