Chapter 7

Synthesis of Configurations

Since the Garp array design is novel, there was no existing synthesis flow for it. This was a mixed blessing. While it required the effort for construction of a complete synthesis flow, it allowed the exploration of alternative flows. Particularly, a new design approach became obvious: one that was compatible with a fully spatial approach; one that was compatible with the Garp array's fixed clock; and one that simultaneously optimized both grouping of operations into complex modules as well as relative placement of the modules.

The algorithm used here is specifically suited for a fully spatial implementation. The spatial approach allows merging of contiguous operations from the DFG into optimized compound modules. The approach is adapted from the instruction selection algorithm used for processors with complex instruction sets. In fact the generated datapath can be viewed as a a collection of static instantiations of complex instructions, interconnected by buses.

Motivation for the approach is best introduced by describing the array and examples of some possible modules.

7.1 Garp's reconfigurable array

The structure of the Garp array has many similarities to that of a field programmable gate array (FPGA). 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 whose input-output function depends on the configuration downloaded into it. Each CLB also typically contains one or more storage elements (registers, also called flip-flops) whose connection to the combinational logic also depends on the configuration. A desired combinational or sequential digital circuit can be realized by setting the configuration of the CLBs and the interconnection network.

Like an FPGA, the Garp array is a two-dimensional array of CLBs interconnected by programmable wiring. Like the processor, the array has a fixed global clock synchronizing all array operations. Unlike most uses of FPGAs, the speed of this clock remains constant for an implementation and is not adjusted for a particular array configuration.

It is natural, although not required, that 32-bit integer datapaths in the array be oriented so that operations such as addition span the central CLBs of each row and are stacked, connected to each other by vertical buses (**Figure 7.1**). There are vertical buses of different lengths, spanning 4 rows, 8 rows, 16 rows, etc. up to the full length of the array. The CLBs each contain individual 1-bit registers; the collection of these across the datapath portion of a row is often treated as a composite 32-bit array register. Each row has two such registers: the Z register, usually used to latch the output, and the D register, used to to hold an invariant value or to latch an input from a memory load or from a far module. The extra CLBs on each side of the datapath area are often useful for implementing controllers and computing Boolean data.

Along each row of CLBs, built-in carry chain hardware supports efficient additions, subtractions, and comparisons. Horizontal wiring channels between adjacent rows support shifts. These features together make multiplication by small constants fairly efficient as well. For example, multiplying a 32-bit variable by any 8-bit constant requires at most two rows and two cycles latency. Because of the flexibility of the CLBs, a single row can often implement a compound group of simple operations. For example, the C integer expressions `(a<<10)|(b&c)` and `(a-2*b+c)` can each be implemented in one array row with a latency of one cycle. The term “module” is used to describe such an implementation of one or more operations.

Memory buses provide the path into and out of the reconfigurable array. Garp’s array has

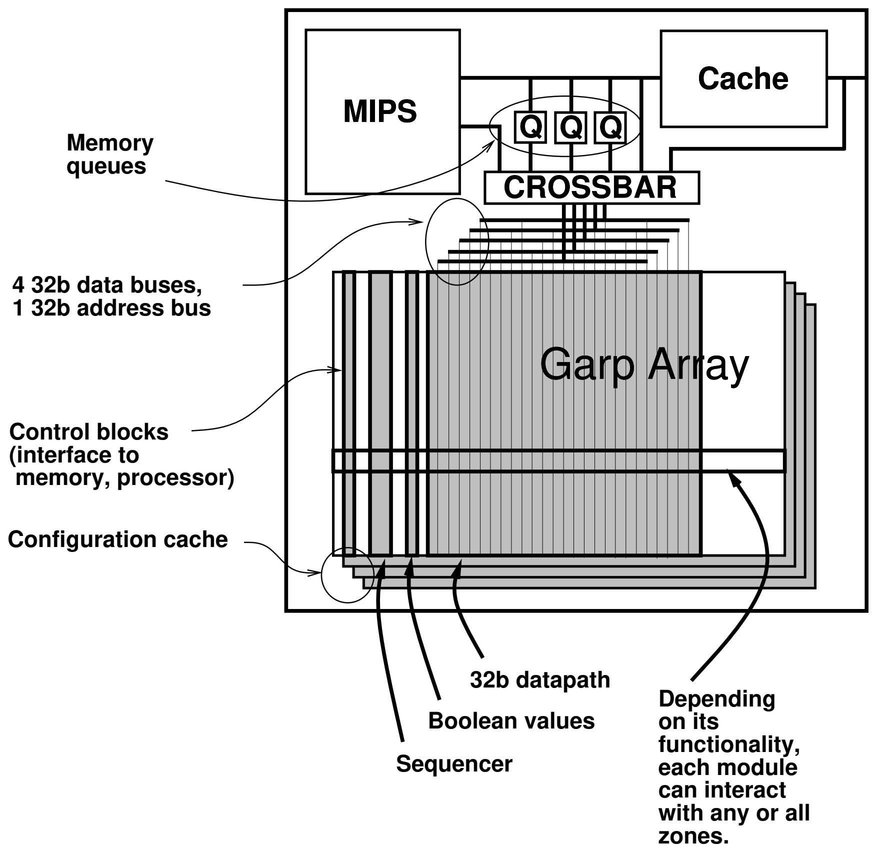


Figure 7.1: Garp array usage conventions.

four 32-bit data buses and one 32-bit address bus. While the array is idle, the processor can use the memory buses to load configurations or to transfer data between processor registers and array registers. While active, the array is master of the memory buses and uses them to access memory.

During execution, the reconfigurable array has access to the same memory system as the main processor, including all caches. To perform a random memory write, one row initiates the write and supplies the address, and another row provides the data. Random reads work similarly: once a row has initiated the read and supplied the address, the data is received into another row after a number of read latency cycles explicitly specified by the configuration. If the memory system cannot respond that quickly, the array stalls automatically, just as a regular processor instruction would interlock on a load that is delayed. At most, one row can initiate a random access each cycle since there is just one address bus. But the array can overlap accesses, initiating a new one every cycle.

With queue memory accesses, the array does not provide an address since the queue itself generates the address sequence after being initialized by the main processor. Typically, the data row initiates each queue access and either provides the next data element that cycle (for a store) or receives the next data element a cycle later (for a load). Since they don't compete for the address bus, multiple queue accesses can transpire in a single cycle.

Control blocks in the array's leftmost column, one per row, provide the row's control interface to the memory or processor. For example, depending on how a control block is configured, asserting one of its inputs might initiate a memory access, sending 32 bits of data from that row as the address. With an alternative configuration, asserting its input will halt array execution, thus acting as a loop exit. Any number of rows can be configured as loop exits, and they can all be evaluated the same cycle—if any is triggered, the array halts.

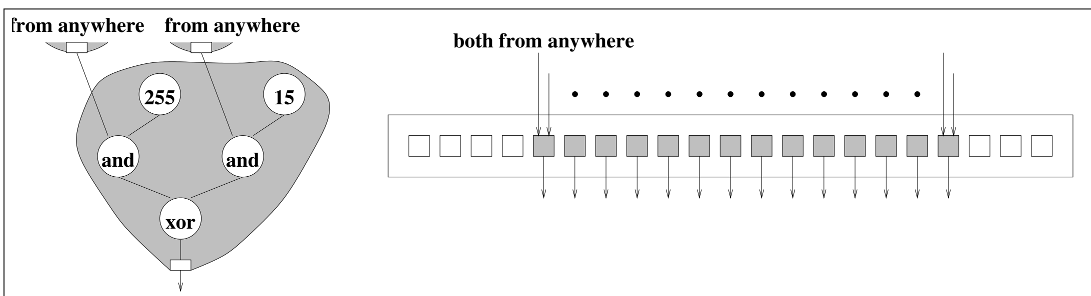
7.2 Examples of modules

This section shows some examples of modules. A module is a minimal unit of placement and scheduling. Two modules can be linked in placement but not scheduling, such as when one must be adjacent to another to facilitate a constant shift. Two modules can also be linked in scheduling but not placement, such as the two portions of a random memory access.

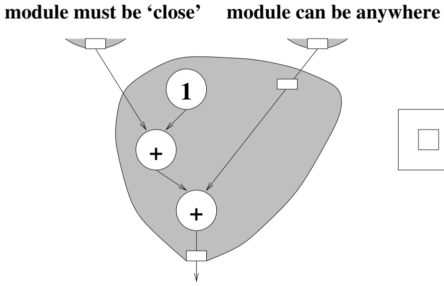
The modules are constructed so that no combinational delay between two registers exceeds one cycle. This is simplified by the fact that the Garp array timing model clearly states which combinations of transfer and computation have delays fitting into one cycle. In the model there are only two classifications of wire length: 'short' and 'long'. Among vertical wires/buses, those spanning 4 or 8 rows are 'short', while longer ones are 'long'. The path from a register, over a long wire, and through a simple logical function to another register takes one cycle. The path from a register, over a short wire, and through a carry chain operation to a register also takes one cycle. However, the path over a long wire followed by a carry chain computation exceeds one cycle and thus must be broken up by inserting an additional register between the long wire and the carry chain calculation.

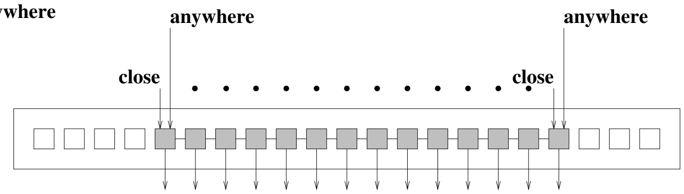
The DFG is the same as that described in Chapter 4 except that random memory accesses are expanded to two nodes: the (possibly predicated) 'memory' node which initiates the access and supplies the address, plus either a 'store' node supplying the data to write or a 'load' node to receive a loaded value. The two halves of an access will become parts of two different modules.

In the diagrams below, the left shows the DFG overlayed with a gray region indicating the region of nodes merged into a module. Small rectangles show register placement. On the right is a simplified depiction of how the module is implemented on the array. The usage conventions of **Figure 7.1** are followed; while this convention seems sensible for constructing a set of tileable modules, it is not the only one that is possible using the Garp array.

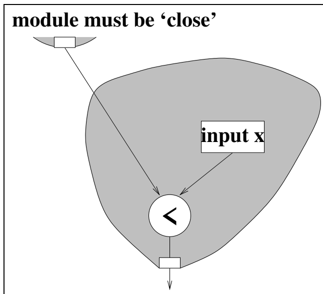


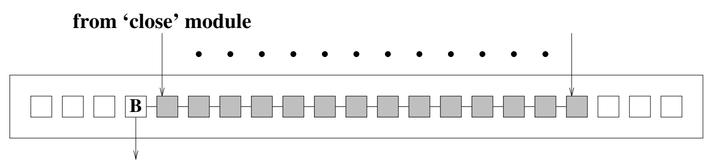
This module computes a bitwise-logical computation on two variable inputs and two constant inputs. It utilizes the CLBs' combinational logic and output Z register.



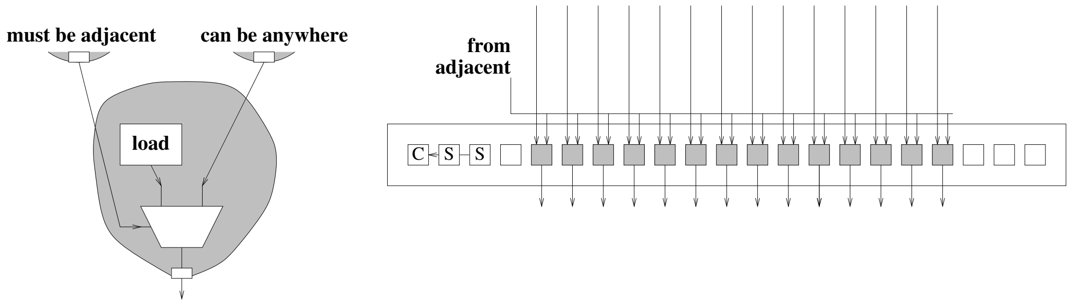


This module sums three values. Because the carry chain is used, inputs must either be buffered locally (latched in a register) or arrive over a short wire. The constant input is hardcoded into the configuration. One other is not buffered and therefore must arrive over a short wire from a close module. The final input is buffered using the row's D register and thus can arrive from a far module; however this path incurs two cycles of latency from the source module's output to this module's output. A module cannot have two buffered inputs since there is just one D register.

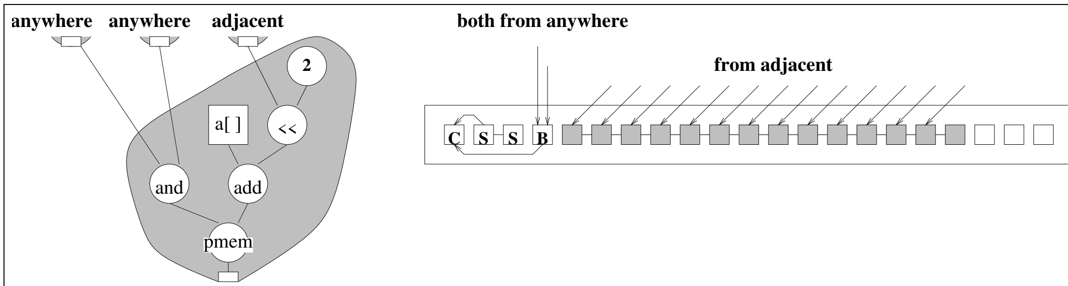




This module performs a less-than function using the row's carry chain. Because the carry chain is used, both inputs must either be registered locally or arrive over a short wire. One input is the kernel-invariant value of variable '`x`'; it is loaded into the row's D register before execution starts. The other input must arrive over a short bus, and so the supplying module must be 'close' The result of the comparison is available in the Boolean column of the array ('B').



This module implements a multiplexor. One of the data inputs is the value from a load; the data is received into the row's D register when the sequencer ('S') triggers the row's control block ('C'). The other data input can arrive over a short or long bus. The module supplying be control input must be immediately above this module; the source module directly drives a global wire between the modules, used by each CLB in the mux to select the correct data input. For a case where the control input is not available from the adjacent module, the control input would need to be latched in the D register of the CLB in the Boolean column and then driven across a global wire, adding an extra cycle of latency.



This module initiates a predicated memory store, providing the address. Some of the predicate and address calculations are performed within the module. The data input must be produced in the adjacent module so that the shared wiring channel can be used to implement the constant shift. The sequencer ('S') ensures the memory access is initiated only during the correct cycle, while the calculated predicate ensures that it is initiated only during the appropriate iterations. When both occur to trigger the control block, the 32 bits in the module's output Z register are driven on the address bus to the memory system. At most one row should drive the address bus any given cycle.

7.3 Existing approaches to FPGA synthesis

Even though no automated design flow existed for the Garp array, it was worth considering existing techniques for FPGAs because of the similarities in their structure.

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 connections between CLBs. The traditional CAD flow performs these tasks separately in the order described.

As the capacity of FPGAs has increased, they have been used more and more in datapath-intensive applications consisting primarily of multibit logical and arithmetic operations. The gate/CLB-level CAD flow, however, performs poorly with datapath designs.

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 circuit to the gate-level design flow. Because the placement step often utilizes a pseudo-random simulated annealing approach, it is unlikely that an efficient bit-slice layout will be rediscovered (**Figure 7.2** a). 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 mapping and placement time. Also, 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. Finally, the Garp array's interface with memory and main processor greatly complicate things: placement constraints would have to be added to ensure that the collective bits of some input or output value for address or data would line up along the appropriate columns of a row. This final consideration alone makes the gate-level approach impractical.

A better approach for datapath circuits is to map each node to a predesigned, preplaced 'hard macro'. This approach can either use vendor's macro generators such as Xilinx's X-BLOX, as did PRISM-II [WAL+93], or build a new generator system such as MARGE [GG97] used by the NAPA-C compiler [GS98]. 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. This approach can lead to the underutilization of FPGA computation resources, especially with coarse-grained architectures (**Figure 7.2** b).

A hybrid synthesis approach might use hard macros for operations needing specific placement for interfacing, then use gate level optimization for regions of other Boolean and arithmetic operations. Yet the optimized regions would still suffer from irregularity as well as difficulty in exploiting special array features.

An academic research project, Koch's Structured Design Implementation [Koc96], used a novel approach to create optimized yet regular datapaths. After performing simple module selection and placement, it performs a module compaction step using standard gate-level tools to optimize one bit slice of the datapath and then tiles the results 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.

7.4 The GAMA approach

The approach implemented in `garpcc`'s datapath mapping tool GAMA[CCDW98] 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 DFG. Because it operates directly at the module level, it can intelligently utilize datapath-level features of the reconfigurable fabric such as fast carry logic. Directly creating optimized, stackable modules also allows easy construction

(a)

(b)

(c)

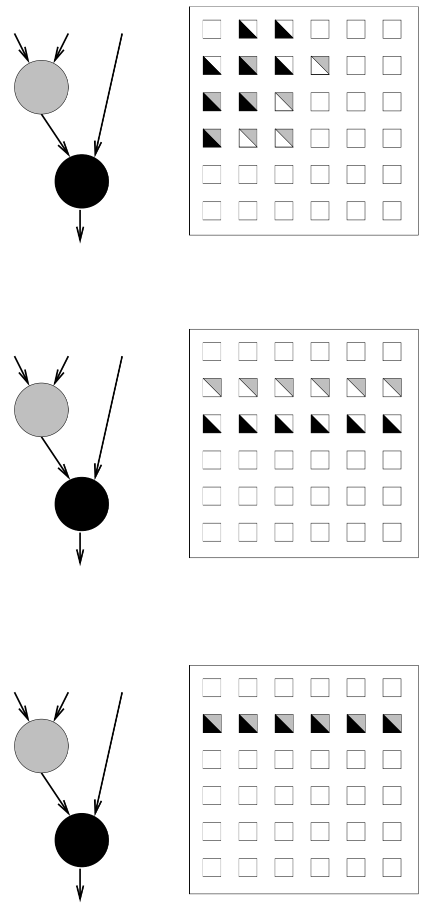


Figure 7.2: Different approaches to implementing datapaths. (a) Implementing each operation as basic gates and then feeding to traditional flow. Regularity is lost. (b) Implementing each operation as a hard macro. Computation resources are underutilized. (c) Ideal approach of merging operations while maintaining regularity. Merged module is ready to tile with other modules in bit-slice datapath.

of a regular bit-slice datapath. In a novel extension to module mapping, GAMA simultaneously considers linear module placement in the 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 different FPGA fabrics: the Garp chip's reconfigurable array as described here, and Xilinx 4000 series FPGAs as described further in [CCDW98]. Although both are based on 4-input lookup tables capable of implementing any 4-input Boolean function, these two arrays present very different mapping problems.

When mapping to the Garp array, GAMA generates a heavily registered circuit so that no register-to-register combinational path delay (including both logic and interconnect delays) exceeds one cycle of the Garp array's fixed-frequency clock. 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. Using these registers contributes no additional setup or hold 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 does not directly handle graphs containing cycles or nodes with fanout, the input DFG must be split into a forest of trees. Each tree is fed to the tree covering algorithm, and the results ultimately reconnected. Cycles are broken by removing loop carried edges. This produces a directed acyclic graph (DAG), which must be further split into trees. The simplest approach is to split the DAG at the output of each multiple fanout node (**Figure 7.3**(a)). GAMA also considers duplicating a shared subtree if it is small, since duplication can lead to faster and smaller mappings in some cases (**Figure 7.3**(b)). The decision to duplicate is based on an ad-hoc heuristic that considers the size of the subtree as well as the opcodes of the nodes around the split point. Note that even if each tree covering is optimal, the overall solution is likely not optimal using this approach. However, optimal covering of DAGs is

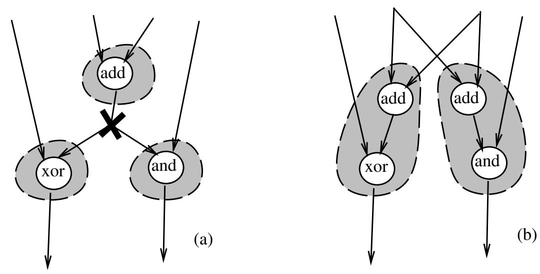


Figure 7.3: DAG splitting alternatives. (a) Force split (**X**) at every point of fanout. (b) Duplicate subtree; can lead to smaller and faster mappings.

NP-complete [AJU77] and so is not directly attempted.

Tree covering As has been shown, 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 each were implemented separately. Typically a compound module consumes a single row 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 the number of possible coverings of a tree can be exponential in the number of nodes in the tree, a dynamic programming approach is used 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.

This phase adds annotations to the DFG indicating module boundaries, inter-module placement constraints, and other information needed for generating the modules correctly.

Post-covering rearrangement and optimizations A pass over the nodes after covering is used to perform some localized optimizations. Opportunities for these optimizations often arise at boundaries between different trees when they are reconnected after the covering. For example, an adder's input assumed to be far may turn out to be close, allowing the removal of the buffering of that input—eliminating a cycle of delay. These optimizations update the annotations on the DFG.

To increase opportunities for such optimizations, this phase first globally rearranges the modules after they have been locally 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, placement constraints from the tree covering phase, such as adjacency of two modules, must be respected during the global rearrangement. The global rearrangement will also consider placing unrelated Boolean and datapath modules in the same row when it is easy to verify they do not interfere with each other.

Besides increasing opportunities for locality-based optimizations, module rearrangement also leads to a more routable datapath.

Module generation After modules and placement have been finalized, each specified module must actually be generated. A rich variety of functions can be implemented using a row of 4-input LUTs augmented with fast carry chain circuitry such as in the Garp array. 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, annotations on the nodes, values of constant inputs, etc., creates the module.

7.5 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 [AG85], and was first used for the analogous problem of technology mapping in Boolean circuits by Keutzer in DAGON [Keu87]. GAMA utilizes `lburg`, a tool developed for the task of code generation in the `lcc` compiler [FH95]. Some modifications to `lburg` were necessary as described in Subsection 7.5.4. This modified version of `lburg` translates a target-specific grammar representing the module library into the actual tree covering subroutine that gets compiled into GAMA.

7.5.1 Basic algorithm

For review, this section 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 7.5**. The following definitions, illustrated in **Figure 7.4**, 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 subgraph. Because of this correspondence, *module* and *pattern* are used 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 *fan-in 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. The *fan-in trees* are the subtrees rooted at the fan-in nodes.

The best cover at node N is calculated as follows. Every pattern in the library is compared at node N to see if it matches. For each matching pattern P, the cost of the resulting cover is calculated by combining the cost of pattern P with the costs of the best covers at each fan-in 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. The ‘cost’ contains separate fields for area and delay; Subsection 7.5.4 will discuss costs further.

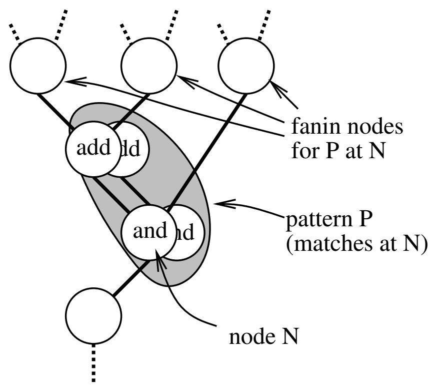


Figure 7.4: Definitions for tree covering.

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 fan-in node for P, and then recursively descending, finding the pattern that is the best match at each fan-in node, etc. This corresponds to the 'emit' phase of instruction selection [FH95], but instead of printing out assembly instructions, it adds annotations to nodes in the tree indicating module boundaries and other information needed to generate modules that interact with their neighbors correctly.

7.5.2 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

```
function coverTree(N) {
    /*** postorder traversal of tree ***/
    foreach node K in children of N {
        status = coverTree(K);
        if (status == ERROR_NO_COVER) return(ERROR_NO_COVER);
    }
    /*** now label N ***/
    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 = P.costFunction(faninCosts[]);
        if (BETTER(C,curBestCost)) {
            curBestCost = C;
            curBestMatch = P;
        }
    }
    if (curBestMatch == null) return(ERROR_NO_COVER);
    N.bestCost = curBestCost;
    N.bestMatch = curBestMatch;
    /*** information has been stored on node N ***/
    return(SUCCESSFUL);
}


/*** sample cost calculation (can be unique for each pattern) ***/
function P.costFunction(faninCosts[]) {
    cost.area = P.area + ∑i faninCosts[i].area;
    cost.delay =
        /*** note that local latency depends on input ***/
        maxi (P.inputToRootLatency[i] + faninCosts[i].delay);
    return(cost);
}
```

Figure 7.5: Basic tree covering algorithm

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 size of the representation of the pattern library, thereby increasing mapping efficiency, will be described in Subsection 7.5.6.

7.5.3 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 at some point. This placement ordering could be determined in a separate step, following the module mapping. But with this approach, the module mapping step would not have the benefit of knowing whether a given input comes from a module that is far, close, or adjacent, and so it is not known when a specific input must be buffered. Thus delay estimates along all paths are inexact when deciding which inputs to favor in the grouping of nodes. Placement considerations also impact the consumption of CLB resources, particularly input buffering resources, in ways that affect how much computation can be mapped to a CLB/module. For example, if the D register is used to buffer an input, then it cannot be used to receive a load. In the absence of placement information, either optimistic or conservative mappings must be made, with a subsequent peephole optimization pass either to correct illegal over-subscribed mappings or to improve under-subscribed mappings. Either approach leads to suboptimal results.

To address these problems arising from separate module mapping and placement phases, GAMA instead performs them simultaneously. As the covering proceeds from bottom up, the cover of the subtree rooted at each node specifies not only the grouping into modules but also the relative placement of those modules. Thus mapping has exact information regarding distances between modules.

Module layout construction follows the usual approach of dynamic programming: each new candidate solution is formed by combining the best solutions of the subproblems. Specifically, a new layout is always formed by placing the new module at the bottom, then placing the layouts for the fan-in trees above it in some order. The only degree of freedom is the ordering of the fan-in tree layouts. Note, however, that unlike many dynamic programming algorithms that combine optimal sub-solutions in a way that preserves optimality, this layout approach combines sub-solutions that in no way guarantees the optimal solution.

The resulting layout can be called a "postfix placement" since it corresponds to some postfix traversal of the module tree, with the freedom of visiting children in any order.

The key idea for seamlessly integrating placement determination into tree covering is that each module in the library is replaced with multiple copies, each specifying a different relative placment of its fan-in trees (**Figure 7.6**). With this GAMA extension, the best cover of a tree also specifies the linear order of all modules in the cover. That is because each (sub)tree root specifies the relative placement of its constituent subtrees, and so on recursively to the leaves. When GAMA is trying each potential module pattern, it is simultaneously evaluating not just alternative node groupings but also alternative layouts.

When a pattern and its implied layout is evaluated, exact inter-module distances are deduced by combining the following information: (i) the root module's specified fan-in ordering (ii) the fact that the output of every subtree layout is available at its bottom edge, and (iii) the known area of each subtree layout. The distance from a non-adjacent fan-in is simply the sum of the rows in intervening subtree layouts.

Because exact inter-module distances are known, there is no need for either conservative or optimistic guessing. Both optimistic and conservative versions of modules are in the library. Optimistic modules need constraints on the distance from inputs—specifically when an input needs to be adjacent or close. If an optimistic module meets all input constraints, it can be used, and will be used when it is cheapest; otherwise a more conservative module is used. An 'adjacent' constraint need not be explicitly checked since it follows immediately from which input is placed closest to the root module. Similarly, a 'close' constraint need not be checked when the fan-in is the one placed closest to the root module. For a non-adjacent fan-in, a 'close' constraint requires a check of the actual distance; if too far, the module/pattern is not a match and is indicated as so by returning infinite cost. The pseudocode for placement-sensitive cost evaluation is shown in **Figure 7.7**.

After the final best cover has been found, the 'adjacent' and 'close' constraints are added

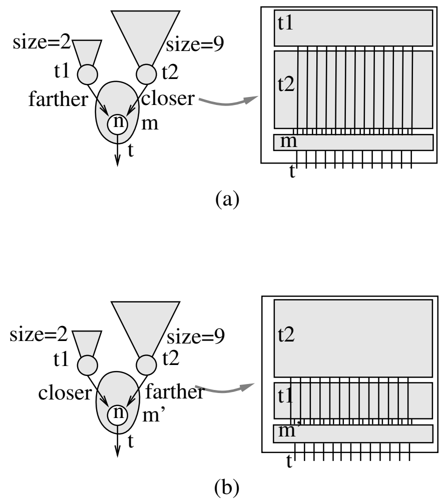


Figure 7.6: Two alternative coverings of node n by two modules, m and m′, that differ only by their fan-in ordering. The different layouts that result are shown to the right. Note that routing lengths and thus delays are different.

to the inter-module edges. This is true even for constraints that did not have to be checked explicitly during covering. A simple example is given in **Figure 7.8**. It shows four modules identical in grouping but differing in fan-in ordering and/or constraints.

Obviously the layout policy used during covering is very restrictive. Likely the real best layout will intermingle the modules from different subtrees. This limitation is compensated to a large degree by global rearrangement of modules after the initial covering-based mapping and placement. In general, the tree covering algorithm for grouping and relative placement mainly helps the critical path. Subsequent global rearrangement and peephole optimization steps then help tidy up areas off of the critical path, improving delay, routing locality, and sometimes area.

7.5.4 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

```
function placementCostFunction(P, fanInCosts[]) {
      cost.area = P.area + ∑i fanInCosts[i].area;
      for i {
            dist[i] = sum of sizes of fan-in trees
                        placed between fan-in[i] and P,
                        according to P's fan-in ordering;
            /*** throw out matches not obeying placement restrictions ***/
            if (P.mustBeClose[i] and dist[i] > closeThreshold) return(∞);
      }
      /*** P.inputToRootLatency[] factors in any buffering delays ***/
      cost.delay = maxi (P.inputToRootLatency[i] + fanInCosts[i].delay);
      return(cost);
}
```

Figure 7.7: Placement-aware cost function used for evaluating the cover and layout resulting from matching pattern P.

there are two important metrics: delay (critical path delay from any primary input to that node) and size (number of rows of CLBs utilized). Even when optimizing just critical path delay, the sizes of the coverings of subtrees must be known 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, `lburg` was extended 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 [Tji86], 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 fan-ins 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 7.5** or placementCostFunction() in **Figure 7.7**, 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 (or nearby nodes) can also be used in the cost calculation. For example, the cost calculation for a constant multiplication finds the actual value of

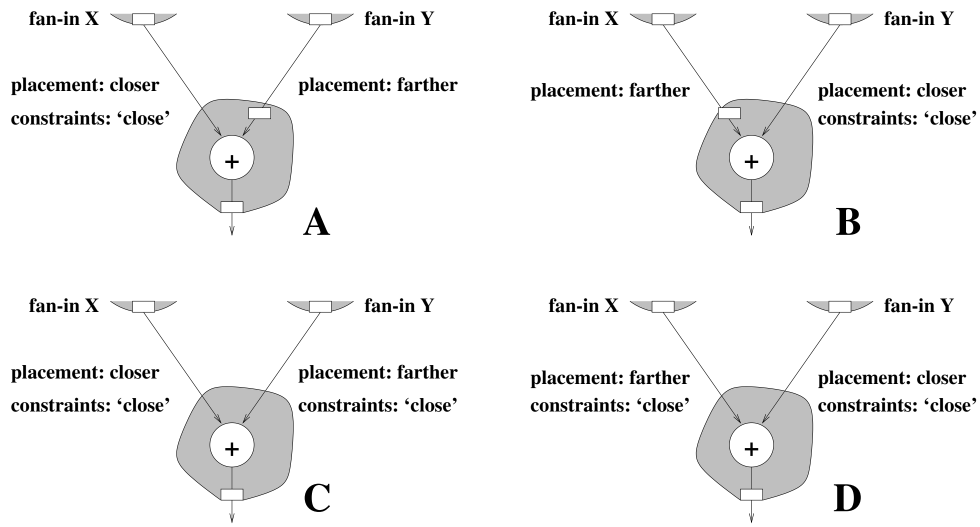


Figure 7.8: Example of modules with same grouping but different in placement ordering and constraints. Because the addition uses the carry chain, inputs must be 'close' or buffered locally. Modules A and B are the conservative versions and will match in any situation, although an extra cycle of latency is added to the path from the far fan-in. Modules C and D are more aggressive, assuming both inputs are close. Consider one specific case where an addition node has fan-in X with best cover [delay:6, area:10] and fan-in Y with best cover [delay:6, area:5]. Modules A and B will both match with delay 8 since one of the inputs in each case has a total latency of 2. Module C will not match because with fan-in X adjacent, fan-in Y is far and thus does not meet the required 'close' constraint. Module D does meet all constraints and results in a delay of 7, so among these four modules, it would be chosen. However in a real case, it is likely that a module that groups more nodes would be the best match at the addition node unless the addition node is a leaf of the tree.

the multiplication node's constant input and uses it to determine how many rows will be required to implement the multiplier.

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. While minimizing just area or just delay is straightforward, minimizing both simultaneously or trading off between the two is not.

7.5.5 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.

The approach used by GAMA is to first cover the entire tree to minimize delay. This gives an estimate of the ASAP (*as soon as possible*) delay 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 once again, as usual, from leaves to root. As each node is reached for covering, the first attempt at calculating a cover uses 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 greater 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 [Fra92], although the details of the implementation are slightly different.

7.5.6 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. This number is further increased by the introduction of placement variations for each module. GAMA uses two techniques to keep the size of the representation of the library 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. 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 7.9**). 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 representation 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.

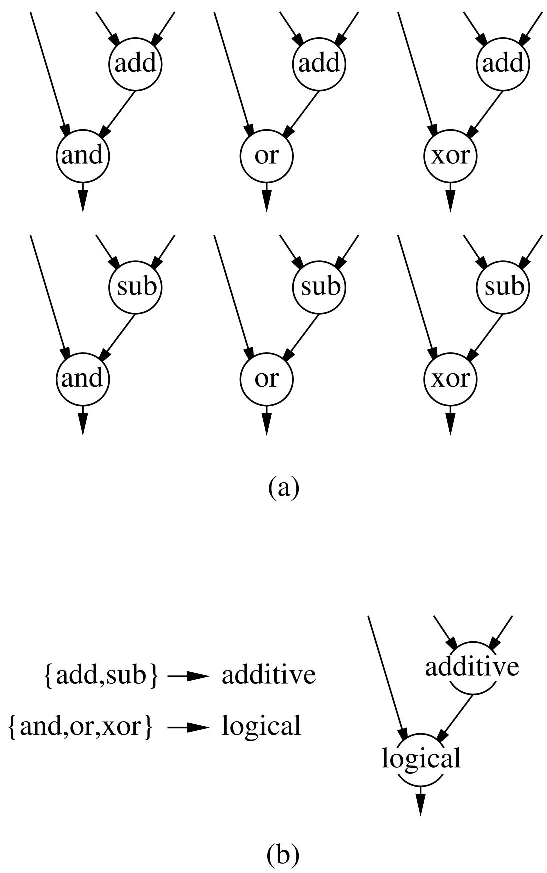


Figure 7.9: Example of library representation size reduction via equivalence. In (a), not exploiting mapping equivalence, there must be a pattern for each combination of opcodes. In (b), the equivalences between addition and subtraction and among bitwise-logical operations are exploited so that this single pattern can replace the six patterns above.

7.6 Scheduling and execution (non-pipelined)

Some modules such as those involved in Exits and memory accesses must be triggered only during the appropriate cycle of each iteration, since execution before their inputs are valid would have undesired consequences. Also, the Hold modules used with non-pipelined execution need a trigger at the end of each iteration so that they load the loop-carried values for use during the next iteration. These triggers are provided by a simple *sequencer* that is synthesized within the configuration. Its duty is to keep track of the current cycle within the iteration and provide triggers to modules as needed. The implementation of the sequencer is trivial since every iteration has the same fixed schedule of SL cycles where SL is the schedule length. Conceptually it is simply a Boolean shift register of SL stages, hooked back on itself, passing around a single '1' bit. The actual implementation is slightly more complicated; it may in fact have side chains forking off of the main chain to allow for more localized routing.

Modules such as simple adders need no trigger from the sequencer as they can execute constantly with no ill side effects. However, it is useful to think of these non-triggered modules as also being scheduled for a specific cycle: the cycle of the iteration that their output first becomes valid.

In most cases a module is "scheduled" exactly at its ASAP (as soon as possible) value. This is equal to the greatest total latency along any path (composed of any combination of distance-0 data, precedence, and/or liveness edges) from a Hold module to that module's output. The one exception occurs when there are multiple non-queue memory accesses. Since each such access requires exclusive use of the address bus, they cannot be scheduled for the same cycle. If two accesses have the same ASAP value, the one with the least slack is given preference and gets scheduled in that slot, while the other is delayed to the next available cycle; consumers of the delayed access are also then delayed from their ASAP value. Non-queue memory accesses present the only case where per-cycle resource constraints must be considered.

With non-pipelined execution, a module's output is guaranteed to be the correct value for that iteration from its scheduled cycle until the end of the iteration. Outside of that range, the module's output must be assumed to be incorrect for either the prior or current iteration, since in general the module's inputs may be a combination of values from the prior and current iteration. This hints at a key idea for pipelined execution (Chapter 8), with which a module's inputs always arrive synchronized for a particular iteration.

7.7 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.

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. This could be improved by keeping instead the set of solutions forming the best curve of tradeoffs between area and delay (discarding any clearly subtoptimal solutions). When evaluating a module cover, all combinations from the fan-in covers' sets would have to be evaluated to form the new best curve of solutions.

The restricted module placement forcing a module subtree's output to always be at the bottom edge of its layout similarly limits the number of potential solutions considered, likely excluding the best one from consideration. This could also be improved by keeping multiple solutions at each node, with different solutions having the output at different locations within the layout. This could allow bidirectional assembly of datapaths. Perhaps all variations could be kept—with the output in row 1, row 2, etc. upt to the last row—or just a subset—first, last, and middle—could be kept.

Finally, since GAMA can only choose from modules in the library, the solution can only be as good as the module library. This is considered in the next subsection.

7.8 Comparing module library to instruction set

The final datapath can be viewed as a group of static, complex instructions interconnected by buses. This inspired the derivation of the module mapping algorithm from the instruction selection algorithm. These both highlight the similarity between designing the module library and designing an instruction set. This section compares and contrasts in factors in designing "set of instructions in instruction set architecture (ISA)" with "set of modules the generator knows how to build".

Both module set and instruction set design hope to identify groups of operations that can be implemented as an optimized unit to give better performance. However, differences in how the groups are implemented make a great difference regarding when it makes sense to add a new grouping.

Instruction set designers must weigh negative aspects of each potential new addition. Every new instruction impacts the microprocessor's implementation even when it is not used. The control/decode logic and possibly the datapath must be modified to support each new instruction. This in turn affects design complexity and perhaps performance.

The module library, in contrast, only comes into play during compile time, and thus adding a new module has no negative impact on run-time performance. Essentially the module library attempts to capture a good designer's expertise regarding how datapath operations can be packed into a small module, and more knowledge is better. The main drawback to a large module library instead is increased complexity, making it more difficult to maintain the library and associated module generator. Also, the speed of GAMA in performing the tree covering is also impacted by increased size of the module library; however, GAMA mapping is so fast compared to other compilation phases that it is very unlikely that this consideration would become significant.

With instruction set design, the total number of different instructions is bounded by their encoding into typically 32-bit instructions including immediate data values and register. Thus inclusion of one opcode may preclude the inclusion of another. There is no such encoding limitation with the module library.

Another related difference is that module library design does not have the legacy problem. Superseded modules can be removed from the library at any time with no ill effects; datapaths synthesized using the old module library still execute perfectly. With instruction set design, in contrast, compatibility requirements mean that an ill-chosen instruction must be supported over lifetime of that and any derived ISAs, complicating future generations of implementations.