Performance Optimization Using Template Mapping for Datapath-Intensive High-Level Synthesis

Miguel R. Corazao, Marwan A. Khalaf, *Member, IEEE*, Lisa M. Guerra, *Student Member, IEEE*, Miodrag Potkonjak, and Jan M. Rabaey, *Fellow, IEEE*

***Abstract*— This paper introduces a new approach to performance-driven template mapping for high-level synthesis. Template mapping, the process of mapping high-level algorithmic descriptions to specialized hardware libraries or instruction sets, involves template matching, template selection, and clock selection. Efficient algorithms for each are presented, and novel issues such as partial matching are addressed. The paper focuses on datapath-intensive ASIC design, though the concepts are also highly applicable to compiler development. Experimental results on examples from real applications show significant improvements in throughput with limited area overhead.**

I. Introduction

A. Performance Optimization Using Template Mapping

As a result of the increase in demand for high-speed integrated circuits, there is an increasing need for synthesis tools targeted toward performance optimization. Techniques for throughput improvement can also be used to achieve other goals. For example, in an ASIC design with fixed throughput constraints, optimizing to reduce the critical path can enable voltage scaling to reduce power [1].

One task in the high-level synthesis and compilation processes with great potential for improving performance is template mapping. In template mapping, at the behavioral level, groups of primitive operations are replaced with more complex and powerful operations. At the architectural level, complex units are used in place of primitive ones. The more complex execution units can represent special purpose hardware, or they may represent chained units. Specialized hardware units are often designed to implement common operations (e.g., multiply-add) and are often optimized for low area, power, or delay. Chaining [2], the removal of intermediate registers between primitive hardware units, can improve the total delay of the units combined.

Manuscript received April 8, 1994; revised December 6, 1995. This work was supported by ARPA, and by grants from ONR and AT&T. This research was performed while M. Potkonjak was with the Computer and Communication Research Laboratories, NEC USA, Princeton, NJ, and all other authors were with the Electrical Engineering and Computer Sciences Department, University of California, Berkeley. This paper was recommended by Associate Editor K. Keutzer.

M. R. Corazao is with the Intel Corporation, Santa Clara, CA 95052 USA.

M. A. Khalaf is with the Altera Corporation, San Jose, CA 95134 USA.

L. M. Guerra and J. M. Rabaey are with the Electrical Engineering and Computer Sciences Department, University of California, Berkeley, CA 94720 USA.

M. Potkonjak is with the Computer Science Department, University of California, Los Angeles, CA 90095 USA.

Publisher Item Identifier S 0278-0070(96)05418-8.

Interestingly, template mapping in high-level synthesis has received little attention while the parallel task at the logic level, technology mapping, was recognized [3] and thoroughly studied early in the development of logic synthesis. One reason may be that while many of the same concepts apply, significant differences, such as the use of hardware sharing in high level synthesis, dramatically complicate the relationship between design metrics such as throughput and area. In logic synthesis, this relationship is significantly more direct.

One of the main intentions of this paper is to give impetus to the exploration of template mapping's potential for performance improvement and to demonstrate its importance in the synthesis process. The paper presents an approach for performance-driven template mapping. Algorithms for detecting the possible matches (template matching) and the selection of a subset of these to cover the graph are presented. While template matching on general cyclic graphs is itself known to be difficult [4], the problem is further complicated when partial matching (Section II-A-3) is considered. The algorithms presented overcome the complexity of the problem by using a novel representation of the solution space which does not require enumerating graph isomorphisms, and yet represents the complete space. Using this representation, high quality graph transformations can be obtained in an efficient manner. In addition to matching and selection, the algorithms implement a method for selecting the optimal clock frequency. These algorithms are sufficiently modular so that they can be used as platforms for investigating other optimization targets (e.g., area and power).

B. Motivational Examples

Fig. 1(a) shows a simple computational structure represented by a control/data flowgraph (CDFG) [5]. Associated with each edge in the CDFG is a delay of 5 ns for accessing the register implied by the edge. Originally, the operations in the CDFG are each implemented by using primitive adder units. The optimal clock period (in terms of speed) for this implementation is 45 ns. For this clock period, the additions complete execution in one clock cycle (5 ns to access register operands +40 ns to execute).

The template shown represents a 3-input adder (or two dual-input adders chained together). By replacing the primitive additions in the CDFG with the template and adjusting the clock period, the total execution delay is improved from 180 ns to 110 ns. It is important to note that had the

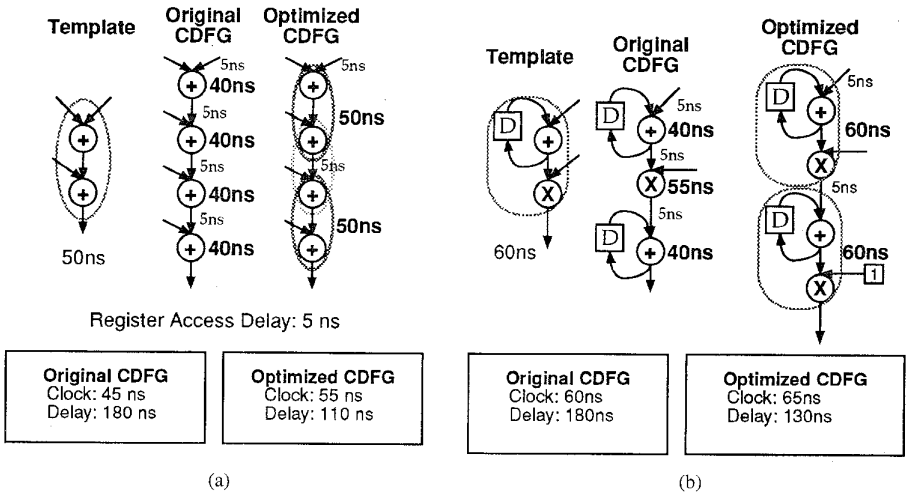

Fig. 1. Motivational examples. (a) Delay improves by 70 ns, but only if the clock period is selected properly. (b) Delay is improved and the CDFG is simplified (to contain only one type of operation).

clock period remained unchanged, the delay would not have improved (since each of the 3-input adders would require two clock cycles to execute). This fact demonstrates the importance of proper clock selection. Also note the fact that before optimization, only a single dual-input adder is required to implement the complete graph (since all additions occur sequentially). After optimization, a 3-input adder must be allocated. Since it can be assumed that the 3-input adder requires more area than the 2-input adder, it is likely that the total active area (execution units, registers, multiplexors) will increase. This demonstrates the important tradeoff between area and performance inherent in template mapping.

Fig. 1(b) shows a slightly more complex example. The template represents an accumulator with a multiplier at its output. At first glance, it seems that the template matches the graph in only one place. However, if the input of the multiplier in the template is set to one, the output of the multiplier is equivalent to the adder output. This means that the multiplier can effectively be treated as a bus. In this way, the template partially matches the second half of the graph. While this second match does not affect the total throughput, the CDFG is simplified so that it contains only one operation, the operation represented by the template. In this way, area can be saved by allowing the same hardware to be reused.

While these examples are relatively simple, they demonstrate the potential of template mapping and some of its complexities.

C. Previous and Ongoing Research

Although template and pattern matching are addressed in many branches of science and numerous engineering fields [6]–[8], this section restricts its attention to compiler, computer-aided design (CAD), and high-level synthesis work.

While the first attempts to apply template matching to compilers were done in the early Seventies [9], [10], the major impetus for the widespread use of pattern matching was a result of a code generation scheme suggested by Hoffman and McDonnel [11]. They proposed that pattern matching could be implemented efficiently using tree-pattern matching by extending the Aho and Corasick multiple-keyword algorithm for template matching [12]. The importance of template matching in compilers has become most apparent recently with the rapid growth of the digital signal processor (DSP) industry. This market is dominated by architectures designed to match common features of DSP algorithms. A major failing of today's DSP compilers is their inability to explore these special instructions effectively [13], [14]. For the same reason, pattern matching is also an essential part of any retargetable compiler for application specific signal processors (ASSP's).

Pattern matching has also become an integral part of logic synthesis. IBM pioneered some of the first work in technology mapping for logic synthesis using a heuristic application of local transformations [15]. SOCRATES [16] addressed the same problem using a rule-based approach. More recently, based on the suggestion in [17] that compiler techniques can be transferred to CAD, Keutzer [3], [18] presented an efficient technique using the tree-processing language Twig [19] (which is based on the tree-pattern matching method in [19], [20]). Currently, several algorithms for logic synthesis technology mapping based on the fast tree dynamic programming techniques are widely used [3], [21], [22]. Technology mapping techniques using binary decision diagrams and Boolean algebra laws have also been proposed [23]. With the proliferation of programmable gate arrays, several mapping algorithms to both table look-up [24], [25] and multiplexor architectures [26] have been developed.

Pattern matching has been recently recognized as an important and powerful method for area optimization at higher levels of abstraction. The IMEC high level synthesis group was one of the first to attempt pattern matching in high-level synthesis by addressing issues of application specific functional units [27], [28] using integer linear programming technology [29]. More recently, Landwehr *et al.* used a 0/1 integer linear

programming model to solve the scheduling, allocation, and binding problems in high level synthesis taking into account multifunctional units and chained operations [30]. Rao and Kurdahi proposed the use of template matching within a framework of regularity extraction for addressing partitioning, scheduling, and allocation in high level synthesis [31], [32]. Chu and Rabaey used a simulated annealing based template matching method for optimizing area and clock period [33]. The functionality recognition problem in component selection (a gray area between high-level synthesis and logic synthesis) was addressed using pattern matching by Rundensteiner *et al.* [34], Ang and Dutt [35], Praet *et al.* [36], and Lanneer *et al.* [37].

The research presented in this paper not only addresses a novel goal (performance optimization), but also provides novel techniques for removing some of the limitations of the previously mentioned CAD and compiler approaches. These techniques include incomplete pattern matching and the matching of cyclic graphs of an arbitrarily general structure. In addition, methodologies are presented for selecting clock periods optimally and efficiently. Since the approaches presented in the following sections separate the template matching and template selection tasks, the proposed algorithms can be easily extended for use in many compiler and CAD domains.

II. Synthesis Algorithms

Before presenting the synthesis algorithms, a few terms should be clarified. **Operators** represent either primitive or complex functions, the later being formed by combinations of the former, and described by a graph called a **template**. A **hardware unit** is the physical device used to execute a particular function. As an example, a carry-look-ahead or carry-select adder can implement addition. Similarly, a carry-save multiplier followed by a carry-select adder implements the complex function of multiply-add. A particular function can clearly be implemented by any one of several different hardware units. For the algorithms described, however, it is assumed that for each function, or operator (whether primitive or complex), one specific hardware unit has been selected for its realization.

The template mapping algorithm can be divided into three major components: template matching, template selection, and clock selection, as shown in Fig. 2. Template matching, the central problem in template mapping, refers to the matching of templates representing complex operators to a CDFG, consisting of only primitive operators. In general, the templates are obtained by one of three methods: 1) they are given in the problem (as would be the case for compilers), 2) they are generated automatically by some algorithm, 3) they are generated manually (as is the case for this paper). Since the interpretation of the templates is application-specific, template generation is not discussed here and, instead, it is assumed that a template library is a given to this problem (indeed, for compiler development, template generation is a nonissue).

Template selection is the actual replacement of groups of nodes in the graph with templates from the library according to matches generated by template matching. Templates are selected so as to minimize the number of clock cycles in the critical path.

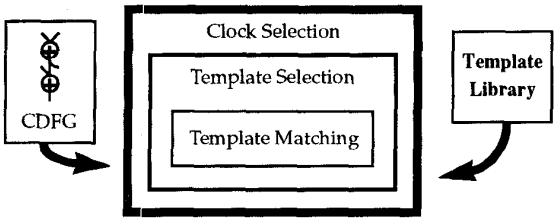

Fig. 2. Structure chart for the template mapping algorithms.

Clock selection refers to the process of selecting a clock period so that the real-time delay can be minimized. The clock selection algorithm iteratively executes the template selection algorithm so as to find an optimal solution (i.e., if we could assume the template selection algorithm were exact, then we would have the optimum clock selection solution).

The following sections describe each of these algorithms.

A. Template Matching

The fundamental difficulty in template matching lies in the fact that the number of template matches can be quite large and prohibitively expensive to enumerate. The proposed template matching algorithm generates a compact representation of the set of potential matches. It handles general graphs with cycles and has polynomial time complexity.

1) Terminology: A few terms must be defined before proceeding. Nodes in the CDFG are referred to as **graph nodes**, while nodes in the templates are referred to as **template nodes.** Inputs and outputs from a template are referred to as **template ports.**

A graph node x and a template node y in template t are said to have a **node match** between them if some group of nodes in the CDFG including x can be replaced by t such that y replaces x. This node match can be represented by the 3-tuple (x, y, t). The set of node matches for a given graph node is referred to as its **match list.** A particular matching between some set of graph nodes and an entire template is referred to as a **template match.**

A node match m is said to have a (mutual) dependency on a node match n if the graph node x_m matched by m is either a parent or a child of the graph node x_n matched by n. If x_n is a child of x_m, then n is said to be a **child dependent** of m. Similarly, if x_n is a parent of x_m, the n is a **parent dependent** of m. A **dependency list** is the list of dependencies of a node match.

2) Template Matching by Relaxation: The output of the proposed template matching algorithm is a match list for each graph node. The total number of node matches for a CDFG is polynomially bounded. In addition, enumerating template matches often requires much more space than simply enumerating node matches. This is a result of the fact that more than one template match may contain the same node match (e.g., in Fig. 3, there are three template matches with

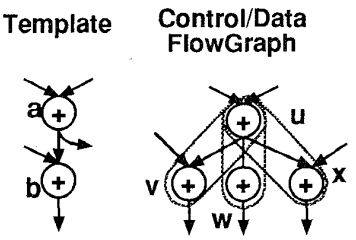

Fig. 3. Template and CDFG with template matches.

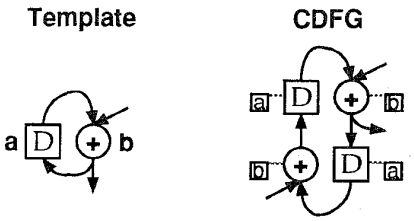

Fig. 5. Degenerate case. The relaxation approach alone would incorrectly allow these invalid matches.

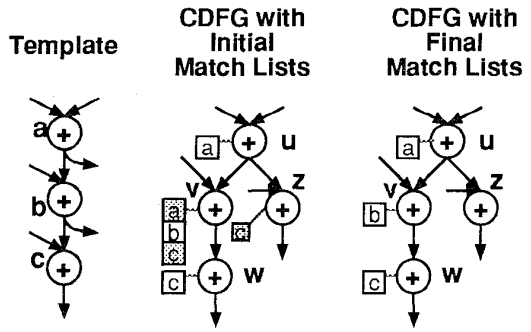

Fig. 4. Template and CDFG with match lists. The algorithm constructs an initial match list and then iteratively strips away invalid matches.

two node matches each. However, there are only four node matches total: $(u, a, t), (v, b, t), (w, b, t), (x, b, t))$.

To demonstrate the algorithm, consider the template and CDFG shown in Fig. 4. The valid node matches to be found by the algorithm are $(u, a, t), (v, b, t)$, and (w, c, t).

The algorithm begins by generating initial node match lists for each graph node x such that a template node y is placed in the match list of x if and only if the following rules are satisfied:

1) x and y must be of the same type.
2) For each parent p_y of y, there must be a parent p_x of x of the same type and at the same input of y as x.
3) There must exist some pairing between children of x and children of y such that for each child c_y of y, there must be a unique child c_x of x of the same type and at the same output of x as y.
4) There must exist a similar pairing such that each child of x is matched unless the corresponding output of y is connected to a template port.

The condition in Rule 4 relating to the template output ports is necessary as can be seen from the following example. In Fig. 4, assume that node w has a child r not shown. If this condition were not in place, the template would not match since there would be no template node matching r. Note also that the pairing of children in Rules 3 and 4 can be easily cast into the maximum bipartite matching problem, a well-solved problem in computer science [38].

Using these rules, the initial match lists shown in Fig. 4 are obtained. The next step in the algorithm is to generate a list of dependencies for each match in each match list. For any two node matches, $m = (x_1, y_1, s)$ and $n = (x_2, y_2, t)$, if s and t are the same template, and graph node x_1 is a parent or child of x_2, then m and n must have dependencies on one another (that is they satisfy the definition of dependency). In the example, (u, a, t) and (v, b, t) have dependencies on one another because b is a child of a. Similarly, (v, b, t) and (w, c, t) also have dependencies on one another.

During the process of generating dependencies, the algorithm detects and prunes invalid matches. Invalid matches are determined by applying the above four rules with an additional stipulation. When comparing parents and children of x and y, they must not only have the same type (as specified in Rule 4), but they must be matched by node matches.

In Fig. 4, (z, c, t) is invalid since c's parent b does not match z's parent u. (v, a, t) and (v, c, t) are similarly invalid. Note in Rules 3 and 4 that the algorithm must verify not only that a valid pairing exists, but also that there exists a valid pairing containing every node match between the children.

The invalid matches are eliminated by a process of relaxation. In general, the term relaxation refers to an approach to solving nondiscrete problems [39, Sec. 8.3–8.6]. Relaxation algorithms generally begin with an initial guess for the solution, and then iteratively improve the solution using local optimization techniques until the result is sufficiently close to the exact solution. The same concept can be used to solve many discrete problems with the added benefit that an exact solution can often be found. This technique applies nicely to the problem of eliminating invalid matches. As each invalid match is eliminated, its dependencies are checked to find matches that have become invalid because of the elimination (local optimizations). This process continues until all invalid matches are eliminated. Since comparisons are always local (i.e., only the parents and children of a node), this approach can be viewed as a relaxation to solve the global problem of invalid match elimination.

After the process is complete, invalid matches may still remain. The example in Fig. 5 shows a typical degenerate case where this may occur. The matches shown in the CDFG do not violate any of the rules, and yet this template obviously does not match. Such cases can be handled by a simple postprocessing step. It can be observed that a particular node match may have many descendant node matches which match the same template node to more than one graph node. However, the ancestors must match template nodes uniquely. This results from the fact that commutativity is not directly treated by the algorithm (if commutativity were treated, then this algorithm would be a solution to the graph isomorphism

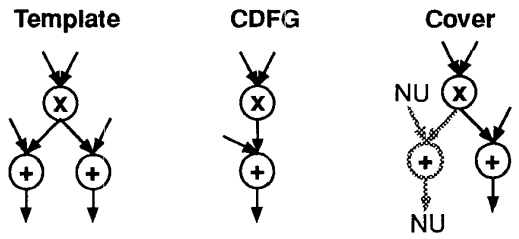

Fig. 6. Partial matching by unused resources.

problem, a well-known unsolved problem). Note that because commutativity is not treated directly, a separate template must be created for each commutative permutation of an algorithm which is expected to occur in the CDFG (e.g., both the patterns "$a + b^*c$" and "$b^*c + a$" may occur in a given CDFG even though they are essentially the same algorithm). Because of the uniqueness requirement for ancestor template nodes, the degenerate node matches of the kind shown in Fig. 5 can be eliminated by removing node matches which have ancestors matching the same template node to more than one graph node. All remaining node matches are valid.

The number of node matches generated by the algorithm is bounded by the product of the size of the CDFG, the number of templates, and the size of the templates. Since all steps in the relaxation are local and the postprocessing step consists of only one depth-first search for each node match, the time complexity of the algorithm is polynomial with respect to the number nodes in the original CDFG (and, in fact, has behaved linearly for all test cases as shown in Fig. 13).

3) Partial Matching: The matches found by this simple approach are **complete** matches. The approach can be extended to include **partial** matches as well. These are cases in which portions of the template remain unmatched. By using these types of matches, the same operators can be used more often allowing for higher resource utilization. Partial matching techniques can be divided into two classes: partial matching by unused resources, and partial matching by identities. These are described in the following sections.

a) Partial matching by unused resources: This class of partial matching stems from a relatively simple observation. While each child of a graph node may need to be matched, no child of a template node has to be matched (except as required to match the children of the graph node).

Fig. 6 shows a simple example. By simply eliminating rule 3 from the basic algorithm described above, partial matches by unused resources can be easily handled. The modified algorithm, then, generates a solution which takes into account all complete matches as well as all partial matches by unused resources.

b) Partial matching by identities: This second class of partial matching results from the observation that by setting an input of an operator to a particular constant value, the output of the operator becomes equivalent to the other input. In this way, a node can be effectively *short-circuited* in the CDFG. Mathematically, such a node is referred to as a neutral element. Fig. 7 shows a typical example of this. While the problem of finding partial matches is difficult in general, it can be made tractable by applying simple heuristics. Unlike the case of complete matches, however, these heuristics cannot be guaranteed to find all cases of partial matching.

The key to the algorithm is the concept of **bypassability.** A template node is said to be **bypassable** on some input if its output value can be set equal to this input by setting the other inputs to constants without inducing side-effects. More specifically, bypassing a template node y should not affect any outputs of the template which are descendants of y. For example, in Fig. 7, if node b were somehow connected to node d, then node c would no longer be bypassable on its second input. By determining bypassability during preprocessing, it is possible to check the validity of node matches involving partial matching by identities without having to consider all the nodes in the templates (which would make the algorithm quite inefficient).

Two other heuristics are used, as well, to obtain efficient operation. First, during preprocessing, only one bypassable input should be selected for each template node. This simplifies many steps in the matching process. Second, during postprocessing, node matches must be checked to ensure that no node can be both bypassed and matched in the same template match.

The entire template matching algorithm can be implemented as a polynomial time algorithm which has been found to be reasonably efficient in practice.

B. Template Selection

Given that the node matches for each node have been found by the template matching algorithm, template matches can be constructed to replace groups of primitive operators (nodes in the CDFG) with more complex operators (represented by the templates). The template selection phase of the algorithm generates a cover for the original graph which optimizes the delay critical path length. The cover contains nodes representing both primitive operations and more complex operations represented by the templates. The complete template selection algorithm is summarized by the following pseudocode. The details of the algorithm are described in the following sections.

```
while critical path improves and there are node matches
      left, do
    Evaluate min/max completion times of all nodes
    S ← set of ε-critical nodes
    C_min ← ∞.
    for each v ∈ S, do
        for each node match m of v, do
            M ← optimally constructed template
                  match containing m
            C ← cost(M)
            if C < C_min, then
                M_opt ← M
    Replace graph nodes in M_opt with template
        (M_opt)
    Remove invalid matches resulting from placing
        M_opt
    if critical path increases or M_opt introduces a
        combinatorial cycle, then
        Restore the CDFG
```

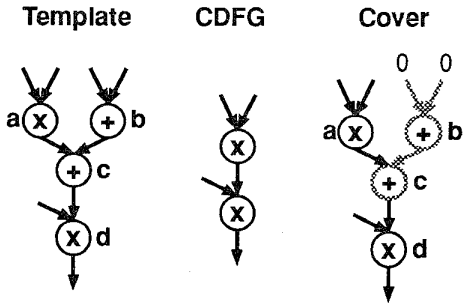

Fig. 7. Partial matching by identities.

1) ε-Critical Network: Although the critical path is the optimization target, optimizing nodes in critical paths only is ill-advised since the critical paths change as nodes are replaced by templates. A common means by which to overcome this difficulty is to use the concept of the **ε-critical network** [40]. The set of nodes lying in paths having lengths within some empirically derived constant ε of the critical path length (a value of approximately 0.1 was found to give the best results for our benchmarks). These nodes presumably are those which are likely to become critical as well as those that already are. The epsilon-critical network is calculated in a manner similar to calculating critical paths using a modified version of Dijkstra's shortest path algorithm [41]. In this way, global information can be taken into account with no significant loss in computation time.

2) Construction of a Template Match: Using the match lists generated by the template matching algorithm, it is possible to construct any valid template match. Nodes in the ε-critical network are used as *seed nodes* for template match construction. The goal is to construct optimal template matches which contain node matches corresponding to seed nodes.

Starting with a given node match, a template match is constructed by recursively collecting other node matches from the dependency lists. Since the dependency list of a node match may match a particular graph node or template node more than once, the following rules must be applied during template match construction.

Assume there exists a node match $m = (x, y, t)$ which matches graph node x to template node y, and is already in some unfinished template match construct.

1) If some match n is a parent dependent of m, then n must be in the construct.
2) For each output o of x, if the corresponding output of y is not connected to a template output port, then for every child c of x connected to o, there must be a node match n in the construct which is a dependent of m and matches c.
3) Assume the construct contains node matches $n_1, n_2, n_3, \cdots, n_k$ which are child dependents of m and whose graph nodes are connected to output o of x. Then n which is a similar child dependent of m with its graph node connected to output o, can only be included in the construct if there exists a pairing of children of x and y which utilizes matches $n_1, n_2, n_3, \cdots, n_k$ and n.
4) Any node match n which matches x or y cannot be added to the construct.

These rules do not necessarily uniquely determine the template match to be constructed for a given node match. Specifically, a node match m, defined as above, may have more than one dependency which matches the same child of x or y. The proposed algorithm uses the following heuristic rules for selection of node matches among the children. Given a node match m defined as above

1) if two dependent node matches n_1 and n_2 of m match the same graph node x, the choice between n_1 and n_2 is arbitrary;
2) if two dependent node matches n_1 and n_2 of m match the same template node y but different graph nodes, the node match whose graph node has a smaller match list is favored.

The second rule is based on the observation that the fewer node matches a given graph node has, the less likely it is that this node will be covered by a template if not included in the current template. The first rule was judged to be adequate simply because at this level, it is impossible to be sure which match would give a better solution and since most template matches are small, they are probably equal.

3) Selection of the Best Template Match: Template matches are constructed to cover the entire ε-critical network. From these an optimal template match is selected which is most likely to reduce the total execution delay. The ε-critical network is then recomputed and a new set of template matches is constructed. This process continues until the delay can no longer be improved. Note that delays are measured in terms of the number of clock cycles.

The key to this algorithm, then, is to select template matches which improve the critical path length the most. Once constructed, the optimal template match is selected using the cost function given in (1) (note that empirical constants are omitted for simplicity)

$$\text{Cost} = \frac{1 - e^{-((M/N)\overline{M})}}{\alpha i_{\text{act}} + (1-\alpha) i_{th}}. \tag{1}$$

The form of the numerator $(1 - e^{-x})$ was selected heuristically to obtain a value in the range 0 to 1. The numerator term itself represents the degree to which using the specified template match will disallow other (possibly better) template matches (close to 0 represents few node matches are covered; close to 1 represents far more than average are covered). M represents the total number of node matches for all graph nodes covered by the template match. N represents the number of graph nodes in the template match. $\overline{M}$ represents the average number of node matches per node in the CDFG.

The denominator term represents the degree of improvement in the delay locally from replacing the primitive nodes with the complex operator represented by the template. α represents the fraction of nodes in the ε-critical network which represent complex operators (i.e., places where primitive operators have

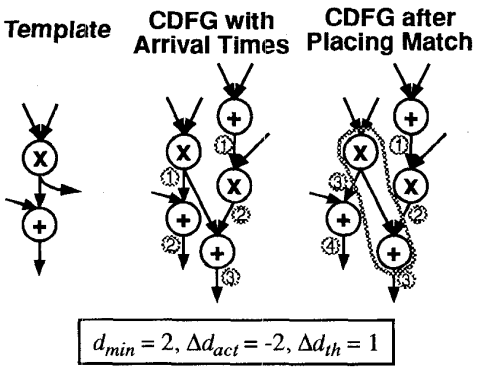

Fig. 8. Calculation of cost function parameters. Assume one clock cycle for each operation.

already been replaced by complex operators). i_{act} and i_{th} represent the *actual* and *theoretical* improvements locally expected to be gained from replacing the primitive operators with the more complex operator represented by the template. More specifically, we wish to incorporate both the actual immediate improvement in the total delay (which can be obtained by using the complex operator) and the improvement which could theoretically be obtained if all the inputs are ready at the same time (which could occur later as template matches are placed). The rationale behind the weighting by α is that initially, when most nodes in the CDFG have not yet been covered, the arrival times of the inputs to an operator are highly uncertain as the CDFG may change dramatically. Therefore, it is important to consider the potential theoretical gains. When most of the CDFG is covered by complex operators, however, the theoretical improvements no longer apply since the CDFG is not likely to change much more.

These improvements i_{act} and i_{th} are computed as shown in (2) and (3). Note that these definitions are somewhat arbitrary, but were selected because of certain valuable properties. More specifically, i_{act} tends to be negative if any (not just the latest finishing) output finishes later using the complex operator than with primitive operators

$$i_{\text{act}} = \frac{\Delta d_{\text{act}}}{d_{\min}} \tag{2}$$

$$i_{th} = \frac{\Delta d_{th}}{d_{\min}}. \tag{3}$$

$d_{\min}$ is a theoretical calculation of the minimum possible delay the graph nodes in the template match could have (this may not be their actual delay in the CDFG since external delays may delay some nodes more than others). Δd_{act} is the number of clock cycles by which the delay will locally improve (or possibly get worse) by replacing the graph nodes with the template. The actual value used is the sum of the amounts by which the arrival times decrease at all template output ports. Δd_{th} is a theoretical computation of the maximum amount of time by which the delay could improve regardless of whether it actually does. Fig. 8 shows an example of the calculation of $d_{\min}, \Delta d_{\text{act}}$, and Δd_{th}.

Note that while Δd_{th} will always be positive (or else the template should not ever be used), Δd_{act} can be negative, as it is in Fig. 8. Assume each node has a one clock cycle delay as does the template. Originally the CDFG has a delay of three clock cycles, but when the substitution is made in the final cover, the total delay increases to four clock cycles. One would expect, however, that the delay would decrease since the template has a lesser delay than the two primitive operations combined separately (as indicated by Δd_{th}). Indeed, if the delays of the other paths in the CDFG can be shortened, then it may actually be advantageous to use this template as shown in Fig. 8 in the CDFG on the right.

If a selected template match in the CDFG actually does worsen the critical path length, the algorithm reverts to the last version of the CDFG and tries a few more template matches. If after a few tries no improvement can be found, the algorithm terminates.

C. Clock Selection

1) Minimal Iteration Method: The covering algorithm described above can only be applied for a specific clock frequency since the delays of the operators (and templates) must be specified as integer multiples of the clock period. It is readily apparent that selecting the proper clock is crucial to obtaining the best performance [Fig. 1(a) and (b)]. It is also apparent that the clock selection and graph covering cannot be performed independently without sacrificing the quality of the results. The clock selection for critical path optimization problem can be defined in the following way:

Given a CDFG and a library of available hardware units with their corresponding execution times, find a clock cycle period which will result in a minimal critical path after the application of an arbitrary, but given template selection algorithm.

In order to improve the overall efficiency of the algorithms, the following observations are used to limit the number of frequencies considered.

1) Based on current and foreseeable technologies, it can be assumed that a resolution of 1 ns for the clock period is sufficient.
2) If clock period $T_{\min}$ is the smallest clock period which yields a delay of one clock cycle for all hardware units, no clock period larger than $T_{\min}$ can yield the least total delay for the CDFG.
3) Since having all hardware units require multiple clock cycles would yield an impractical implementation, no such clock periods merit consideration.

Observation 1 gives the resolution of the range and Observations 2 and 3 give upper and lower bounds, respectively. Observation 2 is easily proven by noting that for all clock periods larger than this bound, the delay is directly proportional to the clock period. Note that in these observations there is no direct assumption on the number of cycles a particular operation will take.

The clock selection algorithm compares all possible clock periods using a three-phase branch-and-bound strategy. The algorithms use two initial *pruning* strategies based on the

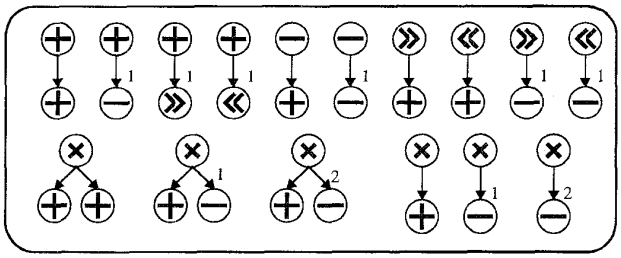

Fig. 9. Template library. The templates shown comprise the template library applied in all of the experiments (note that many of the diagrams shown represent multiple templates based on the different possible input orderings resulting from commutativity). In principal, larger templates could be used as well, but generally with higher area penalties. The numbers on the inputs to the subtractors indicate the input number.

following two observations:

1) Clock Multiple Elimination Strategy: A clock period of length T can always yield a real-time delay critical path at least as short as that resulting from a clock period of length $k \times T$, where k is any integer > 1.
2) Inferior Clock Elimination Strategy: Given two clock periods, T_1 and T_2, if T_1 yields a shorter real-time latency for **all** hardware units than T_2, then the real-time critical path for T_1 is always better than for T_2.

The first observation is easily shown by observing that if the number of clock cycles for each node in the graph is multiplied by k, the period can be reduced by k to obtain the same total delay. The second is shown by observing that for any template-matched cover, if all hardware units have a shorter real-time latency for one clock period than for another, the sum of the delays of those units on any path must be less for that clock period. The reader may observe a possible consequence of applying the first observation blindly. If clock periods T and $k \times T$ both yield the same real-time delay, $k \times T$ is most likely the better choice since larger clock periods usually have less overhead. This fact is discussed below.

By applying these strategies to the set of possible clock periods, the size of the set can be quickly and significantly reduced. In particular, the second observation generally reduces the number of candidates to a very small number. It should be noted that strictly speaking, these observations only apply if the template selection (i.e., graph covering) is the exact optimum result. Experimental results indicate that the covering algorithm indeed consistently generates high quality results so that these apply. It should be noted that by applying the first observation, the algorithm does not consider the fact that when two clock periods could yield the same performance, the longer one is generally preferable. In order to alleviate this, the lower bound on the clock period was set to the delay of the fastest unit in generating the experimental results. This change has only second-order effects on throughput, while it preserves the small size of controller.

For the remaining candidate clock periods, a table is constructed of the execution times for each unit versus the clock period. The covering algorithm (Section II-B) is run for each clock period and the clock period yielding the minimum total delay is selected as optimal. As a final postprocessing step, given an optimal clock period T was found, multiples $k \times T$ are tried starting at $k = 2$ until $k \times T$ yields a worse delay than T. The largest clock multiple of T which still has the same delay as T (i.e., the largest optimal multiple of T) is the final solution used. The reasoning behind this is that larger clock periods generally have less overhead (area and power) than smaller ones, and therefore, are more desirable whenever possible. In our experiments, however, we have found that $k = 1$ usually yields the only optimal solution.

The total clock selection algorithm has been found to increase the run-time complexity of the overall optimization process by only a small constant multiple, regardless of the size of the problem. As will be shown in the next section, clock selection itself is quite useful for many applications, even if template selection is not used.

III. Experimental Results

The algorithms of Section II have been implemented as part of the module selection facilities of the Hyper high-level synthesis environment [5]. The benchmark set used for testing and evaluation represents a variety of CDFG structures. The designs include elementary functions (sqrt, sine), linear controllers of various structures and numbers of states (LinearCn5mat, LinearCn5ellip, LinearCntrl3), several eighth-order Avenhaus IIR filters of different structures (wdf8, cascade8, parallel8, gmladder8), a modem, volterra filters, a differentiator, several transforms (Hilbert, Wavelet, Winogradfft11, Winogradfft13), a convolution (conv5), a seventh-order IIR filter, and several other FIR filters of various structures and sizes (DSrect25, DSfir51, fir133, DSkais55, fir100). The examples were obtained through several sources, including commercial DSP handbooks, the research literature, and IC manufacturers. The results were generated using a library of 44 templates representing groups of chained units as shown in Fig. 9.

Table I shows the throughput improvement for the sample benchmarks. The original clock period is selected to be the minimum clock period for which all templates require one clock cycle to execute. For the designs shown, the overall throughput doubles on average (27% increase from clock selection and 73% increase from template selection). The minimum and maximum improvements for clock and template selection are 21% and 276%, respectively. The mean and standard deviation are 122% and 80%, respectively. Fig. 10 shows the relative contributions of clock selection and template selection to the overall improvement for each of the benchmarks. While template selection tends to have the most impact, clock selection generally contributes significantly and is, in some cases, the dominant contributor to the overall improvement in throughput.

Table II shows the impact on area for these benchmarks. The active area estimates are obtained after the scheduling and assignment step, and represent only arithmetic/logic units, registers, multiplexers, and I/O. The minimum and maximum area overhead are 14% and 225%, respectively. The mean and standard deviation are 66% and 47%. In terms of the number of busses required, the maximum **reduction** is 30%, while the maximum **increase** is 782%. The mean and standard

TABLE I
PERFORMANCE IMPROVEMENT ON BENCHMARKS

Design Name	Original / After Clock Selection / After Clock Select & Instr. Select									Throughput Improvement (%)
	Number of Clock Cycles in Critical Path			Clock Period (ns)			Minimum Sample Period (ns)			
sqrt	11	16	6	62	31	63	682	496	378	80
LinearCn5mat	4	6	5	77	51	51	308	306	255	21
wdf8	10	13	8	83	61	81	830	793	648	28
volterra (s/a)	14	17	7	65	25	37	910	425	259	251
volterra (mult)	12	14	7	89	68	68	1068	952	476	124
iir7	10	11	6	87	69	95	870	759	570	53
cascade8	10	11	8	69	42	42	690	462	336	105
LinearCn5ellip	5	7	5	77	51	51	385	357	255	51
modem	20	30	18	74	25	35	1480	750	630	135
parallel8	9	11	8	68	39	39	612	429	312	96
conv5	10	11	7	77	52	74	770	572	518	49
sine	16	22	23	62	32	27	992	704	621	60
Hilbert	12	13	7	77	54	54	924	702	378	144
gmladder8	34	43	47	72	47	31	2448	2021	1457	68
Wavelet	14	15	9	77	52	57	1078	780	513	110
LinearCntrl3	6	8	5	77	51	51	462	408	255	81
DSrect25	15	16	7	92	75	75	1380	1200	525	163
Differentiator	19	20	9	74	47	47	1406	940	423	232
Winogradfft11	11	12	10	77	52	59	847	624	590	44
Winogradfft13	11	12	10	77	52	59	847	624	590	44
DSfir51	28	29	11	92	75	75	2576	2175	825	212
fir133	67	67	20	88	86	86	5896	5762	1720	243
DSkais55	30	31	11	89	68	68	2670	2108	748	257
fir100	103	103	28	88	86	86	9064	8858	2408	276

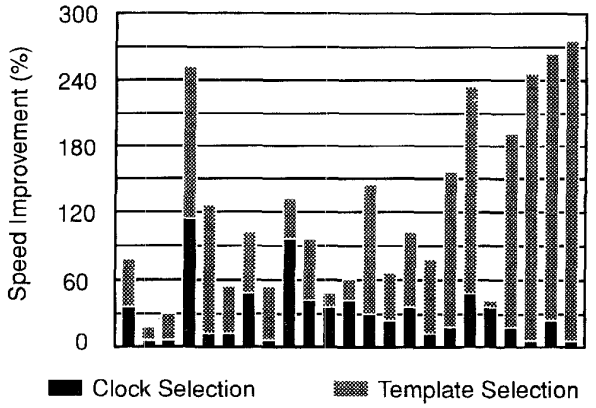

Fig. 10. Relative contributions of clock selection and template selection. While template selection generally contributes more to performance than clock selection, clock selection is still very important.

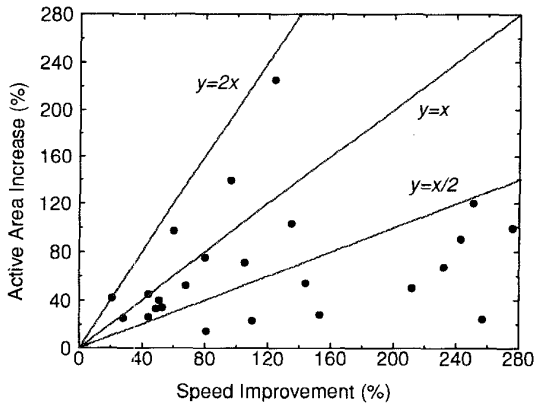

Fig. 11. Speed/Area tradeoff. In most cases, the gains in speed outweigh the penalty in area. In many of these cases, the increase in area is less than half the gain in speed.

deviation are a 64% and 166% increase, respectively. While the worst case increase is high, 9 of the 24 designs actually had a decrease in the required number of busses. Also, note that only four designs had a bus count increase greater than 100%, while 12 out of 24 had an increase in throughput greater than 100%. For the four designs in which the bus count increased, the templates selected caused the slack on many of the graph edges to decrease. This caused a higher degree of data transfer parallelism and thus a larger required number of busses for implementation.

Although the active area increases by 48% on average, the area-time (AT) product overall decreased by 44%. The area-

TABLE II
AREA IMPACT ON BENCHMARKS

Design Names	Original vs. Clk. Selection & Instr. Sel								Active Area Increase (%)	Interconnect Increase (%)
	Number of Nodes		Number of Operators		Active Area (mm^2)		Number of Busses			
sqrt	11	6	3	2	0.52	0.91	4	5	75	25
LinearCn5mat	21	18	3	4	4.12	5.87	27	26	42	-4
wdf8	23	19	4	6	2.05	2.56	23	27	25	17
volterra (s/a)	30	24	5	6	1.68	3.69	20	34	120	70
volterra (mult)	30	23	4	5	2.13	6.92	15	37	225	147
iir7	33	34	5	7	5.46	7.32	28	26	34	-7
cascade8	34	27	4	5	2.41	4.12	29	27	71	-7
LinearCn5ellip	35	19	3	4	6.41	8.98	48	34	40	-30
modem	39	31	4	5	1.39	2.82	12	27	103	125
parallel8	39	37	4	4	2.85	6.82	37	47	139	27
conv5	45	37	4	7	3.60	4.79	68	60	33	-12
sine	49	42	6	7	1.45	2.86	30	38	97	27
Hilbert	49	42	5	6	3.81	5.86	44	57	54	30
gmladder8	50	45	4	7	2.53	3.84	31	42	52	35
Wavelet	53	38	5	7	4.33	5.31	55	55	23	0
LinearCntrl3	56	33	3	4	13.03	14.89	79	57	14	-28
DSrect25	62	53	4	5	7.46	9.52	66	70	28	6
Differentiator	80	69	5	7	4.56	7.63	68	93	67	37
Winogradfft11	104	90	4	9	8.85	12.87	158	148	45	-6
Winogradfft13	116	100	4	8	10.56	13.35	181	168	26	-7
DSfir51	127	108	4	6	10.88	16.33	137	146	50	7
fir133	130	83	3	5	11.96	22.76	18	76	90	322
DSkais55	137	117	4	6	13.12	16.28	158	161	24	2
fir100	302	226	3	6	20.59	41.00	17	150	99	782

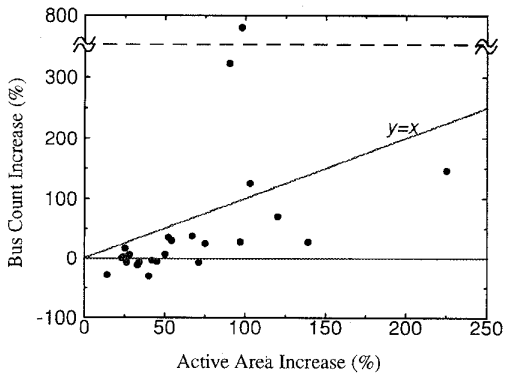

Fig. 12. Interconnect versus active area increase. In most cases, the increase in active area is larger than the increase in the bus count (which sometimes decreases).

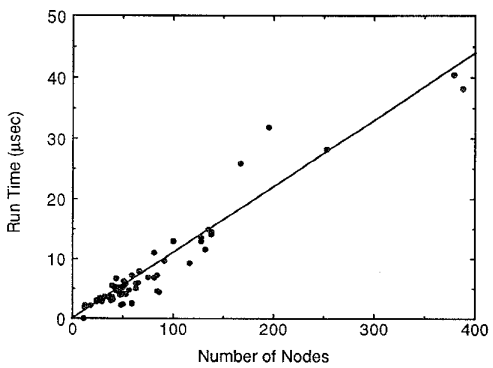

Fig. 13. Execution time versus CDFG size. Execution times obtained from runs on a Sun SparcStation 2. The line shown was derived by a least squares regression. The slope is approximately 0.11 ms/node.

speed tradeoff is shown graphically in Fig. 11 which plots the percentage improvement in speed versus the percentage increase in active area. This plot reflects only active area, not interconnect. Clearly, in most cases, the speed is improved more than the area is hurt (i.e., most points are below $y = x$, the *even tradeoff* line). Fig. 12 shows the relative increases in active area and bus count. In most cases, the increase in bus count is less than the increase in active area indicating that the predicted area increase from active area alone is generally conservative. Although the bus count is not necessarily proportional to the interconnect area, there is experimental evidence that the two are strongly correlated [42].

Fig. 13 shows a plot of the execution time versus CDFG size (number of nodes) for the template selection routines (without

clock selection). The software was run on a Sun SparcStation 2 and exhibited linear time behavior largely resulting from the fact that templates normally have a small number of nodes with small degree.

IV. Conclusion

A methodology and a set of algorithms for performance optimization using template mapping have been presented. The proposed algorithms incorporate a new template matching approach, as well as methods for partial matching and clock selection. Experimental results show significant improvements in performance without unreasonable area penalties. The ideas presented provide an ideal starting point for the investigation of many template matching problems in CAD and compiler domains.

Acknowledgment

The authors wish to thank Prof. R. Newton and S.-H. Huang for their insight, advice, and comments.

References

[1] A. P. Chandrakasan, M. Potkonjak, J. Rabaey, and R. Brodersen, "Hyper-LP: A design system for power minimization using architectural transformations," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1992, pp. 300–303.

[2] R. M. Russell, "The Cray-1 computer system" *Commun. ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[3] K. Keutzer, "DAGON: Technology binding and local optimization by dag matching," in *Proc. ACM/IEEE Design Automatic Conf.*, New York, 1987, pp. 341–347.

[4] M. Garey and D. Johnson, *Computers and intractability: A guide to the theory of np-completeness.* San Francisco, CA: Freeman, 1979.

[5] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of data path intensive architectures," *IEEE Design Test*, vol. 8, pp. 40–51, 1991.

[6] S. Karlin, F. Ost, and B. E. Bliasdell, "Pattern in DNA and amino acid sequences and their statistical significance," in *Mathematical Methods for DNA Sequences,* M.S. Waterman, Ed. Boca Raton, FL: CRC, 1989.

[7] A. S. Perelson, "Theoretical immunology," in *Proc. Lect. Complex Syst.* Redwood City, CA: Addison-Wesley, 1989, pp. 465–499.

[8] *Patterns in Protein Sequence and Structure*, W. R. Taylor, Ed. New York: Springer-Verlag, 1992.

[9] S. G. Wasilew, "A compiler writing system with optimization capabilities for complex order structures," Ph.D. dissertation, Northwestern Univ., Evanston, IL, 1972.

[10] S. W. Weingart, "An efficient and systematic method of compiler generation," Ph.D. dissertation, Yale Univ., New Haven, CT, 1973.

[11] C. W. Hoffman and M. J. O'Donnel, "Pattern matching in trees," *J. ACM*, vol. 29, no. 1, pp. 68–95, 1980.

[12] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Comm. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[13] E. A. Lee, "Programmable DSP architectures: Part I," *IEEE ASSP Mag.*, vol. 5, pp. 4–19, 1988.

[14] E. A. Lee, "Programmable DSP Architectures: Part II," *IEEE ASSP Mag.*, vol. 6, pp. 4–14, 1989.

[15] J. Darringer, W. Joyner, C. L. Berman, and L. Trevilyan, "Logic synthesis through local transformations," *IBM J. Res. Develop.*, vol. 25, no. 4, pp. 272–280, 1981.

[16] A. J. de Geus and W. Cohen, "A rule based system for optimizing combinational logic," *IEEE Design Test*, vol. 2, pp. 22–32, 1985.

[17] S. C. Johnson, "Code generation for silicon," in *Proc. 9th Symp. Program. Lang.*, ACM, New York, 1983, pp. 14–19.

[18] K. Keutzer and W. Wolf, "Anatomy of a hardware compiler," *ACM SIGPLAN Notices*, vol. 23, no. 7, pp. 95–104, 1988.

[19] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 4, pp. 491–516, 1989.

[20] A. V. Aho and S. C. Johnson, "Optimal code generation for expression trees," *J. ACM*, vol. 23, no. 8, pp. 488–501, 1976.

[21] E. Detjens, G. Gannot, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology mapping in mis," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1987, pp. 116–119.

[22] R. Rudell, "Logic synthesis for VLSI design," Ph.D. dissertation, Univ. California, Berkeley, CA, Memo. UCB/ERL M89/49, Apr. 1989.

[23] F. Mailhot and G. De Micheli, "Algorithms for technology mapping based on binary decision diagrams and on Boolean operations," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 599–620, 1993.

[24] R. J. Francis, J. Rose, and K. Chung, "Chrotle: A technology mapping program for lookup table based field programmable gate arrays," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1990, pp. 613–619.

[25] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Improved logic synthesis algorithms for table look up architectures," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1991, pp. 564–567.

[26] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, "An improved synthesis algorithm for multiplexor-based PGA," in *Proc. ACM/IEEE Design Automation Conf.*, 1992, pp. 38–387.

[27] S. Note, F. Catthoor, and J. Van Meerbergen, "Definition and assignment of complex data-paths suited for high throughput applications," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1989, pp. 108–111.

[28] S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral-III: Architecture-driven high level synthesis for high throughput DSP applications," in *Proc. ACM/IEEE Design Automation Conf.*, 1991, pp. 597–602.

[29] W. Geurts, F. Cathoor, and H. De Man, "Time constrained allocation and assignment techniques for high throughput signal processing," in *Proc. ACM/IEEE Design Automation Conf.*, 1991, pp. 124–127.

[30] B. Landwehr, P. Marwedel, and R. Domer, "OSCAR: optimum simultaneous scheduling, allocation and resource binding based on integer programming," in *Proc. ACM/IEEE Euro. Design Automation Conf.*, 1994, pp. 90–95.

[31] D. S. Rao and F. J. Kurdahi, "Partitioning by regularity extraction," in *Proc. ACM/IEEE Design Automation Conf.*, 1992, pp. 235–238.

[32] D. S. Rao and F. J. Kurdahi, "An approach to scheduling and allocation using regularity extraction," in *Proc. Euro. Conf. Design Automation Euro. Event ASIC Design*, 1993, pp. 557–561.

[33] C. M. Chu and J. M. Rabaey, "Hardware selection and clustering in the hyper synthesis system," in *Proc. EDAC*, Los Alamitos, CA, IEEE Computer Society, 1992, pp. 176–180.

[34] E. A. Rundensteiner, D. D. Gajski, and L. Bic, "The component synthesis algorithm: technology mapping for register transfer description," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1990, pp. 208–211.

[35] R. Ang and N. Dutt, "A representation for the binding of RT-component functionality to HDL behavior," in *Proc. CHDL*, Apr. 1993, pp. 251–266.

[36] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man, "Instruction set definition and instruction selection for ASIP," in *Proc. High-Level Synth. Workshop*, 1994, pp. 11–16.

[37] D. Lanneer, M. Cornero, G. Goossens, and H. De Man, "Data routing: A paradigm for efficient data-path synthesis and code generation," in *Proc. High-Level Synth. Workshop*, 1994, pp. 17–22.

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms.* New York: McGraw-Hill, 1990, pp. 600–604.

[39] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis.* New York: Springer-Verlag, 1980.

[40] K. J. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Timing optimization of combinational logic," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1988, pp. 282–285.

[41] E. W. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.

[42] R. Mehra and J. Rabaey, "Behavioral level power estimation and exploration," in *Proc. Int. Workshop Low Power Design*, Napa, CA, 1994, pp. 197–202.

Miguel R. Corazao received the B.S degree in electrical engineering from the University of Texas, Austin, in 1992, and the M.S. degree in electrical engineering and computer-aided design from the University of California, Berkeley, in 1994.

Since 1994, he has been with the Intel Corporation, Santa Clara, CA. He is currently working as a CAD engineer on the development and support of physical design environments for the corporation's microprocessor design teams.

Marwan A. Khalaf (S'91–M'95) received the B.S. degree in computer science and engineering from the University of California, Davis, in 1992, and the M.S. degree in electrical engineering from the University of California, Berkeley, in 1994.

Since then, he has been with the Altera Corporation, San Jose, CA.

Lisa M. Guerra (S'95) for photograph and biography, see p. 744 of the July 1996 issue of this TRANSACTIONS.

Miodrag Potkonjak for biography, see p. 165 of the February 1996 issue of this TRANSACTIONS.

Jan M. Rabaey (S'80–M'83–SM'92–F'95) for photograph and biography, see p. 587 of the June 1996 issue of this TRANSACTIONS.