INSTRUCTION SET MAPPING FOR PERFORMANCE OPTIMIZATION

M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak†, J. Rabaey

Dept. of EECS, University of California at Berkeley, CA 94720
†C&C Research Laboratories, NEC USA, Princeton, NJ 08540

Abstract

Performance optimization is the primary design goal in most digital signal processing (DSP) and numerically intensive applications. The problem of mapping high-level algorithmic descriptions for these applications to specialized instruction sets has only recently begun to receive attention. In fact, the problem of optimizing performance has yet to be addressed directly. This paper introduces a new approach to instruction set mapping (and template matching in general) targeted toward performance optimization. Several novel issues are addressed including partial matching and automatic clock selection.

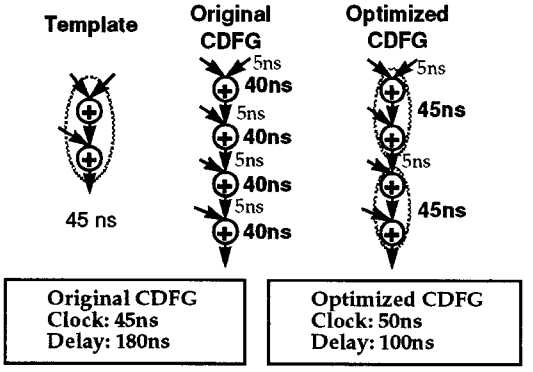

Figure 1: Illustrative Example

1 Introduction

As a result of the increase in demand for high-speed integrated circuits, an increasing need for synthesis tools targeted toward performance optimization exists. The increasing complexity of application specific integrated circuits (ASICs), in general, indicates that performance oriented high-level synthesis tools will become especially important.

Different tasks in high-level synthesis have varying degrees of effect on performance. One potentially powerful transformation in the synthesis process is instruction set mapping. Instruction set mapping refers to replacing groups of primitive hardware units (instructions) with more complex and more powerful units as shown in Figure 1. By replacing primitive instructions with the more complex instructions which execute faster, overall performance can be improved.

The most fundamental problem in instruction set mapping is pattern, or template, matching. One of the most influential works on template matching was provided by Aho et al. who suggested an optimal tree matching approach [1,2] for instruction set mapping in compilers. Keutzer later noted in [3] that this approach could be extended with some heuristics to technology mapping for logic synthesis. In high-level synthesis, the first attempt at template matching came from IMEC [4] and utilized an integer linear programming technique. Rao and Kurdahi devised a different approach to template matching for addressing partitioning in high-level synthesis using regularity extraction [5]. Chu and Rabaey used a simulated annealing based algorithm for optimizing area and clock speed during chaining and module selection [7]. This paper proposes a novel approach to template matching in high-level synthesis which does not require the harsh trade-offs between speed and quality demanded by previous approaches and, further, has the capability to handle general cyclic flowgraphs. The paper also discusses the more general problem of mapping hardware instruction sets using a novel optimization goal (performance).

2 Template Matching

The instruction set mapping algorithm can be divided into three major components as shown in Figure 2: template matching, instruction selection, and clock selection. Template matching, the central problem in instruction set mapping, refers to the matching of templates representing

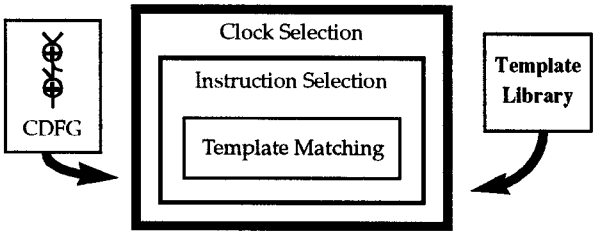

Figure 2: Structure Chart for the Instruction Set

complex instructions (specialized hardware, chained units) to a control/data flowgraph, or CDFG, consisting of only primitive instructions. Since the interpretation of the templates is application-specific, template generation is not discussed here. The fundamental difficulty in template matching lies in the fact that the number of template matches can be quite large and expensive to enumerate.

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**. 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. A matching between some set of graph nodes and an entire template is referred to as a **template match**.

The output of the proposed template matching algorithm is a list of node matches (**match list**) for each graph node. The total number of node matches for a CDFG is polynomially bounded. In addition, many template matches share node matches meaning that the node matches often require less space. The first step in the proposed algorithm is simply to create initial (potentially incorrect) match lists for all graph nodes by comparing only the types of the nodes being matched, and the types and number of parents and children. Invalid node matches are then systematically eliminated by finding node matches for which either a parent or child of the matched graph node does not have a proper node match to a parent or child of the template node. Figure 3 shows a simple example of the execution of the basic algorithm. The node match between graph node v and template node a is invalid because the child w of v does not match template node b, the child of a. This step is repeated until no other invalid node matches are found. Some invalid matches may remain in certain degenerate cases, but are eliminated by a simple post-processing step.

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

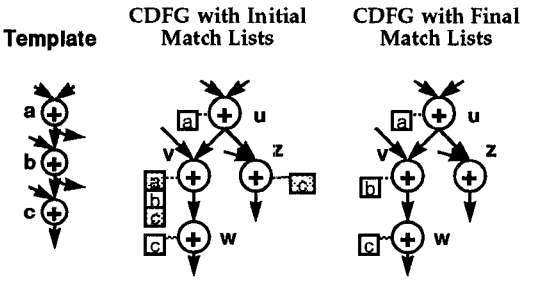

Figure 3: Execution of the Basic Template Matching Algorithm

of the template remain unmatched. By using these types of matches, the same instructions can be used more often allowing for higher resource utilization. Partial matching can be differentiated into two classes. The simplest is partial matching by unused resources. 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). This class of partial matching requires only minor changes to the basic algorithm.

The second class of partial matching is by identities, algebraic or otherwise, as shown in Figure 4. While this problem is difficult in general, it can be made tractable by applying simple heuristics. Unlike the case of complete matches, these heuristics cannot be guaranteed to find all cases of partial matching.

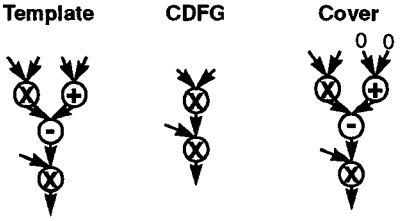

Figure 4: Partial Matching by Identities

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. By determining bypassability during pre-processing, node matches can be verified independently during the main portion of the algorithm without considering the entire template. Also, during pre-processing, at most one bypassable input should be selected for each template node. This simplifies many steps in the matching process. In addition to pre-processing, post-processing is

required to insure that nodes are not both matched and bypassed.

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

3 Instruction 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 instructions (nodes in the CDFG) with more complex instructions (represented by the templates). These template matches must be selected so that the overall critical path length is optimized.

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 **epsilon-critical network**: the set of nodes lying in paths having lengths within some empirically derived constant ε of the critical path length. These nodes presumably are those which are likely to become critical as well as those that already are.

For each graph node in the epsilon critical network ($\varepsilon \approx 10\%$), the proposed algorithms construct a single template match for each of the node matches of the graph node. An optimal match is then selected from among these and is placed in the CDFG. The process is then repeated until the critical path length can no longer be improved.

A single node match does not necessarily uniquely determine the template match to be constructed, particularly when partial matching is considered. The algorithm applies heuristics to select node matches from among the possible choices. The criteria for inclusion in the template match include maximizing the number of epsilon critical nodes in the template match as well as maximizing the improvement (locally) in the total execution time for nodes in the CDFG. Once constructed, the template matches are compared by a heuristic cost function. This function favors template matches which locally improve the execution delays of the nodes and which cover graph nodes with few node matches (since these graph nodes are less likely to be covered by other template matches).

This algorithm generates a complete graph cover which optimizes the critical path. However, before the delays of the nodes and templates can be known, a clock period must be selected as is discussed in the next section.

4 Clock Selection

Selecting the proper clock frequency can have a dramatic effect on performance. The proposed algorithm for clock selection iteratively executes the instruction selection algorithm for different clock frequencies using the best result as the final solution. This iterative approach is necessary since the optimal clock frequency depends on how the CDFG is finally covered and vice versa. The key to the approach is to try as few clock frequencies as possible so as to minimize the running time. The minimum clock period considered is the minimum latency of any operator (since smaller clock periods would generally be impractical even if the delay were theoretically less). The maximum clock period and the resolution between clock periods are assigned values appropriate to the technology.

The number of clock periods to be considered can be reduced significantly by the following observations.

1. A clock period T_1 which is a multiple of another clock period T_2 cannot yield a shorter critical path than T_2.
2. If all operators and templates have a shorter delay for clock period T_2 than for T_1, then T_2 will yield a shorter critical path than T_1.

These observations are used to *prune* obviously inferior clock periods. In addition, the algorithm uses min-bound estimation techniques to quickly eliminate clock periods which could never have a shorter delay than the delay already computed for some other clock period.

5 Experimental Results

These algorithms have been implemented as part of the module selection facilities of the Hyper high-level synthesis environment [6]. The benchmark set used for testing and evaluation represents a wide variety of CDFG structures (regular, irregular, recursive, and non-recursive). The results were generated using a library of 45 templates representing groups of chained units (some of which are commutative permutations of others).

Table 1 shows the throughput improvement of the sample benchmarks. The original clock period is selected to be the minimum clock period for which all instructions require one clock cycle to execute. For the designs shown, the throughput doubled on the average overall (27% increase from clock selection and 57% increase from instruction selection).

Table 2 shows the impact on area for these benchmarks. The active area estimates represent only arithmetic/ logic units, registers, multiplexers, and I/O. While the active

Table 1: Performance Improvement on Benchmarks

Design Name	Original / After Clock Selection / Clk Sel & Instr. Sel								
	# Clock Cycles			Period (ns)			Delay (ns)		
iir7	10	11	6	87	69	95	870	759	570
cascade	10	11	8	69	42	42	690	462	336
LinCnellip	5	7	5	77	51	51	385	357	255
parallel	9	11	8	68	39	39	612	429	312
conv5	10	11	7	77	52	74	770	572	518
Hilbert	12	13	7	77	54	54	924	702	378
gm	34	43	47	72	47	31	2448	2021	1457
Wavelet	14	15	9	77	52	57	1078	780	513
LinCn3	6	8	5	77	51	51	462	408	255
fft11	11	12	10	77	52	59	847	624	590
DSfir51	28	29	11	92	75	75	2576	2175	825
DSkais55	30	31	11	89	68	68	2670	2108	748

Table 2: Area Impact on Benchmarks

Design Names	Original / Clk. Selection & Instr. Sel							
	Number Nodes		Number Instr's		Active Area (mm^2)		Number Interconn	
iir7	36	25	5	7	5.46	7.32	28	26
cascade	37	25	4	5	2.41	4.12	29	27
LinCnellip	38	22	3	4	6.41	8.98	48	34
parallel	42	35	4	4	2.85	6.82	37	47
conv5	40	32	4	7	3.60	4.79	68	60
Hilbert	48	41	5	6	3.81	5.86	44	57
gm	57	43	4	7	2.53	3.84	31	42
Wavelet	53	38	5	7	4.33	5.31	55	55
LinCn3	60	37	3	4	13.03	14.89	79	57
fft11	93	79	4	9	8.85	12.87	158	148
DSfir51	126	107	4	6	10.88	16.33	137	146
DSkais55	136	116	4	6	13.12	16.28	158	161

area increased by 48% on average, the area-time (AT) product overall decreased by 44%. Even considering the fact that clock selection has little impact on area, instruction selection did not increase the AT product significantly. Also, since the number of interconnects did not change significantly, and since interconnect has been shown to impact area significantly [8], the area increase predicted by these estimates can be considered conservative.

Figure 5 shows a plot of the execution time vs. CDFG size (number of nodes) for the instruction 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.

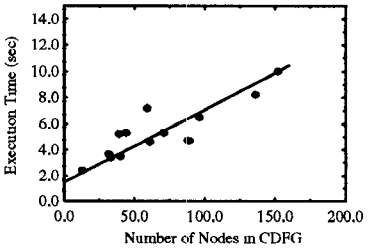

Figure 5: Execution Time vs. CDFG Size

6 Conclusions

A methodology and a set of algorithms for performance optimization using instruction set 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.

Acknowledgments

The authors would like to thank Prof. Richard Newton and Shan-Hsi Huang for their insights, advice, and comments. Also special thanks to Kimberly Knoepke for her help in editing the final manuscript.

References

[1] A.V. Aho, S.C. Johnson: "*Optimal code generation for expression trees*", Journal of ACM, Vol. 23, No. 8, pp. 488-501, 1976.

[2] A.V. Aho, M. Ganapathi, S.W.K. Tjiang: "*Code Generation Using Tree Matching and Dynamic Programming*", ACM Trans. on Programming Languages and Systems", Vol. 11, No. 4, pp. 491-516, 1989.

[3] K. Keutzer, W. Wolf: "*Anatomy of a Hardware Compiler*", ACM SIGPLAN Notices, Vol. 23, No. 7, pp. 95-104, 1988.

[4] S. Note, W. Geurts, F. Catthoor, H. De Man: "*Cathedral-III: Architecture-Driven High Level Synthesis for High Throughput DSP Applications*", ACM/IEEE DAC'91, pp. 597-602, 1991.

[5] JD.S. Rao, F.J. Kurdahi: "*Partitioning by Regularity Extraction*", 29th ACM/IEEE DAC'92, pp. 235-238, 1992.

[6] R.W. Brodersen: "*Anatomy of a Silicon Compiler*", Kluwer Academic Publishers, Norwell, MA, Ch. 16, 1992.

[7] C.-M. Chu, J.M. Rabaey: "*Hardware Selection and Clustering in the HYPER Synthesis System*", EDAC-92, pp. 176-180, IEEE Computer Society Press, Los Alamitos, CA.

[8] M.C. McFarland, T.J. Kowalski: "*Incorporating Bottom-Up Design into Hardware Synthesis*", IEEE Trans. on CAD, Vol. 9, No. 9, pp. 938-950, 1990.