Efficient Custom Instructions Generation for System-Level Design

Huynh Phung Huynh[1] Yun Liang[2] Tulika Mitra[3]
[1]A*STAR Institute of High Performance Computing Singapore
[2]Advanced Digital Sciences Center Illinois at Singapore
[3]School of Computing National University of Singapore
huynhph@ihpc.a-star.edu.sg, eric.liang@adsc.com.sg, tulika@comp.nus.edu.sg

***Abstract*—Customizable embedded processors, where the processor core can be enhanced with application-specific instructions, can provide high performance similar to custom design circuits with the flexibility of software solutions. The acceptability of customizable processors, however, critically hinges on the availability of design automation tools that can identify high-quality custom instructions from the software specification of an application. Automated customization has enjoyed significant research and commercial progress in the recent past. However, this process is currently not closely coupled with the overall system-level design flow. We propose an iterative solution that enables rapid feedback between the custom instructions generation and the system-level design decision. A key component of our solution is an efficient algorithm inspired by multi-level graph partitioning that can quickly produce high-quality custom instructions for the critical regions and thereby alleviate the system performance bottleneck.**

I. Introduction

Customizable processors are slowly emerging as the preferred choice for many embedded systems. Instruction-set customization extends the current instruction set of the processor core with application specific instructions. Custom instructions capture frequently executed computation patterns of an application. They are synthesized as custom functional units, which are tightly coupled to the existing functional units in the datapath of processor core.

The success of customizable processors critically hinges on the presence of tool chains that can expose high-quality custom instructions to the designer from the software level specification of an application. The customization process, however, has largely remained decoupled from the system-level design flow. Let us consider a canonical embedded application consisting of a set of concurrent tasks mapped to a single customizable processor. A typical design flow to accelerate this application with customization takes a bottom-up approach. The designer first generates a set of custom instructions for each individual task with the help of automated tool chains. This is followed by a system-level design space exploration to select a subset of custom instructions for each task such that the overall performance and/or energy objectives of the system are satisfied [3].

Obviously, the bottom-up approach spends enormous effort in generating custom instructions for *all the tasks*. However, many of these tasks do not contribute to the system performance bottleneck and indeed the custom instructions generated for such tasks are effectively ignored in the global selection phase. Therefore, we investigate iterative custom instruction generation scheme that provides a close coupling between the customization process and the system-level design flow. We advocate a top-down approach where the system level performance requirements guide the customization process to zoom into the critical tasks and the critical paths within such tasks. Our approach is "iterative" in the sense that we generate custom instructions in an on-demand basis. In other words, the iterative approach can quickly come up with a first-cut solution that can be iteratively refined (through inclusion of additional custom instructions) at the request of the designer.

It is relatively easy to identify critical tasks and the critical paths within such tasks in an embedded system. The main challenge for our iterative approach is quick generation of a set of quality custom instructions for the critical region. As customization process has traditionally been used in an off-line fashion, most techniques available in the literature are not suitable for our purpose. Custom instruction generation algorithms typically expose computational patterns at all possible granularity levels. In particular, these algorithms are computationally expensive as they generate many small patterns (consisting of a few native operations) with the hope that such patterns will recur multiple times within the scope of the application. Instead, our goal is to quickly identify large patterns that can give us the required performance boost. Therefore, we design an algorithm that can efficiently partition the dataflow graph corresponding to the critical region into few large custom instructions. Our algorithm is named MLGP as it is inspired by multilevel graph partitioning algorithms [4] and satisfies the constraint of generating high-quality custom instructions with minimal effort.

II. Custom Instructions Generation

We consider a system consisting of a task graph and a timing requirement. Without loss of generality, let us assume that the task graph cannot be scheduled to meet the timing requirement. Under this scenario, processor customization can provide the requisite performance boost to help meet the deadline. The objective of our iterative approach in this context is to quickly come up with a set of custom instructions CI so as to meet the deadline. The set CI is returned to the designer as the first working solution. If the designer so desires, our scheme will successively introduce additional custom instructions to CI so as to further improve the performance. In case it is infeasible to meet the deadline, the iterative approach improves the performance as much as possible.

The input to our custom instructions generation algorithm are: (i) a subsequence of basic blocks S along the critical path of the program corresponding to the critical task T_i, (ii) the amount Δ by which we need to reduce the execution time of S through customization, and (iii) the set the custom instructions already created $\mathcal{CI}$. *The last input is required to identify isomorphic custom instructions generated during different iterations and take advantage of hardware area sharing*, which is similar to the hardware reuse in conventional custom instruction generation techniques.

The goal of the custom instruction generation routine is to reduce the execution time of the basic blocks sequence S by amount Δ. If further performance gain is not achievable from the current task T_i, it is excluded from the task set. If we fail to meet the timing requirement even after exploring all the tasks, then we simply return the set of custom instructions selected so far.

A. Definitions

A Data Flow Graph (DFG) $G(V, E)$ represents the computation flow of data within a basic block. The nodes V represent the operations and the edges E represent the dependencies among the operations. $G(V, E)$ is always a directed acyclic graph. The architectural constraints may not allow some types of operations (e.g., memory access and control transfer operations) to be included as part of a custom instruction. These operations are considered as invalid nodes. We let the invalid nodes partition the DFG into multiple regions. Given a DFG $G(V, E)$, we define a region $R(V', E')$ as a maximal

subgraph of G such that (1) V' contains only valid nodes, (2) there exists an undirected path between any pair of nodes in V', and (3) there does not exist any edge between a node in V' and a valid node in $(V - V')$. Invalid nodes do not belong to any region. Note that DFG of a region is not necessary to be convex.

A custom instruction CI is a subgraph that belongs to a region within a DFG. Let $IN(CI)$ and $OUT(CI)$ be the number of input and output operands of CI, respectively. Also, for any custom instruction, let N_{in} and N_{out} be the maximum number of input and output operands allowed, respectively. This constraint arises due to the limited number of register file ports available on a processor. Any legal custom instruction CI must satisfy the constraints $IN(CI) \leq N_{in}$ and $OUT(CI) \leq N_{out}$. Moreover, a custom instruction must be a convex subgraph as non-convex subgraphs cannot be executed atomically. CI is convex if there exists no path in the DFG from a node $m \in CI$ to another node $n \in CI$, which contains a node $p \notin CI$.

B. Region Selection

Given a subsequence of basic blocks S along the critical path of a task, we explore the basic blocks in S in descending order of weight. That is, the basic block with the highest weight is selected for custom instruction generation first. Recall that the weight of a basic block is defined by its contribution (in terms of execution time) to the critical path. We partition the selected basic block into multiple regions. These regions are again sorted in descending order based on their weights. The weight of a region is defined by the number of operations contained within that region. Then, we select the region with highest weight for generating custom instructions. Intuitively, we are selecting the region that has the maximum potential to reduce the length of the critical path by Δ amount. Now, the objective of our problem is to generate a set of custom instructions from the selected region so as to reduce the most execution time. We describe a solution to this problem in the next subsection.

If the custom instructions generated for the selected region can reduce the execution time by at least Δ, then we can simply return those custom instructions along with the *gain*. Otherwise, we continue custom instruction generation for the next highest weight region of the current basic block or the next highest weight basic block if current basic block has been fully explored.

C. MLGP Algorithm

Overview: Given a critical region, the goal of our custom instruction generation algorithm is to quickly reduce the execution time of the region as much as possible. As analysis time is a major concern for our iterative scheme, we cannot spend substantial effort required in exploring all possible custom instructions corresponding to the region and then selecting the optimal ones. Thus, our objective is to generate a set of coarse-grained but legal custom instructions from the region. We observe that our goal can be achieved by *partitioning* the data flow graph (DFG) corresponding to the region into one or more partitions. Each partition should satisfy the number of input and output operands constraints as well as the convexity constraint. That is, each partition can be treated as a custom instruction.

Graph partitioning is a well studied problem in the algorithm research community. In particular, the closest problem to ours is the k-way graph partitioning problem where the vertices of a graph are partitioned into k roughly equal partitions such that the number of edges connecting vertices in different partitions is minimized. However, our problem differs from k-way graph partitioning problem in several important aspects. First of all, we do not have any basis to choose a particular value of k — any value of k is fine with us as long

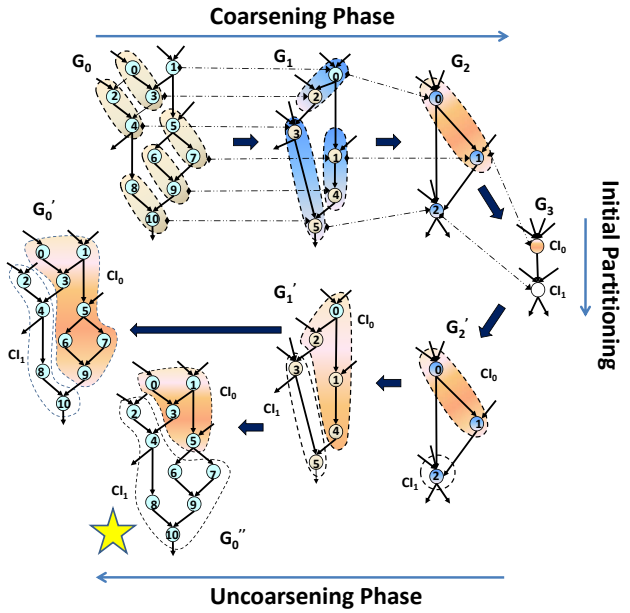

Fig. 1. Illustration of Multi-Level Graph Partitioning. The dashed lines show the projection of a vertex from a coarser graph to a finer graph.

the corresponding partition maximizes the performance gain. Second, the partitions in k-way graph partitioning problem are *not* constrained by input, output, convexity constraints. Third, instead of generating equal-sized partitions and minimizing edge-cut, our objective is to maximize the performance speedup. Finally, we are dealing with a directed graph and not an undirected graph as expected by k-way partitioning problem.

Nevertheless, it turns out that the basic structure used by multilevel recursive bisection algorithms employed to solve k-way graph partitioning problem can be quite effective in our context. Specifically, our custom instruction generation algorithm is inspired by a recently proposed multi-level algorithm due to Karypis and Kumar [4]. The basic structure of the algorithm is as follows. The graph G is first coarsened down to a small number of vertices (coarsening phase), the coarsest graph is partitioned into k parts (partitioning phase), and then this partitioning is projected back towards the original graph by periodically refining the k-partitioning (un-coarsening phase). The k-partitioning is refined on finer graphs as finer graphs have more degrees of freedom and hence provide more opportunity to improve the partitioning.

We adapt this multi-level paradigm to partition a directed graph into a small number of legal partitions so as to maximize the performance gain. We call our algorithm Multi-Level Graph Partitioning (MLGP). To avoid artificially binding k to a particular value, we eliminate the k-partitioning phase from the MLGP algorithm. Instead, we simply set the number of partitions as the number of vertices in the coarsest graph. Figure 1 shows an illustration of the MLGP algorithm applied on the DFG of a region. The original graph has 11 vertices, which are coarsened into 2 vertices or 2 partitions. These partitions are successively refined in the uncoarsening phase to generate the final custom instructions.

Coarsening phase: During the coarsening phase, a sequence of smaller graphs $G_i = (V_i, E_i)$ are constructed from the original directed graph $G_0 = G = (V, E)$ such that $|V_{i+1}| < |V_i|$. A vertex $v' \in V_{i+1}$ in a coarse graph G_{i+1} is formed by either combining two vertices $v, u \in V_i$ of finer graph G_i or by simply setting it to vertex $v \in V_i$ of G_i. In addition, a directed edge is built between two coarse vertices v and u in coarse graph G_{i+1} if there exists directed edge(s)

between their constituent vertices in graph G_i.

Each vertex v' in a coarse graph is a subgraph of G_0 when projected from the constituent vertices of v' in the finer graph. A coarse vertex can potentially become a candidate for custom instruction. Therefore, when combining two vertices v and u to form v', we have to ensure that the subgraph corresponding to v' projected into the original graph G_0 satisfies input, output, and convexity constraints. Let $IN(v')$ and $OUT(v')$ be the number of input and output edges, respectively of the projected subgraph of v' in G_0. Note that $IN(v')$ and $OUT(v')$ are not the sum of input and output edges of coarse vertices v and u.

Our matching heuristic visits the vertices of G_i in random order. If a vertex $u \in V_i$ has not been matched yet, then we select it for matching to form a vertex v' in the coarser graph G_{i+1}. First, we identify the adjacent unmatched vertices of u that when combined with u will satisfy all the three constraints. Then we match u with the adjacent vertex v such that the ratio of performance gain to hardware area (*gain-area ratio* in short) of v' is maximum. We define performance gain: $gain = sw\_ltc(v') - hw\_ltc(v')$. $sw\_lts(v')$ is the software latency of v' by summing up the software latency of all the vertices in the subgraph of v'; $hw\_ltc(v')$ is the hardware latency of v' estimated from the critical path of the subgraph of v'. Hardware area is the sum of hardware area of all vertices in the subgraph of v'. On the other hand, if u cannot find a feasible matching, $v' = u$. In Figure 1, vertices 0 and 2 of G_1 are matched to form vertex 0 of G_2.

The coarsening phase ends when $G_{i+1} = G_i$. Let $G_m = (V_m, E_m)$ be the coarsest graph. Initial partitioning simply selects each vertex $v \in V_m$ as a custom instruction. These initial custom instructions will be refined as we go through un-coarsening phase to project back to G_0. In Figure 1, coarsening phase creates a sequence of coarse graphs $\{G_0, G_1, G_2, G_3\}$ and the initial partitioning partitions G_3 into two custom instructions CI_0 and CI_1.

Uncoarsening Phase: During the uncoarsening phase, the partitioning of the coarsest graph G_m is projected back to the original graph by going through a sequence of finer graphs $G_{m-1}, \ldots, G_0$. In Figure 1, we label the graph during uncoarsening phase with G'_m for easy explanation. But in reality, $G'_m = G_m$. Consider a graph $G_i = (V_i, E_i)$. Its partitioning is represented by a partitioning vector P_i of length $|V_i|$ where for each vertex $v \in V_i$, $P_i[v]$ is an integer between 1 and $|V_m|$ (the number of partitions defined by the number of vertices in the coarsest graph). $P_i[v]$ indicates the partition to which vertex v belongs in graph G_i. In the coarsest graph G_m, each vertex belongs to its own partition.

Let V_i^v be the set of vertices of G_i (in our case one or two vertices) that have been combined to form a single vertex v in the next level coarser graph G_{i+1}. Then during un-coarsening, P_i is initially obtained from P_{i+1} by simply assigning the partitioning of v in the coarser graph ($P_{i+1}[v]$) to the partitioning of each vertex in V_i^v. That is, $P_i[u] = P_{i+1}[v], \forall u \in V_i^v$.

However, at every level of un-coarsening, we have the option of improving the projected partitioning P_i by moving some vertices from one partition to another. It is possible to improve the partitioning P_i compared to P_{i+1}. This is because, G_i is a finer graph and it allows more degrees of freedom to move the vertices. Local refinement based on Kernighan-Lin (KL) [5] or Fiduccia-Mattheyses (FM) [2] partitioning algorithms tend to produce good results for bi-partition. However, using KL or FM for refining multiple partitions is significantly more complicated because vertices can move from a partition to many others. Therefore, we propose a simple and efficient refinement algorithm (similar in spirit to the greedy refinement proposed in [4]) to target the objective and constraints of our problem.

Given $G_i = (V_i, E_i)$ with partitioning solution P_i, a vertex $v \in V_i$ is a boundary vertex of partition $P_i[v]$ if it has at least one adjacent vertex $u \in V_i$ such that v and u belong to different partitions, i.e., $P_i[v] \neq P_i[u]$. Otherwise, v is an internal vertex. For G'_1 in Figure 1, vertices {2,4} are the boundary vertices of partition CI_0 while {0,1} are the internal ones. Note that G'_1 is at the same coarse level of G_1. Our refinement algorithm visits boundary vertices in random order. If v is selected, let p_v is the subgraph of G_i w.r.t. current partition containing v and $NP[v]$ be the set of subgraphs of G_i w.r.t. neighborhood partitions to which vertices adjacent to v belong. Algorithm 1 tries to move v to neighborhood partitions if it is possible.

Algorithm 1: **Moving vertex** v

```
   Input: G_i = (V_i, E_i), P_i and NP[v]
   Result: Update P_i
1  best_ratio_improv = 0;
2  p'_v ← p_v \ {v};
3  if p'_v satifies all constraints then
4      for p ∈ NP[v] do
5          p' ← p ∪ {v};
6          if IN(p') > N_in then
7              Reduce_number_inputs(p');
8          if OUT(p') > N_out then
9              Reduce_number_outputs(p');
10         if p' satisfies all constraints then
11             ratio_improv ← gain(p')/area(p') − gain(p)/area(p) + gain(p'_v)/area(p'_v) − gain(p_v)/area(p_v);
12             if ratio_improv > best_ratio_improv then
13                 best_ratio_improv ← ratio_improv;
14                 Update best_solution;
15 if best_ratio_improv then
16     Update P_i;
```

Let p' be the resulting partition after moving v to a neighborhood partition p. p' may violate constraints. If input constraint is violated by adding v (line 6), we try to reduce the number of inputs (line 7) by continuously adding vertices (in breadth first traversal order) of the *backward* subgraph rooted at v to p'. At each level (in breadth first traversal), the vertices are ordered w.r.t. the number of edges connecting the vertex to the partition p'. If a vertex is connected with p' via multiple edges, it has highest potential to reduce the number of inputs of p'. We define *permanent* inputs as the inputs of the original graph G_0. We stop adding vertices to p' if either (1) input constraint is surely violated because number of permanent inputs of p' is more than N_{in}, or (2) input constraint is satisfied, or (3) either convexity or output constraint is violated. In G'_1 of Figure 1, if we move vertex 2 to CI_1, input constraint is violated (i.e., IN(CI_1) is 5> 4). Then, we continue adding vertex 0 and number of permanent inputs is greater than 4, we stop adding vertices. G'_0 is the finer graph which is projected from G'_1. In G'_0, after moving vertex 9 to CI_1, we continue adding vertices 6,7 which results in valid subgraph CI_1 in G''_0. Because G''_0 has higher gain-area ratio improvement than G'_0, G''_0 is the result of multi-level partitioning instead of G'_0.

Similarly, if output constraint is violated by adding v (line 8), we try to reduce the number of outputs (line 9) by continuously adding vertices of the *forward* subgraph rooted at v in order of breadth first traversal. Then, if p' is a valid subgraph, we compute its ratio improvement, $ratio\_improv$ (lines 10-11). Note that performance gain of p'_v is equal to 0 if p'_v is invalid custom instruction. If ratio improvement is better than best ratio improvement ($best\_ratio\_improv$) so far, we update best ratio improvement and the corresponding solution (lines 12-14). If there exists best ratio improvement, we update P_i (lines 15-16). Intuitively, we move vertex v to the neighborhood partition which has the best ratio improvement.

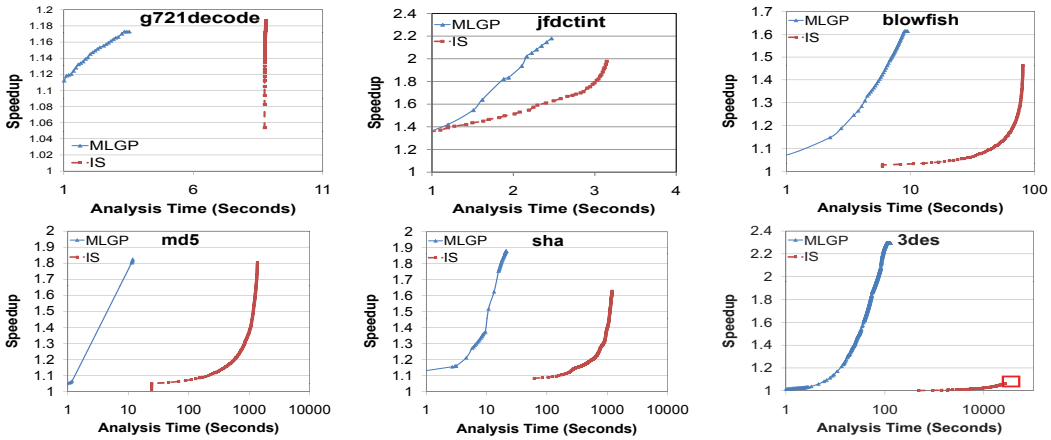

Fig. 2. Speedup versus Analysis Time

III. Experimental Evaluation

A. Experimental Setup

We modify Trimaran 4.0 compiler infrastructure [1] to generate custom instructions. Given an application, we first compile the application and generate the intermediate machine code. Then, we build the program control flow graph and corresponding syntax tree from the intermediate machine code. Subsequently, custom instructions generation is completed. Custom instructions do not include memory references or conditional branches. We assume a single-issue, in-order core with perfect cache and branch prediction as the baseline processor. We use Synopsys design tools with 0.18 micron CMOS cell libraries to synthesize the primitive operations (e.g., addition, multiply, etc.) and get their hardware area and latency. Based on the values of the primitive operations, we estimate the latency and area of the custom instructions. Each custom instruction can have at most 4 input operands and 2 output operands. Execution cycles of a custom instruction is its latency normalized against a MAC operation, which takes 1 cycle to execute in our baseline processor running at 120MHz. We conduct all the experiments on a 3GHz Pentium 4 CPU system with 2GB memory.

B. Efficiency of MLGP Algorithm

In this section, we show that MLGP can quickly generate high quality custom instructions. To substantiate this claim, we compare MLGP with a state-of-the-art custom instruction generation algorithms, called IS algorithm [6]. IS generates almost the optimal set of high quality custom instructions in practice without paying for the exponential computational complexity of the optimal algorithm [6].

We have implemented both the algorithms (MLGP, IS) in the Trimaran infrastructure as discussed in Section III-A. The same synthesis tool and cell libraries have been used for all the algorithms. Moreover, two algorithms have been restricted to generate connected custom instructions with at most 4 input register operands and 2 output register operands. For each benchmark (from MiBench, MediaBench and Trimaran benchmarks), we profile separately with representative inputs and annotate each basic block with its execution frequency. MLGP and IS both work on the "hot" basic blocks in terms of execution frequency. Each benchmark is considered as a task.

Let $\mathcal{B}$ be the set of basic block in a task and let x_i and s_i be the execution frequency and software execution time of the basic block B_i, respectively. Then the software execution time $SW = \sum_{\mathcal{B}} x_i \times s_i$. Let h_i be the execution time of B_i after applying processor customization. Then the reduced execution time $HW = \sum_{\mathcal{B}} x_i \times h_i$. The speedup of the task due to processor customization is then $speedup = \frac{SW}{HW}$.

Figure 2 plots the progress of MLGP and IS as they generate custom instructions for a task. X-axes show the analysis time of the algorithms each time they generate new custom instructions. For IS, a custom instruction is generated after each iteration while MLGP generates a set of custom instructions after processing a region in the basic block. This partially explains the faster analysis time of MLGP compared to IS.

The first observation is that MLGP returns a set of quality custom instructions within one second and more custom instructions are quickly added as analysis time progresses. For most tasks, MLGP returns the complete set of custom instructions within 10 seconds. On the other hand, IS takes much longer to return the first custom instruction for complex tasks (in the order of 1,000 seconds). Subsequent custom instructions are generated slowly. Indeed, for 3des with 2,745 instructions in a basic block, IS fails to generate the full set of custom instructions even after running for half a day. Therefore, we show only partial results (see the red highlighted rectangles). The second observation is that MLGP even obtains better speedup for some tasks (e.g., 3des, sha, blowfish, jfdctint).

IV. Conclusions

We propose an iterative scheme to generate custom instructions in an on-demand basis guided by the system-level performance requirements. Our approach zooms into the critical region that is causing the performance bottleneck and starts the customization process from that region. We provide a close coupling between the system-level design and the customization algorithm. Our efficient algorithm based on multi-level graph partitioning can generate the custom instructions on-the-fly. Experimental results validate that MLGP is quite effective in quickly producing good quality solutions. Therefore, MLGP is a perfect match with iterative scheme.

V. Acknowledgements

This work was partially supported by research grants R-252-000-387-112 & R-252-000-409-112.

References

[1] L. N. Chakrapani et al. *Languages and Compilers for High Performance Computing*, chapter Trimran: An Infrastructure for Research in Instruction-Level Parallelism. 2005.

[2] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC*, 1982.

[3] H. P. Huynh and T. Mitra. Instruction-set customization for real-time embedded systems. In *DATE*, 2007.

[4] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 1998.

[5] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 1970.

[6] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE TCAD*, 25(7), July 2006.