BURS-based Instruction Set Selection

Dmitri Boulytchev
St.Petersburg State University
Universitetskii pr., 28
St.Petersburg, Russia
db@tepkom.ru

ABSTRACT

Application-specific processors (ASIPs) look very promising in the domain of embedded systems since they comprise both flexibility of programmable device and efficiency of application-specific hardware. A number of approaches for automatic application-specific instruction set design were introduced during last years. We apply BURS technique that is commonly used for retargetable codegeneration to this problem. As a result the simple algorithm is presented that generates both instruction set and assembler code from the source program; this algorithm can be used for retargetable codegeneration as well.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.4 [**Programming Languages**]: Processors—*retargetable compilers*

General Terms

Algorithms, Design

Keywords

BURS, embedded systems, instruction set design, retargetable compilation

1. INTRODUCTION

Two main tasks have to be solved during development of embedded system based on application-specific instruction set processor. On the one hand the suitable architecture has to be developed; on the other hand compiler has to be retargeted to this architecture. Solving these tasks independently one may surprisingly discover that the problem of efficient code generation became undesirably hard. The reason is obviously that the compiler has to generate good code for *any* program; so compiler retargeting to application-specific architecture is an over-generalization of the initial problem.

The one way to avoid this glitch is to synthesize machine code together with an appropriate instruction set directly from source application; therefore the instruction set selection problem comes rise. Many different approaches have been developed in this area; we suggest yet another one.

Bottom-Up Rewrite Systems (BURS) is a simple conventional model for retargetable codegeneration. Crafting BURS to synthesize both machine code and instruction set we came up with an algorithm that seems to be good for prototyping: for given instruction-set constraint it provides an instruction set as a set of tree patterns that minimizes the cost of the machine program among all instruction sets satisfying that constraint. The algorithm can be used without any detailed knowledge of hardware properties. It is not a drawback since these properties are not yet clarified due to lack of the hardware.

2. RELATED WORKS

In general sense one may consider instruction set selection as a mapping

$$P \to (A, IS)$$

where P and A — source and target programs correspondingly, IS — instruction set (machine language). We also need this mapping to be semantic correct, that is

$$[P]_L = [A]_{IS}$$

where $[P]_L$ and $[A]_{IS}$ denote semantics of a program P in the source language L and semantics of a program A in target language IS respectively. Finally an instruction-set selection is said to be *optimal* iff it minimizes the value

$$\alpha \times C(A) + (1 - \alpha) \times C'(IS)$$

where $C(A)$ is a cost of target program, $C'(IS)$ is a cost of instruction set, and $0 \leq \alpha \leq 1$ is some coefficient to control a tradeoff between simplicity of an instruction set and simplicity of a program.

The above formulation is not conventional; however many works on instruction-set generation fit it. Instruction set synthesis for piplined architectures is addressed in [14, 13]. For given parameterized pipelined microarchitecture and the set of benchmarks the design of instruction set is considered as a sheduling problem. To control the tradeoff between instruction set and program quality number of cycles in the program and number of instructions in the instruction set

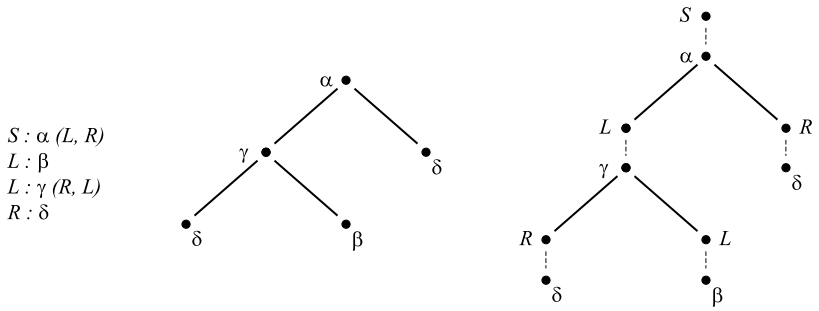

Figure 1: An example of tree grammar, ground tree and its partition.

are used. Sheduling problem is solved by simulated annealing algorithm.

Another approach [19] uses combined representation of datapath and instruction set model. Bundling technique is explored to couple sequencies of micro-operations and construct datapath. Initial datapath parts are taken from a predefined library; profiling is performed to determine frequently occuring operation sequencies.

Template generation as a way for instruction set selection is considered in [15, 3]. Templates are repeated occurencies of possible interdependent nodes and edges in a dataflow graph. Two types of templates are identified: sequential and parallel. Templates are built iteratively starting from pairs of adjacent nodes; initially most frequent pairs are selected. All selected templates are combined into supernodes; therefore DAG isomorphism algorithm is used to detect higher-order templates.

Another graph-based approach to instruction-set selection is presented in [6]. All reasonable patterns with regard to the set of constraints are enumerated to collect instruction candidates. Instruction set selection guided by some cost function is then performed. Finally the target application is mapped into the selected patterns by binate covering algorithm. The similar approach is used in [4].

3. TREE GRAMMARS AND BURS

Our approach to instruction set selection is essentially based on tree grammars and BURS theory so we have to introduce some formal notions.

Tree grammar is a context-free grammar $G = (N, T, S, R)$ where N and T are finite sets of nonterminals and terminals, $S \in N$ — starting nonterminal, R — set of rules of the form

$$K : p$$

where $K \in N$ — nonterminal, p — *tree pattern*, defined as follows:

1. x is a tree pattern where $x \in T$;
2. K is a tree pattern where $K \in N$;
3. $x(p_1, \ldots, p_k)$ is a tree pattern where $x \in T$ and all p_i are tree patterns.

Informally speaking tree pattern is an (ordered) tree with interior nodes labeled with terminals and leaves labeled with terminals or nonterminals. A *ground tree* is a pattern without nonterminal-labeled leaves.

While ordinary "linear" grammars define languages as a sets of words tree grammars define languages as a sets of ground trees. Similarly to the "linear" case one may define transition relation $p \xrightarrow{r} q$ for rule r and two patterns p and q as follows:

$$p \xrightarrow{r} q \; iff \; q = p[l \leftarrow t]$$

where r is $K : t$, l — leaf of p with label K, $p[l \leftarrow t]$ — substitution of l with t. The language defined by tree grammar G is the set of ground trees

$$\mathcal{L}(G) = \{t : S \rightarrow^* t\}$$

where "$\rightarrow^*$" is a reflexive-transitive closure of "$\rightarrow$". More datailed description of tree languages and automata theory can be found in [5].

For any ground tree $t \in \mathcal{L}(G)$ its derivation corresponds to some partition by patterns of G (see Figure 1). This observation explains why tree grammars are used as a convinient formal model for instruction selection problem: if we treat patterns of grammar as machine instructions and nonterminals as storage classes then any derivation of the tree corresponds to some legal instruction selection. Under these considerations the problem of optimal instruction selection for tree is essentially the problem of finding *least-cost derivation* in tree grammar with weighted rules.

Least-cost derivation can be found by dynamic programming algorithm that is in fact generalization of Aho-Johnson algorithm for finding optimal code for expression trees [2]; unlike the latter one it does not yield optimal register allocation. Hereafter we assume that tree grammar is presented in the following simplified form:

1. there are no rules that differ only by cost;
2. there are no chain rules;
3. each pattern is either x or $x(L_1, \ldots, L_k)$, $x \in T$, $L_i \in N$.

This assumption actually does not restrict class of defining languages but slightly simplifies the presentation.

The search is performed in two stages. During the first one, *labeling*, all possible rules are applied in bottom-up manner and costs of all derivations are calculated as presented below:

Label (t) :
if *t - leaf with label x*

then
mark t with all triples (K, r, c)
where r is rule K:x with cost c
else if *t - interior node with label x*
and immediate descendants $t_1, \ldots, t_k$
then
for each t_i **do** *Label* (t_i);
mark t with all triples (K, r, c)
where r is rule $K : x(L_1, \ldots, L_k)$ *with cost* c_0,
each t_i *is marked with triple* (L_i, r_i, c_i) *and*
$c = c_0 + c_1 + \cdots + c_k$ — *minimal cost*
among all such rules for K

It can easily be shown that some node v is marked by this algorithm with triple (K, r, c) if and only if the subtree with root v is derived from nonterminal K with minimal cost c and the first rule of this derivation is r.

During the second stage, *reduce*, least-cost derivation is reconstructed by top-down walk:

Reduce (t, K) :
select triple (K, r, c) from all marks of t;
add application of rule r to t into derivation;
if *t - interior node with label x*
and immediate descendants $t_1, \ldots, t_k$,
and r is $K : x(L_1, \ldots, L_k)$
then
for each t_i **do** *Reduce*(t_i,L_i)

Here we assume that the root of the tree is reduced with starting nonterminal.

Application of tree grammars to pattern matching and codegeneration is discussed in more details in [12, 1, 9, 10]. Due to work of Pelegrí-Llopart and Graham [18] this approach is oftenly being referred to as BURS (Bottom-Up Rewrite System) although some authors claim they are different [17, 16].

4. INSTRUCTION SET SELECTION AS A BURS PROBLEM

As we have seen in the previous section optimal instruction selection for a tree can be considered as a search of least-cost derivation in weighted tree grammar. Patterns of the grammar play role of individual machine instructions; any derivation introduces partition of the source tree. The inverted claim obviously does not always hold: there may be no derivation in given grammar that corresponds to arbitrarily choosen partition. Since any partition corresponds to some instruction selection with regard to some instruction set our first task is to build for given tree a grammar that allows all its patritions.

Definition 1. A tree p is said to be *patrial subtree* of the tree t if and only if $p[l_1 \leftarrow t_1, \ldots, l_k \leftarrow t_k]$ is a subtree of t for some leaves l_i and some trees t_i. In particular any subtree of t is its partial subtree. Clearly, the elements of arbitrary partition are partial subtrees.

Definition 2. Let $\{p_i\}$ be a set of trees. A pair (C, μ) where C is a tree, $\mu : nodes(C) \to \{p_i\}$ is said to be *partition* of tree t if and only if

1. $\forall x \in nodes(C)\ deg(x) = |leaves(\mu(x))|$;
2. $C[\mu] = t$ where $C[\mu]$ is defined as follows:
   - $x[\mu] = \mu(x)$ if $x \in leaves(C)$;
   - $x(t_1, \ldots, t_k)[\mu] = \mu(x)[l_1 \leftarrow t_1[\mu], \ldots, l_k \leftarrow t_k[\mu]]$ where $l_1, \ldots, l_k$ are all leaves of $\mu(x)$.

Informally speaking the partition of t is a tree C such that "expansion" of all its nodes into patterns of $\{p_i\}$ turns it into t; each node v of C is expanded into pattern $\mu(v)$. The height of partition (C, μ) is the height of C.

THEOREM 1. *Let* $S \to p_1 \to p_2 \to \cdots \to t$ *be a derivation sequence of tree t in grammar G. Then there exists partition* (C, μ) *of t by patters of G.*

PROOF. By straightforward induction on length of derivation sequence. □

Definition 3. Tree grammar $G_t = (N, T, S, R)$ is said to be *enumerating grammar* for t if and only if

1. for each patrial subtree p of t there is a rule
$$S : p[l_1 \leftarrow S, \ldots, l_k \leftarrow S]$$
in G_t where $l_1, \ldots, l_k$ are all leaves of p;
2. there is a rule $S : x$ for any $x \in T$ which is used as a label of some node of t.

THEOREM 2. *For any partition of tree t there exists corresponding derivation of t in enumerating grammar* G_t.

PROOF. By straightforward induction on partition height. □

This property of enumerating grammars can be interpreted as follows: for given tree t they describe *all possible* instruction selections in *all possible* instruction sets. So least-cost derivation in enumerating grammar is the least-cost instruction selection among all instruction sets.

The main restriction of the approach being discussed is introduced by limitation of cost function that has to be compliant with dynamic programming algorithm. It is quite simple to estimate cost of the program and quite hard to estimate the cost of instruction set with function of that kind. On the other hand we can not just skip instruction set cost. For example the standard cost function of program is its length; obviously the shortest program for given tree among all possible instruction sets contains one instruction — the tree itself. To avoid such a degenerative instruction sets from being considered we restrict the set of patterns that can be used in enumerating grammars by mean of some constraint. So such a constraint is a parameter of our algorithm.

To construct enumerating grammar we first construct auxilliary grammar $\tilde{G}_t$ with the following properties:

1. $\tilde{G}_t$ contains the rule $S : x$ for each $x \in T$;
2. for each nonterminal K, $K \neq S$, $\tilde{G}_t$ contains exactly one rule;
3. for arbitrary partial subtree p the pattern
$$p[l_1 \leftarrow S, \ldots, l_n \leftarrow S]$$
has a derivation in $\tilde{G}_t$ where $\{l_1, \ldots, l_k\}$ are all leaves of p.

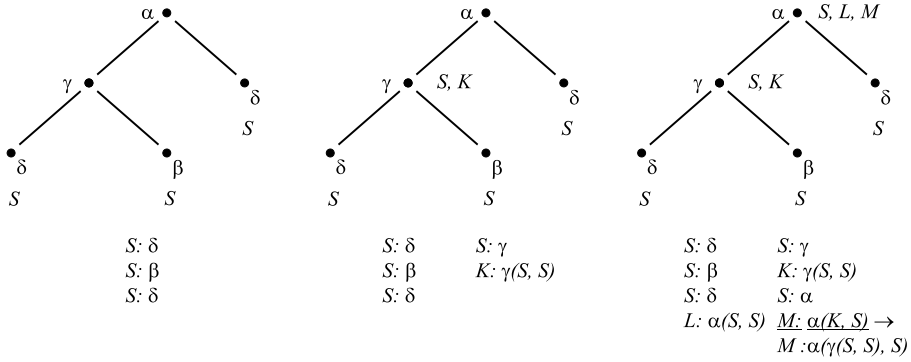

Figure 2: An example of enumerating grammar construction.

The grammar $\tilde{G}_t$ can easily be built via bottom-up breadth-first traversal of t. For each node v with terminal label x we first add a rule $S : x$ into $\tilde{G}_t$; then we apply all rules exactly like during labeling stage of BURS algorithm; finally we enrich $\tilde{G}_t$ with the rules of the form $K : x(L_1, \ldots, L_k)$, where K — new nonterminal, L_i — terminal mark of i-th immediate successor of v and there is no rule with the same pattern yet. In addition we skip all patterns that do not satisfy instruction set constraint. The exact procedure is shown below:

Enumerate (t, $\mathcal{C}$):
$R = \emptyset$;
for all $v \in t$ *in bottom-up breadth-first traversal* **do**
  if v — *leaf with terminal label* x
  then
    $R = R \cup \{S : x\}$;
    label v *with nonterminal* S
  else if v — *interior node with terminal label* x
      and immediate descendants $t_1, \ldots, t_k$
    then
      for all $L_1, \ldots, L_k$ *where* L_i *labels* t_i **do**
        if $K : x(L_1, \ldots, L_k) \in R$ *for some* K
        then *label* v *with* K
        else if $x(L_1, \ldots, L_k)$ *satisfies* $\mathcal{C}$
          then
            $R = R \cup \{K' : x(L_1, \ldots, L_k)\}$
            where K' — *new nonterminal;*
            label v *with* K'

The grammar built by above algorithm contains a separate nonterminal for each nontrivial patrial subtree. We can easily estimate number of all patrial subtrees in the tree of degree d in which all paths from root to leaves have length h.

Let $\mathcal{S}(t)$ be the number of all partial subtrees in tree t, $\mathcal{R}(t)$ — the number of those patrial subtrees that contain root of t. Then

$$\mathcal{S}(t) = 1 + \mathcal{R}(t) + \sum_{i=1}^{d} \mathcal{S}(t_i)$$

$$\mathcal{R}(t) = \prod_{i=1}^{d} \mathcal{R}(t_i)$$

where t_i are all subtrees of root of t; for leaf t_0 $\mathcal{R}(t_0) = 1$ and for a tree t_1 of height 1 $\mathcal{R}(t_1) = 2$. So,

$$\mathcal{R}(t) = \begin{cases} 1 & , \quad h = 0 \\ 2 & , \quad h = 1 \\ \underbrace{(\ldots((2^d + 1)^d + 1)^d + \cdots + 1)^d}_{h-1} & , \quad h \geq 2 \end{cases}$$

that yields a recurrent formula for number of all partial subtrees and the complexity estimation of the algorithm.

Clearly for each nonterminal K in $\tilde{G}_t$ except S the following is true:

1. there is the only one rule for K;
2. there is the only one partial subtree p such that

$$K \rightarrow^* p[l_1 \leftarrow S, \ldots, l_k \leftarrow S]$$

where $\{l_1, \ldots, l_k\}$ are all leaves of p.

So replacement rule for each nonterminal $K \neq S$ with rule

$$K \rightarrow^* p[l_1 \leftarrow S, \ldots, l_k \leftarrow S]$$

finalizes construction of the grammar G_t. An example of algorithm run and final transformation is shown on Figure 2. The underlined rule is the only rule in this example that has to be replaced.

After construction of the enumerating grammar one may apply conventional BURS algorithm described in the previous section to perform instruction selection[1]. The costs of the rules may be assigned in the commonly used manner; for example to minimize code length cost of 1 has to be assigned to the all rules. Least-cost derivation for enumerating grammar yields best instruction selection among all instruction sets with regard to constraint used during grammar construction. To select instruction set we finally have to move all the rules not participating in the least-cost derivation out of the grammar.

[1]Note that the grammar G_t formally does not satisfy the restrictions we have introduced for BURS algorithm; however the grammar $\tilde{G}_t$ does so we may use it instead with the same result.

Table 1: Evaluation Results for the DSPstone Benchmarks

benchmark	size	*triad*			*dual mem*			*mem/mul*			*no constraint*		
		I	A	R	I	A	R	I	A	R	I	A	R
complex_multiply	58	5	38	11	5	36	3	5	38	11	4	34	9
complex_update	78	6	62	14	6	50	4	6	62	14	5	46	13
convolution	70	6	42	8	5	39	2	6	42	8	5	39	7
dot_product	78	6	36	9	6	33	4	6	36	9	5	32	8
fir	83	8	54	12	7	48	2	8	54	12	7	48	11
fir2dim	149	8	147	23	8	136	4	8	147	23	6	132	21
biquad_one_section	86	8	57	14	7	48	2	8	57	14	7	48	14
biquad_N_sections	117	12	91	20	12	76	4	12	91	20	12	73	19
lms	156	9	77	15	9	65	4	9	77	15	8	64	14
matrix	76	9	87	18	9	74	4	9	87	18	8	72	17
matrix1x3	130	6	38	10	6	35	4	6	38	10	5	34	9
n_complex_updates	89	7	101	20	7	89	4	7	101	20	6	85	19
n_real_updates	69	6	56	11	6	53	4	6	56	11	5	52	10
real_update	66	5	28	5	4	25	4	5	28	5	3	24	4
fft	125	38	235	41	42	189	4	35	230	41	42	184	40
g721	866	84	1547	75	149	1049	3	101	1332	75	136	1002	73

With regard to formal definition given in Section 2 the approach in question provides an optimal instruction set selection for any BURS-compliant cost function for target program and the following cost function for instruction set:

$$C(IS) = \begin{cases} 0 & , \quad IS \text{ satisfies the constraint } \mathcal{C} \\ \infty & , \quad \text{otherwise} \end{cases}$$

where $\mathcal{C}$ is the constraint used during enumerating grammar construction phase. We may also note that our algorithm finds instruction set with minimal number of instructions if any nonterminal has unique occurence in the derivation defined by enumerating grammar; unfortunately this observation has rather a theoretic value.

5. EVALUATION

Described algorithm was implemented on top of ASDL-port of the lcc compiler [8, 7, 11]. Additionally to instruction set selection we had to implement local register allocator to store intermediate values and build a program from partition. The classic algorithm of Aho and Johnson [2] was used for this purpose.

We ran our algorithm for the DSPstone benchmark [20]. Four types of constraints were used:

1. *triad* — the number of nodes in pattern is limited by 3 that forces instruction set selector to generate triad-based instruction set;
2. *no constraint* — any pattern is allowed so each separate tree becomes machine instruction;
3. *dual mem* — no more than one occurence of multiplicative, logic or shift operations and no more than two occurencies of memory access operations are allowed in the pattern;
4. *mem/mul* — same as the above, but no simultaneous memory access operation and multiplication as allowed.

The length of the generated program was used as its' cost function.

The results of the evaluation are shown in the Table 1. Here I, A, and R stand for instruction-set size, machine program length, and number of registers. The number of registers is relatively large since we did not perform any clevel global register allocation and assign a dedicated register for each local variable, parameter or temporary. Such a naive allocation does not affect the instruction set selection results. Since our algorithm works only on basic blocks we use some predefined set of control flow instructions to express the control flow in the generated program. Despite the complexity of the algorithm running time for all these benchmarks did not exceed few seconds.

In the Figure 3 the examples of generated code for *convolution* benchmark are shown. Choosing various constraints one may perform fine-tuning of the instruction set in any desirable way; the algorithm is completely insensitive to the nature of the constraint so any test function may be implemented for this purpose.

6. DISCUSSION AND FUTURE WORK

Several drawbacks are natural to the presented approach.

First of all the model of tree covering looks restrictive since even within basic blocks any program is generally represented by a DAG. One has to perform common expression elimination to turn this DAG into forest. We may argue that due to simplicity of the algorithm it is desirable to try various ways of common subexpression elimination prior to instruction set selection.

The second drawback is that each instruction is implied to be a tree. We think there is no hope to adjust this algorithm to handle instructions as DAGs; however we may suggest to perform instruction set optimization to join simpler instructions into complex ones. So our algorithm may be considered as a mean of micro-code generation that is less *ad-hoc*.

To illustrate the possible way to improve generated instruction set by postpass optimization consider the following function:

```
int f (int x[], int y[], int i) {
  return x[i] + y[i];
}
```

Figure 3: An Example of Codegeneration Results

Source Code:

```
int main()
{
  static int x[16], h[16];

  int y, i, *px = x, *ph = &h[15];

  y = 0;

  for (i = 0; i < 16; ++i)
    y += *px++ * *ph--;

  return y;
}
```

Generated Code Under Different Constraints:

no constraint

```
       r2 = $x
       r3 = $h + %60
       r4 = %0
       r0 = r4
       r1 = r4
$loop: r5 = r2
       r2 = r5 + %4
       r6 = r3
       r3 = r6 + %-4
       r0 += [r5] * [r6]
       r1 += %1
       r1 ? %16
       jlt $loop
       return
```

no simultaneous indirection and multiplication allowed

```
       r2 = $x
       r3 = $h + %60
       r4 = %0
       r0 = r4
       r1 = r4
$loop: r5 = r2
       r2 = r5 + %4
       r6 = r3
       r3 = r6 + %-4
       r7 = [r5]
       r8 = [r6]
       r0 += r7 * r8
       r1 += %1
       r1 ? %16
       jlt $loop
       return
```

no simultaneous addition and multiplication allowed

```
       r2 = $x
       r3 = $h + %60
       r4 = %0
       r0 = r4
       r1 = r4
$loop: r5 = r2
       r2 = r5 + %4
       r6 = r3
       r3 = r6 + %-4
       r7 = [r5]
       r8 = [r6]
       r7 *= r8
       r0 += r7
       r1 += %1
       r1 ? %16
       jlt $loop
       return
```

The following machine program is generated by our algorithm by now:

```
r0 = r3 << %2
r4 = [r0 + r1] + [r0 + r2]
return
```

Note that the common subexpression `i << 2` occupies dedicated register `r0`; note also that the instruction `r4 = [r0 + r1] + [r0 + r2]` in the presented program depends on values of *four* registers while generally contains *five* arguments. Taking these considerations into account we may first specialize the instruction and then substitute occurence of register `r0` with corresponding expression. This transformation yields new instruction

```
rx = [ ry << %2 + rz] + [ry << %2 + rt]
```

where the subexpression `ry<<%2` is shared.

The cost function for instruction set in the current implementation is quite poor; it does not even allow to select shortest instruction set among all sets that yeild the same program quality. We think though that the algorithm can be extended to handle more sensible instruction-set costs.

All of these issues are subjects of future research.

On the other hand presented approach has some merits. It allows fast synthesis of instruction set and machine code without any special knowledge of microarchitecture; so it can be used to discover some intrinsic properties of the algorithm.

It produces proven-optimal code with regard to all formulated restrictions. Note that register allocator may be adjusted to aviod introducing anti-dependencies into generated code; so the scheduling will not require instruction selection change.

Finally for fixed instruction set IS this algorithm can easily be turned into code generator: it is enought to specify constraint of the form $p \in IS$, where p is an instruction candidate.

7. REFERENCES

[1] A. V. Aho, M. Ganapathi, and S. W. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.

[2] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, 1979.

[3] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 262–269, 2002.

[4] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. *Proc. 36th International Symposium on Microarchitectures*, 2003.

[5] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata/, 2002.

[6] J. Cong, Y. Fan, G. Han, and Z. Z. Z. Application-specific instruction generation for configurable processor architecture. *Proc. of the 12th International Symposium on Field Programmable Gate Arrays*, pages 183–189, 2004.

[7] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *ACM SIGPLAN Notices*, 26(10):29–43, 1991.

[8] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[9] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engeneering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, (3):213–226, 1992.

[10] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.

[11] D. R. Hanson. Early experience with ASDL in lcc. *Software — Practice & Experience*, 29(5):417–435, 1999.

[12] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1983.

[13] I. J. Huang and A. M. Despain. Generating instruction sets and microarchitectures from applications. *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, pages 391–396, 1994.

[14] I. J. Huang and A. M. Despain. Synthesis of instruction sets for pipelined microprocessors. *Proc. of the 31th ACM/IEEE Design Automation Conference*, pages 5–11, 1994.

[15] R. Kastner, A. Kaplan, S. O. Memik, and E. Bozorgzadeh. Instruction generation for hybrid reconfigurable systems. *ACM Trans. on Design Automation of Electronic Systems*, 7(4):605–627, 2002.

[16] A. Nymeyer and J. P. Katoen. Code generation based on formal BURS theory and heuristic search. *Acta Informatica*, 34(8):597–635, 1997.

[17] A. Nymeyer, J. P. Katoen, Y. Westra, and H. Alblas. Code generation = A* + BURS. In *Computational Complexity*, pages 160–176, 1996.

[18] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Proc. of the Fifteenth Ann. ACM Symp. on Principles of Programming Languages*, pages 294–308, 1988.

[19] J. V. Praet, G. Goossens, D. Lanneer, and H. de Man. Instruction set definition and instruction selection for ASIPs. *Proc. of the 7th ACM/IEEE International Symposium on High-Level Synthesis*, pages 11–16, 1994.

[20] V. Zivojnovic, J. M. Velarde, C. Schlaeger, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. *Proc. International Conference on Signal Processing Applications and Technology*, 1994.