SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(1), 1-12 (JANUARY 1991)

Hard-coding Bottom-up Code Generation Tables to Save Time and Space

CHRISTOPHER W. FRASER*
AT& T Bell Laboratories 2C-464, 600 Mountain Avenue, Murray Hill, NJ 07974-2070, U.S.A.

AND

ROBERT R. HENRY†
Department of Computer Science FR-35, University of Washington, Seattle, WA 98195, U.S.A.

SUMMARY

Code generators based on bottom-up rewrite systems (BURS) are automatically generated from machine-description grammars. They produce locally optimal code for expression trees, but their tables are large and require compile-time interpretation. This paper describes a program that compiles BURS tables into a combination of hard code and data. Hard-coding exposed important opportunities for compression that were previously hidden in the tables, so the hard-coded code generators are not just faster but also significantly smaller than their predecessors. A VAX code generator takes 21.4Kbytes and identifies optimal assembly code in about 50 VAX instructions per node.

KEY WORDS Code generation Compiling Pattern matching Retargeting Table compaction

INTRODUCTION

BURS code generators produce locally optimal code for expression trees. They walk a tree bottom-up and label each node with a numeric *state* that identifies the optimal assembler string to generate for that node. The states are computed by table look-up, in which the indices are the node's operator and the states already computed for the node's children. No dynamic programming is needed at compile time. Once all nodes have been labelled, a top-down tree-walk uses the states to emit code. Complex instructions can cover multiple nodes, so the states also tell which nodes to skip when emitting code.

The tables are generated from a machine-description grammar labelled with additive static costs. Table compaction is vital: for the VAX, the uncompacted data for *each* binary operator would take 2Mbytes. Even with compaction, a typical BURS

***Internet address: cwf@research.att.com**

†Partial support provided by NSF grant CCR-88-01806. Internet address: rrh@cs.washington.edu

0038–0644/91/010001–12$06.00

Received 12 January 1990
Revised 20 June 1990

code generator for the VAX takes over 43Kbytes, and further reductions are generally regarded as desirable. Compaction complicates table interpretation: what is logically a simple three-dimensional array access ends up taking about 40 VAX instructions.

This paper shows that it is better to represent BURS tables with a combination of hard code and data. A table encoder has been written, tested on complete machine-description grammars for the VAX and 68000, and integrated into a retargetable C compiler. Predictably, the hard-coded generator is faster. A typical compilation with interpreted tables took 49 VAX instructions/node for labelling and 188 more to identify the code to output. With hard code, these dropped to 13 and 35.

Less predictably, hard-coding saves space. The smallest hard-coded BURS code generator for the VAX takes 21.4Kbytes, less than half of its interpreted predecessor. Hard-coding exposed important opportunities for compression that were previously hidden in the tables, and it allowed tailoring of encodings to the values at hand. The table encoder offers options that trade space for time. The paper also describes some promising optimizations that turn out to be ineffective.

BACKGROUND

This paper is about representing BURS tables, not generating them, so readers interested in the latter should see the literature. Some papers describe basic tree-matchers, [1-4] and others describe their adaptation to code generation. [5-7] Briefly, the generator manipulates structures like LR(0) items, except that they record partial *tree* matches and also the cost of the associated code. The generator uses dynamic programming to compute optimal matches. The generator takes about 2.4G VAX instructions to build tables for optimal VAX code generation. A deeper understanding of table generation is not a prerequisite here, because the table encoder follows the table generator. A few modest changes were made to the table generator, but virtually all of the techniques that ultimately proved important are performed by a simple post-processor.

Code generators may assume that all nodes have at most two children. The table generator produces one two-dimensional table for each binary operator. At each binary node, a state is computed by indexing the appropriate table using the states already computed for the node's two children. Unary operators can be viewed as degenerate binary operators: a one-dimensional table replaces the two-dimensional one, and it is indexed by the state already computed for the node's lone child. Leaf operators can be viewed as degenerate unary operators: a zero-dimensional table—that is, a constant—replaces the one-dimensional table, so no look-up is needed. In other words, states labelling leaves are fixed by their operator.

Figure 1 gives ANSI C code to interpret uncompressed tables and label the tree at address `p`. Each `descriptor` structure records the arity, table, and number of table rows for one operator. When developing a new machine-description grammar, debugging code may be added to confirm that the operator and the row and column indices are legal. Once the grammar is complete, such code is superfluous, because the front end creates only valid trees, even if the source code contains errors.

Uncompressed tables are useful only for explanatory purposes. Typical tables driving a VAX code generator use about a thousand states, so the naïve representation for each two-dimensional table would require about a million 16-bit words.

```
label(struct node *p) {
  int k;
  struct descriptor *d = descriptors [p->op] ;
  switch (d->arity) {
  case 0: k = 0; break;
  case 1: k = label(p->left); break;
  case 2: k = label(p->left)*d->rowwidth + label(p->right); break
  }
  return p->state = d->table[k];
}
```

Figure 1. An interpreted labeller

In practice, however, most such tables would have only a few distinct rows and columns, so all BURS table builders have eliminated duplicates during table generation. This operation leaves a much smaller table, which is indexed indirectly via a vector of indices called an *index map.*[3] Each index map accepts a state and produces an index for the compressed table. With index maps, the expression in the binary case from Figure 1 becomes

```
d->lmap [label (p->left )]*d->rowidth + d->rmap [label (p->right )]
```

Figure 2 illustrates the introduction of index maps.

After this transformation, the index maps take most of the space. Different BURS table generators [5] have compressed them differently. One of the published code generators packs the elements of each index map into the smallest acceptable bit field whose length is a power of two. For example, if an index map stores row indices of a table with only six rows, then each index map element is stored in four bits. The values would fit in three bits, but three-bit chunks would cross byte boundaries and thus complicate random access to the index maps. Even with power-of-two field widths, unpacking the index maps complicates the table interpreter.

The base code generator for the current research [6] stores all index map elements in bytes (no machine-description grammar has produced a table with more than 256 distinct rows or columns) and saves space by *overlaying* index maps. When the table generator produces a new index map, it compares it with all existing maps. If it finds

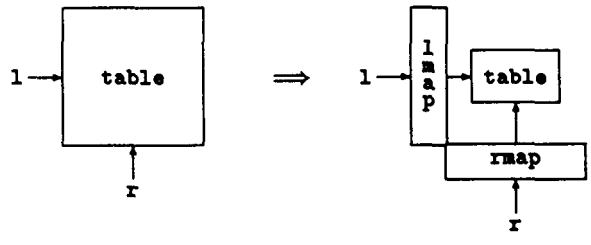

Figure 2. Remove duplicate rows and columns. Access compressed table via index maps

one that differs in fewer than, say, four positions, then the two descriptors are made to share the same index map. Each of the (up to) two index maps in each descriptor is augmented with a list of (up to) four exceptions, each of which is a state–index pair. The table interpreter scans the exception list before reading the index map. If it finds the state in the exception list, it uses the paired index instead of the value in the index map. This scan complicates the table interpreter, but it can reduce the number of index maps by over 20 per cent.

Once the labelling pass finishes, an output pass walks the tree top-down. It propagates down the tree *goal* symbols, which are numbers representing non-terminals in the machine-description grammar. These grammars are roughly analogous to a grammar for a machine's assembly language, so typical non-terminals define instructions and addressing modes. For example, the rule

instruction → `sub13` *src*, *src*, *dst*

uses non-terminals *instruction*, *src* and *dst* to define a subtraction instruction. Each rule comes with a tree pattern that describes the semantics of the instruction in terms of the operators in the compiler's intermediate language. The BURS tables are generated from these tree patterns.

The state that labels each node encodes the set of non-terminals that match the subtree rooted at that node. A state represents a set because some operators may be implemented several ways, depending on context. For example, the code generator may have a choice for some additions: to use an ordinary addition instruction or to fold the addition into the address calculation of another instruction. Thus each node's state specifies several assembler strings, and the goal symbol selects one of them.

HARD-CODING THE LABELLING PASS

The hard-coded labeller uses packing, exception lists and other techniques as well. As a first step, it replaces the descriptors by jumping to code tailored to the operator. Figure 3 gives a hard-coded labeller with an initial case for the subtraction operator. The code computes the row and column indices (`l` and `r`, respectively), which it then uses to access the state table for subtraction. The multiplication that supports the table access is placed with the reference to the index map so that it may be avoided when an exception is found (this particular case has no exceptions for the left subtree). An option that performed *all* multiplications at compile–compile time (by, for example, multiplying each entry in `xmap8` by four) was abandoned when it was determined to reduce the number of instructions executed by the code generators by less than 2 per cent on the VAX. On machines without multipliers, the encoder might be changed to restore this option or to implement the multiplications with shifts and additions.

The original BURS table generator was modified to enumerate states so that states in the table for an operator fall into a compact range. The range for the subtraction operator starts at 900, so the BURS table builder subtracts 900 from all elements in the table, and the labeller adds it back in.

This modification alone allows all but one table to use eight bits per element instead of sixteen; the table for `Plus` remains as a large table because there are

```
label (struct node *p) {
  int l, r;
  switch (p->op) {
  ...
  case 156:  /* sub */
    l = label (p->left)              /*label left sub-tree */
    l = 4*xmap8[l];                  /*map state to row index */
    r = label(p->right               /*label right sub-tree */
    switch (r) {                     /*implement exceptions . . . */
    case 2: r = 0; break;           /* ... map state 2 to index 0 */
    case 5: r = 2; break;           /* ... map state 5 to index 2 */
    default: r = xmap2[r]; break; /* ... otherwise use index map */
    }
    l = sub_table[l + r] + 900; /* fetch the final state */
    break;
  ...
  }
  return p->state = 1;
}
```

Figure 3. A hard-coded labeller

approximately 450 different ways to do addition optimally on the VAX, depending on context. Large tables are converted to small tables using a heuristic. The rows and columns are permuted so that infrequently occurring elements migrate to one corner; index maps preadjusted to follow the permutation. The table is divided into three rectangular *slabs* so that each slab contains no more than 256 distinct elements, and so the number of elements common between slabs is minimized. States are enumerated so that states in a slab fall into a compact range. The final state is determined by indexing the table and then using a short if–then–else chain to determine the slab and thus the value to add back in. Figure 4 illustrates the division of a large state table into slabs.

Figure 4. Permute and divide large state tables into slabs of 256 or fewer distinct values. Replace one range of 16-bit codes with several ranges of 8-bit codes

Next, degenerate cases are optimized. If the table has only one row, then | must be zero, so the assignments to | are dropped; the recursive call is retained, but the

assignment of the return value and the switch are not. A similar optimization holds for assignments to r when a table has only one column. In both cases, the corresponding index map can hold only zeros, so it is deleted. These optimizations collaborate to make unary cases look like the binary case above, without the r. Leaf tables have only one row and column, so their cases end up as a single assignment, of a constant to | .

Unary cases (and binary cases that degenerate to unary cases) are optimized still further. The transformations above would have them end with something like

```
l = xmap7 [l] ;              /* elements between 0 and m */
l = indir_table[l] + 720;    /* elements between 0 and n, n <= m */
```

This code is improved by replacing the index map with one that composes the original one (here xmap7) with the operator's state table (here indir_table). Then the table and the reference to it are deleted. Composition does not widen xmap7 because the state enumeration order ensures that the values in indir_table form a compact range. Figure 5 illustrates this composition.

Degeneracies make the code much smaller than the subtraction case suggests. For the tested VAX description, 11 of the 27 unary tables and 23 of the 53 binary tables have only one element, and 9 of the remaining binary tables have only one row or column. Moreover, most exception lists are empty, and all tables with only one row or column vanish. As a result, the code above is actually smaller than the descriptors that it replaces. Of course, it is also faster than the interpreter.

Finally, an encoder option controls a modest time–space trade-off. The integer codes that represent the operators are ordered so that the leaves come first, then the unary operators, and finally the binary operators. The option moves the labeller's recursive calls out of the cases to before the switch:

```
if (p->op >= MINBINARY) {
  l = label (p->left);
  r = label (p->right );
} else if (p->op >= MINUNARY)
  l = label (p->left );
```

The modest increase in time is due to the comparisons and branches that implement the if–then–else chain above.

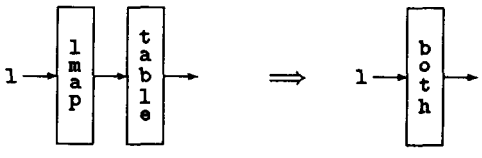

Figure 5. When a state table has only one row or column, compose it with its index map

HARD-CODING THE OUTPUT PASS

In interpretive BURS code generators, the output pass starts by using the goal symbol to index a table and fetch an index map and possibly an exception list, both just like those used during labelling. It then indexes *these* structures using the node's state number. The result is the number of a grammar rule with the same left-hand side as the goal symbol. The rule in turn specifies the assembler template, the registers to allocate and free, the descendants to visit and the goal symbols to pass down to those recursive calls.

The hard-coded code generator replaces these indexing operations with two switches. The first switch replaces the first table with a switch on the goal symbol. The second switch includes one case for each rule. The skeletal output routine appears in Figure 6. The cases in the first switch compute the rule number in much the same way that the labeller computes a state table index. A nested switch implements the exception list, if there is one, and its default case uses an index map. Exceptions can branch directly into the second switch because their values for rulenumber are constant.

The cases in the last switch make the recursive calls, generate the output, and allocate and free registers. A typical case might be

```
opcode = "sub13";
output (p->left, 33);    /* 33 is goalsym for left child */
output (p->right, 33) ;  /* 33 is goalsym for right child */
free any registers allocated to the children
allocate and use any registers needed by the destination for the current node
emit opcode and the addressing strings from the children and the destination
```

Many of these cases share suffixes, so careful cross-jumping is performed as they are generated, and steps are taken to improve opportunities for cross-jumping. For example, the variable opcode above is used so that the cases for the other binary operators can be implemented by assigning their own value to opcode and jumping to the second line above.

PACKING THE INDEX MAPS

Once the index maps for both the labeller and the output pass have been built and overlaid, the remaining maps are packed. Earlier techniques [5] packed each map independently, and used a totally different scheme for the output pass. The encoder packs all index maps into a single vector, in which the elements are C structures of bit fields, one field per map. For example, suppose that a code generator needs three maps, which index rows or columns of 6, 16 and 3 elements, respectively. The encoder would thus represent them as shown in Figure 7. Expressions such as the xmap2[r] in the case for subtraction are replaced with xmap[r] .f2. The random access is not across a packed dimension, so there is no need to round field sizes up to a power of two. Figure 8 illustrates index-map packing.

For the VAX description used in this research, the table generator starts with 209 different index maps. Using exception lists, overlaying and discarding index maps

```
output (p, goal)
struct node *p;
{
  switch (goal) { /* compute rulenumber from goal and p->state */
  case 1:
    switch (p->state) {
    case 23:  goto R43;
    case 26:  goto R38;
    default:  rulenumber = xmap17[p->state] + 19; break;
    }
  break;
  ...
  }
  switch (rulenumber) {
  case 1: R1:  /* generate code for rule number 1 */
  ...
  }
}
```

Figure 6. Skeletal output routine

```
struct {
  unsigned int f1:3; /* xmap1: range is 0..5 */
  unsigned int f2:4;  /* xmap2: range is 0..15 */
  unsigned int f3:2;  /* xmap3: range is 0..2 */
} xmap[NSTATES] = {
  {0, 4, 2}, /* state 0 */
  {5, 1, 1}, /* state 1 */
  ...
};
```

Figure 7. A packed index map

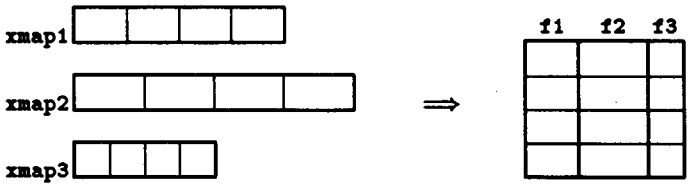

Figure 8. Transpose index maps and pack them into a single structure

used in degenerate cases retains only 38 maps. Each map is 963 bytes, for a total of 36,594 bytes. Packing the maps into structures yields 963 20-byte structures—that is, the encoder packs 38 map elements, one from each map, into 20 bytes—for a total of 19,260 bytes.

To translate xmap[r].f2, compilers emit code to multiply r by the width of a single element. For example, a map structure of 20 bytes requires multiplying r by 20. To reduce costs, the encoder actually divides the map structures into one-word (four-byte) chunks. For example, the 20-byte or 5-word structure used for the VAX compiler is actually represented by five one-word structures called *planks.* It replaces what would otherwise be a multiplication by 20 with one by 4, which may be realized with a shift or, on some machines, absorbed into an address calculation. Figure 9 illustrates this transformation.

Once the structure initializer in Figure 7 was generated, it was observed that the rows were not unique. The redundancy can be exploited in two distinct ways. One option exploits these redundancies at a low level, late in the encoder. This option directs the encoder to store only the distinct map structures, and to access them indirectly through an *indirection vector,* which accepts a state and produces an index into the list of distinct structures. For example, the VAX map has five planks, with 227, 148, 116, 178 and 63 distinct values. Thus the planked structure can be represented with 732 four-byte structures plus five indirection vectors of 963 bytes each, for a total of 7743 bytes, down from the original 36,594. Indices fit in one byte because each plank has fewer than 256 distinct values. Indeed, to keep the indices to a byte, the encoder breaks an almost-full plank and moves on to the next one when adding another field would give the plank more than 256 distinct values. It will resort, however, to 16-bit indices before it will leave a plank more than a third empty; more experience is needed to evaluate this heuristic. This form of indirection is called *late* because the encoder introduces it late (after planking) and because the table interpreter executes it late (immediately before the value is needed).

An independent option exploits the redundancy at a high level, via a modification to the original table generator. A state number is used in three places: twice in the labeller, to characterize a left or right subtree (see Figure 3), and once in the output pass, to specify a grammar rule (see Figure 6). No two states behave identically in all three places, but two states may cause identical labeller behaviour when they label a child. For example, the operators ‘<’ and ‘>’ need different state numbers because the output routine needs to emit slightly different strings for them, but the labeller treats these states the same because the overall code templates are so similar. The modified table generator detects such conditions and produces three sets of index maps: two for the labellers — one each for mapping the labels from left and

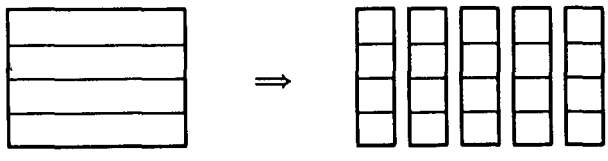

Figure 9. Divide the single-structure map of n *-byte elements into* [n/4] *maps of 4-byte elements*

Table I.

Indirection	*Trade-off*	*Space*					*Instructions executed*		
		label		*output*		*total*	*label*	*output*	*total*
		code	*data*	*code*	*data*				
none	space	2912	27,700	1740	0	32,352	16·8	35·4	52·2
early	space	2816	16,020	1784	7828	28,448	16·7	37·4	54·1
late	space	3208	16,188	2004	0	21,400	17·5	36·8	54·3
both	space	3060	15,036	2032	4880	25,008	17·2	38·8	56·0
none	time	4292	27,700	1740	0	33,732	13·4	35·4	48·8
early	time	4476	16,020	1784	7828	30,108	13·3	37·4	50·7
late	time	4604	16,188	2004	0	22,796	14·1	36·8	50·9
both	time	4808	15,036	2032	4880	26,756	13·9	38·8	52·7

right sub-trees—and one for the output routine. Each map holds only distinct values, and each comes with its own indirection vector. This form of indirection is called *early*, because the encoder may arrange to execute it earlier: the indirection is placed with the recursive calls, which may be done before the switch.

MEASUREMENTS

The encoder can produce eight variations on the same code generator. It can use early indirection, late indirection, both or neither. Independently, it can put the labeller's recursive calls before the switch or down in the individual cases. Table I gives the sizes (in bytes) and the average number of VAX instructions executed per node.

Thus the smallest version uses late indirection and one set of recursive calls before the switch, and the fastest version uses no indirection and calls down in the individual cases. The difference between them is about 12Kbytes, which could be important in the presence of limited memory, and about seven instructions per node, which is probably insignificant alongside the rest of the compiler. By comparison, a suboptimal greedy code generator for the VAX takes 10Kbytes.

Different specifications, even for the same machine, give different results, but the general patterns above remain. For example, the eight variants of the 68000 code generator hosted on the VAX are about 25 per cent smaller; the smallest still uses late indirection and one set of recursive calls before the switch. Nor does switching to a RISC host change matters. On the MIPS R3000, code sizes increase 30–50 per cent, but data sizes change little and continue to dominate the figures; late indirection with factored calls remains the smallest variation.

Early indirection was expected to save the most space, because it was closely connected with the internals of the table generation algorithms. It turned out, however, that late indirection's naive post-processor performs better. One reason may be that late indirection operates on planks, which are narrower and thus take on fewer distinct values. Another may be that late indirection produces a single set of planked index maps and their associated indirection vectors. Early indirection

produces three sets—one for the output pass and two for the labeller—with redundancy and fragmentation that cancel the advantage of their superior data.

The table omits the costs of building the tree in the first place and all costs from the second switch of the output pass, except for the recursive calls. These costs are omitted because they are constant across the eight versions and indeed are incurred by even non-BURS code generators. These costs are, however, not insignificant—indeed, they dominate the figures above. Code to lay down the output characters almost doubles the total sizes above, and it more than doubles the instruction counts.

Typical C compilers yield good object code for the hard-coded code generators. Changing the encoder to implement the hard-coded code generators in assembly code[8] might save a few instructions per node but probably not enough to materially improve the overall compilation rate. Changing the emitted C code, however, might be considered in a few cases. If the compiler does not assign crucial locals and formals to registers, then the encoder should use register declarations. If the compiler uses a slow calling convention, then standard techniques might be used to eliminate the recursion from the labeller and output pass. If the compiler rejects large switches or implements them badly, then it may be necessary to divide them.

The encoder is implemented as an appendage to the table generator for an earlier, interpretive BURS code generator. Its implementation is straightforward. It adds less than 10 per cent to the cost of generating the initial tables, so tuning seemed pointless. It takes the same amount of time to produce all eight versions of the hard-coded code generator.

DISCUSSION

BURS code generators run fast and yield locally optimal code for trees. The table generation algorithms are neither the simplest nor the least costly of the current generation of automatically produced code generators, but only one drawback has been exposed after compiler-generation time: the size of the code generators. Our table encoder largely solves this problem. It brings BURS code generators within a few thousand bytes of competing techniques, and it does so without sacrificing speed. Indeed, direct measurements show that the code generators above are smaller and faster than all previously published BURS code generators. The key is a selective use of hard code.

Late binding, flexibility and uniform treatment of operators make interpreters attractive during development and debugging. Once the tables are right, however, hard code is faster and easier to tailor to the data in hand. Although large switch statements with many branches interfere with optimizers, pipelines and caches, and the mere sizes can stress some compilers, eliminating a level of interpretation can offer considerable compensation.

The generated code, though not intended for human consumption, is actually easier to read than the original tables, and it should be easier to extend if the pattern matcher is used in different applications. For example, exception lists stand out more clearly as switches than as lists of numbers submerged in a huge initialized data structure, and the outer switches clearly delimit the handling of the individual operators.

The generated code benefits from many compiler optimizations: global dead code elimination, loop unrolling, cross-jumping, code hoisting and careful implementation

of switches. Regrettably, it was necessary to implement at least some of these in the encoder to compensate for some host compilers.

The encoder uses a few simple, general tricks:

1. Eliminate arrays that hold only a single value.
2. Eliminate replicated rows and columns. Access the reduced array indirectly via a vector of indices.
3. Use exception lists to eliminate nearly identical rows and columns. Compile them into hard code so that they impose no execution overhead when they are unneeded.
4. Pack multiple equal-length arrays of small numbers into a single structure to waste fewer bits.
5. Transpose tables to avoid random access along a packed dimension.
6. When tables are indexed solely by values from other tables, permute rows and columns to expose more regularities.

An interpreter could use some of the transformations, but it would have to accommodate more representations and thus incur more overhead. Hard code makes the overhead vanish.

These tricks apply to more than just BURS tables, and the hard code generated to support them resembles that generated for very different compiler tables. [8-10] It is thus perhaps time for a general-purpose table encoder, which could accept tables and automatically perform some of the transformations above.

REFERENCES

1. H. H. Kron, 'Tree templates and subtree transformational grammars', *Ph. D. Thesis*, Information Sciences Department, University of California, Santa Cruz, December 1975.
2. Christoph Hoffmann and Michael J. O'Donnell, 'Pattern matching in trees', *Journal of the ACM*, **29** (1), 68–95 (1982).
3. David R. Chase, 'An improvement to bottom-up tree pattern matching', *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, January 1987, pp. 168-177.
4. Jürgen Börstler, Ulrich Mönche and Reinhard Wilhelm, 'Table compression for tree automata', *Technical Report Aachener Informatik-Berichte No. 87-12*, RWTH Aachen, Fachgruppe Informatik, Aachen, Federal Republic of Germany, 1987.
5. Eduardo Pelegri-Llopart and Susan L. Graham, 'Optimal code generation for expression trees: an application of BURS theory', *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, January 1988, pp. 294–308.
6. Robert R. Henry and Peter C. Damron, 'Algorithms for table-driven code generators using tree-pattern matching', *Technical Report 89-02-04*, Department of Computer Science, University of Washington, February 1989.
7. Robert R. Henry, 'Encoding optimal pattern selection in a table-driven bottom-up tree-pattern matcher', *Technical Report 89-02-04*, Department of Computer Science, University of Washington, February 1989.
8. Thomas J. Pennello, 'Very fast LR parsing', *Proceedings* of *the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, **21** (7), 145–151 (1986).
9. Christopher W. Fraser and Alan L. Wendt, 'Automatic generation of fast optimizing code generators', *Proceedings of the SIGPLAN '88 Symposium on Compiler Construction, SIGPLAN Notices*, **23** (7), 79–84 (1988).
10. Christopher W. Fraser, 'A language for writing code generators', *Proceedings of the SIGPLAN '89 Symposium on Compiler Construction, SIGPLAN Notices*, **24** (7), 238–245 (1989).