Chapter 6

BAM Transformations

1. Introduction

After compiling the program from kernel Prolog into BAM code, a series of optimizing transformations is performed. The transformations performed are: (1) duplicate code elimination, (2) dead code elimination, (3) jump elimination, (4) label elimination, (5) synonym optimization, (6) peephole optimization, and (7) determinism optimization. This chapter first gives two definitions and then presents the transformations.

2. Definitions

The following two definitions are useful:

> **Definition DB:** A *distant branch* is a branch that always transfers control to an instruction other than the next in the instruction stream.

According to this definition, there are exactly four distant branches in the BAM: fail, return, jump, and switch. All other branches do not satisfy the definition since they can fall through to the next instruction.

> **Definition BB:** A *contiguous block* is any sequence of instructions that terminates with a distant branch.

According to this definition, a contiguous block can start with any instruction and can contain conditional branches with a fall through case. Therefore the code contains a large number of overlapping contiguous blocks. This is useful to get maximum optimization when looking for contiguous blocks that satisfy some property. The individual transformations mentioned in this chapter will usually only look at contiguous blocks satisfying certain constraints, for example, the contiguous blocks that begin with a label.

3. The transformations

Seven transformations (Figure 6.1) are done on the BAM code generated for each predicate by the kernel to BAM compilation stage. A transitive closure is performed on the sequence of seven transformations, i.e. they are applied repeatedly until there are no more changes. Each transformation is carefully

coded to result in code that is better (i.e. faster or shorter) than its input, so the closure operation terminates.

Figure 6.1 – BAM Transformations

3.1. Duplicate code elimination

All duplicate contiguous blocks except the last occurrence are replaced by a jump to the last one. This optimization is also known as *cross-jumping*. It tightens up loose code generated by the type enrichment transformation (Chapter 4). It is implemented by first creating an table indexed by all contiguous blocks that (1) begin with a label, (2) do not contain any other labels (but they are allowed to contain branches), and (3) are not degenerate blocks that consist of only a single jump, return, or fail instruction (but a single switch is allowed). The table contains the label of the last occurrence of the block. All con-

tiguous blocks in the code, including those that do not begin with labels, are looked up in the table and replaced by jumps if they are not the last occurrence. The result of this optimization is to reduce code size at the price of slightly slowing down execution.

3.2. Dead code elimination

All code that is not reachable from the entry point of a predicate is removed. This is done in two steps: First, all the labels that are reachable through any number of branches are calculated by doing a transitive closure. Second, a linear traversal of the code is done and the instructions following a distant branch up to the next reachable label are eliminated.

3.3. Jump elimination

Rearrange contiguous blocks to minimize the number of jump, call, and return instructions. This optimization is a variant of the *jump chaining* optimization. A transitive closure is done on the following replacements:

(1) Replace a jump by the contiguous block it points to if the block is only pointed to by one branch or if the block is shorter than a preset threshold. The threshold can be changed by a compiler directive. The replacement is not done if the block is part of write mode unification or unification with an atom, since these two cases are hurt by the transformation.

(2) Replace a call to a dummy predicate by the code for the predicate if it is straightline code, i.e. its code consists only of non-branches, call instructions, and branches all of whose destinations are `fail`. The predicate's code must be terminated by a `return` or `fail` instruction.

(3) Replace a conditional branch to a conditional branch by a new conditional branch if possible. The only case currently recognized is:

```
test(ne,tvar,V,L).
 ...
label(L).
switch(Tag,V,fail,L2,L3).
```

which causes the test instruction to be replaced by:

```
switch(Tag,V,L1,L2,L3).
label(L1).
```

(4) Replace a branch one of whose destinations is a jump or fail instruction by a new branch identical to the original one except that the destination label is replaced by the destination label of the jump or by fail.

3.4. Label elimination

Remove all labels that are not jumped to by any branch in the code. This is done in two steps: First, the set of all destinations of all branch instructions is collected. Second, the labels not in this set are removed from the code.

3.5. Synonym optimization

This transformation is similar to strength reduction. It does a linear traversal of the code and replaces every addressing mode by the cheapest addressing mode that contains the same value. For example, if `p(1)` and `r(0)` contain the same value, then an occurrence of `p(1)` can be replaced by `r(0)`. The following cost order (from cheapest to most expensive) is used by default and is based on the cost in the VLSI-BAM architecture:

Addressing mode	Reason for cost	Overhead (cycles)
`r(b)`	Promotes creation of `cut(r(b))` which is a no-op	0
`r(I)`	Usable without overhead	0
`Atom`	Requires ldi (load immediate) instruction	1
`Tag^X`	Tagged pointer creation needs lea (load effective address) instruction	1
`p(I)`	Permanent variable needs ld (load) instruction	1
`[r(I)]`	Indirection needs ld (load) instruction	1
`[r(I)+N]`	Offset indirect needs ld (load) instruction	1
`[p(I)]`	Indirect permanent needs 2 ld (load) instructions	2
`[p(I)+N]`	Offset indirect permanent needs 2 ld (load) instructions	2
`r(void)`	Most expensive because it must not be changed	–

The reason given for the cost describes the instructions necessary to implement the addressing mode for the VLSI-BAM. More information on the instruction set of the VLSI-BAM is given in [34]. The addressing mode `r(void)` is created by the register allocator. It corresponds to a void variable, i.e. a variable that occurs only once in a clause and whose value may therefore be ignored. It is made the most

expensive because it must remain unchanged so that peephole optimization can remove the instruction containing it.

The synonym optimization is implemented by maintaining a set of equivalence classes at all points of the program, where each equivalence class is a set of addressing modes whose values are identical. Labels in the code cause the set of equivalence classes to be reset to empty. A future extension of this module could eliminate this restriction by following the labels and performing a transitive closure, resulting in a slight performance gain.

3.6. Peephole optimization

A transitive closure is performed on a peephole transformation with a window of three instructions. The set of patterns was determined empirically by looking at the compiler's output and adding patterns to fix obvious inefficiencies. Each pattern is implemented as a single clause in the optimizer. The patterns are one, two, and three instructions long. However, the window is extended to arbitrary size for one pattern, a generalized last call optimization:

```
call(N/A).
deallocate(I). % Arbitrary number of deallocate instructions.
 ...
deallocate(J).
return.
```

which is transformed to:

```
deallocate(I). % Same sequence as above.
 ...
deallocate(J).
jump(N/A).
```

3.7. Determinism optimization

A choice instruction is removed if it is followed by a sequence of instructions that cannot fail and a cut instruction. This simple-looking optimization significantly increases determinism—many predicates (e.g. Warren's quicksort benchmark) containing a cut become deterministic that would otherwise be compiled with a choice point.

A similar optimization is performed by the simplification transformation of kernel Prolog (Chapter 4). For example, it transforms `(!,p ; q)` into `(!,p)`. The determinism optimization extends simplification—if the goal `s` compiles into instructions that cannot fail then it is able to successfully optimize the BAM code of `(s,!,p ; q)` even when simplification cannot determine that `s` always succeeds.

Consider this predicate, which contains no cut:

```
:- mode((max(A,B,C) :- uninit(C))).  % C is unbound and unaliased.

max(A, B, C) :- A<B, B=C.            % No cut here.
max(A, B, C) :- A=C.
```

It is compiled into the following BAM code (slightly simplified for readability):

```
procedure(max/3).
        deref(r(0),r(0)).
        deref(r(1),r(1)).
        jump(lts,r(0),r(1),l(max/3,1)).  % Conditional branch A<B.
        move(r(0),[r(2)]).               % A<B is false.
        return.
label(l(max/3,1)).
        choice(1/2,[0,2],l(max/3,4)).    % A<B is true.
        move(r(1),[r(2)]).
        return.
label(l(max/3,4)).
        choice(2/2,[0,2],fail).
        move(r(0),[r(2)]).
        return.
```

When `A<B` is true, a choice point is created to try both clauses. If a cut is inserted into the first clause:

```
:- mode((max(A,B,C) :- uninit(C))).  % C is unbound and unaliased.
max(A, B, C) :- A<B, !, B=C.         % Cut is added here.
max(A, B, C) :- A=C.
```

then the code becomes deterministic:

```
procedure(max/3).
        move(b,r(3)).
        deref(r(0),r(0)).
        deref(r(1),r(1)).
        jump(lts,r(0),r(1),l(max/3,4)).  % Conditional branch A<B.
        move(r(0),[r(2)]).
        return.
label(l(max/3,4)).
        cut(r(3)).
        move(r(1),[r(2)]).
        return.
```

Measurements done by Touati [70] justify this optimization. He finds that it makes about half of all choice point operations avoidable.