Software Optimization Using Hardware Synthesis Techniques

Bret Victor, *bret@eecs.berkeley.edu*

Abstract— *Although a myriad of techniques exist in the hardware design domain for manipulation and simplification of control logic, a typical software optimizer does very little control restructuring. As a result, there are missed opportunities for optimization in control-heavy applications. This paper explores how various hardware design techniques, including logic network manipulation, can be applied to optimizing control structures in software.*

I. INTRODUCTION

SOFTWARE running in an embedded system must often examine and respond to a large number of input stimuli from many different sources. Because a processor is a time-multiplexed resource, it cannot process these input signals in parallel as a hardware-based design can. Thus, a significant percentage of computing time is spent traversing control structures, determining how to respond to the given set of inputs. It is possible that an especially control-heavy embedded application might spend more time figuring out what to do than actually doing it!

However, the optimization phase of a typical compiler is primarily directed at data flow, with the intention of speeding up data-processing applications and loop-based structures. And while "code motion" is certainly a valid and utilized concept in software optimization, nowhere do we see the sort of radical control restructuring that a typical hardware optimizer performs. A logic manipulation package intended for hardware design will rewrite logic equations and create and merge nodes with wild abandon, whereas the output of a software compiler is generally true to the control structures in the original source code.

This paper discusses various ways in which techniques from the realm of hardware design can be applied to the optimization of control structures in software. First, two local optimization techniques, the Software PLA and Switch Encoding, are presented. These allow for a more efficient evaluation of complex logic equations and large *if/else* structures respectively. Then, a general method for restructuring a software routine using logic networks is introduced, along with a discussion of the software package that has been developed as an implementation.

II. LOCAL HARDWARE-INSPIRED TRANSFORMS

Logic Minimization

Consider the expression in the following *if* statement:

```
if ((a&&c)||(b&&(c||d))||(d&&a))
```

A compiler would parse this into an expression tree, and generate code that would directly traverse the tree and evaluate the expression as given. "Short-circuit" branching might be used, but no inspection and modification of the boolean expression itself would typically take place.[1]

However, no hardware compiler would implement this expression without first running it through a logic minimizer. If we give this expression to ESPRESSO and factor the result with a factorization algorithm, we see that the above can be rewritten equivalently as

```
if ((a||b)&&(c||d))
```

This new expression takes half as many boolean operations to evaluate as the previous one. While simple logic minimization may seem like an obvious technique, neither compilers nor programmers generally do it. It should be noted that expression simplification should only be attempted on expressions where evaluation of the operands causes no side effects, such as function calls. Otherwise, the programmer may be relying on the short-circuit semantics of C's boolean operators to conditionally modify the state of the system.

Software PLA

In evaluating the above expression, each variable is treated as boolean, representing either a true or false value. It may seem wasteful that on a machine with a 32-bit (or even 8-bit) data word, only one bit is being used at a time. The Software PLA is a method for evaluating logic expressions that attempts to use more of the data word width, and effectively evaluate parts of the expression in parallel. It

[1] Short-circuit boolean evaluation is less useful in embedded applications than in general-purpose computing. An embedded application typically has to meet a set of realtime constraints, and additional performance past these constraints is not beneficial. Thus, the speed of embedded software must be measured with the worst case performance, which is not affected by short-circuit operators.

is modeled after a PLA (programmable logic array), a hardware structure which breaks an expression into a sum-of-products form, calculates all the product terms in parallel, and then ORs them together. Consider this boolean expression, in sum-of-products form:

$$abc + \bar{a}cd + b\bar{c}$$

The first step is to create a table of "bit masks". There is one row for each unique literal that appears the expression, and the columns of the table correspond to the product terms. A row has a 0 in a particular column if that product term contains that literal, and a 1 if it does not. The bit masks for this example are shown in Figure 1.

We assume that all of the bits of an input variable are either 0 or 1, depending on that variable's value. If this is not the case, they can be trivially transformed into such a representation. The evaluation procedure begins with initializing an evaluation valuable to all 1's. This variable is then ANDed with an input value ORed with its bit mask. This is done for each input variable, in both the positive and inverted sense if necessary. At the end of this process, if the evaluation variable is all 0's, the expression evaluates to 0; otherwise it evaluates to 1. This last step, equivalent to the OR stage in a PLA, can be implemented with a simple test for equality to zero. The first few steps of an example procedure are shown in Figure 1. Note that this same technique, with slight modifications, can be used to evaluate an expression in product-of-sums form, in case that is handier for the particular expression.

Evaluation of a Software PLA requires two instruction for inputs used in the positive sense and three instructions for inverted inputs. Evaluation of an expression in the conventional manner requires one instruction per boolean operation, which includes ANDs, ORs, and inverting. If we have an sum-of-products expression with n literals and m product terms, and assume half the literals are inverted inputs and each product term contains half the literals, we find:

$$ops_{PLA} = 2.5n + 1$$
$$ops_{SOP} = (0.75n - 1)m + (m - 1)$$
$$ops_{PLA} < ops_{SOP} \text{ when } m > 4$$

Thus, the Software PLA is better than a direct evaluation of a sum-of-products expression when there are a large number of product terms. How it compares to the best factored form of a given expression cannot be determined in general, but there are cases when it does provide an improvement, especially for expressions that do not factor well. For example, a four-input XOR, implemented with ANDs and ORs, requires

- 47 operations in sum-of-products form
- 32 operations in factored form
- 21 operations as a Software PLA.

	a b c	a' c d	b c'
a	0	1	1
a'	1	0	1
b	0	1	0
c	0	0	1
c'	1	1	0
d	1	0	1

```
s = 111       ; initialize s

x = a OR  011 ; mask for a
s = s AND x   ; apply mask

x = a XOR 111 ; invert a
x = x OR  101 ; mask for a'
s = s AND x   ; apply mask

x = b OR  010 ; mask for b
s = s AND x   ; apply mask
```

FIGURE 1: BIT MASKS AND EVALUATION CODE

Switch Encoding

Consider the following software structure:

```
if (X)       { func_0 (); }
else if (Y)  { func_1 (); }
else if (Z)  { func_2 (); }
else         { func_3 (); }
```

where *X*, *Y*, and *Z* are expressions using some set of input variables. The function call statements are mutually exclusive — only one will be executed. The worst-case evaluation time of this structure is slow, because *X*, *Y*, and *Z*, potentially complicated expressions, all have to be evaluated in order to execute *func_3 ()*.

A switch structure is another mechanism for executing one of a set of mutually exclusive statements.

```
switch ( i ) {
  case 0:   func_0 ();  break;
  case 1:   func_1 ();  break;
  case 2:   func_2 ();  break;
  default:  func_3 ();  break;
}
```

This structure, at least when switching on close-to-consecutive values, executes much faster than an *else* chain because it is implemented with a table lookup. However, it requires an integer operand to switch on. If we think of this integer as simply an array of bits, then we can generate it in much the same way that a hardware designer implements a state machine. In this four-case example, *i* is effectively two bits wide. Bit 0 is high when *i* is 1 or 3, which correspond to *func_1 ()* and *func_3 ()* respectively, and bit 1 is high when *i* is 2 or 3. The conditions when each statement should be executed can be derived from *X*, *Y*, and *Z*, and boolean equations for each bit can be generated:

$$i\_bit\_0 = \bar{X}Y + \bar{X}\bar{Y}\bar{Z}$$
$$i\_bit\_1 = \bar{X}\bar{Y}$$

These equations can be minimized with a logic minimization tool in terms of the primary inputs, and implemented simply as:

```
i = (i_bit_1 << 1) | (i_bit_0);
```

It is important to note that when constructing the switch structure, the statements need not be numbered consecutively. Any distinct value of *i* can be assigned to any statement, or equivalently, the statements can be reordered

arbitrarily. Thus, we can attempt to find a numbering that minimizes the logic required to generate *i*. This is exactly what hardware designers do when devising a state encoding based on a set of next-state equations. Thus, any algorithms that address the state encoding problem can also be applied to this sort of software optimization.

III. PROCEDURE OPTIMIZATION BY LOGIC NETWORK SIMPLIFICATION

In the design phase, a digital circuit is generally represented as a network of nodes, each of which performs a particular logic function. These nodes can be manipulated with CAD tools until a representation of the circuit that is optimal in some sense is reached. The nodes can then be mapped into digital gates, and a circuit that performs the desired logic function is thus implemented.

Whereas hardware can be easily viewed as a network of nodes, the connection between a software program and a logic network is less apparent. Nevertheless, if software could somehow be put into this representation, there is the potential to leverage the large amount of research, not to mention CAD tools, in the area of logic network optimization.

For this purpose, a tool named BRO was developed.[2] BRO attempts to decompose the control structure in a software procedure into logic network, optimize it, and then rebuild the procedure using the new control structure. It uses the excellent SUIF compiler and intermediate representation format[3], and is partially implemented as modules that run within the SUIF environment. The following process is used when optimizing a software program using BRO:

- C source code is compiled to SUIF format
- The BRO frontend creates a logic network from the SUIF file
- The SIS package is used to simplify the network
- The BRO backend reconstructs the SUIF code using the new network
- SUIF outputs the new program as C source[4]

BRO Frontend

The task of the BRO frontend is, given a software procedure with a possibly hierarchical collection of *if/then/else* structures, to extract a logic network that completely captures the control flow and is flexible enough to allow for optimization.

The inputs to the logic network are the control variables in the procedure, that is, all variables that are used within the condition expression of an *if* statement. The outputs of the logic network correspond to blocks of non-control statements in the code. An output evaluates to 1 if the block of code that it represents should be executed, given the current set of input values.

It may be helpful to refer to Figure 2 throughout the following discussion. BRO's network generation process begins at the top of the procedure. When it encounters an *if* statement, it generates two nodes, a *then* node and an *else* node. These are named *node_n_t* and *node_n_e* respectively, where *n* is a number that is incremented with each *if* statement. These nodes represent whether the *if* statement's *then* or *else* clause should be executed, and consist of the condition expression and its inverse, ANDed with the parent node if this *if* statement is nested within the *then* or *else* clause of another *if* statement. When BRO comes to a non-*if* statement, it gathers as many consecutive such instructions as possible into a block, and generates an output node. The output represents whether this block should be executed, and its node equation is simply the parent node if the block is within a *then* or *else* clause, or a constant 1 otherwise. Outputs are named *output_n*, where n is again incremented consecutively. When BRO has finished traversing the procedure, it has generated a set of node equations that represents a logic network.

Using these equations, it is possible to determine the set of input values for which any given statement in the procedure will be executed. The network is thus a complete representation of the procedure's control structure, and together with the statement blocks associated with each output, contains enough information to construct a software procedure with functionality identical to the original.

Control Variable Modification

Due to the sequential nature of software execution however, a complication arises. Consider the code in Figure 3. The variable *b* is used in the conditions of two *if* statements, but it is modified between them. If BRO ignored this, it would generate a network that assumed that both uses of *b* had the same value, which is not necessarily true. This could lead to incorrect behavior after the network had been manipulated and then transformed back into software.

Although it would be convenient to assume that

`if ( a && b) {`	*node_1_t = a & b, node_2_e = !(a & b)*
`  if (c)`	*node_2_t = c & node_1_t,*
	node_2_e = !c & node_1_t
`    x = 1;`	*output_1 = node_2_t*
`  else`	
`    x = 2;`	*output_2 = node_2_e*
`}`	
`else`	
`  x = 3;`	*output_3 = node_1_e*

FIGURE 2: CODE EXAMPLE AND GENERATED NODES

[2] This tool, which uses logic networks to optimize software, was seen as somewhat of a duel to SIS, which uses logic networks to optimize hardware. Hence, the name.

[3] See *http://suif.stanford.edu* for information on SUIF.

[4] The standard SUIF distribution does not come with an assembly language backend.

```
if ( a && !b)
  x = 1;
b = new_b();
if ( !b && c)
  x = 2;
```

FIGURE 3

```
if ( a && !b)
  b = 1;
else {
  if ( b )
    z = 3;
  b = new_b();
  if ( b )
    y = 7;
}
```

FIGURE 4

control variables are never modified in the body of a procedure, this is certainly not the case, and such an assumption would place an unfair restriction on the programmer.[5] The solution lies in realizing that although both condition expressions refer to a variable *b*, the value used in one expression has no connection to the value used in the other, and thus they are effectively independent inputs to the network. BRO makes sure that they go by different names by labeling each input with the output number in which its variable was last modified. The variable starts with the name *b_0*, which is used in the nodes generated by the first *if* statement. When the statement that modifies *b* is encountered, the name is set to *b_2*, because that statement is part of *output_2*. The nodes generated by the second *if* refer to the variable by its new name.

Now, consider the code in Figure 4. In the *then* clause, the input's name is set to *b_1*. In the *else* clause, the input starts out as *b_0* (because what occurs in the *then* clause cannot affect it), and becomes *b_3*. Thus, the question arises of what the name should be after the *if* statement ends. If the *then* clause were executed, the name should be *b_1*; if the *else* clause were executed, the name should be *b_3*. Fortunately, BRO has that information available in the form of *node_1_t* and *node_1_e*, and it generates a node:

$$node\_b\_1 = (b\_1 \,\&\, node\_1\_t) + (b\_3 \,\&\, node\_1\_e)$$

This node is then used as the new name for the variable *b*. Thus, whenever BRO leaves an *if* statement where a variable was modified in one or both of the clauses, it generates a node that consolidates the two possible names for the variable at that point. This technique allows the network inputs to be tracked as they go through the control structure, and gives the network minimizer enough information to successfully simplify the network.

Integer Control Variables

So far, all control variables have been boolean. But it is certainly desirable in many applications to use integer variables inside *if* conditions. For example:

```
if ( (i < 4) || (i == 7) )  { … }
```

There are three ways of dealing with the issue of integer control variables. The first is to simply disallow them. This places no restriction upon the range of programs that can be implemented, because the programmer can always rewrite the above line as:

```
temp1 = ( i < 4 );
temp2 = ( i == 7);
if ( temp1 || temp2 )  { … }
```

That is, temporary boolean variables can be created with the results of the comparisons and used in the condition expression. However, this is undesirable for several reasons. First of all, we can clearly see that *temp1* and *temp2* cannot both be true simultaneously, because *i* cannot be both less than four and equal to seven. However, for BRO to determine this from the above code, it would have to have overly extensive dataflow analysis capabilities. So this simple information, which is vital for network simplification, is lost. Secondly, it is simply inefficient and tedious for the programmer to have to write code in this form.

A possibility that would allow for integer control variables would be to use a multi-valued (MV) network. All integer comparisons then could be expressed in terms of MV literals. For instance, the example above would generate the following node:

$$node\_1\_t = i\_0^{\{0,1,2,3\}} + i\_0^{\{7\}}$$

This approach has the advantage that all information about the use of the integer variable is captured within the representation, so an optimal network simplification is possible. There are practical downsides, however. It would be necessary to explicitly specify a range for every integer variable so an MV input of the appropriate width could be constructed. This is not easy in C, and would require either a mechanism external to the language or an awkward substitution of *enum*s for *int*s with all control variables. Again, this would entail a tedious modification of existing source code. Another disadvantage of using a MV network is that it requires the use of a MV network manipulation package.

The solution that is implemented in BRO is "comparison inputs". When BRO encounters our example line above, it generates the following node:

$$node\_1\_t = i\_0\_compare\_lt\_4 + i\_0\_compare\_eq\_7$$

That is, it creates two new boolean inputs to the network, and names them according to the required comparison. However, BRO can examine the comparisons (or even just the names of these inputs) and see that they cannot both be true because of a satisfiability constraint. It then is able to express this information to the network minimizer through the use of a don't care network. It simply adds the cube:

$$i\_0\_compare\_lt\_4 \,\&\, i\_0\_compare\_eq\_7$$

to the network's external don't care set, and the minimizer can then optimally simplify the network. For example, if we have the code in Figure 5, the generated don't care between

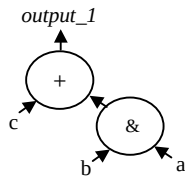
[5] State-based languages, such as Esterel, indeed force this restriction in some cases. C programmers, however, are not used to being restricted.

```
if ( i < 3 )
  x = 1;
else if ( i >= 3 )
  x = 2;
```

FIGURE 5

```
if ( i < 3 )
  x = 1;
else if ( i == 3 )
  x = 2;
else if ( i > 3 )
  x = 3;
```

FIGURE 6




```
node_1 = a && b;
node_2 = c || node_1;
output_1 = node_2;
if ( output_1 ) {
  /* statements */
}
```

FIGURE 7: CODE PRODUCED BY SIMPLE BACKEND

the two comparisons would allow the *(i >= 3)* condition to be removed, because it is implicit with the *else*.[6]

If there are more than two distinct comparisons computed for a particular integer variable, BRO generates the don't care cubes for every pair of comparison inputs in the set. Unfortunately, this implies that in a code sequence such as Figure 6, the *(i > 3)* condition, even though it is redundant, would not be removed. The reason is that such a removal would require a don't care cube

i_0_compare_lt_3 & *i_0_compare_eq_3* & *i_0_compare_gt_3*

which is not in the set because the don't cares are only generated pairwise. It is certainly possible to generate the complete set of don't cares for every combination of comparison inputs. However, the algorithm to do so is much more complicated, so only pairwise don't cares are currently implemented in BRO. Here, the advantage of using a MV network is evident, as it expresses such relations automatically without the need to generate don't cares.

The BRO Backend

After the logic network has been constructed, it is given to SIS, a network manipulation package. SIS performs the task of removing redundancies, extracting common sub-expressions, and in general, massaging the network into something more efficient. But exactly what sort of network SIS should output depends strongly on what the BRO backend is expecting, so it can properly perform the transformation back to software.

The simplest backend that can be conceived is one that simply generates statements to recursively evaluate the fanins from an output node in topological order, and then perform a conditional branch, to execute the statements associated with each output only if that output node evaluates to 1. This is done for each output node, skipping shared intermediate nodes that have already been computed. This produces code such as that in Figure 7.

This backend, assuming a reasonable SIS script, effectively performs expression minimization and common sub-expression extraction on the original program. However, it has the effect of flattening the entire control structure. In particular, the backend will never generate *else*s or nested *if*s, and these hierarchical elements are essential in most cases to producing efficient code.

Fortunately, it is possible to devise algorithms to determine when it is possible to insert *else*s and nested *if*s. *Else* generation is based on the observation that two expressions are mutually exclusive if their onsets do not intersect. Thus, BRO could create a node which ANDs an output node with the subsequent output:

node_test = *output_1* & *output_2*

and request that SIS simplify this node. If the node simplifies to zero, then the second output will never be true if the first one is. The evaluation of the second output node, its conditional branch, and the statements associated with that output can be placed inside an *else* clause of the first output's conditional branch. This has two benefits. At runtime, if the first output is true, then the second output will not even be evaluated, which improves performance. Also, the logic to evaluate the second output can be minimized using the first output's onset as a don't care space. This leads to a faster evaluation of the second output when the first output is false.

Nested *if* generation follows a similar procedure. One expression is said to completely contain another when the offset of the first and the onset of the second do not intersect. Again, a node can be created to compute this:

node_test = !*output_1* & *output_2*

If this node simplifies to zero, then the second output can only be true when the first one is. Thus, the evaluation of the second output and its associated statements can be placed within the *then* clause of the first output's conditional branch, after the first output's statements. This has similar benefits — the second output need not be evaluated if the first is false, and the logic for the second output can be minimized using the first output's offset as a don't care space.

BRO currently only implements the simple backend, but it is expected that the use of these advanced techniques could produce efficient, hierarchical control structures. However, it could still fall short of handwritten code (including the original source being optimized) because it would only generate *if*s and *else*s at output nodes. That is, every *then* and *else* clause would have to begin with a statement block. This does not in general lead to the most efficient code. Thus, it is necessary to perform *else* and nested *if* generation at intermediate nodes as well as output nodes. This in turn requires SIS to produce intermediate

[6] Actually, the BRO frontend expresses all comparisons in terms of "equal-to" or "greater-than". This is possible because the inverted sense of the input can be used. Thus, *(i < 3)* actually refers to !*i_0_compare_gt_2*. As long as the proper don't cares are generated, this makes no difference to the network optimizer. However, it is somewhat easier for the BRO backend to deal with.

nodes that are meaningful for this purpose. The methods for going about this are currently unclear.

IV. CONCLUSION

Various techniques from the area of hardware synthesis can be used to optimize control flow in software. This idea can be applied at the local level, with methods such as the Software PLA and Switch Encoding. Or, it can be applied globally, through the process of decomposing a procedure into a logic network, manipulating the network, and transforming it back into software in an optimal manner. A tool has been developed to accomplish this, and although much further work is required, it shows promise for optimizing control-heavy software applications.

REFERENCES AND RELATED RESEARCH

[1] E. M. Sentovich, et al., “SIS: A System for Sequential Circuit Synthesis”. *Technical Report of the UC Berkeley Electronics Research Lab*, May 1992

[2] F. Balarin, et al., “Synthesis of Software Programs for Embedded Control Applications”. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol 18, pp. 834–849

[3] G. Berry, et al., “Esterel: a Formal Method Applied to Avionic Software Development”. *Science of Computer Programming*, vol 36, pp. 5-25