Operating Theory of the Series 32000® GNX™ Version 3 Compiler Optimizer

National Semiconductor
Application Note 583
Series 32000 Applications
January 1989

1.0 INTRODUCTION

The main difference between the GNX-Version 3 compilers and other compilers is the optimizer. Recompiling and optimizing with a GNX-Version 3 compiler will result in a 10% to 200% speedup for most programs, with an average improvement of over 30%. This chapter describes some of the advanced optimization techniques used by the compiler to improve speed or save space. The most important techniques are:

- Value propagation
- Constant folding
- Redundant-assignment elimination
- Partial-redundancy elimination
- Common-subexpression elimination
- Flow optimizations
- Dead-code removal
- Loop-invariant code motion
- Strength reduction
- Induction variable elimination
- Register-allocation by coloring
- Peephole optimizations
- Memory-layout optimizations
- Fixed frame

The following sections describe these techniques in more detail.

2.0 THE OPTIMIZER

The optimizer, shared by all the GNX-Version 3 compilers, is based on advanced optimization theory developed over the past 15 years. Central to the optimizer is an innovative global-data-flow-analysis technique which simplifies the optimizer's implementation. It allows the optimizer to perform some unique optimizations in addition to all the standard optimizations found in other compilers. Optimizations are performed globally on the code of a whole procedure at a time and not just in a local context.

The optimizer is implemented as a multi-step process. Each step performs its particular optimizations and provides new opportunities for the optimizations of the next step.

2.1 STEP ONE

The first step in the optimization process is to read in the source program one procedure at a time and to partition this procedure into basic blocks. A basic block is a straight line sequence of code with a branch only at the entry or exit. Some of the optimizations performed during this step are:

- **Value Propagation**

  Value propagation (or copy propagation) is the attempt to replace a variable with the most recent value that has been assigned to it. This optimization is primarily useful in the special case of constant propagation. It is important because it creates opportunities for other optimizations. Value propagation can be turned off by the /CODE_MOTION optimization flag (–Om on UNIX® systems).

- **Constant Folding**

  If an expression or condition consists of constants only, it is evaluated by the optimizer into one constant, thereby avoiding this computation at run-time. The optimizer, using algebraic properties such as the commutative, associative and distributive law, sometimes rearranges expressions to allow constant folding of part of an expression.

  The GNX-Version 3 C compiler also folds floating-point constant expressions. This feature can be turned off using the /NOFLOAT_FOLD option (–Oc on UNIX systems) of the optimizer.

- **Redundant-Assignment Elimination**

  The optimizer detects and eliminates assignments to variables which are not used later in the program or which are assigned again before being used. This optimization can often be applied as a result of value propagation.

  Value propagation, constant folding, and redundant assignment elimination are illustrated in *Figure 1*.

RRD-B30M105/Printed in U. S. A.

The program sequence

```
a = 4;
if (a*8 < 0) b = 15;
else b = 20;
...code which uses b but not a...
```

is translated by the GNX-Version 3 C compiler front end into the following intermediate code

```
a ← 4
if (a*8 ≥ 0) goto Ll
b ← 15
goto L2
Ll: B ← 20
L2: ...
```

which is transformed by ''value propagation'' into

```
a ← 4
if (4*8 ≥ 0) goto Ll
b ← 5
goto L2
Ll: b ← 20
L2: ...
```

which after ''constant folding'' becomes

```
a ← 4
if (true) goto Ll
b ← 15
goto L2
Ll: b ← 20
L2: ...
```

''dead code removal'' results in

```
a ← 4
goto Ll
Ll: b ← 20
L2: ...
```

which is transformed by another ''flow optimization'' into

```
a ← 4
b ← 20
```

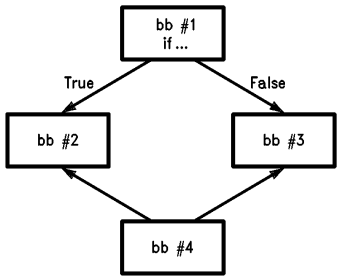
Since there is no further use of a, a ← 4 is a ''redundant assignment:''

```
b ← 20
...
```

FIGURE 1. Relationship between Various Optimizations

2.2 STEP TWO

The second step in the optimization process is the construction of the program's ''flow graph.'' This is a graph in which each node represents a basic block. A basic block is a linear segment of code with only one entry point and one exit point. If there is a path in the program that leads from one basic block to another, then an ''arrow'' is drawn in the graph to represent this path. *Figure 2* illustrates a flow graph, representing an ''if-then-else'' sequence.



TL/EE/10344–1

FIGURE 2. Flow Graph

During the construction of the flow graph, additional optimizations can be performed:

- **Flow Optimizations**

  Flow optimizations reduce the number of branches performed in the program. One example is to replace a branch whose target is another branch with a direct branch to the ultimate target. This often makes the second branch redundant. At other times, code is reordered to eliminate unnecessary branches. Branches to ''return'' are replaced by the return-sequence itself.

- **Dead Code Removal**

  Flow optimizations are also designed to help the optimizer discover code which will never actually be executed. Removal of this code, called ''dead code removal'', results in smaller object programs.

2.3 STEP THREE

Step three of the optimization process is called ''global-data-flow-analysis''. It identifies desirable global code transformations which speed program execution. Many of these concentrate on speeding up loop execution, since most programs spend 90% or more of their time in loops. Global-data-flow-analysis is the computation of a large number of properties for each expression in the procedure.

Unlike most optimizers, which employ unrelated and separate techniques, the optimizer centers around one innovative technique which involves the recognition of a situation called ''partial redundancy''. This technique is so powerful that many other optimizations turn out to be special cases. The central idea is that it is wasteful to compute an expression, say a * b, twice on the same path; it is often faster to save the result of the first computation and then replace the fully redundant second computation with the saved value. More common, however, is the case in which an expression is partially redundant; there is one path to an expression, which already contains a computation of that expression, but another path to that same expression does not.

The following optimizations are performed by a common technique:

- **Elimination of Fully Redundant Expressions**

  This optimization is often called ''Common Subexpression Elimination''. It is relatively simple to avoid the recomputation of fully redundant expressions. The optimizer saves the result of the first computation (usually in a register variable) and uses the saved value in place of the second computation. Performance-conscious programmers sometimes do this themselves, but many cases, such as array index and record number calculations, are recognized only by the optimizer.

- **Partial Redundancy Elimination**

  A partially redundant expression can be eliminated in two steps. First, insert the expression on the paths in which it previously did not occur; this makes the expression fully redundant. Second, save the first computations and use the saved value to replace the redundant computation. An example of this optimization is shown in *Figure 3*.

  Partial redundancy elimination sometimes results in slightly larger code, but execution is not harmed, since all inserted expressions are in parallel and only one is actually executed.

- **Loop Invariant Code Motion**

  If an expression occurs within a loop and its value does not change throughout that loop, it is called ''loop invariant''. Loop invariant expressions are also partially redundant. This can be understood by realizing that there are two paths into the loop body: one is through the loop entry (the first time the loop is executed), and the other is from the end of the loop, while the exit condition is false. Loop invariant computations are, therefore, removed from the loop in the same way: the expression is first inserted on the entry path to the loop, and then the expression is saved on the entry path in a register, while the redundant computation in the loop is replaced by that register.

- **Strength Reduction**

  This optimization globally replaces complex operations by simpler ones. This is primarily useful for reducing complex array-subscript computations (involving multiplication into simpler additions).

```
for (i = 0; i < 15; i+ = 0)
a [i] = 0;
```

  is transformed into:

```
for (i = 0, p = a; i < 15; i+ = 1, p+ = 4)
    *p = 0;
```

- **Induction Variable Elimination**

  Induction variables are variables that maintain a fixed relation to other variables. The use of such variables can often be replaced by a simple transformation. For instance, the example given for strength reduction can be reduced to the following:

```
for (p = a; p < a + 60; p+= 4)
*p = 0;
```

In the following code, a*b is ''partially redundant'' (computed twice only if C is true):

```
if (C)
   x = a*b;
else
   b = b + 10
y = a*b;
```

It is first transformed into a ''fully redundant'' expression

```
if C = 1
   x ← a*b
else
   b ← b + 10
   temp ← a*b
y ← a*b
```

Then, as in the simple case of ''redundant expression elimination,'' this is reduced to

```
if C = 1
   temp ← a*b
   x ← temp
else
   b ← b + 10
   temp ← a*b
y ← temp
```

Now, the expression a*b is computed only once on any path.

FIGURE 3. Example of Partial Redundancy Elimination

2.4 STEP FOUR

The fourth optimization step performed by the optimizer, and possibly the most profitable, is the ''register allocation'' phase. Register allocation places variables in machine registers instead of main memory. References to a register are always much faster and use less code space than respective memory references.

The algorithm used by the optimizer is called the ''coloring algorithm''. First, global-flow-analysis is performed to determine the different live ranges of variables within the procedure. A live range is the program path along which a variable has a particular value. Generally, an assignment to a variable starts a new live range; this live range terminates with the last use of that assigned value.

The optimizer subsequently constructs a graph as follows: each node represents a live range; two nodes are connected if there exists a point in the program in which the two live ranges intersect. The allocation of registers to live ranges is now the same as coloring the nodes of the graph so that two connected nodes have different colors. This is a classic problem from graph theory, for which good solutions exist. If there are not enough registers, more frequently used variables have higher priority than less frequently used ones. Loop nesting is taken into account when calculating the frequency of use, meaning that variables used inside of loops have higher priority than those that are not.

Most optimizing compilers attempt register allocation only for true local variables, for which there is no danger of ''aliasing.'' An alias occurs when there are two different ways to access a variable. This can happen when a global variable is passed as reference parameter; the variable can be accessed through its global name, or through the parameter alias. A common case in C is when the address of a variable is assigned to a pointer.

The optimizer takes a more general approach by considering all variables with appropriate data types as candidates for register allocation, including global variables, variables whose addresses have been taken, array elements, and items pointed to by pointers. These special candidates cannot reside in registers across procedure calls and pointer references and, therefore, normally have lower priority than local variables. However, instead of completely disqualifying the special candidates in advance, the decision is made by the coloring algorithm.

Additional important optimizations performed by the register allocator are:

- **Use of Safe and Scratch Registers**

  The Series 32000 machine registers are, by convention, divided into two groups: registers R0 through R2 and F0 through F3, the so-called ''scratch'' registers which can be used as temporaries but whose values may be changed by a procedure call, and the ''safe'' registers (R3 through R7 and F4 through F7) which are guaranteed to retain their value across procedure calls. The register allocator spends a special effort to maximize the use of scratch registers, since it is not necessary to save these upon entry or restore them upon exit from the current procedure. The use of scratch registers, therefore, reduces the overhead of procedure calls.

- **Register Parameter Allocation**

  The register allocator attempts to detect routines, whose parameters can be passed in registers. This is possible for static routines only, since by definition all the calls to such routines are visible to the optimizer. Calls to other (externally callable) routines are subject to the standard Series 32000 calling sequence. Passing parameters in registers in another way to reduce the overhead of procedure calls.

2.5 STEP FIVE

The last optimization step consolidates the results of all previous steps by writing out the optimized procedure in intermediate form for the separate code generator. Some reorganizations take place during this step. Local variables which have been allocated in registers are removed from the procedure's activation record (frame), which is reordered to minimize overall frame size.

3.0 THE CODE GENERATOR

The back end (code generator) attempts to match expression trees with optimal code sequences. It applies standard techniques to minimize the use of temporary registers, which are necessary for the computation of the subexpressions of a tree. The main strength of the code generator lies in the number of ''peephole optimizations'' it performs.

Peephole optimizations are machine-dependent code transformations that are performed by the code generator on small sequences of machine code just before emitting the code. Some of the most important peephole transformations are listed below:

- The code for maintaining the frame of routines which have no local variables, or whose variables are all allocated in registers, is removed.
- Switch statements are optimized into binary search, linear search or table-indexed code (using the Series 32000 CASE instruction), in order to obtain optimal code in each situation.
- The stack and frame areas are always aligned for minimal data fetches.
- Reduction of arithmetic identities, i.e., x*1 = x, x+0 = x, etc.
- Use of the ADDR instruction instead of ADD of three operands.
- Some optimizations performed in the optimizer, such as the application of the distributive law of algebra, i.e., (10+i)*4 = 40+4*i, provide additional opportunities to the code generator to fully exploit the Series 32000's addressing modes.
- Use of ADDR instead of MOVZBD of small constant.
- Strength Reduction Optimizations. Use of MOVD instead of MOVF from memory to memory; use of index addressing mode instead of multiplication by 2, 4 or 8; use of combinations of ADDR instructions or shift and ADD sequences instead of multiplication by other constants up to 200.

- Fixed Frame Optimization. An important contribution of the code generator is its ability to precompute the stack requirements of a procedure in advance. This allows the generation of code which does not use (nor update) the FP (frame pointer), resulting in cheaper calling sequences.

  This optimization is most useful when the procedure contains many procedure calls because it is not necessary to execute code to adjust the stack after every call. Parameters are moved to the pre-allocated space instead of pushing them on to the stack using the top-of-stack addressing mode. Note that when using this optimization, the run-time stack pointer stays the same throughout the procedure, and all references to local variables are relative to it and not the FP. Also note that the evaluation order of parameters is unpredictable because parameters that take more space to evaluate are treated first to save space.

While most optimizations are beneficial for both speed and space, some optimizations favor one over the other. The default setting of the optimizer switch favors speed over space in trade-off situations. The following optimizations are trade-off situations which are affected by an optimization flag.

- Code is not aligned after branches.
- All returns within the code are replaced by a jump to a common return sequence.
- Certain space-expensive peephole transformations are not performed.

4.0 MEMORY LAYOUT OPTIMIZATIONS

The following memory layout optimizations are performed by the GNX-Version 3 C compiler:

- Frame variables that are allocated in registers are removed from the frame.
- Internal, static routines whose parameters are passed in registers have smaller frames.
- The stack alignment is always maintained. Stack parameters are passed in aligned positions.
- Frame variables are allocated in aligned positions. The compiler reorders these variables to save overall frame space.
- Code is aligned after every unconditional jump.

Lit. # 100583

National Semiconductor Corporation
1111 West Bardin Road
Arlington, TX 76017
Tel: 1(800) 272-9959
Fax: 1(800) 737-7018

National Semiconductor Europe
Fax: (+49) 0-180-530 85 86
Email: cnjwge@tevm2.nsc.com
Deutsch Tel: (+49) 0-180-530 85 85
English Tel: (+49) 0-180-532 78 32
Français Tel: (+49) 0-180-532 93 58
Italiano Tel: (+49) 0-180-534 16 80

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductor Japan Ltd.
Tel: 81-043-299-2309
Fax: 81-043-299-2408