Control Flow Graph Reconstruction for Assembly Language Programs with Delayed Instructions *

Nerina Bermudo and Andreas Krall
Christian Doppler Laboratory
Institut für Computersprachen – Technische Universität Wien
{nerina,andi}@complang.tuwien.ac.at

Nigel Horspool
Department of Computer Science
University of Victoria, Canada
nigelh@uvic.ca

Abstract

Most software for embedded systems, including digital signal processing systems, is coded in assembly language. For both understanding the software and for reverse compiling it to a higher level language, we need to construct a control flow graph (CFG). However CFG construction is complicated by architectural features which include VLIW parallelism, predicated instructions and branches with delay slots.

We describe an efficient algorithm for the construction of a CFG, where the parallelism has been eliminated, instructions are reordered and delay slots have been eliminated. The algorithm's effectiveness has been demonstrated by its use in a reverse compiler for the Texas Instruments C60 series of digital signal processors.

1 Introduction

Most software for embedded systems processors, especially digital signals processors, has been coded in assembly language. Such programs are difficult to understand and hard to maintain. Furthermore, they are not easily ported to a new processor. If the software can be reverse compiled to a higher level language such as C, it becomes much easier to understand and to port.

An important first step in reverse compilation is to construct a control flow graph (CFG) for each subroutine. However, the task is made complicated if the processor has instructions with delay slots, and it is made especially complicated if the delay slots of branch instructions can contain other branch instructions.

We show how to transform the assembly language to a form which has no delay slots. From this form, we can easily generate the CFG.

*This work was supported by the Christian-Doppler Forschungsgesellschaft and Infineon.

2 Computer Architectures with Delay Slots

Most modern general purpose computer architectures maintain an illusion that the instructions in an assembly language program are executed in the order in which they are written. However it is an illusion because (a) different kinds of instructions may require different numbers of clock cycles to complete, (b) instructions may have to wait for operands to become ready, and (c) instructions may have to wait for an appropriate functional unit to become available. Modern architectures therefore have complicated logic to look ahead in the instruction stream to find instructions which can be executed out of order to keep all the functional units busy while preserving the correct semantics of the program.

In contrast, some other architectures, especially those for embedded systems and DSP applications, do not attempt to maintain such a strong illusion. To reduce the complexity of the processor (and implicitly, therefore, to reduce power consumption), instructions may not always produce their results in the order in which they are fetched. For example, the Texas Instruments C6000 series [10] requires a single cycle to complete a simple fixed-point instruction such as addition (the ADD opcode). That means that if the instruction begins its execution in cycle i, it will read its operands at the beginning of the cycle and will store the answer in the result register at the end of the same cycle. However, the multiply instruction (opcode MPY) cannot complete its execution in a single cycle; it requires two cycles. That is, if the MPY instruction begins execution in cycle i then the two operands are fetched at the beginning of cycle i but the result register is not updated with the product of the multiplication until the end of cycle $i+1$. Cycle $i+1$ is known as the *delay slot* for that MPY instruction. Stated differently, the MPY instruction has one delay slot. The programmer can code another instruction to be executed in that delay slot as long the instruction does not need the result of the MPY instruction and does not need the same functional unit. Sometimes the logic of the program will be such that nothing further can be

usefully computed until the result of the MPY is available, in which case the programmer should place a no-operation instruction (opcode NOP) in the delay slot.

On the Texas Instruments C6000 series, many fixed-point instructions, such as the ADD instruction, do not have any delay slots. Multiply has one delay slot; the various instructions to load from memory have four delay slots – i.e. if the load instruction begins execution in cycle i then the value obtained from memory is not stored into the result register until the end of cycle $i + 4$. Finally, the various branching instructions have five delay slots. The effect is that if a branch instruction is executed in cycle i then the program counter will keep following its previously determined sequence for five more cycles. The program counter will not be updated to refer to the destination of the branch until the end of cycle $i + 5$. (Floating-point instructions on the TI C6000 series have various numbers of associated delay slots which range between zero and nine.)

The association of delay slots with branch instructions makes assembly code particularly difficult to read by humans. A human must remember that the control transfer does not take place until several instructions later. The problem is compounded further by another complication – the delay slots can themselves contain branch instructions.

3 Choices for Removing Delay Slots

In the general case, each instruction I would have an associated number of execution cycles I_X. As we have demonstrated above, it can be hard for a human reader to make sense of the code. Perhaps, more importantly, the delay slots present a major complication to tools which attempt to disassemble the code or to construct a control flow graph.

The assembler code, as written, shows the instructions in the order in which they are dispatched. However, for the purposes of understanding control flow, we need to see the instructions in the order in which they are completed. It would be desirable, therefore, to remove the effect of delay slots from the code. We see two plausible alternatives for achieving that goal.

1. Transform the code sequence to an equivalent version where all delay slots are occupied by no-op instructions. For example, a branch instruction which takes six cycles and therefore has five delay slots should appear in the transformed program followed by five no-op instructions.
2. Transform the code sequence to a version for an idealized computer which has the same instruction set, but where all instructions complete execution in a single cycle.

We discuss each one in turn.

3.1 Filling Delay Slots with No-Ops

This approach is appealing because the result should be a program which is still executable on the same platform, but where the code is human readable and also easy for tools to process. Unfortunately, it is impossible to complete the transformation without imposing some arbitrary instruction orderings – orderings which are not implied in the original program. For example, consider the following group of three instructions:

```
inst1   ;  has two delay slots
inst2   ;  has one delay slots
inst3   ;  has zero delay slots
```

In this example, all three instructions complete their execution at the same time (as would be possible if they employ different functional units on the processor). If we present the transformed code as follows:

```
inst1   ;  has two delay slots
NOP
NOP
inst2   ;  has one delay slots
NOP
inst3   ;  has zero delay slots
```

then we have imposed an ordering which was not implied in the original code.

3.2 An Ideal Computer with No Delay Slots

Our alternative approach, and the approach adopted in the remainder of this paper, is to transform the code to a version where all instructions are assumed to complete their execution in one cycle and where several instructions may be executed in parallel. For example, the group of three instructions shown earlier may be transformed to the following form:

```
inst1 || inst2 || inst3
```

The new code sequence accurately shows all three instructions as completing simultaneously. However, the code can be misleading or, perhaps, wrong because it also shows the three instructions fetching their operands in the same cycle, whereas the original code sequence showed them being fetched in different cycles. It is easy to construct an example where the timing of an operand fetch is significant. Suppose the original program is as follows where we assume that multiply (MPY) requires two delay slots and loading a constant (MVK) has no delay slots:

```
MPY  R2, R3, R1    ;  R1= R2*R3
MVK  20, R2        ;  R2 = 20
NOP                ;  no delay slots
```

```
push <entry point,∅> onto Stack;
while Stack is not empty do
   <PC,State> = popped value from Stack;
   I = instr[PC];
   succ = PC+1;
   if instr[PC] is a branch then
       add <PC,delay,target> to State
       // where target = branch destination,
       // delay = # delay slots for a branch
   fi;
   nextSt = ∅;
   origin = dest = -1;
   foreach <i,c,d> ∈ State do
      if (c = 0) then
         origin = i;  dest = d;
         if instr[i] is unconditional then
             succ = -1
         fi
      else
         add <i,c-1,d> to nextSt
      fi
   od
   if dest≥0 and <dest,nextSt> is new then
       add <origin,PC,dest> to CFG;
       push <dest,nextSt> onto Stack
   fi;
   if succ≥0 and <succ,nextSt> is new then
       add <-1,PC,succ> to CFG;
       push <succ,nextSt> onto Stack
   fi
od
```

Figure 1. CFG Edge Construction Algorithm

If ordered according to the order in which the instructions complete their execution (and eliminating the NOP), we would have

```
MVK  20, R2
MPY  R2, R3, R1
```

and that is clearly incorrect because R2 is changed before the MPY instruction has read its value. To preserve correctness, we must insert instructions to make copies of operands when needed. A correct solution would be

```
MV   R2, T1
MVK  20, R2
MPY  T1, R3, R1
```

where the idealized architecture is extended with additional temporary registers to hold operand copies.

4 Constructing the CFG

The CFG construction algorithm has four phases:

1. edge recognition,
2. basic block construction,
3. delay resolution, and
4. data conflict resolution.

Each phase is explained in detail below.

4.1 Edge Recognition

Initially, we create a CFG where each instruction corresponds to a node in the graph. Subsequently, groups of nodes for consecutive statements are coalesced into basic blocks, resulting in the usual form for a CFG.

The analysis performed by the edge recognition algorithm is equivalent to traversing all paths through the program. Whenever the traversal reaches a program point where a conditional branch is ready to complete execution, the path forks. The algorithm continues following one path while stacking the other path for analysis later. While following a path, the algorithm keeps track of program state which, for the purposes of our analysis, consists of information about pending delayed branches (i.e. at the current point being analyzed in the program they have been initiated but have not yet completed). The algorithm must remember which program states have been analyzed at which program point so that program loops do not cause the algorithm to repeat work and to never terminate. The result of the algorithm is a set of CFG edges.

For the purposes of describing the edge construction algorithm, we consider the program to be a vector of instructions which is indexed by a range of consecutive integers starting at 0. In particular, *instr[i]* denotes the instruction at position i. We represent each CFG edge by a triple $< i, s, d >$ where

- i is either the position of the branch instruction in the program which causes the control transfer or is -1 if the control transfer is the result of normal sequential execution;
- s is the source of the edge and is the position of the last instruction to be executed before the control transfer occurs;
- d is the destination of the edge and is the position of the next instruction to be executed after the control transfer.

We represent the delayed branch state at a point P as a set of triples $< i, c, d >$ where

- i is the position of a pending branch instruction (a branch which has been fetched but has not yet completed execution);

```
 0  LabA:        inst1
 1       [cond]  B  LabB  // 3 delay slots
 2               inst2
 3       [cond]  B  LabD  // 3 delay slots
 4               inst3
 5               inst4
 6               B  LabA  // 3 delay slots
 7               B  LabC  // 3 delay slots
 8               inst5
 9               inst6
10  LabB:        inst7
11               B  LabC  // 3 delay slots
12  LabC:        inst8
13               inst9
14  LabD:        inst10
15               inst11
16               inst12
```

Figure 2. Running Example

c is a count of how many instruction cycles must occur before the control transfer occurs;

d is the destination of the control transfer.

The stack used by the algorithm contains entries which are pairs consisting of a position in the program and a delayed branch state,

Pseudocode for the program is shown in Figure 1. The tests of the form *<x,nextSt> is new* are checks as to whether that pair has been previously pushed onto the stack.

Consider the sample program shown in Figure 2, where each branch is assumed to have 3 delay slots. Two of the branches, as indicated by the *[cond]* prefix, are conditional. If this program is processed by the edge construction algorithm, it will explore and discover the control flow paths as diagrammed in Figure 3. In this diagram, arrows indicate where the algorithm has followed a (delayed) branch. If there is pair of arrows emanating from a program point, the branch was conditional. The algorithm stops following a path when a repeated program point and state combination is encountered. To stress these repeated points, they are flagged with superscript asterisks in the diagram.

The CFG edges that are created by the algorithm for this example should be apparent from Figure 3 too. Each arrow in that figure corresponds to a CFG edge. In addition, because we assumed that each instruction in the original program represented a node in the CFG, there are many simple edges that correspond to sequential control flow. For example, the sequence of positions `0 1 2 3 4` in the figure represents 5 nodes which are connected by sequential control flow.

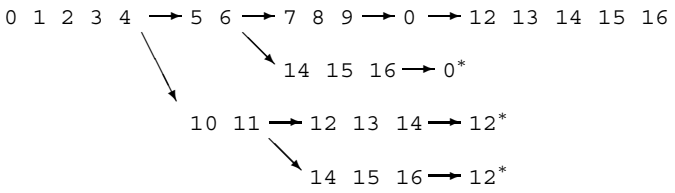

Figure 3. Paths Explored in Running Example

4.2 Basic Block Construction

Cooper et al. [5] gave an algorithm for constructing the CFG for an assembly language program with delayed branches. Their algorithm produces a different kind of CFG from ours because the delayed branches remain in the program. The program is simply partitioned into basic blocks with the necessary control flow edges being inserted. The control flow logic to show which tests cause which changes in the sequence of execution are hard to discern.

Our approach which generates a program version where branch instructions do not have delay slots is more amenable to further program analysis. However, in general, if we want to translate the program into a version for our idealized instruction set then we must duplicate some code.

Fortunately the edge construction algorithm can identify which nodes must be duplicated. Considering again Figure 3, as an example, the contents of this figure are easily redrawn as a CFG with basic block nodes and where instructions have been duplicated as necessary. The diagram appears in Figure 4.

The critical observation is that if an instruction in the program can be executed in more than one branch delay state (recall that the branch delay state is a set of the pending delayed branch instructions) then that instruction must be replicated for each such state. To construct the corresponding CFG, the edge construction algorithm must be modified in a minor way.

- Whenever a pair of the form $<PC, state>$ is pushed onto the stack in this algorithm, the algorithm checks to see if the pair has been pushed previously. If this is a new pair, then we augment the algorithm to create a new node in the CFG. (A new node must also be created for the initial $<$*entry point*$, \emptyset >$ pair that is pushed at the start.)
- Whenever a new CFG edge is created, it should be related to the CFG nodes that are created by the algorithm. Thus the statement *add <origin,PC,dest> to CFG* in the algorithm needs to be modified. The source of the edge is the node corresponding to the pair *<PC,State>*, while the destination node corre-

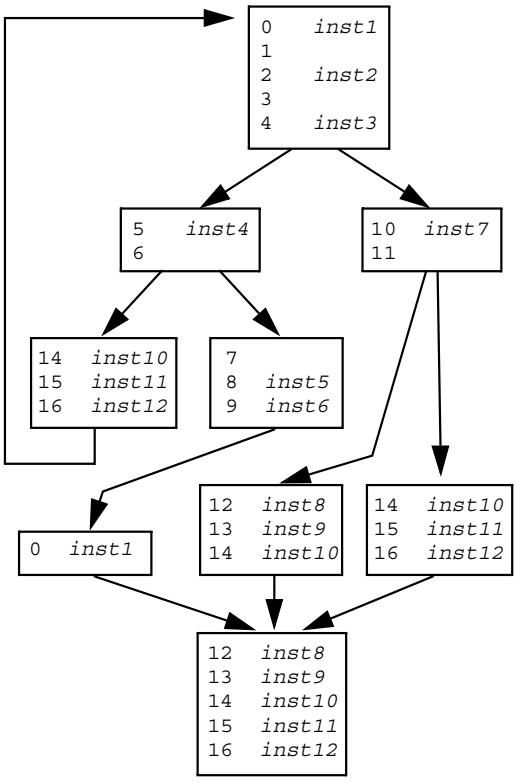

Figure 4. CFG for Running Example

sponds to the pair <*dest,newState*>. The edge can be labelled with *origin* to indicate which branch instruction in the original program causes this control transfer. The statement *add <-1,PC,succ> to CFG* is modified similarly. The edge may be labelled with *-1* to indicate sequential flow of control.

- Any branch instruction should not be listed (or shown as NOP) when the node containing that instruction is created in the new CFG.

Finally, we note that the nodes of the CFG correspond to single statements in the original program. To coalesce these simple nodes into basic blocks is a simple matter of checking all the edges. If a node A has a unique successor B and if the edge construction algorithm created an edge of the form <*-1,A,B*> (i.e. an edge for sequential flow) then nodes A and B can be merged.

4.3 Delay Resolution

Once the control flow graph has been created, it is relatively simple to resolve any other instructions which have delay slots. Nevertheless it may be necessary to move instructions across CFG edges and to duplicate code. (Similar problems arise in global instruction scheduling and have been described elsewhere [8].)

Consider an instruction I at position P which has d delay slots. As discussed in the introduction, I should be moved to a new position d instructions later, where the instruction is completed. A NOP instruction is left in place of I at its original position in order to maintain the instruction slot structure of the program. The NOP instructions can be safely removed as the very final step, after delay resolution and data conflict resolution have been completed.

If the new position occurs within the same basic block, then the moved instruction is inserted at its new position in parallel with any other instructions located at that position.

If the new position does not occur within the same basic block, then I must be inserted in each successor block at the appropriate position.

If the number of delay slots is sufficiently large and the length of a successor block sufficiently short, then I might have to be propagated to successors of the successor, and so on. Whenever an instruction should be moved to a block with several predecessors, this block has to be duplicated and the instruction is moved to the copy. Furthermore, whenever a instruction I traverses a block with several predecessors before reaching its final destination, all blocks from the one having more than one predecessor until the destination of I must be copied. The algorithm in figure 5 takes care of this problem.

4.4 Data Conflict Resolution

When moving an instruction to a different position in the program, it is important to take the possible data dependence conflicts into account and resolve them to maintain the semantic meaning of the original program.

Let I be an instruction at position p with a delay $d > 0$. For now, we assume the destination of I after delay resolution to be in the same block as I. Moving I from p to $p + d$ will cause a conflict if in the range $[p \ldots p + d]$ there is any instruction which completes a redefinition of any of the operands of I.

Consider the following example where we assume that multiplication (opcode MPY) has one delay slot and the load instruction (opcode LDW) has two.

```
0       MPY  B7, B9, B13  // 1 delay slot
     || ADD  A14, B4, B7
1       ADD  A10, A7, A13
     || LDW *A0++, B9     // 2 delay slots
2       NOP
3       NOP
```

The new location for the MPY instruction will be position 1. This instruction defines register B13 and uses registers B7 and B9. We find the instruction `ADD A14, B4, B7` at position 0, which redefines B7. In order to preserve the

```
function processBlock(bb, delaySet,
                      pred, copySet)
begin
  nPreds = number of predecessors of bb;
  if nPreds > 1 then
    add all instructions of delaySet
      to copySet;
  fi
  if copySet ≠ ∅ then
    copybb = copy of bb;
    insert copybb with single predecessor
           pred in original CFG;
    bb = copybb
  fi
  mark bb as visited;
  foreach position P in basic block bb do
    B = copy of bundle at position P;
    foreach instruction I' in bundle B do
      I = copy of instruction I';
      delay = delay of I;
      if I is not a NOP instruction then
        add triple <I,delay,P> to delaySet;
        replace instruction at position P
          in original CFG with NOP;
      fi
      foreach triple <J,d,oP> in delaySet do
        if I defines input reg r of J then
            handleDataConflict(r, <J,d,oP>);
        fi
      od
    od
    newDelaySet = ∅;
    foreach triple <J,d,oP> in delaySet do
      if d=0 then
        add J at position P in new CFG;
        if J∈copySet then
          remove J from copySet
        fi
      else
        add <J,delay-1,oP> to newDelaySet
      fi
    od
    delaySet = newDelaySet;
  od
  foreach successor S of basic block bb do
    if delaySet≠∅ or
       S has not been visited yet then
      processBlock(S,delaySet,bb,copySet)
    fi
  od
end
```

Figure 5. Instruction Movement / Data Conflict Elimination

```
function handleDataConflict(reg, <J,d,P>)
begin
  t = getNewTemporaryRegister();
  m = instruction "MOV reg,t";
  add m to bundle at position P in new CFG;
  if reg is postincremented then
    a = instruction "ADD reg,1,reg";
    add a to bundle at position
      P in new CFG;
    remove postincrementing from J
  fi;
  Handle other addressing modes similarly
  replace reg by t in J
end
```

Figure 6. Handling Of Data Conflicts

result of the multiplication, we invent a new temporary register *temp0*, then we add an instruction initializing *temp0* at position 0 (the old position of MPY), and we modify the MPY instruction to be `MPY temp0,B9,B13`. Note that LDW instruction at position 1, which redefines the register B9, does not create a conflict because it does not change that register until two instruction cycles later.

The LDW instruction does, however, have a side-effect other than loading a value into register B9. In this particular case, register A0 is post-decremented and that side-effect occurs in the same cycle as when the instruction is initiated. If we wish to move the LDW instruction two positions later, we should decrement the A0 register immediately (in case other instructions use A0 as an input operand) while using the old value for the LDW instruction. We can achieve the desired effect with a second new temporary register, *temp1*. The final version of the code sequence where no instructions have any delay slots is therefore as follows.

```
0       MV   B7, temp0
      || ADD  A14, B4, B7
1       MPY  temp0, B9, B13
      || ADD  A10, A7, A13
      || MOV  A0, temp1
      || ADD  A0, 1, A0
2       NOP
3       LDW  *temp1,  B9
```

We propose a simple approach where we make copies of *all* input register operands used by the delayed instructions which cause conflicts. Later passes in the reverse compiler should include copy elimination to remove the redundant copy operations.

With our simplification, resolution of the data conflicts can be conveniently combined with moving instructions to their final positions in the delay-free version of the code. It traverses the CFG as produced by the combined edge detection and basic block construction algorithm, and creates

a new CFG.

The algorithm is initiated by making the function call `processBlock(0,∅,-1,∅)`, where the number of the basic block where the program begins execution is assumed to be 0. The second argument to `processBlock` is a set of triples of the form <*instruction, delay, original position*>, where:

instruction is a copy of an instruction in the original CFG,

delay is the number of instruction cycles which must be completed after entry into the basic block before the instruction finishes execution, and

original position is the position where the instruction was declared. In case of data conflict this is the position where the temporary variable must be initialized.

The third argument of this function is the predecessor basic block, the block which which was being processed when processBlock was recursively called. The fourth argument is an instruction list. Whenever the algorithm reaches a block with several predecessors, *copyList* holds all instructions whose original position was before this block. These instructions are the ones which force us to create copies of blocks, as explained in section 4.3.

The algorithm appears in Figures 5 and 6.

4.5 Sequentialization

After eliminating delays and removing data conflicts, we have a form of program where several instructions may occupy the same execution slot. That is, the instructions are organized into bundles. In order to remove this instruction level parallelism, the instruction bundles need to be rewritten in a sequential form. However we need to be careful because one instruction in the bundle can define a new value for a register while another instruction uses the same register as an input operand.

A simple approach to sequentialize the bundle is to write the instructions in the same order as they appear in the bundle, but modifying the instructions to use copies of input registers whenever needed to avoid a conflict. The function *sequentializeBundle* shown in Figure 7 implements the simple sequentialization.

```
function sequentializeBundle(bun)
begin
  foreach register r do
    registerMap[r] = r
  od;
  S = ∅;
  defRegs = ∅;
  foreach instruction I in bun do
    inRegs = { r | I uses register r };
    foreach r ∈ inRegs ∩ defRegs do
      // resolve the conflict
      if registerMap[r] = r then
         t = getNewTemporaryRegister();
         m = instruction "MOV r,t";
         registerMap[r] = t;
         S = m + S
      fi;
      replace occurrences of r in I used as
         an input register by registerMap[r]
    od;
    outRegs = { r | I defines register r };
    defRegs = defRegs ∪ outRegs;
    S = S + I
  od;
  return S
end
```

Figure 7. Sequentialization Algorithm

5 Related Work

As far as we know, we were the first to describe a CFG construction algorithm for delayed branches which removes delay slots. The work closest to ours is the CFG construction algorithm by Cooper et al. [5]. They present an algorithm for building a correct CFG from scheduled assembly code that includes branches in branch delay slots. A significant difference from our algorithm is that they do not replace delayed branches but just insert all control flow graph edges to the existing instructions. This algorithm is helpful for program understanding. However, if the CFG is used as the basis for further analyses, the delay slots of instructions have to be taken into account.

Computing the CFG in a compiler from a function's intermediate representation is straightforward and described in any compiler book [1, 3]. Computed branches and mixing of code and data make CFG construction more difficult or even impossible for binary programs or programs written in assembly language.

Horspool and Marovac [9] state that the separation of instructions from data for most computer architectures is equivalent to the Halting Problem and therefore unsolvable in general. They describe an algorithm that traces the control flow in a program seeking a disassembled version of the program with a maximal number of instructions.

Flow graph construction from binaries is also necessary in binary translators, reverse compilers and link time or binary optimizers. Sites et al. [14] describe CFG construction in the Alpha binary translator, Christina Cifuentes [4] describes in detail CFG construction in her thesis about reverse compilation, Debray et al. [6] describe CFG construction for a binary optimizer, Sutter et al. [15] present techniques for improving analysis of indirect branches and Kiss et al. [12] discuss slicing of binaries.

Program	#Branches (Cond)	#Loops	Loop length	#Blocks	#Block copies	#Edges	#Bundles original	#Bundles CFG	#Stmts Original	#Stmts CFG	#Temp registers
autcor	2 (1)	1	8	4	1	5	30	38	116	211	13
bitrev	1 (1)	1	7	4	1	5	25	32	76	129	9
blk_move	3 (3)	1	2	7	4	7	33	33	46	51	0
dotprod	6 (5)	1	1	9	6	13	18	25	81	128	0
gouraud	2 (2)	1	4	5	2	5	29	29	97	99	3
idct	2 (2)	1	11	7	2	10	53	75	202	384	11
iir	2 (2)	1	5	6	3	7	37	42	119	179	12
iircas4	2 (1)	1	4	6	3	7	23	27	84	126	3
latanal	2 (1)	1	3	6	3	7	21	24	83	121	5
latsynth	3 (1)	1	2	5	2	6	14	17	52	55	0
max	2 (2)	1	3	6	3	7	25	28	92	145	11
minerror	1 (1)	1	9	4	1	5	27	36	99	206	36
vecsumsq	6 (6)	1	1	9	6	13	20	28	86	133	0
w_vec	3 (3)	1	2	7	4	7	32	35	88	101	5
dct	2 (2)	2	11	7	2	10	48	70	194	368	25
fir4	3 (3)	2	16	10	4	14	34	56	130	282	18
radix2	3 (3)	2	19	10	4	14	41	67	176	355	13

Table 1. Experimental Results

Basic block duplication also occurs at control flow graph normalization [2]. CFG normalization facilitates program transformations, program analysis and automatic parallelization. Goto statements can be eliminated and transformed to structured control flow [7]. Janssen and Corporaal minimize the number of duplicated nodes with controlled node splitting [11].

Code motion across basic block boundaries leads to code duplication. This problem is common in compilers and well studied. Gupta [8] presents a code motion framework for global instruction scheduling which eliminates delay slots.

Krall et al. [13] describe an ultra fast compiled emulator for an architecture with delayed instructions. Emulation of delayed instructions leads to the execution of parts of an instruction in succeeding basic blocks. To keep code duplication small a combination of basic block duplication and conditional execution is used.

6 Experimental Results

This section presents some results obtained by the implementation of the algorithms explained in this paper. Our evaluations have been performed on hand-written DSP code.

Table 1 shows the results of the CFG construction algorithm on several test programs. All the programs used contain at least one branch. Column 1 shows the total number of branches, and the number of conditional branches in parentheses. Columns 2 and 3 give the number of loops in the program and their lengths. Column 4 shows the number of basic blocks in the constructed control flow graph, and column 5 states how many of those blocks were caused by instruction copying while creating the basic blocks of the CFG or while moving delayed instructions. Column 6 gives the number of edges in the CFG. Finally, columns 7 to 11 provide a code size comparison between the original assembler code and the code described by the CFG, as well as the number of variables that have been added in order to resolve data conflicts while removing the delays.

The following section introduces a concrete example showing the control flow graph obtained using the techniques presented in this paper as well as a simple form of C translation of the code.

6.1 Example

Figure 8 shows the assembler code of a lattice filter for the TIMS320C60x architecture. To simplify the presentation, the code omits execution unit information.

This example contains 3 delayed branches, all of which jump to label *LOOP*. The CFG generated by our program is shown in figure 9. We can observe that the second and third blocks of the graph contain the two first iterations of the loop. They are originated by the first two branches in the assembler code. The fourth block contains the actual loop body. Due to the delays associated with the multiplication and load instructions, the two first iterations of the loop are not identical to the remaining iterations.

Due to a data conflict between instructions `MV A3, A0` and `MPYHL A4, A0, A2`, temporary variables have been added when moving the latter instruction.

Finally, figure 10 shows the C code that we would obtain from our CFG after sequentialization of the instruction bundles using a simple direct translation from our IR into C. (An optimizing compiler should find many opportunities for simplifying the code.)

```
latsynth:
          B      LOOP
||        ZERO   A7
||        ZERO   B7
          MVK    7, A1
||        ADD    B4, B4, B1
          B      LOOP
||        ADD    B1, A6, A6
||        ADD    B1, B6, B6
          ADD    3, B4, B0
LOOP:
   [B0]   B      LOOP
||        SHR    A2, 16, A5
||        MPY    1, B1, B5
||        MPYHL  A4, A0, A2
||        MV     A3, A0
||        LDH    *--A6, A3
||        LDH    *--B6, B7
   [A1]   ADD    -1, A1, A1
|| [B0]   ADD    -1, B0, B0
|| [!A1]  STH    B4, *+B6[7]
||        ADD    A5, B5, B4
|| [B0]   SUB    A4, A7, A4
||        MPY    1, B7, B1
||        MPY    B7, A3, A7
          SHR    A4, 16, A6
          STH    A6, *+B6[6]
```

Figure 8. Example Assembler Code

7 Conclusions and Further Work

We have presented an algorithm for computing the CFG of an assembly language program where instructions may have associated delay slots. The final version of the CFG is formed from basic blocks whose instructions have no associated delays. This program version is much more amenable to further analysis and, in particular, to reverse compilation processing than the original form.

The algorithm has been incorporated into a reverse compiler for the TI C60 digital signal processor and applied to the reverse compilation of some DSP kernels.

The biggest limitation of the algorithm is that the destinations of the branch instructions must be determinable by simple inspection of the assembler code. An enhancement to the algorithm so that it performs abstract interpretation, tracking register contents along execution paths, would enable the analysis to handle some uses of indirect branches. They are commonly used to implement switch statements and computed goto statements.

Acknowledgements

This work was supported in part by the Christian Doppler Forschungsgesellschaft and Infineon.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992.

[3] A. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[4] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, School of Computing Science, July 1994.

[5] K. D. Cooper, T. J. Harvey, and T. Waterman. Building a control flow graph from scheduled assembly code, June 2002.

[6] S. K. Debray, W. Evans, R. Muth, and B. D. Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000.

[7] A. M. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings: 5th International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, May 1994.

[8] R. Gupta. A code motion framework for global instruction scheduling. In *International Conference on Compiler Construction*, LNCS 1383, pages 219–233, Lisbon, Portugal, 1998. Springer Verlag.

[9] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[10] T. I. Inc. Tms320c6000 cpu and instruction set reference guide, 2000.

[11] J. Janssen and H. Corporaal. Controlled node splitting. In T. Gyimothy, editor, *Compiler Construction, 6th International Conference*, volume 1060 of *Lecture Notes in Computer Science*, pages 44–58, Linköping, Sweden, April 1996. Springer.

[12] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy. Interprocedural static slicing of binary executables. In T. Gyimothy, editor, *Source Code Analysis and Manipulation*, pages 118–127, Linköping, Sweden, September 2003. IEEE.

[13] A. Krall, S. Farfeleder, and N. Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. In J. Takala, editor, *SAMOS 2005*, LNCS 3553, pages 222–231, Samos, July 2005. Springer.

[14] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993.

[15] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In H. R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, pages 1013–119, Las Vegas, USA, June 2000. CSREA Press.

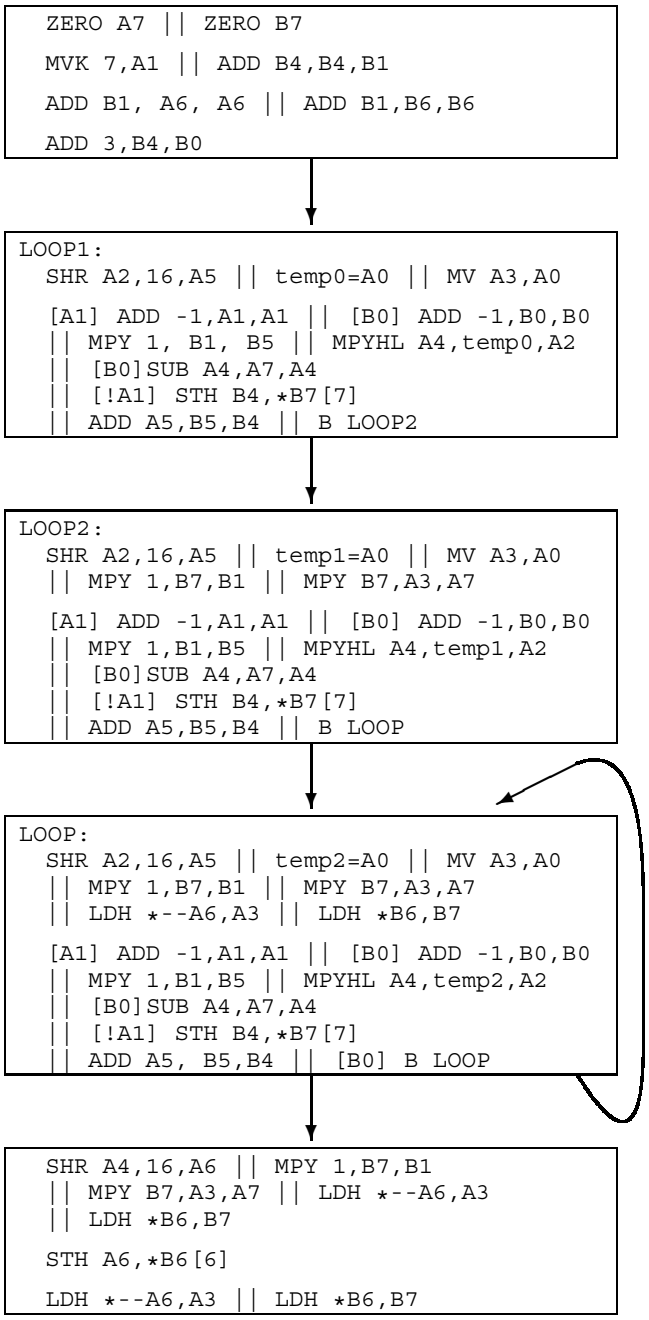

Figure 9. CFG for Figure 8 after Delay Resolution

```
        A7 = 0;
        B7 = 0;
        A1 = 7;
        B1 = B4+B4;
        B0 = 3+B4;
        temp0 = 0;
        A5 = A2>>16;
        A0 = A3;
        if (A1) A1 = (-1)+A1;
        if (B0) B0 = (-1)+B0;
        B5 = 1*B1;
        A2 = (A4>>16)*(temp0 & 0xFFFF);
        if (B0) A4 = A4-A7;
        if (!A1) *(B7+7) = B4;
        B4 = A5+B5;
        goto LOOP2;
LOOP2:
        temp1 = 0;
        A5 = A2>>16;
        A0 = A3;
        B1 = 1*B7;
        A7 = B7*A3;
        if (A1) A1 = (-1)+A1;
        if (B0) B0 = (-1)+B0;
        B5 = 1*B1;
        A2 = (A4>>16)*(temp1 & 0xFFFF);
        if (B0) A4 = A4-A7;
        if (!A1) *(B7+7) = B4;
        B4 = A5+B5;
        goto LOOP;
LOOP:
        temp2 = 0;
        A5 = A2>>16;
        A0 = A3;
        B1 = 1*B7;
        A7 = B7*A3;
        A3 = *(--A6);
        B7 = *B6;
        if (A1) A1 = (-1)+A1;
        if (B0) B0 = (-1)+B0;
        B5 = 1*B1;
        A2 = (A4>>16)*(temp2 & 0xFFFF);
        if (B0) A4 = A4-A7;
        if (!A1) *(B7+7) = B4;
        B4 = A5+B5;
        if (B0) goto LOOP;
        A6 = A4>>16;
        B1 = 1*B7;
        A7 = B7*A3;
        A3 = *(--A6);
        B7 = *B6;
        *(B6+6) = A6;
        A3 = *(--A6);
        B7 = *B6;
```

Figure 10. C Code for Figure 8