Building a Control-flow Graph from Scheduled Assembly Code

Keith D. Cooper, Timothy J. Harvey and Todd Waterman*

ABSTRACT

A variety of applications have arisen where it is worthwhile to apply code optimizations directly to the machine code (or assembly code) produced by a compiler. These include link-time whole-program analysis and optimization, code compression, binary-to-binary translation, and bit-transition reduction (for power). Many, if not most, optimizations assume the presence of a control-flow graph (CFG). Compiled, scheduled code has properties that can make CFG construction more complex than it is inside a typical compiler. In particular, branch-to-register operations can introduce spurious edges into the CFG. If branch delay slots contain other branches, the classic algorithms for building a CFG produce incorrect results.

This paper uses two simple examples to explain the problem. It presents an algorithm for building correct CFGs from scheduled assembly code that includes branches in branch delay slots. The algorithm works by building an approximate CFG and then refining it to reflect the actions of delayed branches. If all branches have explicit targets, the complexity of the refining step is linear with respect to the number of branches in the code.

1. INTRODUCTION

Increasingly, systems are applying compiler technology to previously compiled code. The Dynamo system interprets statically-compiled, executable code to improve performance by using dynamic information to improve scheduling and cache management [3]. Link-time systems perform whole-program analysis and optimization; they start from the compiled code for each procedure or module [13]. Just-in-time compilers for Java take compiled bytecodes as input and rapidly produce machine code for performance-critical regions [21]. Binary translation systems read in executable code and rewrite it for another instruction set [9]. In the past, such systems have been used for emulation; in the future, they will be used to perform load-time tailoring in Grid environments [11, 4]. Each of these systems reads and manipulates previously compiled code.

The control-flow graph (CFG) is a fundamental data structure needed by almost all the techniques that compilers use to find opportunities for optimization and to prove the safety of those optimizations. Such analysis includes global data-flow analysis [17, 16], the construction of an SSA-graph [8], and data-dependence analysis [15, 12]. Other techniques use the CFG to guide a more local analysis and replacement phase [22, 6, 19]. These techniques all assume the existence of a CFG. If the source code for the transformation is compiled, scheduled code, then the CFG construction must handle the additional complexity that can arise in such code.

Compiled code differs from the intermediate forms used inside most compilers. Two particular features can complicate CFG construction. Branches that target an address held in a register (as opposed to an immediate constant) introduce a level of uncertainty that can produce spurious edges in the CFG. The compiler can avoid such branches in its intermediate representations; they are more likely to appear in compiled, scheduled, assembly code. Branch delay slots exacerbate the problem of finding the first and last operation in each block. If branches can occupy the delay slot of another branch, the problem becomes much more complex.

Branch-to-register operations complicate CFG construction because the compiler may be unable to determine the branch targets. When this happens, the compiler must add an edge from the block containing the branch

*Authors' address: Department of Computer Science; Rice University, MS 132; Houston, TX, USA 77005. Corresponding author: waterman@rice.edu

to every block that it might reach. Naively, this set contains every block. The compiler can narrow the set by finding all of the labels that the program loads into registers. (This is safe unless the program performs arithmetic on a label value and branches to the result.) It can perform more precise analysis, similar to that required in call-graph construction with function-valued parameters [7]. First, however, it needs an approximate CFG that overestimates the set of potential paths.

Branch delay slots complicate the task of finding the first and last operation in each block. If the delay slots contain ordinary operations (no control-flow into or out of the delay slots), then this just requires an additional counter to track where in the instruction stream the branch takes effect. If the delay slots contain branch operations, then the compiler must maintain counters for all pending branches. When multiple branches target the same label (*i.e.*, the block has multiple predecessors in the CFG), the compiler must handle the effects of multiple sets of pending branches. Each of these pending branches can terminate a block and add one or more edges to the CFG. These effects cause the classic algorithms for CFG construction [1, 16, 17] to fail—building a CFG that does not correctly reflect the potential flow of control in the code.[1] Sorting out all of these effects adds significant complication to the CFG constructor.

In a more traditional setting, the compiler writer can avoid these problems. Careful design of the intermediate code can let the compiler avoid using the branch to register construct internally, for most source language constructs. When such a construct must be used, the compiler can annotate the operation with labels that correspond to the source-code statements that might be targets of the branch. Similarly, since most uses of a CFG occur before scheduling, the compiler can avoid dealing with delay slots completely in the CFG construction. (If the compiler performs allocation after scheduling, it may can preserve the scheduler's CFG for later use in the allocator.) In a system that handles scheduled code, however, the compiler cannot avoid these problems.[2]

We first encountered this problem while building an assembly-to-assembly translator for Texas Instruments' TMS320C6000, a high-performance DSP chip. As with other emerging architectures, the C6000 allows branches to issue in the delay slots of other branches. Since the branch latency on the C6000 is five cycles, the compiler has many delay slots to fill. The compiler uses this feature to generate efficient, albeit cryptic, code [23]. When the body of a loop is shorter than the branch latency, the compiler can pre-schedule multiple loop-ending branches to create an efficient loop. The resulting loop begins with several consecutive branches. The branches are followed by the instruction or instructions in the loop body, and another loop-ending branch. At run-time, every loop-ending branch, after the first, will execute in the delay slot of another branch. This leads to code that executes efficiently, but is difficult to analyze.

As branch delays become longer, we expect that more architectures will experiment with this feature. Some commodity architectures, such as the Sparc V.9, already include it [24].

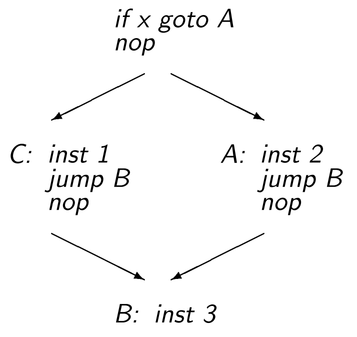
Systems that consume and analyze compiled code must be prepared to handle correctly branches in the delay slots of other branches. The main result in this paper is a worklist algorithm that constructs a correct CFG for such code. When applied to code that does not use this feature, the algorithm has the same complexity as the classic CFG construction algorithms. Constructing these CFGs provides us with the opportunity to perform meaningful optimizations in a new framework.

2. A SIMPLE EXAMPLE

To illustrate the complexity that arises when branches issue in branch delay slots, consider the following code fragment:

```
if x
    then inst 1
    else inst 2
inst 3
```

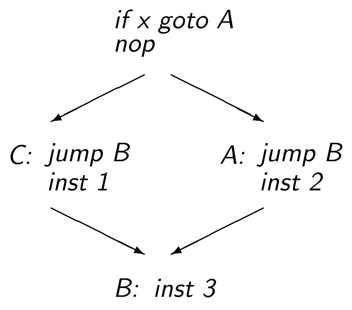
A naive scheduling of this code for a single delay slot architecture produces the following CFG:



[1]These authors assume that the CFG is built from a well-behaved intermediate representation that does not include branches in delay slots. Other authors simply assume that CFG construction is well understood and omit the algorithm entirely [10, 14, 2].

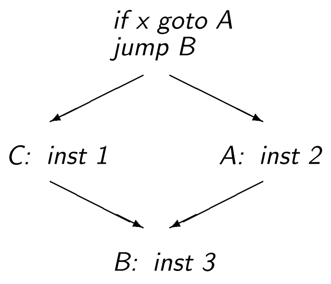
[2]Some systems, such as OM and `alto`, convert scheduled code back to a higher-level representation that does not contain delay slots [20, 18]. With branches in delay slots, this may not always be possible.

A `nop` is inserted after each branch instruction in the code fragment to fill the delay slots. When the compiler tries to fill the delay slots, it can eliminate the delays in blocks A and C:



Unfortunately, the `nop` in the start block remains, since there are no other instructions in the block that can be moved into the delay slot.

If branches can be placed inside of delay slots, an aggressive compiler can trim the schedule even further. The jump instructions at the beginning of blocks A and C (above) can be promoted to the start block and combined since they have the same target. This results in the following CFG and assembly code:



```
    if x goto A
    jump B
    inst 1

A:  inst 2

B:  inst 3
```

The assembly code shows that the existence of branches within delay slots can quickly become confusing. It is not locally evident from examining blocks A and C why control flow proceeds to block B. The common assumption that the instruction that causes the termination of a basic block is located within the same basic block is no longer valid.

The situation quickly becomes more complicated than this simple example. Given an architecture with a large number of delay slots and a program with any number of branch instructions scheduled into the delay slots of other branches, the resulting CFG can become littered with many small basic blocks that do not have a clear or obvious path leading to them. In addition, cycles of branches can be created where the branches are in each others' delay slots. As a result, the CFG construction algorithm cannot complete in a single pass. This necessitates a more complex approach.

```
A:  if x goto B          A:  if x goto B
    inst 1                   inst 1
    inst 2                   inst 2
    jump C                   jump C
    inst 3                   inst 3
    inst 4                   inst 4
B:  if y goto A
    inst 5               B:  if y goto A
    jump C                   inst 5
C:  inst 6                   jump C
    inst 7
    inst 8               C:  inst 6
                             inst 7
                             inst 8

Original Code            After Step One
```

Figure 1: Continuing Example

It may appear that this problem can be solved with replication. This notion is misleading for several reasons. First, such replication can cause significant code growth. Second, replication can easily invalidate the results of register allocation and scheduling. Finally, to understand what to duplicate and where to put it, either the compiler needs the CFG built by our algorithm, or it is forced to duplicate the kind of simulation that the worklist step performs.

Our new algorithm for CFG construction has three distinct steps: detecting and marking labels, adding standard control flow, and adding control flow that originates in delay slots. The first two steps constitute the standard CFG-construction algorithm. They take time that grows linearly with the program's length. If there are no branches in delay slots, they construct a valid CFG. When branches occur in delay slots, the third step is needed to model the program's behavior and construct the corresponding control flow.

3. THE BASE ALGORITHM

Without branches in delay slots, CFG construction takes two steps. The first step partitions the code into a set of basic blocks (maximal length sequences of straight-line code). These become the nodes in the CFG. The second step looks at the branches in the code and fills in the CFG's edges to represent the flow of control. These steps correspond to the two situations that can terminate a basic block—either a label or a branch. If the code has branches in the delay slots of other branches, the CFG construction begins with these same two steps.

We will use the code fragment on the left side of Figure 1 as a continuing example to illustrate each step of the algorithm. It assumes an architecture with two delay slots on each branch.

```
$block\_list$ = initial list of blocks
for each block $b$ in $block\_list$
    remove $b$ from $block\_list$
    $branch\_found$ = false
    for each instruction $i$ in $b$
        if $i$ is a branch
            let $branch\_found$ = true
            let $countdown$ = branch-latency
            break
    if $branch\_found$
        for each instruction $p$ in $b$ after $i$
            decrement $countdown$
            if $countdown = 0$ break
        if $countdown = 0$
            split $b$ at $p$
            let $b'$ = remainder of $b$
            add $b'$ to $block\_list$
            add edges from $b$ to targets of $i$
            if $b$ is conditional add edge to $b'$
    if not $branch\_found$ or $countdown > 0$
        add edge from $b$ to fallthrough of $b$
```

Figure 2: Handling normal control flow

The first step detects labels using a single linear pass that splits each label off to form the beginning of a new basic block and a table is created with the location of each label. The right side of Figure 1 shows how the original code is broken up into basic blocks by the presence of labels. For simplicity, we assume that branches can only target labels and not arbitrary PC addresses. (See the earlier discussion.)

Given the initial set of basic blocks, the algorithm can add normal control flow. It does this in a second linear pass which is detailed in Figure 2. Each branch that is not in a delay slot triggers the creation of a counter with a value equal to the number of delay slots supported by the architecture. The counter is decremented for each additional instruction examined, and no further counters are created until it reaches zero; *i.e.*, subsequent branches are, for now, ignored. When the counter reaches zero, the basic block is split at that point, and edges are added to all possible targets of the branch. This produces the CFG in Figure 3.

If the current block ends before the counter reaches zero, the counter is discarded without adding edges to the branch's targets. (This can only occur when a label occurs in one of the branch's delay slots.) These edges will be added in the algorithm's third pass. Instead, the algorithm adds an edge from the current block to the block begun by the labeled statement.

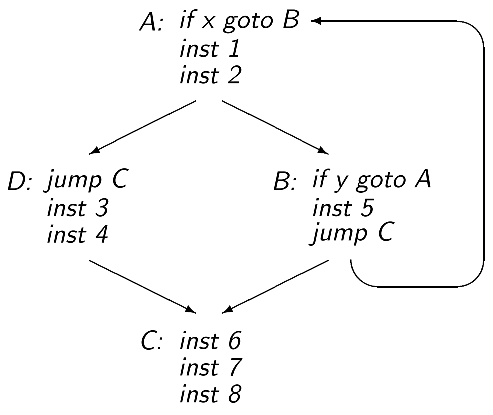


Figure 3: After adding normal control flow

If the target machine does not allow branches in the delay slots of other branches, but does allow a transfer of control to an operation that occupies the delay slot of another branch, this situation can be handled by simply replicating the operations that fall in both blocks. This creates a label-free copy of the code in the delay slots, and a separate copy with the labels from the original code. This requires, at most, one copy of each operation in the delay slots of that branch, so the cost is minimal.

The third and final step adds control flow that results from branches in delay slots that are ignored by the previous steps. The algorithm simulates the control flow of the program, this time taking into account control-flow instructions in delay slots. These branches can necessitate splitting the initial blocks, which, in turn, affects the continuing walk. If the example did not include the jump to label C in block B, then the CFG built in step two (shown above) would be correct and the third step, shown in the next section, would be unnecessary.

4. THE ITERATIVE ALGORITHM

At the completion of the algorithm's second step, the approximate CFG consists of blocks that either end with a branching instruction and up to k instructions in delay slots, or end with no branch. The delay-slot instructions may be ordinary operations, NOPs, or (as yet) unconsidered control-flow instructions. On a given architecture, control-flow instructions take k cycles to *activate* – that is, k cycles after a control-flow instruction issues, control shifts accordingly. If a control-flow instruction, BR_1, executes in, for example, the second delay slot of a control-flow instruction, BR_0, control will shift to one of BR_1's targets two instructions into the block targeted by BR_0. Thus, any block reached through BR_0 will end on its second instruction when BR_1 activates. To model this in the CFG, the CFG-builder must break the targeted block after two instructions and add edges that lead to the block (or blocks) targeted by BR_1.

```
$worklist$ = {start-block:Ø}
while ($worklist$)
    remove element $e$ from $worklist$
    **process-block**($e.block$, $e.list$)

**process-block**($block$, $counter\_list$)
    if $block$ has been seen with $counter\_list$ before
        break
    for each instruction $i$ in $block$
        decrement counters in $counter\_list$
        if $i$ is a branch
            $counter\_list = counter\_list + \{i : \text{branch-latency}\}$
        if any counter in $counter\_list = 0$
            break for
    if $i$ is not at end of $block$
        create new block with remaining instructions in $block$
        add edge from $block$ to new block
    if no counter in $counter\_list = 0$
        let $f$ = $block$'s fall through block
        $worklist = worklist + \{ f : counter\_list \}$
    else
        let $j$ = branch instruction in $counter\_list$ with (counter = 0)
        for each target block $t$ of instruction $j$
            add edge from $block$ to target $t$
            $worklist = worklist + \{ t : counter\_list - \{j : 0\}\}$
```

Figure 4: Pseudo-code for the worklist algorithm

The algorithm proceeds in a symbolic walk over the CFG. As control passes from one block to another, the algorithm passes to the target blocks a list of pending control-flow instructions with a countdown timer for each that shows when it will activate. We call these data structures *branch counters*; each instance is a pair containing the pending branch and a numerical counter that represents the number of cycles remaining before the branch activates. At each block, the algorithm walks through the instructions in the block, in order, counting down each of the branch counters until one reaches zero. When a counter reaches zero, it breaks the block at that point, adds an edge from the shortened block to the remainder of the block,[3] and adds an edge from the shortened block to each of the targets of the activated branch. Any remaining branch counters in the list are replicated and passed to each of the new target blocks.

The algorithm continues in this way, processing blocks until no block has a new branch counter. To make this efficient, we implement the algorithm with a worklist, adding a block to the worklist each time it gets a new branch counter. A block and its associated branch counters represent a specific control-flow path that reached the block. Hence, a block can be on the worklist more than once at a single point in time with each different set of branch counters denoting a different path to the block. It is critical that the algorithm only adds a block when that block is assigned a distinct, new branch counter. This restriction ensures that the algorithm terminates. Pseudo-code for the algorithm is shown in Figure 4.

The worklist algorithm continually calls `process-block` on the first element in the worklist until the worklist is empty. `Process-block` accepts a basic block to examine and a list of branch-counters. The list of counters represents those branches that were still pending when control flow passed to the current block along some path. `Process-block` examines each instruction, adding new branches to the list of counters, and decrementing the counters that already exist. When some counter reaches zero, it creates a new block with the remaining instructions, and adds each target of the branch whose counter reached zero to the worklist with the remaining counters. If no counter reaches zero before

[3]We assume that there is no dead code in the scheduled, compiled code that the algorithm takes as its input. If this assumption is not justified, then the branch from the shortened block to the block created to hold its remainder may be spurious. A simple postpass on the final CFG can detect this situation and remove the dead branch and block.

the end of the block, the block's fall-through block is added to the worklist with the current list of counters. A block's fall-through is the one immediately following the block in the input stream.

Returning to our continuing example, block A, the start block, begins on the worklist. Processing the start block does not change the CFG, because there is no extraordinary control flow; only one branch is encountered and its counter reaches zero when the block ends. Upon completion, block A's successors, blocks B and D, are added to the worklist.

Processing block D causes no changes. Only Block B contains a branch within a delay slot. When `process-block` reaches the end of block B, the branch counter associated with the jump instruction will not have completed. Therefore, the possible successors of the terminating branch, A and C, are placed on the worklist with the outstanding branch counter.

When block A is reexamined, the inherited counter will complete two instructions into the block. This forces the algorithm to split the block after the second instruction and add a new edge from the shortened block A to block C, as shown in Figure 5. In addition, since the branch counter associated with the branch at the beginning of block A has not completed, it is propagated to the newly created block E and to block C.

Block E does not change when it is processed again, but block C is split after the first instruction, and an edge is added back to block B due to the branch from block A. Block B does not need to be placed on the worklist again, since it has already been visited and there are no new branch counters passed in. The newly created block F is processed, but, because processing the block does not deal with any branch counters, it remains unchanged.

Next, block C must be processed again with the branch counter inherited from block B. Since block C has been reduced to a single instruction, the counter is decremented and passed on to block F, which is added to the worklist. Block B is not added to the worklist, because no counter reaches zero when the block completes, so only the fall-through successor is added to the worklist. This correctly conveys the fact that there is no possible control flow path from block B into block C that branches back to block B.

Finally, block F is processed with the branch counter and is split with a branch back to block C. Block G is also added to the worklist and processed, but it has no affect on the CFG.

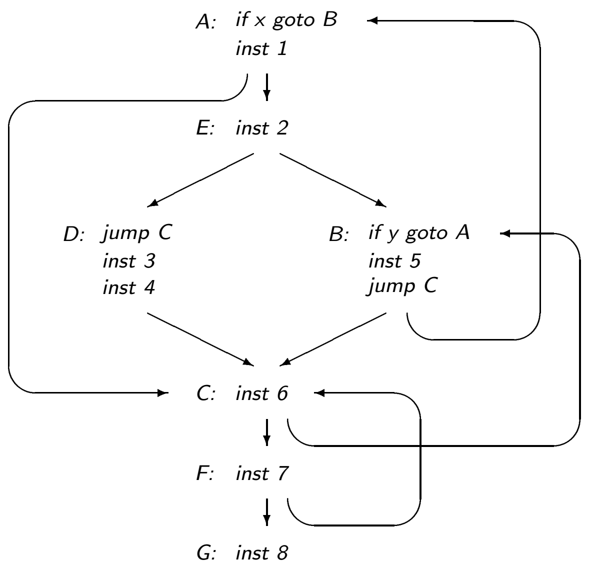


Figure 5: The final CFG

Note that the order of blocks chosen from the worklist is irrelevant. Although the cuts in our example would have happened differently if we had removed the blocks in a different order, the final CFG will be the same in all cases.

5. TERMINATION AND CORRECTNESS

The worklist step terminates because it cannot consider a given ⟨*block,counter list*⟩ pair more than once. The code explicitly checks for this case in lines 7 and 8. The counter list consists of up to k branch counters, where k is the number of delay slots that follow a branch. Each branch counter is a branch operation and a number c in the range $0 \leq c \leq k$. The number of branch counters is finite, $\mathsf{O}(k \cdot b)$, where b is the number of branches. (Of course, $b \leq i$, where i is the number of instructions.) Thus, the number of counter lists is finite. Since the number of blocks is also finite, the set of ⟨*block,counter list*⟩ pairs is finite and the algorithm terminates.

The number of possible ⟨*block,counter list*⟩ pairs looks large. The algorithm considers all paths of length k that start from a branch operation. This allows it to construct the correct and precise CFG. We can speed up the algorithm by having it consider individual branch counters, rather than counter lists. However, that algorithm can add spurious edges to the final CFG—edges that cannot arise in any execution.

Correctness can be proven through contradiction. Assume that there is a non-dead branch statement whose associated edge is not in the final CFG. Line 24 of the algorithm shows that any branch that is added to the counter list has the appropriate edge created; hence, the branch without an edge must not have been added to a counter list. Lines 9 and 11 further show that if a block is encountered by `process-block` all counters within the block must be added to a counter list. Therefore, the block which contains the branch statement must not have been processed. However, since all blocks placed on the worklist are processed by line 4, and all targets of a branch are added to the worklist by line 25, the block must not be reachable from the start block. This contradicts the original assumption that the branch statement is not dead. Therefore, every non-dead branch statement must have an associated edge in the final CFG, and construction of the CFG is correct.

6. COMPLEXITY

The complexity of each pass can be considered separately. The first step examines each instruction once and performs $\mathsf{O}(1)$ work at each instruction. Thus, it takes $\mathsf{O}(i)$ time, for i instructions.

The second step also examines each instruction once. On most operations, it takes $\mathsf{O}(1)$ time. For a branch, however, it must add j edges, where j is the number of potential branch targets—the branching factor. Thus, the time for the second step is $\mathsf{O}(i + j \cdot b)$, where b is the number of branches. If all of the branches of the program have explicit targets, then j is two, and the second step requires $\mathsf{O}(i)$ time. However, branches with ambiguous targets, such as a branch-to-register, produce a higher value of j. For such branches, j is the number of values that the register might have. In the worst case, j is $\mathsf{O}(i)$, and the cost of the second step is $\mathsf{O}(i^2)$. Taken over the entire second step, however, the work will be proportional to the number of edges in the CFG, given by $j \cdot b$.

The third step invokes `process-block` on every ⟨*block, counter list*⟩ pair that appears on the worklist. Thus, an upper bound on its cost is the number of these pairs. We can view the counter list as a list of k elements, where an element is either a branch counter or a token indicating a counter with no branch. (This corresponds to a delay slot that is filled with a non-branching operation.) The number of such counters appears to be $\mathsf{O}(b^k)$.

Fortunately, the structure of the code restricts the set of valid branch counters. Assume that the list is kept in increasing order. If the first slot is occupied by some branch B, the second slot must be occupied by the null token or by a branch that is reachable in one cycle from B. The number of such branches is j, the branching factor used above. The third slot must contain either the null token, or a branch reachable from the second branch, and so on out to the k^{th} position. The number of counter lists that can result from a specific branch B is limited to $\mathsf{O}(j^k)$. Thus, the number of distinct items that can appear on the worklist is $\mathsf{O}(j^k \cdot b)$.

Thus, the complexity of the third pass dominates the overall complexity. The overall complexity of the algorithm derives from this bound. The algorithm calls `process-block` at most $\mathsf{O}(j^k \cdot b)$ times. `Process-block` examines each operation, taking at most $\mathsf{O}(j)$ time per operation. Blocks that `process-block` examines twice can be no longer than k instructions, since the first trip through `process-block` will split the block within k instructions of the entry. Thus, the worst case complexity of the third step is $\mathsf{O}(j^k \cdot b \cdot k \cdot j)$, or $\mathsf{O}(j^{k+1} \cdot b \cdot k)$.

In practice, the worst case complexity depends heavily on the branching factor and the number of delay slots. With branches that have explicit targets, j is usually two. The number of delay slots is typically small. For example, $k = 1$ on the Sparc and $k = 5$ on the C6000. With $j = 2$ and $k = 1$, j^k is a small constant and the algorithm runs in $\mathsf{O}(2^2 \cdot i \cdot 1) = \mathsf{O}(i)$ time. Adding a small number of delay slots without adding ambiguous branches raises the constant, but not the asymptotic limit. Adding ambiguous branches with a single delay slot ($j = i$ and $k = 1$) produces a worst case complexity of $\mathsf{O}(i^2)$. The combination of ambiguous branches and multiple delay slots causes the complexity to explode.[4] However, the increased complexity reflects the number of potential paths that the algorithm must consider. Each of these paths requires a constant amount of work. The increase in complexity in the algorithm, therefore, is solely a function of the increase in the number of these paths.

7. CONCLUSION

Recent years have seen a number of systems that consume as input compiled code that has already been optimized, scheduled, and allocated. These systems perform optimizations that require data-flow analysis computed over the CFG. However, the presence of branches in branch delay-slots complicates the construction of a CFG from compiled code and causes the classic algorithms for building a CFG to produce incorrect results.

This paper presents a method to correctly build the CFG for scheduled code in the presence of branches within delay slots. A three-pass algorithm is used to construct the CFG; the first two passes build the “normal” CFG, and the third pass uses a worklist algorithm to propagate branch information from block to block to construct the control flow associated with branches in delay slots. Decomposing the algorithm into separate steps simplifies its explanation and allows the algorithm to bypass the final step if the code does not include

[4]This provides yet another reason why compilers should avoid ambiguous branches whenever possible!

branches in branch delay slots. The running time of the algorithm is dependent on the complexity of the CFG itself – if all branches have explicit targets, the worklist portion of the algorithm is linear. We have implemented this algorithm in an assembly-to-assembly translator for the TMS320C6000.

Acknowledgements

This work has benefited from the support, suggestions, and encouragement of the entire Scalar Compiler Group at Rice University. Reid Tatge of Texas Instruments has patiently answered our many questions about both the TMS320C62xx and the code produced by TI's compiler. This work has been supported by DARPA through USAFRL contract F30602-97-2-298, by a grant from the Texas Advanced Technology Project, and by the NSF through the GrADS project, grant number 9975020.

8. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, pages 1–12, June 2000.

[4] Fran Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crummey, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS project: Software support for high-level grid application development. *Inernational Journal of High Performance Computing Applications*, 15(4), Winter 2001.

[5] Preston Briggs. Drawing control-flow graphs with style. July 1994.

[6] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, June 1997.

[7] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 22(1):171–179, January 1987.

[9] Paul J. Drongowski, David Hunter, Morteza Fayyazi, David Kaeli, and Jason Casmira. Studying the performance of the FX!32 binary translation system. In *Proceedings of the First Workshop on Binary Translation*, October 1999.

[10] Charles N. Fischer and Richard J. LeBlanc. *Crafting a Compiler with C.* Benjamin/Cummings, 1991.

[11] Ian Foster and Carl Kesselman. *The GRID: Blueprint for a New Computational Infrastructure*. Morgan Kaufman Publishers, Inc., San Francisco, CA, USA, 1999.

[12] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. *SIGPLAN Notices*, 26(6):15–29, June 1991. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[13] David W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. *SIGPLAN Notices*, 32(6):122–133, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[14] Allan I. Holub. *Compiler Design in C.* Prentice Hall, 1990.

[15] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact dependence analysis. *SIGPLAN Notices*, 26(6):1–14, June 1991. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[16] Charles R. Morgan. *Building an Optimizing Compiler.* Digital Press, 1998.

[17] Steven S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kauffman, 1997.

[18] Robert Muth, Saumya Debray, Scott Watterson, and Koen De Bosschere. alto: A link-time optimizer for the Compaq Alpha. *Software Practice and Experience*, pages 67–101, January 2001.

[19] Karl Petis and Robert C Hansen. Profile guided code positioning. *SIGPLAN Notices*, 25(6):16–27, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[20] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, pages 1–18, March 1993.

[21] Sun Microsystems, Inc. *The Java HotSpot Virtual Machine*, 2001. Available online at http://java.sun.com/products/hotspot.

[22] Philip H. Sweany and S.J. Beatty. Dominator-path schedule—a global scheduling method. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[23] Reid Tatge. *Private communication*. Several discussions related to the TMS320C6xxx ISA and the code produced by Texas Instruments' compiler for those processors., 2000.

[24] Daniel L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. PTR Prentice-Hall, 2000.