INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI  48106-1346 USA
800-521-0600

UMI®
RICE UNIVERSITY

Post-Compilation Analysis and Power Reduction

by

Todd Waterman

A Thesis Submitted
in Partial Fulfillment of the Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Keith D. Cooper, Professor, Chair
Computer Science,

Ken Kennedy, Professor
Computer Science

Vijay S. Pai, Assistant Professor
Electrical and Computer Engineering

Houston, Texas
April, 2002
Post-Compilation Analysis and Power Reduction

Todd Waterman

Abstract

Optimization outside of traditional frameworks is emerging as a new research focus in the compiler construction community. Scheduled assembly code is one area of increased interest. Optimization cannot be performed without a control-flow graph (CFG), and current CFG construction algorithms can fail on scheduled code. We present a new construction algorithm that correctly constructs CFGs and permits meaningful optimization for scheduled code.

One potential post-compilation optimization is reducing power consumption by minimizing switching activity on the instruction bus. We designed and implemented an algorithm that attempts to minimize switching activity by renaming registers for Texas Instruments' TMS320C6200 processor. We gathered results using a power simulator developed inside Texas Instruments. We determine that reducing bit transitions on the instruction bus is not a profitable technique for reducing the power consumption of this particular microprocessor.
Acknowledgements

First, I would like to thank Keith Cooper for his direction and insight. His guidance was critical to the creation of this document and developing my understanding of compilers. The remainder of my thesis committee, Ken Kennedy and Vijay Pai, also contributed substantially to the shaping of my thesis. Reed Tatge provided assistance with the C6200 and its power simulation. Tim Harvey's door was always open and he was always willing to discuss my work or anything else.

Finally, I would like to thank my family and Nina. They took an active interest in something they did not understand, and never complained about my not having a real job.
## Contents

Abstract i
List of Illustrations v
List of Tables vi

1 Introduction 1

2 Control-flow Graph Construction 4
  2.1 Introduction ........................................ 4
  2.2 A Simple Example ................................... 7
  2.3 The Base Algorithm ................................ 9
  2.4 The Iterative Algorithm ............................ 13
  2.5 Termination and Correctness ....................... 17
  2.6 Complexity .......................................... 18
  2.7 Conclusion .......................................... 20

3 Reducing Bit Transitions 21
  3.1 Introduction ......................................... 21
  3.2 Related Work ........................................ 22
  3.3 The General Algorithm ............................. 24
  3.4 Architectural Restrictions ......................... 26
  3.5 Results and Analysis ............................... 27
  3.6 Future Improvements ............................... 30
  3.7 Conclusion .......................................... 31
4 Conclusion

Bibliography
Illustrations

2.1 Continuing Example .............................................. 10
2.2 Psuedo-code for creating normal control flow .............. 11
2.3 Example after adding normal control flow ................... 12
2.4 Pseudo-code for the worklist algorithm ..................... 13
2.5 The final CFG .................................................... 16

3.1 Pseudo-code for RAG construction ............................ 25
3.2 Pseudo-code for register renaming ............................ 26
Tables

3.1 Average bit transitions on the instruction bus . . . . . . . . . . . . 28
Chapter 1

Introduction

Compilers transform programs written in high level languages into programs that can execute on a computer. Optimizing compilers further attempt to minimize the resource requirements of the resulting program. Over the past 40 years, a great deal of attention has been directed towards minimizing the time and space requirements of programs. The power requirements of a program, however, have long been neglected by compiler writers.

Power consumption has recently become an issue of increased interest with the proliferation of mobile computing devices: cellphones, PDAs, and laptops. Mobile devices have a limited power supply, and power consumption directly affects the battery lifetime of the device. Hardware designers have spent significant effort designing processors that require less power, while the concept of designing software to consume less power has been left relatively unexploited.

The compiler can play an important role in reducing power consumption, since it can generate observationally equivalent programs that consume different amounts of power. Instruction selection, transitions between instructions, and the use of functional units are just some issues governed by the compiler that impact power consumption. Optimizations geared towards reducing the running time of the program often reduce the energy consumption of the program as well.

We, however, are concerned with transformations dealing purely with power. Once a program has been optimized for time and space, can we further optimize it for power without harming the other resource requirements of the program? For this reason, we chose to investigate the effect of bit transitions on power consumption.
Previous research suggested that bit transitions between the fetch packets of an executable were responsible for a significant amount of power consumption [9, 18]. A fetch packet is a set of instructions that are loaded from memory in parallel. Power consumption can be reduced by minimizing the number of bits that change on the instruction bus when a fetch packet is loaded. Many of the bits in a fetch packet represent the encoding of opcodes and other critical information that is dictated by the architecture and cannot be changed. However, almost half of the bits in a fetch packet represent instruction operands. These operands can be renamed to reduce the number of bit transitions between fetch packets while preserving the meaning of the code.

Renaming operands in an intelligent manner to reduce power consumption first requires data-flow analysis: compile-time reasoning about the run-time flow of values. Unfortunately, the renaming step must be performed after scheduling and allocation – usually the final steps in compilation – and little effort has been expended on how to perform data-flow analysis at this point. Data-flow analysis is commonly performed using a control-flow graph (CFG), a data structure where nodes represent basic blocks and edges represent possible branches and jumps. Some architectural features can cause classic CFG construction algorithms to fail. The Texas Instruments' TMS320C6200 has one such feature; it allows branches to execute in the delay slots of other branches. The Texas Instruments compiler uses this feature to perform software pipelining of short loops.

We were motivated to design a CFG construction algorithm capable of handling scheduled code by our interest in reducing bit transitions. However, reducing bit transitions is just one of many applications that require a CFG for scheduled code. Since almost all non-local compiler optimizations require analysis provided by a CFG, this algorithm is critical for performing optimizations on scheduled code. Even local optimization is complicated by the presence of branches in delay slots of other branches. The compiler must perform analysis equivalent to our CFG construction algorithm
just to find the boundaries of basic blocks. Recent work in a variety of areas has shown the optimization of scheduled code to be an increasingly important area.

This thesis has two main contributions. First, it presents an algorithm that can build an accurate control-flow graph on scheduled code even if the code has branches in delay slots. Secondly, it proposes a technique for renaming registers to reduce power consumption using the CFG algorithm and examines its profitability on the C6200, using power simulation tools provided by the manufacturer. We discovered that renaming registers was not an effective method for reducing power consumption on the C6200, because bit transitions on the instruction bus have little effect on the C6200's power consumption, contrary to the predictions in the literature.

The remainder of this thesis is organized as follows. Chapter 2 describes our CFG construction technique for scheduled code. Chapter 3 explains our algorithm to reduce power consumption by renaming registers to reduce bit transitions. It also shows the results of our algorithm on the TMS320C6200. Chapter 4 summarizes the contributions of this thesis.
Chapter 2

Control-flow Graph Construction

2.1 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 [16]. Just-in-time compilers for Java take compiled bytecodes as input and rapidly produce machine code for performance-critical regions [28]. Binary translation systems read in executable code and rewrite it for another instruction set [12]. 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 [5, 14]. 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 [21, 22], the construction of an SSA-graph [10], and data-dependence analysis [15, 19]. Other techniques use the CFG to guide a more local analysis and replacement phase [7, 24, 29]. 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 explicit or immediate constant) introduce a level of uncertainty that can add spurious edges to the CFG. Branch delay slots exacerbate the problem of finding the boundaries of 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 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 [8]. 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, 21, 22] 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.

---

1These 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 [2, 13, 17].
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 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.\footnote{Some systems, such as OM and alto, convert scheduled code back to a higher-level representation that does not contain delay slots [27, 23]. With branches in delay slots, this may not always be possible.}

We first encountered this problem while building an assembly-to-assembly translator for the TI 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 [30]. When the body of a loop is shorter than the branch latency, the compiler can preschedule multiple loop-ending branches to create an efficient loop. This produces a loop that 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 efficient execution, but the code is difficult to analyze.

As branch delays become longer, we expect this feature to continue to exist in future architectures. One new architecture, the Sparc V.9, already includes it [32].

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.

2.2 A Simple Example

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

\[
\text{if } x \\
\quad \text{then inst 1} \\
\quad \text{else inst 2} \\
\quad \text{inst 3}
\]

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

[Diagram of the CFG]

A \textit{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:

```
if x goto A
  nop
C: jump B
  inst 1
A: jump B
  inst 2
B: inst 3
```

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
C: 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 with any number of branch
instructions within 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.

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.

2.3 The Base Algorithm

Without branches in delay slots, CFG construction takes two steps. The first step partitions the code into a set of preliminary 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. It fills in the CFG's edges to represent the flow of control, and further divides basic blocks when necessary. 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 2.1 as a continuing example to illustrate each step of the algorithm. It assumes an architecture with two delay slots on each branch.

A: \text{if } x \text{ goto } B \\
\quad \text{inst 1} \\
\quad \text{inst 2} \\
\quad \text{jump } C \\
\quad \text{inst 3} \\
\quad \text{inst 4}

B: \text{if } y \text{ goto } A \\
\quad \text{inst 5} \\
\quad \text{jump } C \\

C: \text{inst 6} \\
\quad \text{inst 7} \\
\quad \text{inst 8}

A: \text{if } x \text{ goto } B \\
\quad \text{inst 1} \\
\quad \text{inst 2} \\
\quad \text{jump } C \\
\quad \text{inst 3} \\
\quad \text{inst 4}

B: \text{if } y \text{ goto } A \\
\quad \text{inst 5} \\
\quad \text{jump } C \\

C: \text{inst 6} \\
\quad \text{inst 7} \\
\quad \text{inst 8}

Original Code \quad \text{After Step One}

Figure 2.1: Continuing Example

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 2.1 shows how the original code is broken up into a preliminary set of 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 2.2. Each branch 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
\texttt{block\_list} = initial list of blocks
for each block \texttt{b} in \texttt{block\_list}
    remove \texttt{b} from \texttt{block\_list}
    \texttt{branch\_found} = false
for each instruction \texttt{i} in \texttt{b}
    if \texttt{i} is a branch
        let \texttt{branch\_found} = true
        let \texttt{countdown} = branch-latency
        break
    if \texttt{branch\_found}
        for each instruction \texttt{p} in \texttt{b} after \texttt{i}
            decrement \texttt{countdown}
        if \texttt{countdown} = 0 break
    if \texttt{countdown} = 0
        split \texttt{b} at \texttt{p}
        let \texttt{b'} = remainder of \texttt{b}
        add \texttt{b'} to \texttt{block\_list}
        add edges from \texttt{b} to targets of \texttt{i}
        if \texttt{b} is conditional add edge to \texttt{b'}
    if not \texttt{branch\_found} or \texttt{countdown} > 0
        add edge from \texttt{b} to fallthrough of \texttt{b}

Figure 2.2: Pseudo-code for creating normal control flow
block is split at that point, and edges are added to all possible targets of the branch. This produces the CFG in Figure 2.3.

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.

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.
worklist = \{\text{start-block}:\emptyset\}

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 2.4: Pseudo-code for the worklist algorithm

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.

2.4 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 $BR_1$.

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, 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 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

\footnote{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.}
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 2.4.

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 a 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 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
Figure 2.5: The final CFG

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 2.5. 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.

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.

2.5 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, \(O(k \cdot b)\), 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 algo-
rithm 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.

2.6 Complexity

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

The second step also examines each instruction once. On most operations, it takes 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 O(i + j · 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 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 O(i), and the cost of the second step is 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 · b.

The third step calls 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 $O(b^k)$.

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 $O(j^k)$. Thus, the number of distinct items that can appear on the worklist is $O(j^k \cdot b)$.

Thus, the complexity of the third pass dominates the overall complexity. The algorithm calls process-block at most $O(j^k \cdot b)$ times. Process-block examines each operation, taking at most $O(j)$ 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 $O(j^k \cdot b \cdot k \cdot j)$, or $O(j^{k+1} \cdot b \cdot k)$.

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 $O(2^2 \cdot i \cdot 1) = O(i)$ 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 $O(i^2)$. The combination of ambiguous branches and multiple delay slots causes the complexity to explode.\(^4\) However, the increased complexity

\(^4\)This provides yet another reason why compilers should avoid ambiguous branches whenever possible!
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.

2.7 Conclusion

Recent years have seen a number of systems that consume as input compiled, optimized, scheduled code. 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 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.
Chapter 3

Reducing Bit Transitions

3.1 Introduction

Over the last 50 years, the driving force behind both computer architecture and compiler design has been improving the running time of programs. Little attention has been paid to the issue of power consumption outside of its interplay with the speed of a program. However, with the explosion of portable computing devices, power consumption has become a concern independent of program speed. Cellphones, PDAs, and the brake controller on a modern automobile all have power limitations that have forced architects to consider the importance of power consumption in their designs.

Compiler writers are currently behind their hardware counterparts in investigating techniques to reduce power consumption. The role of hardware in power consumption is obvious; if a processor runs at a lower voltage it will consume less power. The impact of compilers on power consumption is less apparent, but may still be important. Speed optimizations often reduce energy consumption since the program has less time to consume power. Independent of optimizations for running time, there are several methods of transforming programs to reduce their power consumption. Instruction selection, scheduling of functional units, and instruction ordering all affect power consumption without necessarily impacting performance.

One area where the compiler can impact power consumption is by reducing bit transitions. Switching costs dominate the power consumption of CMOS circuits [9]. A portion of these switching costs arise from bit transitions on the instruction bus. The instruction bus carries groups of assembly instructions, called fetch packets,
from program memory to the processor. Many of the bits in a fetch packet are encodings of instruction opcodes and other critical information that are specified by the architecture and cannot be changed. Almost half of the bits in a fetch packet, however, represent instruction operands and the opportunity for power optimization.

The majority of instruction operands are registers which can be manipulated to reduce bit transitions. Register allocation normally assigns registers to minimize the insertion of spill code. Register naming, however, is performed pseudo-randomly. Register names can be reassigned to reduce bit transitions without adversely effecting running time.

We implemented a register-renaming algorithm to reduce the number of dynamic bit transitions on the Texas Instruments' TMS320C6200 DSP. Constructing our renaming algorithm on an existing architecture was critical for measuring its impact on both bit transitions and power consumption. Virtual machines rarely have as many quirks and peculiarities that constrain the renaming algorithm. In addition, it is difficult to report valid power measurements for a non-existent architecture. In contrast, we tested our results using a power simulator developed inside Texas Instruments and provided to us for this study.

Our algorithm was able to make a modest improvement in the dynamic bit transition count; unfortunately, this did not translate into a significant power reduction. Bit transitions during instruction fetch proved not to be a major contributor to power consumption on the C6200. This algorithm remains beneficial when small variations in power are significant, or for an architecture where switching activity on the instruction bus plays a heavier role in power consumption.

3.2 Related Work

Interest in software techniques to reduce power consumption has increased considerably over the last few years. The correlation of instruction selection and register allocation to power consumption have both been investigated [11]. Previous research
has also been conducted on minimizing bit transitions.

Shin and Kim attempt to reduce bit transitions during instruction fetch on the C6200 processor [25]. Instead of renaming registers, they reorder operations within a fetch packet to reduce transitions. This reordering scheme only works on a VLIW processor that uses compressed encoding: the position of an operation inside a fetch packet does not correspond to a specific functional unit. Our algorithm, while implemented on a VLIW processor, is not specific to that architecture. Their technique successfully reduces the number of bit transitions, but they do not report changes in power consumption. Also, their bit transition per instruction fetch numbers do not agree with those of the Texas Instruments' simulator for unmodified benchmarks.

Mehta et al. devise another algorithm for renaming registers to reduce power consumption [20]. Their algorithm relies on profiling to guide their optimization instead of static analysis. Their implementation is for the DLX architecture, a research architecture without hardware implementation. This prohibits them from examining the limitations of a physical machine on their algorithm, and requires assumptions regarding power consumption when they simulate their results.

A final interesting technique involving bit transitions is proposed by Benini et al. [4]. They also attempt to reduce the bit transitions between instructions. However, instead of transforming code to reduce transitions they propose designing an architecture that minimizes transitions for commonly run programs. Their approach is orthogonal to ours. We attempt to make software work with hardware, while they try to make hardware work with software.

Our algorithm differs from previous work in several ways. We present a compile-time optimization to reduce switching activity that can be applied to any architecture. We implemented the algorithm for a specific processor to expose architectural limitations and provide meaningful power results.
3.3 The General Algorithm

Our algorithm reduces the dynamic count of bit transitions between fetch packets through register renaming. Fetch packets consist of a fixed number of instructions that are each the same size. Each instruction has a standard layout; register operand names always reside in the same location. Therefore, reducing the number of bit transitions between corresponding registers reduces the bit transitions between fetch packets as well.

A register's bit pattern is just a binary encoding of the register number. Therefore, a transition from register seven (0111) to eight (1000) will require four bit transitions, while a transition from register four (0100) to six (0110) would only require a single transition. The goal of this work, then, is to find a method to place register numbers with similar bit patterns in the same location of adjoining fetch packets.

There is a traditional separation of concern in modern register allocators, which make a distinction between the register allocation phase and the register assignment phase. Allocation determines which live ranges are placed in registers. Assignment takes these live ranges, combines live ranges that do not interfere into groups, and gives each group a physical register. Register naming is the portion of register assignment that places groups into physical registers. Our algorithm changes the mapping of groups to registers to minimize bit transitions between fetch packets.

In order to rename registers effectively, the frequency of transitions between each pair of registers must be determined. This enables the renaming algorithm to give registers with the most interaction the closest bit patterns. Transition frequency is encoded in the construction of a register adjacency graph (RAG). A RAG has a node for each original register name, and the edges between nodes are weighted by the number of times the registers occur in the same position in adjacent fetch packets.

Accurately determining weights in the RAG requires knowledge of how often each instruction executes. We use data-flow analysis to statically estimate the execution frequency of each instruction. First, a control flow graph is constructed using the
build CFG
compute loop-nesting depth \((LND)\) of each block

for \(i = 0\) to \(\text{max\_register\_number}\)
  for \(j = 0\) to \(i\)
    let \(weight(i, j) = 0\)

for each block \(b\)
  for each transition between fetch packets \(f\) and \(f'\) in \(b\)
    for each register \(r\) in \(f\)
      let \(r' = \text{register in same position as } r\) in \(f'\)
      if \(r' > r\), \(\text{swap}(r, r')\)
      let \(weight(r, r') = weight(r, r') + 10^{LND_b}\)

Figure 3.1: Pseudo-code for RAG construction

The technique from Chapter 2. Then the loop nesting depth of each basic block is determined using a DJ graph [26]. Each basic block is assumed to execute \(10^{\text{ind}}\) times, where \(\text{ind}\) is the loop nesting depth of the block. This method is certainly not precise, but correctly encapsulates the dominance of loops on execution frequency [31].

Once execution frequencies have been determined, the RAG can be constructed in a single pass over the code. Each basic block is examined and the transitions between fetch packets are discovered. For each register referenced in the first fetch packet, its partner in the second packet is found. The appropriate edge weight in the RAG is then increased by the loop nesting depth of the block. Pseudo-code for the construction of the RAG can be seen in Figure 3.1.

The number of register bit transitions in the program is the summation of each edge's weight in the RAG multiplied by the number of different bits between the two registers. Our algorithm's goal is to rename registers to minimize this cost. An optimal solution to this problem is NP-complete. Therefore, we guide renaming with a heuristic which runs in linear time with respect to the number of edges in the graph.

Register renaming is performed using a simple, greedy scheme: give the most
let worklist = all edges in RAG
while some registers do not have names
    let \((r_1, r_2)\) = edge in worklist with highest weight
    remove \((r_1, r_2)\) from worklist
    if \(r_1\) has not been renamed
        if \(r_2\) has not been renamed
            assign \(r_1\) and \(r_2\) to registers with the closest bit pattern available
        else
            assign \(r_1\) to the register with the closest bit pattern to \(r_2\)'s assignment
    else if \(r_2\) has not been renamed
        assign \(r_2\) to the register with the closest bit pattern to \(r_1\)'s assignment

Figure 3.2: Pseudo-code for register renaming

frequent transition the lowest cost. All of the edges in the RAG are placed in a worklist. The edge with the highest weight is removed from the worklist, and if the registers have not already been reassigned, they are assigned to the two registers with the closest possible bit pattern. This is repeated until all registers have been reassigned. Code for the algorithm is shown in Figure 3.2.

3.4 Architectural Restrictions

The general register renaming algorithm is motivated solely by the desire to reduce bit transitions. Unfortunately, when this algorithm is applied to a specific machine, other issues must be addressed. We implemented the algorithm on the Texas Instruments' TMS320C6200. This processor has several features that require the adaptation of the general algorithm; some of these are common while others are not. Examining these peculiarities is important, since no particular machine is purely general. Even though the C6200 has 32 general purpose registers, a specific register can only be renamed to five alternatives after architectural restrictions are imposed on the algorithm.

The first feature of the C6200 that disrupts the general algorithm is the existence
of register pairs. The C6200 allows operations to be performed on 32-bit integer values which occupy a single register and 40-bit values which require two adjoining registers. Each register has a specified register that it can be paired with. When performing register renaming, register pairs are not broken apart. If two registers constitute a pair before renaming, they must make up a complete pair after renaming. This constraint is enforced by considering register pairs as atomic units. Treating register pairs atomically increases the granularity of the renaming scheme, which can limit profitability. For example, a transition originally between an odd and even register in different pairs can never be reduced to fewer than two bits.

Another machine-dependent property which constrains renaming is the presence of registers with special properties. The C6200 has two separate register banks which can only be accessed by different functional units. A few of the registers can be used as predicates for instructions, and two of the registers can be used by a special addressing instruction. When renaming a register, it is critical that the new register has all of the same properties as the original. This constraint further reduces the renaming options of our algorithm.

Architectural features and peculiarities have a non-trivial effect on register renaming. The C6200 has 32 general purpose registers. After architectural constraints are considered, ten of the registers cannot be renamed. In addition, each of the remaining registers can only be renamed to another register in the same bank and can only reside in the same half of any pair. Each of these limitations reduces by half the number of registers a particular register can be renamed to. Thus, an individual register has no more than five alternative registers for renaming in the C6200, rather than 31 if there were no architectural restrictions.

### 3.5 Results and Analysis

We tested our algorithm using a power-accurate simulator for the C6200 provided by Texas Instruments. The simulator reports the average number of bit transitions on
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>fir4</td>
<td>26.91</td>
<td>26.59</td>
<td>27.38</td>
</tr>
<tr>
<td>fir8</td>
<td>28.90</td>
<td>29.16</td>
<td>29.27</td>
</tr>
<tr>
<td>vecsumsq</td>
<td>24.42</td>
<td>24.40</td>
<td>25.03</td>
</tr>
<tr>
<td>dotprod</td>
<td>24.12</td>
<td>24.11</td>
<td>24.44</td>
</tr>
</tbody>
</table>

Table 3.1: Average bit transitions on the instruction bus

the instruction bus and a relative power index. A simulator was used since it provides power results specific to the processor we used, but does not involve the complications of physically measuring power consumption.

The benchmarks were also provided by Texas Instruments. They consisted of hand-optimized assembly code for various applications. The use of optimized benchmarks is important since any attempt to minimize power consumption should begin with optimizing the code for speed.

The average number of bit transitions for several benchmarks is shown in Figure 3.1. The Original column shows the average number of transitions before optimization, while the Best column shows the average number of transitions after optimization. The Original data point, however, represents a random point in the space of the possible number of bit transitions. When the code was optimized bit transitions were not considered, allowing the original number to fall anywhere in this range. We also run our algorithm in reverse, trying to increase the number of bit transitions, to see how large a variation in bit transitions can exist, and approximate this range. The results of our reverse algorithm are underneath the Worst column.

The results shown in Figure 3.1 are disappointing. The change in bit transitions from best to worst is never greater than three percent. In one case, the fir8 benchmark, there were fewer bit transitions on the instruction bus before our transformation was applied. This is due to the fact that a heuristic is used to rename registers and
the original naming scheme was excellent. Furthermore, the resulting change in the relative power index is less than a tenth of a percent. We believe that these results are due to two distinct factors.

The limited change in bit transitions on the instruction bus is due to the architectural restrictions discussed in Section 3.4. Previous research suggests that renaming registers can significantly reduce the number of bit transitions for a virtual architecture that has no limitations [20]. The architectural limitations on the C6200 are severe. Instead of being able to completely rename 32 registers, which would be the case if there were no restrictions, ten of the registers cannot be renamed and the remainder are divided into two sets of pairs for renaming.

The lack of variation in the power index is due partially to the small change in bit transitions on the instruction bus, but also to a lower correlation between switching costs during instruction fetch and power consumption than originally assumed. If we assume a linear relation between instruction bus bit transitions and power consumption, even a 30 percent reduction in transitions would reduce power consumption less than one percent. Attempts to reduce power by reducing switching activity on the instruction bus are motivated by the observation that the majority of power consumption in a CMOS circuit results from switching activity [9]. Our results indicate that bit transitions on the instruction bus consist of only a small part of the switching activity for the C6200. However, this may differ for other architectures: A study of the ARM7TDMI processor's power consumption showed a 1.2% reduction in power by eliminating one bit transition per instruction [18].

Our algorithm failed to significantly reduce power consumption for two reasons. First, architectural restrictions limited renaming and prevented a large reduction in bit transitions on the instruction bus. Second, and more crucial, switching activity on the instruction bus accounts for only a small portion of total power consumption.
3.6 Future Improvements

The reduction of bit transitions on the C6200 had a negligible effect on the power consumption of programs. This negated the importance of trying to improve our algorithm to further reduce bit transitions. However, if this algorithm was implemented on an architecture where reducing bit transitions was important, there are other potential improvements.

Our register renaming algorithm is limited in two distinct ways. First, it makes conservative assumptions when dealing with machine specific constraints. Secondly, it uses a heuristic to perform renaming. If reducing bit transitions is critical and the original algorithm is not sufficient, there are ways to overcome both of these limitations.

Currently, a linear-time heuristic is used to rename registers after the RAG is built. The heuristic relies on a small number of heavily weighted edges in the RAG. Our test programs exhibited this trait, but, a program with more uniformly distributed weights could be problematic. Worst case behavior could also result in giving a relatively important edge a large transition cost when renaming even heavier weights.

Using a more powerful heuristic could solve these problems. Power optimization is normally performed once for a program that will execute many times. Running time of the compiler becomes a less important issue as a result. A non-linear heuristic that used back tracking to avoid worst-case behavior could improve results. In addition, formulating the renaming heuristic as an integer-programming problem would certainly improve results. A timing mechanism could be included in the algorithm to ensure that running time would not become prohibitive.

Another possible improvement to the renaming algorithm involves relaxing the constraints caused by machine peculiarities. The current algorithm takes a conservative approach to various machine-dependent concerns of the C6200. It assumes that all registers that can be used as register pairs or predicates are actually used as
such. Additional analysis could be performed to determine which registers are used as predicates or pairs. This analysis would allow the renaming algorithm to only place renaming restrictions on a subset of the possible registers.

Eliminating the limitations of the renaming algorithm is one way of possibly improving performance; another method could be to combine the algorithm with other optimizations. For example, Shin and Kim reorder operations in fetch packets to reduce transitions [25]. This algorithm could be enhanced to produce code with a RAG more amenable to optimization. Another optimization that could help produce a better initial RAG is instruction scheduling. Code could be rescheduled with a priority towards giving adjacent instructions a small number of bit transitions between opcodes and producing a better initial RAG. Both of these techniques could directly reduce the switching activity on the instruction bus and assist the register renaming algorithm.

3.7 Conclusion

Due to the emergence of mobile devices, power consumption of programs has recently become an important concern. Much of the power usage in a circuit is derived from switching activity. We devised a post-compilation transformation to rename registers with the goal of reducing bit transitions on the instruction memory bus of a processor. This algorithm was implemented on the Texas Instruments' TMS320C6200 processor.

Unfortunately, our results on the C6200 were not significant. Reductions in the number of bit transitions in the program were less than expected. This arises from architectural restrictions of the C6200 that limit our renaming algorithm. In addition, the reductions in bit transitions lead to almost no change in the power consumption of test programs. We believe that, on the C6200, bit transitions on the instruction bus do not constitute a significant portion of the total switching activity.

The technique we present is capable of reducing switching activity on the instruction bus, though its profitability can be hampered by architectural restrictions.
This does not significantly reduce power consumption on the C6200. This algorithm could be useful on an architecture where bit transitions during instruction fetch had a greater role in power consumption, or if another purpose existed for reducing bit transitions.
Chapter 4

Conclusion

This thesis has two primary contributions. First, it introduces a new control-flow graph (CFG) construction algorithm that can be used for scheduled assembly code. Second, it presents a compiler post-pass to reduce bit switching activity on the instruction bus of a processor.

Optimization after scheduling has become more common in recent years. However, previous CFG construction algorithms, which provide necessary analysis, often fail on scheduled code. Chapter 2 presents a construction algorithm that correctly handles branches in delay slots – the flaw in previous algorithms. Our algorithm adds an additional step to the traditional CFG construction algorithm. It requires no additional running time if branches are not located in delay slots. Otherwise, its complexity directly corresponds to the complexity of the CFG.

Chapter 3 introduces an optimization to reduce switching activity using the CFG construction algorithm to provide necessary data-flow analysis. It renames registers to reduce bit transitions between adjoining fetch packets on the instruction bus, and hopefully reduce power consumption as well. We implemented the algorithm on the Texas Instruments’ TMS320C6200 DSP. We achieved minor reductions in switching activity on the instruction bus, but no consequential reduction in power consumption was achieved. This led us to believe that switching activity on the instruction bus was not a major component for power consumption on the C6200.
Bibliography


[8] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. Constructing the


SIGPLAN '97 Conference on Programming Language Design and Implementation.


