THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

DECREASING PROCESS MEMORY REQUIREMENTS BY OVERLAPPING INSTRUCTIONS

By

Richard L. Bowman

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 1997

The members of the Committee approve the thesis of Richard L. Bowman defended on June X, 1997.

David B. Whalley
Professor Directing Thesis

Charles Kacmar
Committee Member

Theodore P. Baker
Committee Member

Approved:

R. C. Lacher, Chair, Department of Computer Science

CHAPTER 1

INTRODUCTION

In the not too distant past computer systems contained extremely small memories. In fact, computers found on many desktops today contain more megabytes of memory than past computers contained in disk space. Programmers faced with developing programs required to fit within these constrained memory environments were forced to design smaller, less functional systems. To address this constraint, programmers used several techniques to minimize the amount of storage needed by a program.

One of the first high-level programming languages, Fortran, introduced the concept of overlapping multiple variables within a common memory segment. Programmers using the EQUIVALENCE statement could easily introduce subtle errors by not carefully calculating the live ranges of variables being overlapped. Subsequent programming languages, such as C and Pascal, provided programmers with similar capabilities.

Overlays were another popular technique used to overcome the limited sizes of main memories. Overlaying allowed programs to exceed the physical memory limitations by loading sections of code into memory that was previously allocated to another section of the same executable. The development of an appropriate and efficient overlay structure was left up to the programmer.

With the advent of virtual memory and larger physical memories, programmers were alleviated with the burden of economizing the use of memory. Due to these larger memories, the typical size of application programs increased. However, as processor speeds continue to increase at a faster rate than main memory and disk access times, the performance of a memory hierarchy is becoming more significant. Reducing the size of a program on a machine with virtual memory can enhance paging performance. A page fault can easily require 700,000 to 6,000,000 cycles to resolve [1]. Thus, avoiding a single page fault by overlapping instructions can result in a significant performance improvement.

Processors are now being used in an increasing number of applications that are often embedded within some other type of system. These systems frequently have no virtual memory; therefore programmers must develop programs which fit entirely within main memory. Since embedded programmers are faced with such hard limits, even minute reductions in memory requirements may prove beneficial for the development of embedded systems.

This thesis describes a technique for reducing the amount of memory required for a program by overlapping instructions. Overlapping instructions was accomplished using three different transformations. First, cross jumping common trailing instruction sequences preceding unconditional transfers of control are presented in Chapter 4. Chapter 5 then explicates a cross jumping technique employed to overlap common sequences found prior to function calls. Chapter 6 follows with an explanation of a technique used to overlap relocatable code segments. All three of these techniques utilize a reordering algorithm. This reordering algorithm allows the execution of instructions to be reordered when the outcome is guaranteed to be safe and the reordering action is deemed beneficial. A disadvantage to reordering instructions can sometimes result in a slight increase in the dynamic code size, because reordering combined with overlapping can occasionally affect the compiler's ability to successfully fill delay slots.

CHAPTER 2

RELATED WORK

Most compiler optimizations focus on saving cpu time. Although some optimizations may incidentally save space, many actually increase the size of a program. These optimizations include function inlining [2] , loop unrolling [3] , scalar expansion [4] , avoiding unconditional jumps [5] , avoiding conditional branches [6] , and many others. Techniques that compress the size of a program without increasing the dynamic number of instructions would be appealing since the increase in process memory requirements from performing space-increasing optimizations could be partially offset.

There have been a few optimizations designed to save space. Code hoisting moves identical instructions from multiple blocks in different paths to a single dominating block [7]. Cross jumping moves identical instructions from multiple blocks in different paths to a singe post-dominating block [8]. Fraser *et. al.* [9] applied a general cross jumping and procedural abstraction technique on assembly code. They reported a 7% decrease in the number of static instructions when invoking their code compression technique on a range of Unix system utilities. Their technique did not use any data flow information and hence required that each common sequence of assembly instructions be contiguous; unlike the reordering technique described in this thesis. Additionally, each application of their procedural abstraction optimization required an increase in the number of executed instructions. While this abstraction resulted in static code size decreases, the additionally required call and return instructions could adversely affect the speed of the executable. In fact, one advance of a Sparc's register window can require several cycles since a window overflow causes a system trap that stores 16 registers to memory. In contrast, the techniques applied in this thesis for overlapping instructions rarely results in a dynamic instruction increase.[1] Whalley [10] investigated overlapping static data with other portions of static data and instructions. He reported a 7% decrease when overlapping static data with data, and over 22% when overlapping static data and instructions. Within the same report, Ratliff describes compression techniques that overlap run-time stack data after the invocation of procedural inlining. Ratliff diminished the number of bytes used in the run-time stack by an average of 6%.

[1] During each application of the techniques applied in this thesis the number of static instructions is decreased without the addition of any new instructions. However, the number of instructions executed may occasionally increase. This fluctuation is due to the effects of reordering instructions prior to filling delay slots.

CHAPTER 3

ENVIRONMENT

The overlapping algorithms described in this thesis were developed as extensions to the research compiler VPO (Very Portable Optimizer) [11]. The compiler system, VPO, is a highly portable optimizing compiler that is implemented using two passes. The front end, VPCC (Very Portable C Compiler), performs the initial pass over the C source files and emits register transfer lists (RTLs). An RTL represents a machine-independent representation of a machine-dependent operation. For example, the RTL

```
r[1] = r[3] + 4;
```

represents a register-to-register integer add on many machines. While any particular RTL is machine specific, the general form of RTLs is machine independent. The RTLs are then passed to the backend, VPO (Very Portable Optimizer). VPO reads in the basic blocks of RTLs, performs improving transformations, and translates them into the target machine's assembly language. VPO can be ported to new machines by changing a small amount of machine-dependent code within the VPO backend. The machine-dependent code includes grammar and related semantic actions used to recognize machine instructions and translate these instructions from RTLs to assembly code. The code expander, which simply expands the intermediate code to RTLs, must also be updated.

Figure 1 details the interaction between VPCC and VPO.

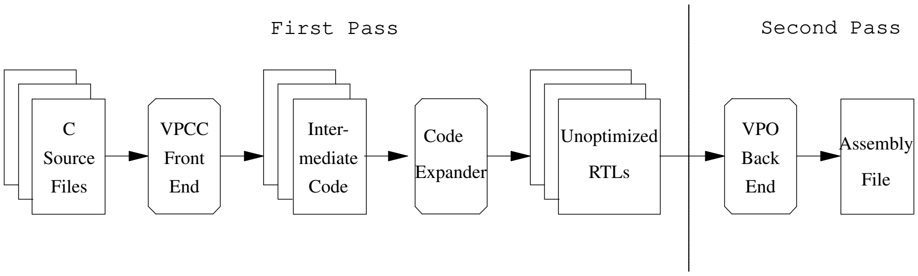

Figure 1: Original VPO Overview

The instruction overlapping algorithms presented in thesis operate interprocedurally.[2] The original version of the compiler, VPO, performed it's optimizations on a per-function basis. Figure 2 contains the order of optimizations performed by the original VPO. After optimizing each function VPO proceeds to translate the optimized RTLs into assembly instructions and dumps them out into an assembly file. Thus, VPO releases all of the information contained in memory for a function before processing the next function. In order to accommodate the interprocedural optimizations presented in this thesis, several modifications had to be made to VPO. The compiler had to be redesigned to allow the program representation for all of the individual source files comprising a program to be *live* in memory simultaneously. Previously, the optimizer was invoked only once for each source file.

[2] An interprocedural optimization performs transformations across functional boundaries. Most transformations performed by compilers are contained within the bounds of an individual function.

Original Version of VPO	Interprocedural VPO described in Thesis

Second Pass
 	
for each source file {
  for each function {
      branch chaining
      useless jump elimination
      dead code elimination
      eliminate unconditional jumps by reordering code
      instruction selection
      evaluation order determination
      global instruction selection
      register assignment
      jump minimization
      instruction selection
      eliminating jumps in loops
      do {
         register allocation
         instruction selection
         common subexpression elimination
         dead variable elimination
         code motion
         recurrences
         strength reduction
         induction variable elimination
         useless jump elimination
         cheaper instruction replacement
         instruction selection
         }
      while change;
      instruction scheduling
      fill delay slots
      useless jump elimination
      transform RTLs into the target machine's assembly language
  }
}
	Second Pass
	
	for each source file {
	  for each function {
	      branch chaining
	      useless jump elimination
	      dead code elimination
	      eliminate unconditional jumps by reordering code
	      instruction selection
	      evaluation order determination
	      global instruction selection
	      register assignment
	      jump minimization
	      instruction selection
	      eliminating jumps in loops
	      do {
	         register allocation
	         instruction selection
	         common subexpression elimination
	         dead variable elimination
	         code motion
	         recurrences
	         strength reduction
	         induction variable elimination
	         useless jump elimination
	         cheaper instruction replacement
	         instruction selection
	         }
	      while change;
	      instruction scheduling
	      useless jump elimination
	      dump out the optimized RTLs
	  }
	}
	
	Third Pass
	
	for each temporary file {
	  read in optimized RTLs
	  create unique labels
	  create unique global variable names
	}
	crossjump unconditional common targets w/ reordering
	crossjump calls to common functions w/ reordering
	overlap relocatable sections of code
	fill delay slots
	transform RTLs into the target machine's assembly language

Figure 2: Order of Compiler Optimizations in VPO

The decision was made to modify the optimizer to dump out the optimized RTLs in place of assembly code during the second pass, and then perform a separate, additional pass accommodating interprocedural optimizations as shown in Figure 2. Stemming from this change VPO was modified to create unique labels, unique global variables, and the filling of delay slots optimization was postponed until after all interprocedural optimizations had completed. The machine-dependent translation step was postponed and invoked as the final step during the third, interprocedural, pass. Figure 3 details the newly formed interaction between the first, second, and third passes.

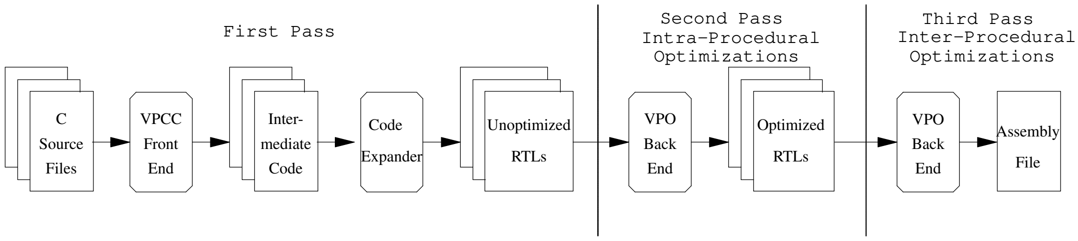

Figure 3: Overview of the Interprocedural version of VPO

The filling of delay slots was postponed until after interprocedural optimizations due to the juxtapositions that arise with reordering instructions. The filling of delay slots attempts to locate RTLs to place after instructions that transfer the flow of control, ie. branches, calls, and jumps. These instructions cause a stall on pipeline architectured machines. If a RTL can be safely moved from above the transfer of control instruction or from it's target and placed in the delay slot, then the pipeline stall is offset with the execution of the delay slot instruction. The action of placing the candidate instruction into the delay slot can sometimes adversely affect the effectiveness of the compression optimizations presented in this thesis.

CHAPTER 4

CROSS JUMPING RTLS PRECEDING JUMPS

The algorithm for performing cross jumping has many similarities to other implementations of this space-saving transformation. For each basic block in the program we examine its immediate predecessors. If any of the immediate predecessor blocks end with an unconditional jump, then they are considered candidates for applying the cross jumping transformation. When identical RTLs are found in common between a subset of candidate predecessor blocks and the Fall Through Block (FTB), the common RTLs can be overlapped.[3] A FTB is defined as a predecessor block which must not end in a transfer of control and who's execution path unconditionally *falls* into the next block. Overlapping occurs by removing the common set of RTLs from the predecessor block that is not the FTB and adjusting the ending unconditional jump's target to jump to its first common RTL within the appropriate FTB. A predecessor having no common RTLs would simply transfer control to its original target.

For example, consider the set of basic blocks in Figure 4. The cross jumping algorithm defines blocks A, B, and C as *unconditional* predecessor's of block E. Block D

[3] VPO's useless jump elimination transformation removes an unconditional jump if it's target is a block that has no FTB. The transformation safely accomplishes this by reordering the target block to follow the current block; thus effectively transforming the current block into a FTB.

is identified as the FTB. After comparing the RTLs contained in the three *unconditional* predecessor blocks with those in the FTB, four common RTLs are detected as overlappable, which are the bold faced RTLs contained in blocks **A** and **C**. During the overlapping action the FTB is split into three blocks, thus providing two new target labels. The common RTLs are deleted and block A and C's unconditional jump RTLs are updated.

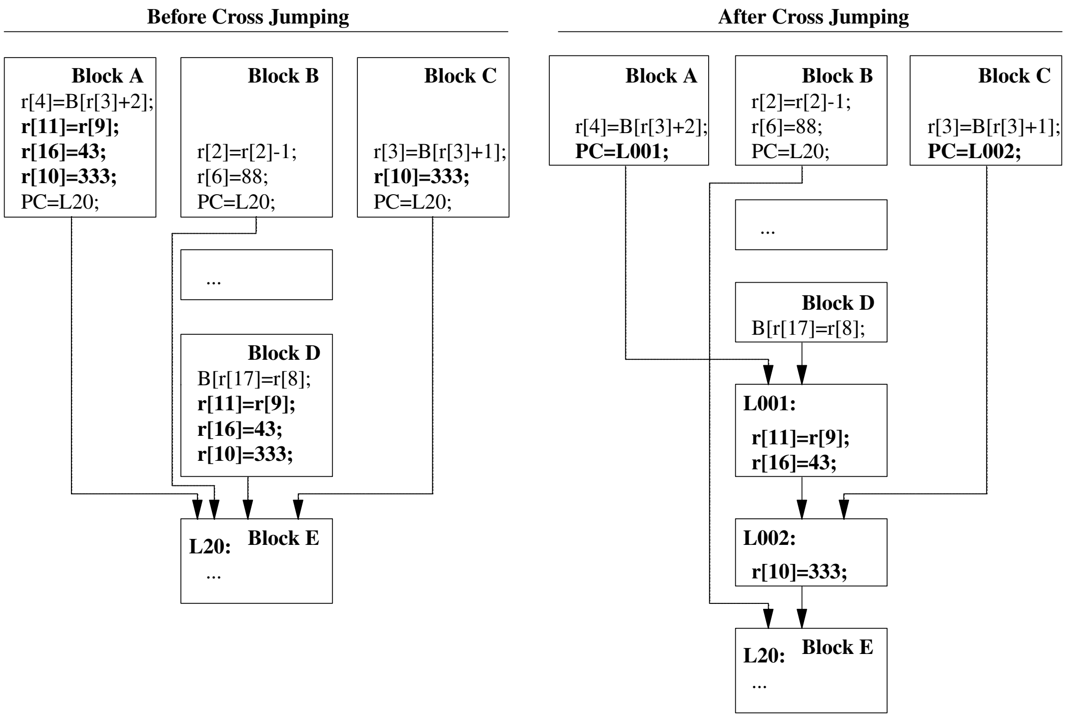

Figure 4: Cross Jumping Unconditional Jump Predecessors Example

Unlike most implementations of cross jumping, this version allows other RTLs to follow common RTLs in the predecessor blocks. Analysis is performed to determine if the RTLs in the predecessor blocks can be reordered so the common RTLs appear last. This analysis was accomplished by first detecting the registers that are set and used in each RTL within the program and efficiently representing these sets and uses as bit vectors. A search within one predecessor block for an RTL that appears in a different predecessor block is terminated when a register the RTL uses or sets is found to be set by a different RTL.

To facilitate fast comparisons of RTLs, a checksum value is calculated for each RTL as it is read in during the initialization of the interprocedural pass. When comparing two RTLs for equivalence, the checksums for each RTL are first compared. A full comparison of the two RTLs is only required when the checksums are identical.

The cross jumping algorithm used to overlap RTLs requires the following steps: 1) identification of candidate predecessor blocks, 2) the determination of the subset of candidate predecessor blocks that will produce the maximum compression, and 3) the application of cross jumping the RTLs found in common. The algorithm for cross jumping RTLs preceding unconditional jumps to a common target is given in Figure 5.

The *Cross_Jumping_Uncond* procedure visits each basic block in a program with the objective of identifying a target block of one or more unconditional jumps. Predecessor blocks meeting this criteria are considered candidates for applying the cross jumping transformation. The task of determining the subset of candidate predecessors, that will produce the maximum savings is fulfilled in the *Find_Best_Subset_Jumps* procedure. *Find_Best_Subset_Jumps* applies a bit-shifting algorithm in order to form each possible subset of blocks from the candidate set. If n is the number of candidate blocks, then there are 2^n-1 unique subsets containing at least one of the unconditional

```
PROCEDURE Cross_Jumping_Uncond()
    FOREACH basic block DO
        FTB:=Fall Through Block of the current block;
        DO
            uncond_pred_blks := list of predecessor blocks ending in an
                Unconditional Jump targeting the current block;
            IF (uncond_pred_blks NOT empty) THEN
                best_subset_pred_blks:=Find_Best_Subset_Jumps(uncond_pred_blks, FTB);
                IF (best_subset_pred_blks NOT empty) THEN
                    new_Fall-Through-Block:=create a new basic block
                              directly after FTB;
                    move all common RTLs from original FTB into
                        the new_Fall-Through-Block;
                    FOREACH block IN best_subset_pred_blks DO
                        remove all overlappable RTLs
                        update Unconditional Jump to target the new_Fall-Through-Block
                    END FOR
                    FTB:=new_Fall-Through-Block;
                END IF
            END IF
        WHILE changes
    END FOR
END PROCEDURE


PROCEDURE Find_Best_Subset_Jumps( uncond_pred_blks, FTB )
    n:=the number of blocks in the uncond_pred_blks list;
    temp_list:=allocate a list of n+2 pointers to basic blocks;
    FOR subset:=2ˆn-1; subset>0; subset-- DO
        temp_list[0]:=FTB;
        FOR y:=0,position:=1; y<n; y++ DO
            mask:=0;
            mask|=(unsigned 1)<<y;
                  IF ((mask & subset)==mask) THEN
                      temp_list[position]:=uncond_pred_blks[y];
                position++;
            END IF
        END FOR
        count:=Find_Common_RTLs_Reordering(temp_list);
        IF (count>previous_maximum) THEN
            save this current temp_list since it
            yields the highest compression
        END IF
    END FOR
    return the list of predecessor blocks which yield
    the highest compression
END PROCEDURE


PROCEDURE Find_Common_RTLs_Reordering( temp_list )
    DO
        FOREACH block in temp_list DO
            Build_Followlist(block);
        END FOR
        IF ( any RTLs are found in common among all followlists )THEN
            consider these as overlappable
            increment overlappable count
        END IF
    WHILE (more matches);
    return overlappable count
END PROCEDURE
```

Figure 5: Cross Jumping RTLs Preceding Unconditional Jumps Algorithm

predecessor blocks.[4] Followlists, which contain a set of RTLs that can be safely reordered downwards in a basic block due to having no conflicts with subsequent RTLs, are constructed for each of the blocks contained in each subset. The number of overlappable RTLs are then determined from these followlists. The optimal subset is determined after exhaustively checking each subset for overlappable RTLs. *Cross_Jumping_Uncond* cross jumps the RTLs found in common among this subset and continues searching for further cross jumping candidates throughout the remainder of the program.

Figure 6 displays an application of the cross jumping algorithm applied on a series of basic blocks extracted from the Unix utility, *grep*. Blocks 48, 49, 53, 55, 58, and 59 are predecessor blocks of block 60. Block 59 is further defined as the FTB. Blocks 49, 55, and 58 are considered candidates for applying the cross jumping transformation since they all unconditionally target block 60.

In order to determine which subset of the three candidate blocks will produce the most savings, 2^3-1 or 7 subsets must be checked for overlappable RTLs. Table 1 displays the bits used by the bit-shifting algorithm to fabricate each of the 7 unique subsets. A variable, name `subset,` is initialized to 7 during the first iteration, and decremented by 1 thereafter. The variable's bits are paired up with the candidate predecessor blocks. If a block's associated bit is set during an iteration, then the block is included in the current subset.

[4] A limiting number of 14 was substituted for *n* during testing.

Figure 6: Cross Jumping Unconditional Jump Predecessors in Grep

subset variable Integer Value	bit 2 block 55	bit 1 block 58	bit 0 block 49	current subset
7	1	1	1	55,58,49
6	1	1	0	55,58
5	1	0	1	55,49
4	1	0	0	55
3	0	1	1	58,49
2	0	1	0	58
1	0	0	1	49

Table 1: Subset Bit Representation

The constructed followlists for each subset's blocks are depicted in Table 2.[5] When the `subset` variable is equal to 1, only block 49 is contained in the current subset of candidate blocks. Block 49's leaderlist contains RTLs 11 and 9, and the FTB's contains RTLs 31 and 29. The matching algorithm detects equivalence between RTLs 9 and 29, and RTLs 11 and 31. Therefore, RTLs 9 and 11 are overlappable with RTLs 29 and 31, respectively. As long as at least one match is found among all of a subset's followlists, then the matching algorithm reiterates. RTLs detected as overlappable are omitted from future leaderlists. After completing four iterations RTLs 9 and 29, 11 and 31, 10 and 30, and 8 and 28 are determined to be overlappable. The *First Transformation* column in Figure 6 shows the set of basic blocks after the RTLs are crossjumped. The original FTB, block 59, is split into two blocks. Overlappable RTLs are moved down into the lower half, while non-overlappable RTLs remain in the upper half. Finally, block 49's copy of the overlappable RTLs is removed and RTL 12's target is updated to branch to label **L00002**. At this point the first transformation on block 60's unconditional

[5] While building the followlist, the *Build_Followlist* procedure attempts to traverse the preceding block. This upward traversal beyond the basic block boundary is only performed when the preceding block is the FTB of the current block. In other words, the preceding block's execution path is guaranteed to fall into the current block.

predecessors is complete.

Subset variable integer value	Uncond. Pred. Blks.	1st Iteration **follow_list**	Overlappable RTLs	2nd Iteration **follow_list**	Overlappable RTLs	3rd Iteration **follow_list**	Overlappable RTLs
7	49 55 58 **59***	11,9 26,25,21 31,29					
6	55 58 **59***	 26,25,21 31,29					
5	49 55 **59***	11,9 – 31,29					
4	55 **59***	 31,29					
3	49 58 **59***	11,9 26,25,21 31,29	9=21=29	11 26,25 31			
2	58 **59***	26,25,21 31,29	21=29	26,25 31			
1	49 **59***	11,9 31,29	9=29 11=31	10 30	10=30	8 28	8=28

*** Fall Thru Block**

Table 2: First Transformation's Followlists

As a result of successfully cross jumping RTLs from block 60's predecessors, the cross jumping algorithm attempts to perform a second transformation on block 60's unconditional predecessors. Block 49 is no longer a candidate block during this attempt, since it is not an unconditional predecessor of block 60. Table 3 displays the second transformation's followlists. Note that without the ability to reorder RTLs, RTL 21 would not have been cross jumped. The *Second Transformation* column in Figure 6 depicts the RTLs after the second transformation is applied.

Subset variable integer value	Uncond. Pred. Blks.	1st Iteration **follow_list**	Overlappable RTLs	2nd Iteration **follow_list**	Overlappable RTLs
3	55 58 **59b***	26,25,21 31,30			
2	58 **59b***	26,25,21 31,30	31=21	26,25 30	
1	55 **59b***	31,30			

*** Fall Thru Block**

Table 3: Second Transformation's Followlists

Following the second and final transformation on block 60's unconditional predecessors, a total of five RTLs are overlapped. When compiled on the SPARC architecture where every instruction is a word in size, twenty bytes of memory are compressed.

CHAPTER 5

CROSS JUMPING RTLS PRECEDING CALLS

Traditionally, cross jumping occurs only on instructions within a single function [8]. Often calls to the same function within a program pass similar arguments. The RTLs required to setup these arguments can often be overlapped. Therefore, the cross jumping algorithm presented in this chapter breaks the functional boundary by interprocedurally cross jumping RTLs from calls to the same function. As with the previous optimization, RTLs are reordered to increase the extent to which common RTLs can be overlapped.

The algorithm used to cross jump RTLs preceding calls requires the following steps: 1) identification of each function's set of predecessor blocks, 2) determination of the subset of predecessor blocks that will produce the maximum compression, and 3) the application of cross jumping the common RTLs. Since the concept of cross jumping common RTLs and these steps closely mirror the cross jumping unconditional predecessor optimization, much of the unconditional cross jumping algorithm and code is reused. Although missing from this optimization is the concept of a FTB; it has been replaced with the concept of a function preheader block. A function preheader block is defined as a basic block that precedes the header block, or top block, of a function and must unconditionally *fall* into the header block. Unlike the FTB, the preheader does not initially exist. When RTLs from call predecessor blocks are determined to be overlappable, the preheader block is created. A unique global name and the set of common RTLs is then placed in the preheader block. The set of common RTLs is removed from each cross jumpable predecessor block and the call RTL is updated to call the preheader block's global name. Call predecessors having no common RTLs continue to call the function's header block.

The step of identifying each function and it's predecessors is satisfied early during VPO's third pass, see Figure 2. As VPO reads in the partially optimized RTLs from the second pass it identifies call RTLs. These call RTLs are used to form a linked list of toplist structures. Each toplist structure contains pointers to the top block of the function being compiled by VPO and it's list of predecessor blocks (blocks that call it).[6]

Figure 7 displays the cross jumping algorithm involving function calls. The algorithm follows the same basic construct as cross jumping unconditional predecessor blocks algorithm presented in Chapter 4, with a few minor modifications. Since the preheader block does not initially exist, matching RTLs have to exist in two or more predecessor blocks in order to yield a savings when cross jumped. Recall when cross jumping unconditional predecessor blocks, the common RTLs had to exist in the FTB. Therefore, even if only a single RTL was found in common between one of the

[6] This optimization can only be applied to functions VPO compiles. Precompiled functions, such as those found in standard libraries, have already been compiled into object code. VPO can not create the required preheader block for such functions.

unconditional predecessor blocks and the FTB, cross jumping produced a savings.

```
PROCEDURE Cross_Jumping_Func_Calls()
    FOREACH function DO
        best_subset_pred_blks:=Find_Best_Subset_Calls(list of current
            function's predecessors );
        IF (best_subset_pred_blks NOT empty) THEN
            pre_header_blk:=create a new basic block
                      preceding the current function's top block
            new_global_name:=generate a new glocal name for the pre_header_blk;
            insert the new_global_name as the first RTL in the pre_header_blk
            insert one copy of the overlappable RTLs into the pre_header_blk
            FOREACH block IN best_subset_pred_blks DO
                remove all overlappable RTLs
                update Call RTL to call the new_global_name
            END FOR
        END IF
    END FOR
END PROCEDURE


PROCEDURE Find_Best_Subset_Calls( func_pred_blks )
    n:=the number of blocks in the func_pred_blks list;
    temp_list:=allocate a list of n+1 pointers to basic blocks;
    FOR subset:=2^n-1; subset>0; subset-- DO
        FOR y:=0,position:=0; y<n; y++ DO
            mask:=0;
            mask|=(unsigned 1)<<y;
                  IF ((mask & subset)==mask) THEN
                      temp_list[position]:=func_pred_blks[y];
                position++;
            END IF
        END FOR
        IF (position<2) THEN
            continue;
        END IF
        count:=Find_Common_RTLs_Reordering(temp_list);
        IF (count>previous_maximum) THEN
            save this current temp_list since it
            yields the highest compression
        END IF
    END FOR
    return the list of predecessor blocks which yield
    the highest compression
END PROCEDURE


PROCEDURE Find_Common_RTLs_Reordering( temp_list )
    DO
        FOREACH block in temp_list DO
            Build_Followlist(block);
        END FOR
        IF ( any RTLs are found in common among all followlists )THEN
            consider these as overlappable
            increment overlappable count
        END IF
    WHILE (more matches);
    return overlappable count
END PROCEDURE
```

Figure 7: Cross Jumping RTLs Preceding Function Calls Algorithm

A second modification was made in the *Build_Followlist* procedure, which builds the followlists for each subsets' blocks. As *Build_Followlist* fabricates each passed in block's followlist, it traverses upwards through the RTLs contained in the current block. Upon reaching the first RTL in the current block, *Build_Followlist* attempts to traverse into the preceding basic block. This upward traversal beyond the basic block boundary is only performed when the preceding block is the fall through block of the current block. This is the same constraint forced upon the *Build_Followlist* procedure by the cross jumping unconditional predecessor blocks optimization. However, two additional constraints are mandated by the cross jumping function calls optimization. The upward traversal must be aborted upon the detection of a *call* , *save*, or *restore* RTL. As a short explanation, consider this simple and unrealistic program:

```
#include <stdio.h>
void main()
{
  int y;

  scanf("%d\n",&y);      /* line 1 */
  printf("y=%d\n",y);    /* line 2 */
  y=foo(y);              /* line 3 */
  printf("y=%d\n",y);    /* line 4 */
  y=foo(y);              /* line 5 */
}
```

Lines 3 and 5 contain calls to the function `foo`. Both calls are preceded by identical calls to the `printf` function. Figure 8 displays the original RTLs, the RTLs following an illegal cross jump, and the RTLs after legally cross jumping a common RTL preceding the calls to `foo`. Blocks 3 and 5 contain the calls to the function

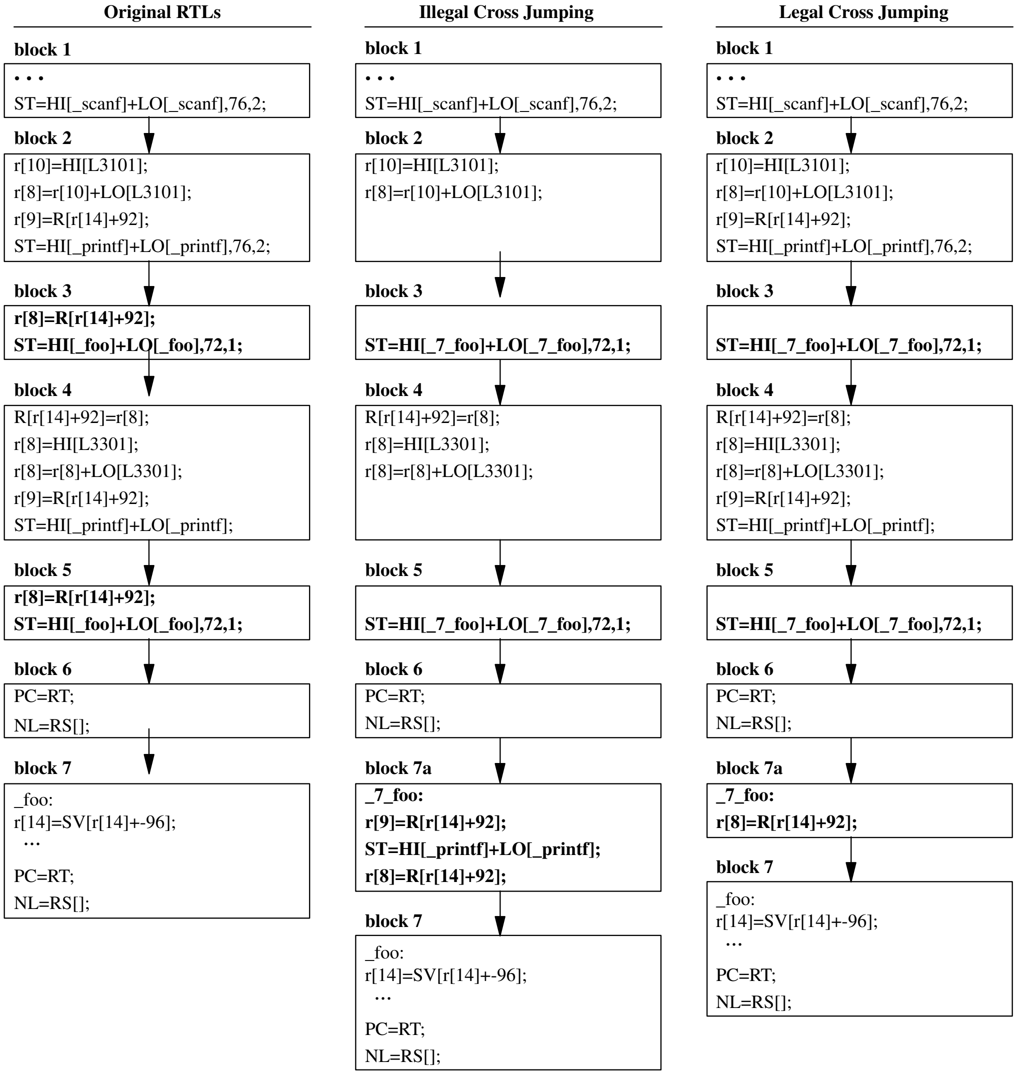

Figure 8: Followlist Example

`foo`.

Without the additional constraint not allowing the inclusion of call RTLs in a block's followlist, the cross jumping algorithm would produce the illegal set of RTLs listed in the *Illegal Cross Jumping* column. On the SPARC architecture a save instruction must be executed in between pairs of call instructions in order to save register values. When a call instruction is performed, register `%o7` receives the address of the call instruction. If another call instruction is executed prior to saving the return value stored in register `%o7` then, the return address will be overwritten. Therefore, the call to the `printf` function cannot be cross jumped down into the preheader block, block 7a. The *Legal Cross Jumping* column contains the set of RTLs after cross jumping the single RTL into the preheader block. In summary, the followlists cannot contain call, save, or restore RTLs when cross jumping RTLs preceding function calls.

Figure 9 shows an example of five calls to the function *pfnote* in the program *ctags*. Three RTLs are found to be the same preceding four of the calls. These three RTLs are overlapped before the original entry of the function. The call RTLs are updated to transfer control to a new global name preceding the overlapped RTLs. Other calls to the function can still transfer control to the original global name, or function name. Without inspecting the sets and uses of the RTLs to allow reordering of instructions, the first common RTL would not be detected and only six RTLs would be compressed instead of nine.

Before Cross Jumping

Call 1

...
r[9]=HI[L166];
r[10]=HI[_lineno];
r[8]=r[9]+LO[L166];
r[9]=M[r[10]+LO[_lineno]];
r[10]=1;
CALL _pfnote();

Call 2

...
CALL _pfnote();

Call 3

...
r[9]=HI[L318];
r[10]=HI[_lineno];
r[8]=r[9]+LO[L318];
r[9]=M[r[10]+LO[_lineno]];
r[10]=1;
CALL _pfnote();

Call 4

...
r[10]=HI[_lineno];
r[8]=r[14]+.16_nambuf;
r[9]=M[r[10]+LO[_lineno]];
r[10]=1;
CALL _pfnote();

Call 5

...
r[10]=HI[_lineno];
r[8]=r[14]+.20_nambuf;
r[9]=M[r[10]+LO[_lineno]];
r[10]=1;
CALL _pfnote();

function entry

_pfnote:
r[14]=SV[r[14]-1120];
...

After Cross Jumping

Call 1

...
r[9]=HI[L166];
r[8]=r[9]+LO[L166];
CALL _newlabel();

Call 2

...
CALL _pfnote();

Call 3

...
r[9]=HI[L318];
r[8]=r[9]+LO[L318];
CALL _newlabel();

Call 4

...
r[8]=r[14]+.16_nambuf;
CALL _newlabel();

Call 5

...
r[8]=r[14]+.20_nambuf;
CALL _newlabel();

Preheader Block

_newlabel:
r[10]=HI[_lineno];
r[9]=M[r[10]+LO[_lineno]];
r[10]=1;

function entry

_pfnote:
r[14]=SV[r[14]-1120];
...

Figure 9: Example of Cross Jumping with Function Calls

CHAPTER 6

OVERLAPPING RELOCATABLE CODE SEGMENTS

The instructions within a program may be divided into relocatable segments of code. Each segment starts with a basic block that is not fallen into from another block and ends with a block containing an unconditional transfer of control, which cannot be a call instruction.[7] This optimization compares each relocatable code segment against other relocatable segments in the program. If one code segment is entirely equivalent to another code segment or a subset of it, then the two code segments are overlapped. Branches and jumps that transfer control to the same relative location within the code segments are considered equivalent even though they reference different labels. VPO was changed to generate symbolic offsets in place of variable names while reading in the partially optimized RTLs generated during the second pass. This change facilitates more matches since two segments of code may reference different local variable names stored in identical stack offsets.

The target distance associated with branches or jumps to blocks within or from a relocated code segment may be significantly increased since the code segments being

[7] With the exception of the **exit** function. Since execution control never returns from a call to the standard library **exit** routine it is considered the end of a relocatable code segment.

overlapped may be in different functions. Figure 10 shows the format of a SPARC branch or jump instruction. The offset from the instruction following the branch is a sign-extended 22 bit value representing a displacement in words (i.e. SPARC instructions), not bytes. Thus, the distance between a branch or jump and its target can be over 2 million instructions. Abstracting relocatable code segments is not performed when the distance between code segments is too great, which rarely occurs for the SPARC architecture.

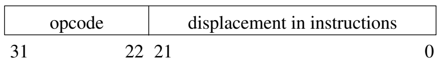

Figure 10: SPARC Branch and Jump Instruction Format

The Overlapping Relocatable Code Segment algorithm is depicted in Figure 11. The algorithm is divided into the following steps: 1) detecting each relocatable segment, 2) searching and identifying a matching relocatable segment, and 3) overlapping the common segments. By definition, the first block of a function begins a relocatable segment. In order to locate the last block of this first relocatable segment, the blocks are traversed in a downward direction. The last block is identified upon reaching either a block ending in an unconditional transfer of control or the last block contained in a function. After detecting this first relocatable segment the algorithm searches the remainder of the program endeavoring to find a matching segment. When control is returned to this first step the detection algorithm repeats, setting the block following the current relocatable segment as the first block of the next relocatable segment.

Once a relocatable segment is detected, the matching step searches the remainder of the program attempting to identify a matching code segment. The search starts with the first block in the program and continues downward until either a match has been found or the top block of the current relocatable segment has been reached. At such time when no match has been found within all of the blocks preceding the current relocatable segment, the algorithm begins searching through the blocks following the current relocatable segment. While visiting each block, the RTLs are examined in an upward fashion starting with the last RTL contained in the block. Each pair of RTLs is checked for equivalence. First, each RTL's checksum value is tested.[8] On success the RTLs are further compared for textual equivalence, which ensures that two RTLs are in fact, equivalent. On the other hand, when the pair's checksums differ then an additional test is made to determine if the RTLs are branch instructions to equivalent relative distances. Consider two otherwise identical code segments that reference a label within each respective code segment. They will reference different labels, but if they skip equal relative distances, then the two segments are identified as overlappable. Thus, two transfers of control are deemed equivalent when either they absolutely reference the same textual label or they relatively jump equal distances in the same direction. An additional constraint is applied to the relative jumps before being determined as equal, the target of each jump must be located within each jump RTL's respective segment.

[8] Equal checksum values does not assure that two RTLs are equivalent, but inequality guarantees that differences exist. The test for checksum equivalence requires a single integer comparison. In contrast, textual equivalence requires a maximum of **n** character comparisons, where **n** is equal to the number of characters in the shorter RTL. Therefore, the execution speed gains derived from using the checksum values outweight the overhead required to calculate each RTL's checksum value.

Otherwise, the overlapping of two relocatable segments would not be guaranteed to be safe, because the overlapper could not relocate a relative jump without also relocating the target of the jump. The matching algorithm continues to traverse upwards through the two segments' RTLs until either the entire current relocatable segment is matched or the current pair of RTLs is identified as being unmatchable.

```
PROCEDURE Overlap_Relocatable_Segments()
    top_block:=First basic block in the Program;
    FOREACH basic block DO
        IF (current basic block ends in an Unconditional Transfer of Control ||
           current basic block ends in a call to the function exit) THEN
           Match_Relocatable_Segments(top_block,current block);
           top_block:=the block directly following the current block;
        END IF
    END FOR
END PROCEDURE

PROCEDURE Match_RTLs(rtl1,rtl2)
    IF (rtl1->checksum_value == rtl2->checksum_value) THEN
        IF (rtl1 and rtl2 are textually equivalent) return TRUE;
    END IF
    IF (rtl1 and rtl2 are Jumps && only the target Label differs) THEN
        IF (rtl2's target is within the Relocatable Segment) THEN
            rtl1_distance:=calc_target_distance(rtl1);
            rtl2_distance:=calc_target_distance(rtl2);
            IF (rtl1_distance == rtl2_distance) return TRUE;
        END IF
    END IF
    return FALSE;
END PROCEDURE

PROCEDURE Perform_Relocatable_Overlap(top_block, last_block, new_top_block)
    current_blk:=top_block;
    new_target_blk:=new_top_block;
    done:=FALSE;
    WHILE (! done) DO
        IF (current_blk has a Label) THEN
            change all of it's predecessor blocks to target new_target_blk;
        END IF
        IF (current_blk == last_block) done:=TRUE;
        temp_blk:=current_blk;
        current_blk:=next block following current_blk;
        remove temp_blk from the program;
        new_target_blk:=next block following new_target_blk;
    END WHILE
END PROCEDURE
```

Figure 11a: Overlapping Relocatable Segments Algorithm

```
PROCEDURE Match_Relocatable_Segments(top_block, last_block)
    match_blk:=first block in program;
    WHILE (match_blk != top_block ) DO
        current_rtl:=last RTL in last_block;
        match_rtl:=last RTL in match_blk;
        WHILE (Match_RTLs(current_rtl, match_rtl)) DO
            # The entire relocatable segment has been matched
            IF (current_rtl == first RTL in top_block) THEN
                # Split the current_blk in order to generate a new target label
                IF (match_rtl != first RTL in it's block) THEN
                    newblock:=create a basic block following match_blk;
                    move all the RTLs from match_rtl to the end of
                    block containing match_rtl into newblock;
                    match_blk:=newblock;
                ELSE
                    match_blk:=block containing match_rtl;
                END IF
                Perform_Relocatable_Overlap(top_block, last_block, match_blk);
                return;
            END IF
            current_rtl:=RTL preceding current_rtl;
            match_rtl:=RTL preceding match_rtl;
        END WHILE
        match_blk:=block following match_blk;
    END WHILE
    match_blk:=block following last_block;
    WHILE (match_blk) DO
        current_rtl:=last RTL in last_block;
        match_rtl:=last RTL in match_blk;
        WHILE (Match_RTLs(current_rtl, matching_rtl)) DO
            # The entire relocatable segment has been matched
            IF (current_rtl == first RTL in top_block) THEN
                # Split the current_blk in order to generate a new target label
                IF (match_rtl != first RTL in it's block) THEN
                    newblock:=create a basic block following match_blk;
                    move all the RTLs from match_rtl to the end of
                    block containing match_rtl into newblock;
                    match_blk:=newblock;
                ELSE
                    match_blk:=block containing match_rtl;
                END IF
                Perform_Relocatable_Overlap(top_block, last_block, match_blk);
                return;
            END IF
            current_rtl:=RTL preceding current_rtl;
            match_rtl:=RTL preceding match_rtl;
       END WHILE
       match_blk:=next block following match_blk or NULL if no more blocks;
    END WHILE
END PROCEDURE
```

Figure 11b: Overlapping Relocatable Segments Algorithm *(continued)*

Prior to overlapping the relocatable segment when a matching segment has been located, a check is made to ascertain if the current segment requires division. The division is necessary when the current relocatable segment's first RTL does not overlap on the basic block boundary. An example is depicted in Figure 12. Assuming that the letters in each block represent RTLs and segments 1 and 2 are relocatable, then segment 1 can be overlapped with segment 2. The first block in segment 2 must be divided in order to generate a label as a target for segment 1's predecessors. Notice the divided block and the savings of five RTLs after overlapping.

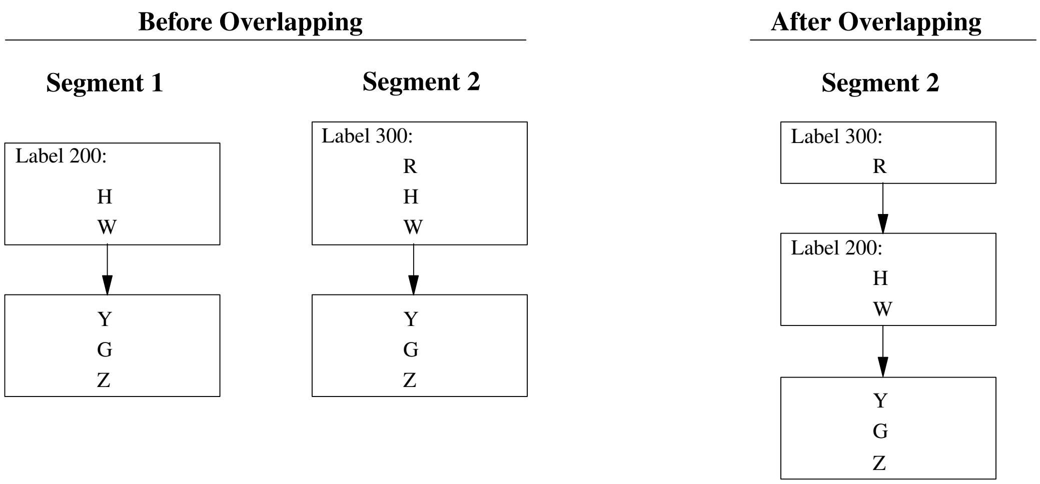

Figure 12: Overlap Relocatable Segment Requiring a Split

In order to complete the overlapping, the third step updates all the current relocatable segment's predecessors to target the matching segment, and removes the current relocatable segment's blocks. Figure 13 shows an example of one relocatable segment being overlapped with a portion of another in the function *yyparse* within the *lex* program.

Block 969 and block 970 are identified as a relocatable segment, since block 968 precedes block 969 and blocks 968 and 970 end with unconditional jumps.[9] Throughout this example we will refer to this relocatable segment as the current relocatable segment.

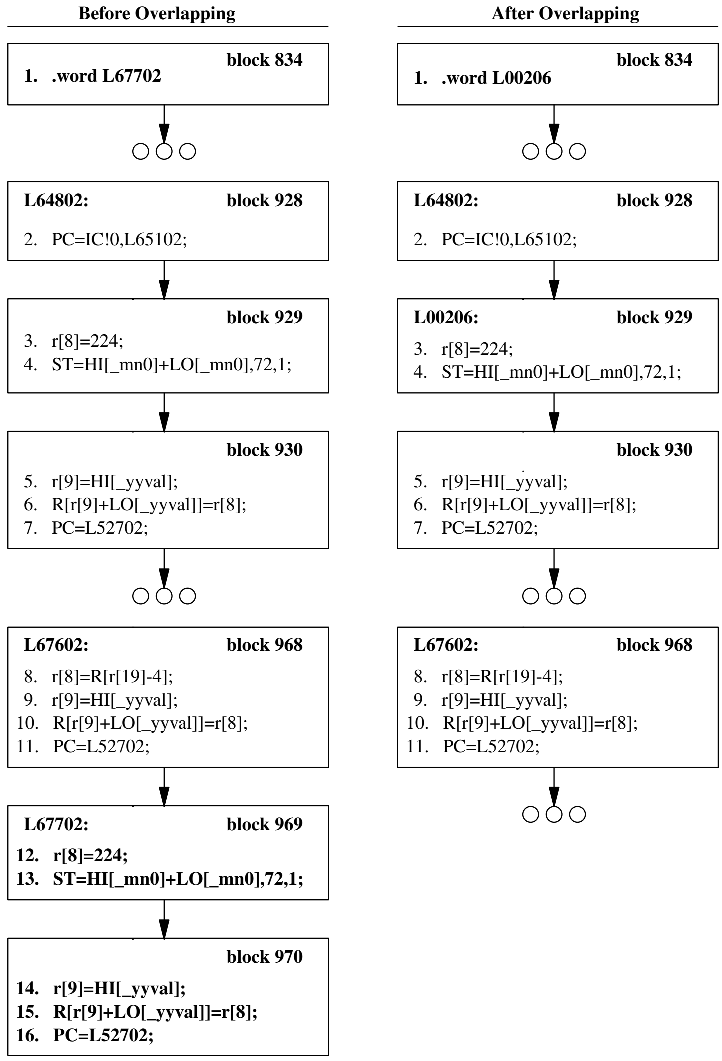

Figure 13: Overlap Relocatable Segment from Lex

[9] Block 969's other predecessor, block 834, contains a RTL used in an implementation of a jump table. The VPO code generator generates jump tables to implement case statements and series of If-Then-Else statements. For our concerns we will simply treat this RTL as a unconditional jump RTL.

The matching step receives the current relocatable segment and searches throughout the remainder of the program attempting to locate a matching code segment. Blocks 929 through 930 are identified as a code segment matching the current relocatable code segment. Since block 969 fully overlaps with block 929, block 929 requires no division. Notice that only part of the relocatable segment bounded by blocks 928 through 930 is used during the overlapping transformation.

In order to overlap the matching code segments, a new label is generated for block 929, which becomes the new target of the jump table entry found in block 834. Next, blocks 969 and 970, comprising the current relocatable segment, are removed. In summary, the relocatable segment contained in blocks 969 and 970 was overlapped with the ending part of the relocatable segment containing the blocks 928, 929, and 930. The overlapping example produces a savings of five RTLs or 20 bytes when implemented on the SPARC architecture.

CHAPTER 7

RESULTS

Table 4 shows the benchmarks, Unix utilities, and application programs that were used to test the extent that instruction overlapping could be applied. The column labeled *Number of Unoptimized Instructions in Bytes* contains the number of instructions in bytes required by each program prior to overlapping instructions. An additional set of test programs was created by performing procedure cloning prior to overlapping instructions. The number of bytes required for library functions is not included since library functions are dynamically linked and not compiled by VPO.

Program	Description	Number of Unoptimized Instructions in bytes	
		Without Cloning	With Procedure Cloning
cal	Calendar Generator	1376	1868
cmp	Comparison of Two Files	1576	1576
csplit	Split a File	7736	7988
ctags	C Tags Generator	9816	10308
dhrystone	Synthetic Benchmark	1956	2000
grep	Pattern Search	4048	4604
join	Relational Join on Files	2736	4280
lex	Scanner Generator	35724	44900
linpack	Linear Algebra Routines	10588	11464
mincost	VLSI Circuit Partitioning	4428	4500
sdiff	Side-by-Side File Display	7476	7972
tr	Translate Characters	1692	1692
tsp	Traveling Sales Person	4776	4788
whetstone	Arithmetic Operations	4812	4812
yacc	Parser Generator	31236	32800

Table 4: Test Programs

Tables 5, 6, and 7 show the number of bytes saved and the percentage decreased for each of the fifteen programs after applying the Cross Jumping Unconditional Predecessors, Cross Jumping Calls, and Overlapping Relocatable Segments optimizations respectively. Table 8 displays the combined instruction overlapping results when all three of the optimizations were invoked during a single compilation.

Notice that the combined results, in Table 8, are not the sum of the three individual optimizations. For example, the *join* program's individual overlapping results (without procedure cloning) were 28, 32, and 8 bytes. The sum of these values is greater than the 56 bytes shown in Table 8. This is attributed to the effects of instruction reordering. A single application of overlapping occurring in one of the instruction overlapping

Program	CrossJump Unconditional Jumps			
	Without Cloning		With Cloning	
	Bytes Saved	Percent Decrease	Bytes Saved	Percent Decrease
cal	0	0.00%	0	0.00%
cmp	0	0.00%	0	0.00%
csplit	40	0.52%	40	0.50%
ctags	12	0.12%	12	0.12%
dhrystone	4	0.20%	4	0.20%
grep	40	0.99%	40	0.86%
join	28	1.02%	16	0.37%
lex	316	0.88%	312	0.69%
linpack	8	0.08%	8	0.07%
mincost	40	0.90%	40	0.89%
sdiff	172	2.30%	188	2.36%
tr	16	0.95%	16	0.95%
tsp	8	0.17%	8	0.17%
whetstone	4	0.08%	4	0.08%
yacc	328	1.05%	328	1.00%
average	67.8	0.62%	67.73	0.55%

Table 5: Results from Cross Jumping Unconditionals

Program	Cross Jumping Calls			
	Without Cloning		With Cloning	
	Bytes Saved	Percent Decrease	Bytes Saved	Percent Decrease
cal	20	1.16%	40	2.14%
cmp	-4	-0.25%	-4	-0.25%
csplit	36	0.47%	28	0.35%
ctags	56	0.57%	12	0.12%
dhrystone	0	0.00%	0	0.00%
grep	8	0.20%	4	0.08%
join	32	1.17%	12	0.28%
lex	184	0.52%	156	0.35%
linpack	260	2.46%	212	1.85%
mincost	72	1.63%	116	2.58%
sdiff	92	1.23%	92	1.15%
tr	4	0.24%	4	0.24%
tsp	16	0.33%	12	0.25%
whetstone	180	3.74%	180	3.74%
yacc	68	0.22%	44	0.13%
average	68.5	0.91%	60.73	0.87%

Table 6: Results from Cross Jumping Calls

Program	Overlapping Relocatable Segments			
	Without Cloning		With Cloning	
	Bytes Saved	Percent Decrease	Bytes Saved	Percent Decrease
cal	0	0.00%	304	16.27%
cmp	0	0.00%	0	0.00%
csplit	56	0.72%	99	1.10%
ctags	20	0.20%	40	0.39%
dhrystone	28	1.43%	36	1.80%
grep	32	0.79%	32	0.70%
join	8	0.29%	8	0.18%
lex	232	0.65%	1244	2.77%
linpack	0	0.00%	212	1.85%
mincost	8	0.18%	8	0.18%
sdiff	20	0.27%	92	1.15%
tr	0	0.00%	0	0.00%
tsp	8	0.17%	12	0.25%
whetstone	0	0.00%	0	0.00%
yacc	120	0.38%	296	0.90%
average	35.5	0.37%	158.87	1.84%

Table 7: Results from Overlapping Relocatable Segments

Program	Overlapping Combined Results			
	Without Cloning		With Cloning	
	Bytes Saved	Percent Decrease	Bytes Saved	Percent Decrease
cal	20	1.16%	344	18.42%
cmp	-4	-0.25%	-4	-0.25%
csplit	132	1.71%	148	1.85%
ctags	80	0.81%	53	0.50%
dhrystone	32	1.64%	40	2.00%
grep	80	1.98%	76	1.65%
join	56	2.05%	40	0.93%
lex	732	2.05%	1700	3.79%
linpack	268	2.53%	330	1.92%
mincost	120	2.71%	164	3.64%
sdiff	272	3.64%	292	3.66%
tr	20	1.18%	20	1.18%
tsp	32	0.67%	28	0.59%
whetstone	184	3.82%	184	3.82%
yacc	476	1.52%	628	1.91%
average	166.9	1.81%	269.7	3.04%

Table 8: Overlapping Combined Results

optimizations may affect the other two optimizations' ability to overlap a series of instructions. Observe the negative savings listed for the *cmp* program when Cross Jumping Calls was invoked. Similarly, this increase in the number of static instructions is due to VPO's inability to fill delay slots after the Cross Jumping Calls optimization successfully overlapped 4 instructions.

These results give interesting insight into the ability to overlap instructions using the three techniques presented in this thesis. All fifteen of the test programs contain overlappable instructions.[10] Since most functions contained in the test suite of programs accept at least one argument, Cross Jumping Calls provided the largest average percentage decrease when compiling without procedure cloning. On the Sparc

[10] Recall that the *cmp* program contained 4 overlappable RTLs. The negative values in Table 2 arose from VPO's inability to fill delay slots fillable prior to overlapping instructions.

architecture registers 8 through 15 are used to pass values to functions. The setup RTLs used to initialize these registers were often overlapped by the Cross Jumping Calls transformation. The Overlapping Relocatable Segments optimization with procedure cloning invoked supplied the largest average savings.

The results show that a small average decrease in the memory requirements for the fifteen test programs was accomplished using the three overlapping optimizations presented in this thesis. Even though the average overall percentage decreased by only 1.81% (without procedure cloning) the savings may be the difference between a process fitting in an embedded system.

CHAPTER 8

FUTURE WORK

There are several areas to investigate that can potentially increase instruction overlapping. The juxtapositions that stem from VPO's inability to fill some delay slots after instruction overlapping could be corrected by a more sophisticated instruction scheduler, which could undo the effects of cross jumping when needed. Another area is to experiment with is register remapping. Register remapping could be used with instruction reordering to facilitate more matches among all three of the instruction overlapping optimizations presented in this thesis. Finally, VPO's intra-procedural optimizations, depicted in Figure 2, could be reinvoked after overlapping instructions. Overlapped instructions may provide new opportunities for applications of intra-procedural optimizations, *i.e.* branch chaining, useless jump elimination.

While the decrease in process memory requirements has been measured, a more detailed analysis of the effect that overlapping instructions has on performance is still needed. Preliminary measurements indicate that little impact was observed on primary instruction caches. While the caching performance of many programs changed, the average performance was almost identical. This indicates that the positioning of instructions probably has a greater caching impact than reductions in size. The effects on paging of concurrent processes also needs to be observed.

CHAPTER 9

CONCLUSION

This thesis presents three separate compiler optimizations for reducing process memory requirements by overlapping instructions. The first optimization attempts to cross jump common instructions preceding unconditional jumps to a common target. The second optimization attempts to cross jump common instructions preceding calls to a common function. Finally, the third optimization attempts to overlap relocatable segments of instructions with other relocatable segments. Unlike many other compiler optimizations, the ones presented in this thesis rarely add additionally executed instructions in order to facilitate more transformations. Analysis of these techniques shows that there are some opportunities for overlapping instructions. Although the average percentage decrease in instructions is small, even minute savings may benefit cache performance and applications targeting embedded systems.

REFERENCES

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition,* Morgan Kaufmann, San Francisco, CA (1996).

[2] J. Davidson and A. Holler, “A Study of a C Function Inliner,” *Software—Practice & Experience* **18**(8) pp. 775-790 (August 1988).

[3] J. W. Davidson and S. Jinturkar, “Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler,” *Proceedings of Compiler Construction Conference*, pp. 59-73 (April 1996).

[4] D. A. Padua, D. J. Kuck, and D. Lawrie, “High-Speed Multiprocessors and Compilation Techniques,” *IEEE Transactions on Computers* **29**(9) pp. 763-776 (September 1980).

[5] F. Mueller and D. B. Whalley, “Avoiding Unconditional Jumps by Code Replication,” *Proceedings of the SIGPLAN ’92 Conference on Programming Language Design and Implementation*, pp. 322-330 (June 1992).

[6] F. Mueller and D. B. Whalley, “Avoiding Conditional Branches by Code Replication,” *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation*, pp. 56-66 (June 1995).

[7] A. V. Aho and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, MA (1977).

[8] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler,* American Elsevier, New York, NY (1975).

[9] C. W. Fraser, E. W. Myers, and A. L. Wendt, “Analyzing and Compressing Assembly Code,” *Proceedings of the SIGPLAN ’84 Symposium on Compiler Construction*, pp. 117-121 (June 1984).

[10] Richard L. Bowman, Emily J. Ratliff, and David B. Whalley, “Decreasing Process Memory Requirements by Overlapping Program Portions,” Technical Report, Florida State University, Tallahassee, Florida (January 1997).

[11] M. E. Benitez and J. W. Davidson, “A Portable Global Optimizer and Linker,” *Proceedings of the SIGPLAN ’88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

BIOGRAPHICAL SKETCH

Richard Bowman was born in Fort Lauderdale, Florida in December 1971. In 1994, he received his Bachelor of Science from Florida State University with a major in Computer Science. After working for part of a year as a Computer Engineer he returned to Florida State University to begin working on his Master of Science. Currently, Richard lives in Melbourne, Florida where he is employed as a Unix System Administrator at Harris Corporation.

ACKNOWLEDGEMENTS

I wish to thank my major professor, Dr. David Whalley. Without his support and guidance this thesis would not have been a reality. I thank my two loving parents, Larry and Carol Bowman, for all of the opportunities they have giving me throughout my life. I would like to thank Julie Windey for continually encouraging me. And finally, I give thanks to my committee members Dr. Kacmar and Dr. Baker for their valuable assistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Most compiler optimizations focus on saving time, sometimes at the expense of increasing size. Yet processor speeds continue to increase at a faster rate than main memory and disk access times. Reducing the size of a program may result in improved memory hierarchy performance. Also, processors are now frequently being used in embedded systems that often have strict limitations on the size of programs it can execute. This thesis describes three separate techniques used to reduce process size and memory requirements by automatically overlapping instructions. Cross jumping instructions preceding unconditional jumps, cross jumping instructions preceding function calls, and overlapping relocatable segments are all optimizations shown to decrease process sizes for a variety of programs with minimal or no dynamic instruction increases.